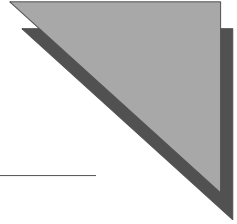

MicroStation®



*MDL™ Function
Reference Manual
Volumes 1 & 2*



DAA009530-1/0004

Trademarks

AccuDraw, Bentley, the “B” Bentley logo, Engineering Links, MDL, MicroStation, MicroStation GeoGraphics and SmartLine are registered trademarks of Bentley Systems, Incorporated.

Bentley SELECT is a service mark of Bentley Systems, Incorporated.

Adobe, the Adobe logo, Acrobat, the Acrobat logo, Distiller, Exchange, and PostScript are trademarks of Adobe Systems Incorporated.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

United States Patent Nos. 5,815,415 and 5,784,068.

Windows is a registered trademark and Win32s is a trademark of Microsoft Corporation.

Other brands and product names are the trademarks of their respective owners.

Copyrights

©1999 Bentley Systems, Incorporated.

MicroStation/J ©1998 Bentley Systems, Incorporated.

MicroStation 95 ©1995 Bentley Systems, Incorporated.

MicroStation Version 5 ©1993 Bentley Systems, Incorporated.

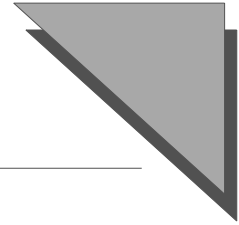
IGDS file formats ©1981-1988 Intergraph Corporation.

Intergraph Raster File Formats ©1993 Intergraph Corporation Used with permission.

Unpublished – rights reserved under the copyright laws of the United States and other countries.

All rights reserved.

Table of Contents



Command Organization with State Control Functions

Sample MDL Command Function	1-3
State Control Functions	1-4

User Interface

Window Management Functions.	2-1
Example	2-3
Window Drawing Functions.	2-19
Example	2-21
Window Docking Functions.	2-37
Resource Management Functions	2-40
Example	2-41
Binary Portable Resource Files	2-41
Machine format	2-42
Data definitions.	2-42
Generating data definitions for dynamically created resources	2-42
Resources with Alias Names.	2-43
Parse Functions	2-67
Example	2-68
Input Handling Functions.	2-73
Example	2-75
Input processors and sequencing.	2-75
MicroStation's main dispatching loop	2-76

Functions That Display Messages (Output)	2-95
Example.	2-96
C Expression Handling Functions	2-102
Example.	2-103

Dialog Box Manager

Menu Bar Item Functions	3-2
Text Pull-down Menu Functions	3-18
Related Structs	3-22
Option Pull-down Menu Functions	3-23
Option Button Item Functions	3-27
Icon and Icon Frame Functions	3-35
List Box Item Functions.	3-41
Text Item Functions	3-56
Scroll Bar Item Functions	3-64
Push Button Item Functions	3-66
Toggle Button Item Functions	3-69
Level Map Item Functions	3-71
Color Picker Item Functions	3-72
Completion Bar Functions.	3-73
Dialog Box General Functions.	3-78
Dialog Box Item Functions	3-95
Dialog Box Drawing Functions	3-112
Dialog Box Rectangle Drawing Functions	3-121

Dialog Box Font Functions.	3-127
Command Queuing Functions	3-130
Track Bar Window Functions	3-136
Busy Bar Window Functions	3-140
Miscellaneous Dialog Box Functions.	3-146
Tab Page List and Tab Page Functions	3-157
ComboBox Item Functions.	3-164
SpinBox Item Functions.	3-168
 System Functions	
System Functions.	4-2
Example	4-5
Configuration Variable Functions	4-38
Examples	4-39
 CAD Engine	
Element Creation Functions	5-2
Example	5-3
Element Information Extraction Functions.	5-17
Miscellaneous Element Functions	5-27
Example	5-28
Element Descriptor Functions.	5-45
Example	5-48
Boolean Functions.	5-81
Measurement Functions	5-84
Element Intersection Functions.	5-93

View Functions.	5-97
Example.	5-100
Sectioning and Hidden Line Viewing Functions	5-142
Auxiliary Coordinate System Functions	5-152
Current Transformation Functions	5-155
Example.	5-157
Transient Element Functions	5-163

Element Linkage Functions

Automatic data conversion using data definitions	6-1
--	-----

Element Location and Manipulation

Scan Functions	7-1
Element Location Functions.	7-11
Examples	7-13
Element Modification Functions.	7-23
Selection Set Processing Functions.	7-30
Example.	7-30
Fence Processing Functions.	7-34
Example.	7-34
Surface Creation Functions (3D Only)	7-37
Example.	7-37
Complex Chain Creation Functions	7-40
Element Clipping Functions.	7-42
Dynamic Buffer Functions.	7-45
Example.	7-45

Text Utility Functions**B-spline Functions**

B-spline Structures	9-1
B-spline Function Error Codes	9-5
B-spline Construction Functions	9-6
Example	9-9
B-spline Modification Functions	9-50
Example	9-52
B-spline Knot Functions	9-67
Example	9-67
B-spline Query Functions	9-74
Example	9-75

Dimension Functions

Dimension types and requirements	10-2
Example	10-4

Line Style Functions

Line Style Names	11-2
The line style engine	11-3
The line style component cache	11-3
The application interface	11-3
Resource Definitions	11-4
Line Style Functions	11-5
Creating Point Symbol Resources	11-13

Multi-Line Functions

Example.	12-2
------------------	------

Element Association Functions

Reference Files

Reference file attachment parameters.	14-2
Clipping boundaries	14-2
Transforming between reference and master file coordinates .	14-2

Cells

Cell Functions.	15-1
Example.	15-2
Shared Cell Functions	15-13

Patterning Functions

Basic Patterning Functions	16-1
Example.	16-1
Associative Pattern Functions	16-5

Color Management Functions

Color Codes, Color Tables, Color Maps and Display Colors . .	17-1
Exact Colors	17-2
Color Descriptors vs. Color Indexes.	17-3
Functions that return Color Descriptors:.	17-3
Functions that return Color Descriptors cast as integers: . .	17-3
Functions that have Color Descriptor arguments:	17-3
Functions that have Color Descriptor arguments cast as integers:	17-4

Arguments:	17-4
Arguments:	17-4
Color Table Functions	17-5
Color Descriptor Functions.	17-8
Color Palette Functions	17-17
RGB Conversion Functions.	17-23
Color Configuration Functions	17-28
Color Map Functions	17-31
 Rendering and Image Utilities	
Functions for Importing, Exporting and Manipulating Raster Images	18-1
Image File Formats	18-6
Sample Applications	18-9
Internal Image Format	18-9
Material Table Functions	18-82
Raster Reference Functions.	18-86
 Settings Functions	
Settings Manipulation Functions	19-1
Example	19-1
Level Functions	19-8
Example	19-10
 Math	
Rotation Matrix Functions.	20-1
Example	20-3

Transformation Matrix Functions	20-12
Example.	20-13
Vector Manipulation Functions	20-19
Example.	20-20

Non-graphical Data

Database Interface Functions.	21-2
Example.	21-4
SQL Functions	21-20
Example.	21-21
Database Settings Functions	21-32
Example.	21-33
Database Dialog Functions	21-38
Tag Functions.	21-52

External Program Communication Functions

External Program Communication Functions	22-1
Windows NT-specific notes.	22-3
Debugging external programs under Windows NT.	22-4
DOS-specific notes	22-4
Debugging protected mode external programs.	22-6
Using mdb.	22-7
Using 386DEBUG or 386 SRCBug.	22-7
Using External Programs Under UNIX	22-8
Portability Issues.	22-8
mdlExternal_terminateProgram	22-9
mdlExternal_messageReceive	22-9
Debugging external programs under UNIX	22-9
Function definitions	22-10

Digitizer Functions**Data Conversion**

Conversion Functions	24-1
Binary Portability Functions	24-8
How the Binary Portability Functions Work	24-9

File Manipulation

File Utility Functions	25-1
Example	25-2
Work File Functions.	25-13
File List Functions	25-18
Text File Functions	25-27
Low Level I/O Support	25-30
Interaction with Dynamic Link Modules	25-31
Record Locking	25-32

Constraint Problem Solving

Components	26-2
Defining and Solving a Constraint Problem.	26-3
Analyzing the Constraint Problem	26-3
Units and the Current Transform	26-4
Constraints Example	26-4
Construction Frame Functions	26-4
Constraint Parameter Functions	26-9
Constraint Functions	26-11
Equation Constraint Functions	26-29
Creating an equation is done in four steps:.	26-29

Attachment Functions	26-32
Constraint Model Functions	26-33
Constraint Object Functions	26-43
Solver Variable Functions	26-46

MicroStation for Windows NT, Clipboard and DDE Interface

Overview of the Windows NT Interface	27-1
Dynamic Data Exchange (DDE)	27-1
What is DDE?	27-1
Sample Programs	27-2
The Clipboard	27-2
MDL Clipboard Interface (Pasting)	27-2
Processing Example	27-3
Clipboard programming guide	27-4
Memory Management Functions	27-7
Windows NT DDE Functions	27-10
wType Details	27-40
File Drag and Drop Utility Functions	27-43
Clipboard Functions	27-45
User's View	27-45
Programmer's View	27-45

Utilities

String List Manager	28-1
String lists as resources	28-2
String Functions	28-19
License Management Functions	28-32
Dynamic Array Functions	28-36
Help Functions	28-42

Function Key Functions	28-44
Undo Functions	28-47
Tutorial Functions	28-52
Example	28-52
Plotting Functions	28-54
BASIC Interface Functions	28-56
Wide Character String Functions	28-60
Examples	28-61
User Preference Functions	28-68
Miscellaneous Functions	28-73
Example	28-73
 HTML Authoring Tool Library	
HTML Authoring Functions	29-1
 Example Source Code	
grphtest.mc	30-1
rasticon.mc	30-10
resmover.mc	30-68
parse.mc	30-94
chngtxt.mc	30-99
input.mc	30-118
output.mc	30-125
cexpr.mc	30-130
calculat.mc	30-134
system.mc	30-135
excfgar.mc	30-144
create.mc	30-151
element.mc	30-162
elemdscri.mc	30-168
doc.mc	30-168
view.mc	30-169
trumpet.mc	30-178
locate.mc	30-192

fence.mc	30-198
dynamic.mc	30-209
bspline.mc	30-218
mline.mc	30-244
lvlnames.mc	30-252
dynamic.mc	30-288
math.mc	30-295
trumpet	30-295
database.mc	30-295
file.mc	30-306
tutorial.mc	30-310
create.mc	30-314
extract.mc	30-323
misc.mc	30-331

1

Command Organization with State Control Functions

MicroStation's state control functions control MicroStation's **command state**. For MDL functions to work like MicroStation functions, they must control their command states the way MicroStation commands control their command states.

MicroStation's command state controls the interaction of different types of commands. For example, state command interaction enables the windowing command to be used during element placement.

The command state determines how MicroStation manages user events such as data points and key-ins. It also determines how dynamics are controlled.

Any MicroStation command can be classified as a **primitive** command, **view** command, **immediate** command or **utility** command.

Primitive commands create, modify, or delete elements. PLACE LINE and DELETE ELEMENT are examples of primitive commands. Any primitive command must call `mdlState_startPrimitive`, `mdlState_startModifyCommand` or `mdlState_startFenceCommand` during its initialization.

View commands modify or update views. ZOOM and WINDOW AREA are examples of view commands. Any view command must call `mdlState_startViewCommand` during its initialization.

Immediate commands modify a setting. LOCK SNAP KEYPOINT is an example of an immediate command. Immediate commands do not call state functions.

Utility commands perform non-interactive functions. COMPRESS is an example of a utility command. Utility commands must call `mdlState_startDefaultCommand` when they finish executing.

Starting a primitive command stops any other commands. A primitive command does not terminate until another primitive command replaces it. Starting a view command can suspend a primitive or replace the current view command. MicroStation terminates an active view command when a primitive command or another view command starts. Also, a view command must terminate itself by calling `mdlState_exitViewCommand` if the user enters Reset. When a view command terminates itself, the suspended primitive

command is resumed. Running an immediate command does not affect the state of primitive and view commands.

MicroStation keeps pointers to functions to handle various user events. The MicroStation command handler calls these functions. The functions for all normal MicroStation commands are called this way. MicroStation keeps a set of pointers for view commands and a set for primitive commands. When a view command starts, the pointers for the primitive commands are not modified. MicroStation begins using the function pointers for view commands. When the view command exits, MicroStation resumes using the function pointers for primitive commands. This capability allows primitive commands to be implemented without being suspended by view commands.

MicroStation allows MDL applications to be called with state function pointers. MDL applications can specify handlers to be called when the user enters a data point, Reset, or key-in. They can also specify functions for restarting the current command.

Through MicroStation's function pointers, **dynamics** are available to MDL applications. The term dynamics refers to the rubber-banding displayed while a user creates or modifies an element. **Simple dynamics** and **complex dynamics** are available. Simple dynamics is used when a uniform operation can be performed on all elements in dynamics. Complex dynamics is used when the dynamic display requires that a different manipulation be performed on certain elements in dynamics. See *userState_dynamicUpdate* and *userState_complexDynamicUpdate* for more information. (View commands cannot use dynamics functions).

Many MDL state functions accept prompt and function numbers as parameters. These numbers refer to strings in *MessageList* resources in the application's resource file. The resource file that contains the strings must remain open the entire time the strings can be needed.

When MicroStation tries to load the string, it needs the string number and resource ID of the *MessageList* that contains the string. The string number is the prompt number or function number specified in the *mdlState_start...* functions. By default, *MessageList*'s resource ID is 0. However, the *mdlState_registerStringIds* function can specify alternate resource IDs.

The function number determines the command name displayed in the MicroStation command window while the command is active. It also determines the command name that the UNDO command displays in the message, "command undone."

The prompt number typically specifies the prompt to display when the command is started.

Many examples distributed with MicroStation illustrate state function use. The *chngtxt* example illustrates how modify and fence commands can use the state functions.

The plashape example illustrates how element creation can use the state functions. It also illustrates how simple and complex dynamics work. Both examples illustrate how message lists should be used. The file state.mc illustrates the calling sequences of functions not illustrated in chngtxt or plashape.

Sample MDL Command Function

```
cmdName void commandInitialization
(
char      *unparsedP      /* => arguments entered by user */
)
cmdNumber CMD_ ..., CMD_ ...
```

This is the first function called when a command is initiated. The function does not have to be called `commandInitialization`; the function name is insignificant to MicroStation.

The standard methods of activating a command in MicroStation can be used to activate this function. If the `cmdNumber` reserved word is present, the user may either key in the command, or, if a palette is hooked up with an icon representing this command, the user may select the command off the palette. If the `cmdName` reserved word is present, the user may also initiate a command by keying in:

MDL COMMAND <commandInitialization>

substituting the function name for the one above. It is recommended that the programmer only use `cmdName` for private testing and not as a long term means of initiating a command.

The two reserved words `cmdName` and `cmdNumber` are optional, however one must be present in order to initiate the command. Use of `cmdNumber` requires that a command table be present. See the “Command Tables” section of the *MDL Programmer's Guide* for more information.



Multiple command numbers can call a single function.



These functions, because of the special reserved words, `cmdName` and `cmdNumber`, vary from ANSI C. However, when prototyping this function in your source code, the prototype will be a standard ANSI format.

MDL ignores the return value of command initialization functions.

State Control Functions

The following table lists state control functions:

Function	Used to
<code>mdlState_clear</code>	reset MicroStation to its initial state when no command is active.
<code>mdlState_startDefaultCommand</code>	start the default command when the current command is finished.
<code>mdlState_checkSingleShot</code>	determine if MicroStation is operating in single-shot mode.
<code>mdlState_startPrimitive</code>	start a primitive command.
<code>mdlState_startPrimitiveAndSetPopupMenu</code>	start primitive command and set pop-up menu simultaneously.
<code>mdlState_startModifyCommand</code>	start a modify command.
<code>mdlState_startViewCommand</code>	start a view command.
<code>mdlState_exitViewCommand</code>	exit a view command.
<code>mdlState_restartCurrentCommand</code>	restart a primitive command.
<code>mdlState_startFenceCommand</code>	perform the standard initialization for primitive commands.
<code>mdlState_registerStringIds</code>	specify resource IDs for the prompt and command name message lists.
<code>mdlState_setKeyinPrompt</code>	specify a string to be used as the prompt in the key-in area.
<code>mdlState_dynamicUpdate</code>	specify an MDL function to be used with simple dynamics.
<code>mdlState_setFunction</code>	designate an MDL function to be called to handle certain user events.
<code>mdlState_setAccuDrawContext</code>	optimize AccuDraw's behavior for the context of the current drawing modification tool.

The following table lists state control user functions. These are user-supplied functions that MDL calls when certain events occur within MicroStation. The application programmer defines user function names. (The names in the table are merely

illustrations). The functions are designated to MDL through function arguments pointing to MDL routines.

Function	MicroStation calls when
userState_commandCleanup	a command is being terminated by a new command.
userState_datapoint	the user enters a data point.
userState_reset	the user enters a Reset.
userState_keyin	the user keys in a string, i.e., enters a key-in.
userState_dynamicUpdate	a simple dynamics function is active and the user moves the cursor.
userState_complexDynamicUpdate	a complex dynamics function is active and the user moves the cursor.
userState_show	an element has been identified but has not yet been accepted.
userState_clean	an element was identified and has now either been accepted or rejected. This function is intended to reverse the effect of a userState_show function.
userState_fenceContent	MicroStation processes each element in a fence.
userState_fenceOutline	the active fence is to be modified.

mdlState_clear

```
void mdlState_clear();
```

Description The mdlState_clear function resets MicroStation's state to "no command active." After mdlState_clear is called, all input that does not start a command is ignored. The cursor is reset to the initial state and the prompt in the key-in field is set back to the default prompt.

Applications usually call mdlState_startDefaultCommand rather than this function to return MicroStation to the default state. The default command is selected in the User Preferences dialog box.

Returns The mdlState_clear function is of type void. It returns no value.

See Also mdlState_startDefaultCommand, mdlState_checkSingleShot.

mdlState_startDefaultCommand

```
void mdlState_startDefaultCommand();
```

Description The mdlState_startDefaultCommand function starts the default command. It also terminates an MDL primitive or utility command. Generally, primitive commands do

not terminate until another command is started. Primitive commands call `mdlState_checkSingleShot` and utility commands call `mdlState_startDefaultCommand`.

The MicroStation COMPRESS command calls `mdlState_startDefaultCommand` after compressing a file.

Returns The `mdlState_startDefaultCommand` function is `void`. It returns no value.

mdlState_checkSingleShot

```
int mdlState_checkSingleShot();
```

Description The `mdlState_checkSingleShot` function determines whether the operator wants the command to operate in single-shot mode. When MicroStation runs in single-shot mode, the command returns to the default command when the current command completes. Otherwise, the current command remains active.

Most primitive commands should call `mdlState_checkSingleShot` when the command is complete.

If `mdlState_checkSingleShot` determines that single shot mode is active, it calls `mdlState_startDefaultCommand`.

Returns `mdlState_checkSingleShot` returns `TRUE` if single-shot mode is being used.

See Also `mdlState_restartCurrentCommand`.

mdlState_startPrimitive

```
void mdlState_startPrimitive
(
    MdlFunctionP    dataFunc,      /* => data point function or NULL */
    MdlFunctionP    resetFunc,    /* => reset function or NULL */
    int             funcName,     /* => index into message list */
    int             promptNum     /* => index into message list */
);
```

Description The `mdlState_startPrimitive` function starts a primitive command. It is called only for element creation commands. For other commands it is called when the application calls `mdlState_startModifyCommand` or `mdlState_startFenceCommand`.

This function terminates active commands. It also clears highlighting remaining from the previous command. If a temporary element remains from a previous command, this function erases it. It resets snap data, clears all state data, and sets up the state data using *dataFunc*, *resetFunc*, *funcName* and *promptNum*.

dataFunc specifies a function to be called if a data point is entered. *resetFunc* specifies a function to be called if Reset is entered. MicroStation also uses the function specified by *resetFunc* as the restart function (see

mdlState_restartCurrentCommand). The values of these arguments can be valid MDL function pointers or NULL.

funcName and *promptNum* are integers that specify strings in MessageList resources in the application's resource file. The strings corresponding to these numbers display as the current command name and the initial prompt. If these strings are zero, nothing displays for the command or prompt. Also, the undo logic records the *funcName* value and uses it to display the command name if the Undo command is used to undo the command.

Returns The mdlState_startPrimitive function is of type void. It returns no value.

See Also mdlState_startPrimitiveAndSetPopupMenu, mdlState_startModifyCommand, mdlState_startFenceCommand, mdlState_startViewCommand, mdlState_setFunction, mdlState_registerStringIds, user functions from this chapter.

mdlState_startPrimitiveAndSetPopupMenu

```
DItem_PullDownList *mdlState_startPrimitiveAndSetPopupMenu
(
    MdIFunctionP      dataFunc,      /* => data point function or NULL */
    MdIFunctionP      resetFunc,     /* => reset function or NULL */
    int               funcName,      /* => index into message list */
    int               prompt,        /* => index into message list */
    ULONG menuType, /* => menu type to insert into view popup, 0=Text*/
    long              menuId,        /* => id of menu to insert */
    void              *ownerMD       /* => usually NULL */
);
```

Description The mdlState_startPrimitiveAndSetPopupMenu function starts a primitive command and sets a pop-up menu simultaneously. See the discussion of mdlState_startPrimitive for the use of the first four parameters.

The menu to add is identified by *menuType* and *menuId*, belonging to the MDL task specified by *ownerMD*. If *ownerMD* is NULL, as is usually the case, the current task is the owner.

Returns mdlState_startPrimitiveAndSetPopupMenu returns a pointer to the inserted menu, or NULL if there is an error.

See Also mdlView_setPopupMenu, mdlState_startPrimitive, mdlWindow_isPopUp.

mdlState_startModifyCommand

```

void mdlState_startModifyCommand
(
    MdlFunctionP    resetFunc,          /* => function in MDL program */
    MdlFunctionP    acceptFunc,        /* => function in MDL program */
    MdlFunctionP    dynamicFunc,       /* => function in MDL program */
    MdlFunctionP    showFunc,          /* => function in MDL program */
    MdlFunctionP    cleanFunc,         /* => function in MDL program */
    int             funcName,          /* => index into message list */
    int             acceptPromptNum,    /* => index into message list */
    int             useSelection,       /* => TRUE or FALSE */
    int             needPoints          /* => 0, 1, or 2 */
);

```

Description The mdlState_startModifyCommand function starts a command that will locate and modify elements. MicroStation's element location logic performs a majority of modify command functionality. The mdlState_startModifyCommand ensures that all modification functions operate similarly.

Users can employ two methods to identify elements for the element modification commands. The command can prompt the user to identify an element (or set of elements) and wait until the user identifies and then accepts the elements. Alternatively, **selection sets** can be used. Selection sets are a group of elements that the user identifies before activating the element modification command. Some modification commands (such as delete, copy, or move element) understand selection sets, whereas others (such as extend line and fillet) do not. When writing element modification commands, you must decide whether you will accept selection sets as element identifiers.

The mdlState_startModifyCommand function automatically establishes the proper sequence for modification commands so the user interface is the same. It uses the following logic:

```

{
    if (useSelection is TRUE and a selection set is active)
    {
        if (needsPoints is 0)
        {
            /* command does not need additional data points */
            call the acceptP function;
        }
        else if (needsPoints is 1)
        {
            /* command needs one additional data point */
            call mdlState_startPrimitive with dataP as the data
            point function and mdlState_restartCurrentCommand as
            the reset function;
            load dynamic buffer with entire selection set;
        }
    }
}

```

```

    }
    else if (needsPoints is 2)
    {
        /* cmd needs more than one additional data pt */
        save accept function, dynamic function, and accept prompt;
        put out "enter first point (selection set)" prompt;
        call mdlState_startPrimitive with an internal function
        that gets the anchor point as the data point function;
    }
}
else
{
    /* useSelection is FALSE, or no active selection set */
    clear selection set if one is active;
    call mdlState_startPrimitive with mdlLocate_identifyElement as
    the data point function;
    put out "Identify element" prompt;
    save info specified as arguments (prompts & function pointers);
    set cursor to the "locate element" cursor;
}
}

```

The application can control the Element Location routine operation by filtering elements in a *userLocate_elementFilter* user function.

The function specified by *acceptFunc* is called when the user identifies and accepts the element to be modified (either with a selection set or through the Element Location logic). Accept functions for element modification commands generally use the modify commands to change the specified element(s). When the accept function is called, the point that the user gives to identify the element is in *statedata.pointstack* [0], passed as the first argument to the data point function. Information about the point's projection onto the accepted element can be obtained from the *mdlLocate_getProjectedPoint* function. The design file's current file position gives the element's file position. You can obtain the current file position by calling *mdlElement_getFilePos*, specifying *FILEPOS_CURRENT* as the first parameter.

cleanFunc specifies a function that cleans up the activity of *showFunc*. This function is called if the user enters Reset after an element is located. *NULL* is commonly specified for *showFunc* and *cleanFunc*. As an example, MicroStation uses *clean* and *show* functions for the fillet command. See *userState_show* and *userState_clean* for more information on these functions.

funcName and *promptNum* specify strings in *MessageList* resources in the application's resource file.

A non-zero *useSel* means that the command works with selection sets. When a selection set is used, MicroStation's modify logic skips the element identification and acceptance logic since the elements have been identified. If *useSel* is non-zero and at least one element is selected, *needPoints* tells MicroStation the number of additional data points required.

If *needPoints* is 0, MicroStation immediately calls the accept function. For example, the MicroStation DELETE ELEMENT command does not need additional data points. Commands use 0 for *needPoints* if they need only to know the elements involved in the operation.

If *needPoints* is 1, MicroStation calls *mdlState_startPrimitive*, specifying *acceptFunc* as the data point function and *acceptPromptNum* as the prompt. The function specified by *dynamicFunc* is set up as the dynamic function if *dynamicFunc* is non-NULL. For example, the MicroStation SCALE ELEMENT command needs one additional data point.

If *needPoints* is 2, MicroStation first calls *mdlState_startPrimitive*, specifying an internal MicroStation function as the data point function. This function puts a data point in *statedata.pointstack* [0].

Once the data point is entered, MicroStation uses *acceptFunc* as the new data point function. It also displays the prompt specified by *acceptPromptNum*. For example, the MicroStation MOVE ELEMENT command needs two additional data points.

When a modify command completes, it calls *mdlLocate_restart*. This function calls *mdlState_checkSingleShot*.

Depending on the return value of *mdlState_checkSingleShot*, *mdlLocate_restart* restarts either the modify or default command.

Returns The *mdlState_startModifyCommand* function is of type *void*. It returns no value.

See Also "Element Location Functions" (*mdlLocate_...*), "Selection Set Processing Functions," "Element Modification Functions" (*mdlModify_...*), user functions from this chapter.

mdlState_startViewCommand

```
void mdlState_startViewCommand
(
    MdlFunctionP    dataFunc,      /* => function in MDL program */
    int             funcName,      /* => index into a message list */
    int             promptNum      /* => index into a message list */
);
```

Description The *mdlState_startViewCommand* function starts a view command.

dataFunc specifies a function to be called when a data point is entered.

funcName is an integer specifying a command name.

promptNum is an integer specifying a prompt. These are both string numbers for elements of `MessageList` resources in the application's resource file. If 0 is used for either parameter, the parameter is ignored.

Returns The `mdlState_startViewCommand` function is of type `void`. It returns no value.

See Also View Functions, `mdlState_setFunction`.

mdlState_exitViewCommand

```
void mdlState_exitViewCommand
(
    int      messageDesired      /* => TRUE or FALSE */
);
```

Description The `mdlState_exitViewCommand` function exits a view command.

If *messageDesired* is non-zero, MicroStation displays “View command exited” in the prompt field.

Returns The `mdlState_exitViewCommand` function is `void`. It returns no value.

See Also `mdlState_startViewCommand`.

mdlState_restartCurrentCommand

```
void mdlState_restartCurrentCommand();
```

Description `mdlState_restartCurrentCommand` restarts a primitive command. It first calls `mdlState_checkSingleShot`. If single-shot mode is not active, it clears the point stack, clears the *dgnBuf* variable, and calls the current command's restart function. If single-shot mode is active, `mdlState_restartCurrentCommand` starts the default command.

MDL applications should call `mdlState_restartCurrentCommand` only from primitive commands.

A command indirectly specifies the address of the restart function when it calls an `mdlState_start...` function. These functions call `mdlState_startPrimitive`. `mdlState_startPrimitive` sets the value of the restart function pointer to the value specified as the reset function pointer. `mdlState_setFunction` never changes this value.

Returns The `mdlState_restartCurrentCommand` function is of type `void`. It returns no value.

See Also `mdlState_checkSingleShot`.

mdlState_startFenceCommand

```

void mdlState_startFenceCommand
(
    MdlFunctionP    contentFunc,      /* => function in MDL program */
    MdlFunctionP    outlineFunc,     /* => function in MDL program */
    MdlFunctionP    dataFunc,        /* => function in MDL program */
    MdlFunctionP    resetFunc,       /* => function in MDL program */
    int             funcName,        /* => index into a MessageList */
    int             promptNum,       /* => index into a MessageList */
    int             clippingMode     /* => clipping mode */
);

```

Description The `mdlState_startFenceCommand` function performs standard initialization for primitive commands.

contentFunc designates an MDL function to be called to process the fence elements. It is called once for every element that satisfies the search criteria.

outlineFunc designates an MDL function to be called to display the new fence outline. This argument is often `NULL`, since it is used only if the entire fence is being moved, scaled, or rotated.

dataFunc designates an MDL function to be called when a data point is entered. If `NULL` is passed, MicroStation establishes `mdlFence_process` as the data point function.

resetFunc designates an MDL function to be called when Reset is entered.

funcName is an integer specifying a command name. *promptNum* is an integer specifying a prompt. These are numbers for strings in `MessageList` resources in the application's resource file.

clippingMode determines the way the command wants MicroStation to treat the elements that overlap the fence when fence lock is turned on. Possible values for *clippingMode* are:

clippingMode	Meaning
<code>FENCE_NO_CLIP</code>	Do not allow command to continue if user has fence clip lock turned on.
<code>FENCE_CLIP_ORIG</code>	If fence clip lock is on, clip original elements before calling <i>contentFunc</i> .
<code>FENCE_CLIP_COPY</code>	If fence clip lock is on, copy the element before clipping it, then call <i>contentP</i> for the clipped copy.

Returns The `mdlState_startFenceCommand` function is of type `void`. It returns no value.

See Also `mdlState_registerStringIds`, "Element Location Functions" (`mdlLocate_...`), `userState_fenceContent`, `userState_fenceOutline`.

mdlState_registerStringIds

```
void mdlState_registerStringIds
(
  int      commandStringId,    /* => ID for command MessageList */
  int      promptStringId     /* => ID for prompt MessageList */
);
```

Description The `mdlState_registerStringIds` function specifies the resource IDs that MicroStation uses when loading prompt and command strings for the application.

Returns The `mdlState_registerStringIds` function is `void`. It returns no value.

See Also `mdlState_startPrimitive`, `mdlState_startModifyCommand`, `mdlState_startViewCommand`, “Message lists” in the “Resources Overview” chapter.

mdlState_setKeyinPrompt

```
void mdlState_setKeyinPrompt
(
  char      *stringP          /* => prompt string */
);
```

Description The `mdlState_setKeyinPrompt` function specifies a string to be used as the prompt in the key-in area in MicroStation’s command window. The next time a primitive command starts, the prompt is automatically restored to MicroStation’s prompt.

If an application uses `mdlState_setKeyinPrompt` and an `mdlState_start...` function other than `mdlState_startViewCommand`, the `mdlState_start...` function must be called first since it restores the prompt to MicroStation’s default prompt. To modify MicroStation’s default prompt, change the contents of the built-in variable *mgdsPrompt*.

`mdlState_setKeyinPrompt` should never be used for a view command. It is intended for primitive commands only.

Returns The `mdlState_setKeyinPrompt` function is of type `void`. It returns no value.

mdlState_dynamicUpdate

```
MdlFunctionP mdlState_dynamicUpdate
(
  MdlFunctionP      dynamicFuncP,    /* => function in MDL program */
  int               validData        /* => TRUE or FALSE */
);
```

Description The `mdlState_dynamicUpdate` function specifies an MDL function to be used with **simple dynamics**. The function determines the screen display as the cursor is moved. Dynamics are used only while a primitive command is active. Starting a

view command suspends the dynamics associated with the active primitive command.

When an MDL application uses simple dynamics, the MDL application's function is called to create or update an element in the design *bufferdgnBuf*. MicroStation draws the element in a temporary mode.

validData designates whether the current design buffer contents are valid. If they are, MicroStation draws them as part of the dynamics initialization. Typically, *validData* is `TRUE` when a modify command uses `mdlState_dynamicUpdate`. Otherwise, it is `FALSE`.

Dynamic functions are valid only when MicroStation is executing a primitive command. The state data function pointer for dynamic functions is reset every time a primitive function is restarted. Elements are drawn by dynamics in a temporary mode. Elements drawn in a temporary mode are erased when state functions are cleared or when a new primitive is started.

MDL also supports complex dynamic updates. With complex dynamic updates, the MDL program must also erase and draw the elements. Complex dynamics are started by a call to the `mdlState_setFunction` function.

Returns The `mdlState_dynamicUpdate` function returns a pointer to the simple dynamic function that was previously set using `mdlState_dynamicUpdate`.

See Also `mdlState_setFunction`, `mdlState_startPrimitive`, `mdlState_startModifyCommand`, `mdlState_startFenceCommand`, `userState_dynamicUpdate`, `userState_complexDynamicUpdate`.

mdlState_setFunction

```
#include <userfnc.h>

MdlFunctionP mdlState_setFunction
(
    int             eventType,      /* => STATE_DATAPOINT */
    MdlFunctionP    stateFunc      /* => function in MDL program */
);
```

Description The `mdlState_setFunction` function designates an MDL function to be called for certain user events.

Possible values for *eventType* are `STATE_DATAPOINT`, `STATE_RESET`, `STATE_KEYIN`, `STATE_COMMAND_CLEANUP`, `STATE_COMPLEX_DYNAMICS`, `STATE_OOPSFUNCTION` and `STATE_DRAG_INITS`.

functionP must be a valid pointer to an MDL function, or `NULL`. If no value designates a function for simple dynamics, use the `mdlState_dynamicUpdate` function.

The designated function becomes associated with the current MicroStation state. If `mdlState_setFunction` is used with a function (such as

`mdlState_startPrimitive` or `mdlState_startViewCommand`) that starts commands, `mdlState_setFunction` must be called after any `mdlState_start...` function since these functions set up the MicroStation state.

This function most commonly modifies the state as the user steps through the command states. For example, if a command requires three data points, the command is likely to have three functions for data points. The `mdlState_start...` function will specify the function for the first data point. The function for the first data point will use `mdlState_setFunction`, in turn, to specify the function for the second data point.

`mdlState_setFunction` does not modify MicroStation's restart function pointer when it modifies MicroStation's reset function pointer. All `mdlState_start...` functions set the restart function pointer to the initial reset function pointer value.

Returns The `mdlState_setFunction` function returns a pointer to the user function (of the same type) that was previously set using `mdlState_setFunction`.

See Also `mdlState_dynamicUpdate`, `mdlState_startPrimitive`, `mdlState_startModifyCommand`, `mdlState_startViewCommand`, user functions from this chapter.

mdlState_setAccudrawContext

```
#include <msstate.fdf>
#include <accudraw.h>

int mdlState_setAccudrawContext
(
    long          flags,          /* hints from accudraw.h */
    Dpoint3d      *originP,      /* or NULL */
    Dpoint3d      *deltaP,       /* or NULL */
    double        *distanceP,    /* or NULL */
    double        *angleP,       /* or NULL */
    void          *orientationP  /* RotMatrix or Dpoint3d, or NULL */
);
```

Description The `mdlState_setAccudrawContext` function allows a command to provide “hints” to AccuDraw, so that it may behave more intelligently depending on the context of the current tool. MDL programmers writing placement or modification tools will find that they can greatly enhance the usability and power of these tools by optimizing them for AccuDraw with this function.

This function is typically called when starting a command, after receiving a data point, or in dialog item hook functions after a setting has been changed. It is not recommended to call it from a dynamics function. The effects of the hints are temporary -- they will last only until a new primitive command is started. If AccuDraw is not active or the “Context Sensitivity” setting is turned off, the function will have no effect. Furthermore, the

hints can be overridden by the user through AccuDraw's settings and shortcut commands.

The function can be used to tell AccuDraw to use an origin point other than the default (last data point placed), to define the orientation (rotation) of AccuDraw's drawing plane, to lock the x, y or z coordinate value or the distance or angle, to force AccuDraw into either polar or rectangular coordinates, or to disable AccuDraw altogether for commands where it may get in the way.

It is recommended that you refer to the adrwdemo MDL example application for example commands which are optimized using this function.

The *flags* parameter is a bit mask indicating which hints are being sent. It is set by combining any number of hints defined in `accudraw.h` with a bitwise OR (`|`).

Hint	Used to
ACCUDRAW_Disable	disable AccuDraw (until a new primitive command is started).
ACCUDRAW_FixedOrigin	force AccuDraw's origin to remain in place until a new command starts, another origin is provided by using <code>ACCUDRAW_SetOrigin</code> , or the user explicitly moves the origin with the 'O' shortcut command.
ACCUDRAW_LockAngle	lock AccuDraw's angle to a specified value. This will only have effect if in polar coordinates, so you may wish to use <code>ACCUDRAW_SetModePolar</code> .
ACCUDRAW_LockDistance	use to lock AccuDraw's distance to the previous distance. You may also wish to use <code>ACCUDRAW_SetDistance</code> to provide a distance. This will only have effect if in polar coordinates, so you may wish to use <code>ACCUDRAW_SetModePolar</code> .
ACCUDRAW_Lock_X	lock AccuDraw's X value. This value locked is the same as the one that appears in AccuDraw's window when in rectangular coordinates, which is measured relative to the origin along AccuDraw's X axis. This will only have effect if in rectangular coordinates, so you may wish to use <code>ACCUDRAW_SetModeRect</code> . You must send the value using the <i>deltaP</i> argument, which is a <code>Dpoint3d</code> whose x member has the value desired.
ACCUDRAW_Lock_Y	see <code>ACCUDRAW_Lock_X</code> .
ACCUDRAW_Lock_Z	see <code>ACCUDRAW_Lock_X</code> .
ACCUDRAW_OrientACS	set AccuDraw's rotation to ACS mode. Unlike other hints, the effect of this hint will remain after the current command terminates. It is often desirable to use this hint in a command which defines an ACS.

Hint	Used to
ACCUDRAW_OrientDefault	restore AccuDraw's orientation to the default (user defined) rotation, which may be View, Top, Front, Side or ACS.
ACCUDRAW_Set3dMatrix	set AccuDraw's drawing plane orientation. This behaves identically to ACCUDRAW_SetRMatrix, except that rotation is defined by an array of 3 unit vectors (Dpoint3d) representing the x, y and z axes, rather than by a RotMatrix. This is often convenient, eliminating the need to call mdlRMatrix_fromRowVectors first.
ACCUDRAW_SetDistance	set AccuDraw's "previous distance" to a specified value. The user can dynamically "index" to this distance, and can recall it later by using the Page Up key. You must send the distance (double) using the <i>distanceP</i> argument.
ACCUDRAW_SetFocus	move keyin focus to AccuDraw's window. This is sometimes convenient in a hook function for a dialog item, so the keyboard focus does not remain in the Tool Settings window.
ACCUDRAW_SetModePolar	set AccuDraw's coordinate system to polar (distance, angle).
ACCUDRAW_SetModeRect	set AccuDraw's coordinate system to rectangular (x, y, z).
ACCUDRAW_SetNormal	define the Z axis, or normal vector, of AccuDraw's drawing plane. AccuDraw will derive the X and Y axes based on the user defined (default) orientation. You must provide a unit vector (Dpoint3d) to the <i>orientationP</i> argument.
ACCUDRAW_SetOrigin	set AccuDraw's origin. You must provide a point using the <i>originP</i> argument.
ACCUDRAW_SetRMatrix	set AccuDraw's drawing plane orientation (rotation). You must provide a rotation matrix (type RotMatrix) using the <i>orientationP</i> argument. The rotation matrix is row based (rather than column based), so it may be necessary to use mdlRMatrix_invert first (such as if using a rotation matrix derived from a circle, arc or text element).
ACCUDRAW_SetXAxis	define the X axis of AccuDraw's drawing plane. AccuDraw will derive the Y and Z axes based on the user defined (default) orientation, which may be View, Top, Front, Side or ACS (auxiliary coordinate system). You must provide a unit vector (Dpoint3d) to the <i>orientationP</i> argument.

The *originP* parameter points to a Dpoint3d to be used as AccuDraw's origin point. This is only used if ACCUDRAW_SetOrigin is sent in the *flags* argument. Otherwise, pass NULL for this parameter.

The *deltaP* parameter points to a Dpoint3d to be used for AccuDraw's x, y and/or z offset value(s). This is only used if ACCUDRAW_Lock_X, ACCUDRAW_Lock_Y, and/or ACCUDRAW_Lock_Z is sent in the *flags* argument.

Otherwise, pass `NULL` for this parameter. It is recommended that unused members of the `Dpoint3d` be set to zero.

The *distanceP* parameter points to a `double` to be used as AccuDraw's distance. This is only used if `ACCUDRAW_SetDistance` and/or `ACCUDRAW_LockDistance` is sent in the *flags* argument. Otherwise, pass `NULL` for this parameter.

The *angleP* parameter points to a `double` to be used as AccuDraw's angle, expressed in radians. This is only used if `ACCUDRAW_LockAngle` is sent in the *flags* argument. Otherwise, pass `NULL` for this parameter.

The *orientationP* parameter points to a `RotMatrix`, an array of 3 unit vectors of type `Dpoint3d`, or a single unit vector of type `Dpoint3d`, to be used to define AccuDraw's drawing plane orientation. This is used if `ACCUDRAW_SetRMatrix`, `ACCUDRAW_Set3dMatrix`, `ACCUDRAW_SetXAxis` or `ACCUDRAW_SetNormal` is sent for the *flags* argument. Otherwise, pass `NULL` for this argument.



This function was implemented in MicroStation 95.

Returns `mdlState_setAccudrawContext` returns `SUCCESS` if AccuDraw is active.

userState_commandCleanup

```
void userState_commandCleanup();
```

Description A command-cleanup user function is designated to MicroStation when the user function name is specified in a call to `mdlState_setFunction`. *userState_commandCleanup* does not need to be the function name. The actual function name is insignificant to MicroStation.

Generally, MicroStation and MDL commands terminate only when another command starts. The old command is not notified that another command is starting. The new command modifies only the state function pointers and the functions for the old command are not called again.

MicroStation's start primitive logic calls the current *userState_commandCleanup* function and then clears the *userState_commandCleanup* function pointer. Therefore, if an MDL command needs to be notified when it is terminated, it designates the user function after calling the appropriate `mdlState_start...` functions.

Returns MDL ignores the return value.

See Also `mdlState_setFunction`.

userState_datapoint

```
void userState_datapoint
(
Dpoint3d    *point,    /* => data point entered by user */
int         view       /* => view point was in */
);
```

Description A data point user function is designated to MicroStation when the user function name is specified in a call to `mdlState_setFunction` or in an `mdlState_start...` function. *userState_datapoint* does not need to be the function name. The actual function name is insignificant to MicroStation.

If the current command state defines a data point user function, the function is called when the user enters a data point.

The current point is passed to *userState_datapoint* in *point* and the view is passed in *view*. Before calling the data point user function, MicroStation adjusts the point that the user enters for the current locks (such as snap, grid and unit) and tentative point information. It then transforms the point to the current coordinate system.

Returns MDL ignores the return value.

See Also `mdlState_setFunction`, `mdlState_startModifyCommand`, `mdlState_startPrimitive`, `mdlState_startViewCommand`, `mdlLocate_setFunction`.

userState_reset

```
void userState_reset();
```

Description A reset user function is designated to MicroStation when the user function name is specified in a call to `mdlState_setFunction` or in an `mdlState_start...` function. *userState_reset* does not need to be the function name. The actual function name is insignificant to MicroStation.

If the current command state defines a Reset user function, MicroStation calls the function when the user enters a Reset.

Returns MDL ignores the return value.

See Also `mdlState_setFunction`, `mdlState_startModifyCommand`, `mdlState_startPrimitive`, `mdlState_startViewCommand`.

userState_keyin

```
void userState_keyin
(
char    *cmdStringP    /* => input string, never NULL */
);
```

Description A key-in user function is designated to MicroStation when the user function name is specified in a call to `mdlState_setFunction`. *userState_keyin* does not need to be the function name. The actual function name is insignificant to MicroStation.

If the current command state defines a key-in user function, the function is called when the user enters a key-in.

cmdStringP points to the key-in string. *cmdStringP* is never `NULL`.

If a tutorial is active, MicroStation makes the first field of the tutorial the current input field after the state data key-in function returns.

Tutorial key-ins are never returned to state data key-in functions.

Returns MDL ignores the return value.

See Also `mdlState_startModifyCommand`, `mdlState_startPrimitive`, `mdlState_startViewCommand`.

userState_dynamicUpdate

```
#include <basetype.h>

void userState_dynamicUpdate
(
  Dpoint3d    *point,          /* => current point */
  int         view             /* => current view */
);
```

Description A dynamic update user function is designated to MicroStation when the user function name is specified in a call to `mdlState_dynamicUpdate` or `mdlState_startModifyCommand`. *userState_dynamicUpdate* does not need to be the function name. The actual function name is insignificant to MicroStation.

If the current command state defines an update function for simple dynamics, MicroStation calls the function when the cursor is moved.

userState_dynamicUpdate functions should use the data point *point*, the view *view*, and the current element in the *dgnBuf* variable to create a new element in *dgnBuf*. MicroStation erases the old element and draws the new one in all visible views.

Returns MDL ignores the return value.

See Also `mdlState_dynamicUpdate`, `userState_complexDynamicUpdate`.

userState_complexDynamicUpdate

```
void userState_complexDynamicUpdate
(
  Dpoint3d    *pointP,        /* => current point */
  int         view,           /* => current view */
  int         drawMode        /* => display mode */
);
```


Description A complex dynamic update user function is designated to MicroStation when the user function name is specified in a call to `mdlState_setFunction`. *userState_complexDynamicUpdate* does not need to be the function name. *userState_complexDynamicUpdate* is only used as an example; the actual function name is insignificant to MicroStation.

If the current command state defines an update function for complex dynamics, MicroStation calls the function when the cursor is moved.

pointP gives the cursor's location.

view gives the cursor's view.

drawMode is `TEMPDRAW` or `TEMPERASE`. These fields are often passed to MicroStation functions and used to draw elements.

Returns MicroStation ignores the return value.

See Also `mdlState_setFunction`, `mdlElement_display`.

userState_show

```
void userState_show();
```

Description A show user function is designated to MicroStation when the user function name is specified in a call to `mdlState_startModifyCommand`. *userState_show* does not need to be the function name. The actual function name is insignificant to MicroStation.

Show functions are associated with modify commands. If a modify command is using the standard sequence of states and the current command state defines a show function, the *userState_show* function is called after the element to be modified is located and loaded in the *dgnBuf* variable.

Returns MicroStation ignores the return value.

See Also `mdlState_startModifyCommand`, `mdlLocate_setFunction`, `userState_clean`.

userState_clean

```
void userState_clean();
```

Description A clean user function is designated to MicroStation when the user function name is specified in a call to `mdlState_startModifyCommand`. *userState_clean* does not need to be the function name. The actual function name is insignificant to MicroStation.

Clean functions are associated with modify commands. Typically, a *userState_clean* function is paired with a *userState_show* function. If the standard sequence of states is used for a modify command, the *userState_clean* function would be called if the user enters a Reset after

identifying the element to be modified. The clean function usually undoes the previous show function's action.

Returns MicroStation ignores the return value.

See Also *mdlState_startModifyCommand*, *mdlLocate_setFunction*, *userState_show*.

userState_fenceContent

```
#include <mselems.h>

int userState_fenceContent
(
    void    *arg
);
```

Description A fence content user function is designated to MicroStation when the user function name is specified in the *contentP* argument to *mdlState_startFenceCommand*. *userState_fenceContent* does not need to be the function name. The actual function name is insignificant to MicroStation.

The *userState_fenceContent* function is called when the *mdlFence_process* function accepts an element. *mdlFence_process* can either be called directly by MDL applications or by MicroStation if *dataP* is NULL for *mdlState_startFenceCommand*.

arg is passed to the *userState_fenceContent* function from the call to *mdlFence_process*. MicroStation does not use it and applications use it for arguments and/or return status to/from *userState_fenceContent*.

userState_fenceContent should call the following to obtain the element's file position:

```
filePos=mdlElement_getFilePos(FILEPOS_CURRENT, &fileNum);
```

Returns If the *userState_fenceContent* function returns SUCCESS, the fence command continues. Otherwise, it stops.

See Also *mdlState_startFenceCommand*, *mdlFence_process*, *userState_fenceOutline*.

userState_fenceOutline

```
#include <global.h>

void userState_fenceOutline
(
    Dpoint3d    *point,          /* => point entered by user */
    int         view             /* => view point was in */
);
```

Description A fence outline user function is designated to MicroStation when the user function name is specified in the *outlineP* argument to *mdlState_startFenceCommand*.

userState_fenceOutline does not need to be the function name. The actual function name is insignificant to MicroStation.

If the current command is a fence command and the command state defines a fence outline user function, the *userState_fenceOutline* function is called when the user enters a data point.

The data point entered by the user is passed to *userState_fenceOutline* in *point* and *view*.

Returns MDL ignores the return value.

See Also *mdlState_startFenceCommand*, *userState_fenceContent*.

2

User Interface

This section describes the methods for obtaining access to windows, manipulating them and drawing to them.

This chapter describes the following functions:

- Window management functions
- Window drawing functions
- Window docking functions
- Resource management functions
- Parse functions
- Input handling functions
- Functions that display messages (Output)
- C expression handling functions

Window Management Functions

All MicroStation graphic output is oriented towards on-screen **windows**. MicroStation uses many types of windows, including view windows, the tutorial window and dialog box windows. Your MDL program can draw graphic primitives only to view windows and dialog box windows.

Every graphic built-in function accepts a **window pointer** as an argument. This is a pointer to an **opaque structure**, a structure for which the members are private to MicroStation's window manager. Opaque structures are used when the details of the underlying structure are subject to implementation differences among different platforms. They are also used when client applications cannot reasonably examine or manipulate the members. In `mdl.h`, the type of `MSWindow` is declared `void` with the `typedef` declaration. Your application should declare a pointer to an `MSWindow` to pass to these routines. For the graphic functions, a pointer (also opaque) to a `DialogBox` can be substituted for a pointer to an `MSWindow`.

The following table lists the window management functions:

Function	Used to
<code>mdlWindow_cursorTurnOff</code>	turn off the cursor.
<code>mdlWindow_flush</code>	flush buffered graphics.
<code>mdlWindow_titleGet</code>	get title of window.
<code>mdlWindow_titleSet</code>	set title of window.
<code>mdlWindow_sink</code>	sink window to lowest priority.
<code>mdlWindow_float</code>	restore window to base priority.
<code>mdlWindow_toBack</code>	send window to the back of its priority grouping.
<code>mdlWindow_toFront</code>	bring window to the front of its priority grouping.
<code>mdlWindow_resize</code>	resize a window.
<code>mdlWindow_extentSet</code>	resize a window.
<code>mdlWindow_hide</code>	hide a window from view.
<code>mdlWindow_show</code>	recall a hidden window.
<code>mdlWindow_close</code>	close a window.
<code>mdlWindow_updateAllWindows</code>	force all open views and dialogs to be redrawn.
<code>mdlWindow_changeScreen</code>	move window from screen to screen.
<code>mdlWindow_maximize</code>	increase the size of the indicated window to its largest possible size.
<code>mdlWindow_minimize</code>	decrease the size of the indicated window to its smallest possible size.
<code>mdlWindow_restore</code>	return the window to the size and position it occupied before the size-changing operation.
<code>mdlWindow_windowEventsProcessAll</code>	force MicroStation to process all queued window resize/update events.
<code>mdlWindow_getFirst</code>	get the first window in the list.
<code>mdlWindow_getNext</code>	get the next window in the list.
<code>mdlWindow_getLast</code>	get the last window in the list.
<code>mdlWindow_getPrevious</code>	get the previous window in the list.
<code>mdlWindow_isVisible</code>	check visibility status.
<code>mdlWindow_isDialogBox</code>	test whether or not the designated window is a dialog box.
<code>mdlWindow_isDisplayed</code>	test whether or not the designated window is visible to the user.

Function	Used to
mdlWindow_displayDescrGet	get information about the screen that the window is on.
mdlWindow_screenNumGet	get the number of the screen that the window is on.
mdlWindow_viewWindowGet	get pointer to view window.
mdlWindow_pointToLocal	transform global to local point.
mdlWindow_pointToGlobal	transform local point to global.
mdlWindow_contentRectGetLocal	get content rectangle in local coordinates.
mdlWindow_contentRectGetGlobal	get content rectangle in global coordinates.
mdlWindow_globalRectGetLocal	get global rectangle in local coordinates.
mdlWindow_globalRectGetGlobal	get global rectangle in global coordinates.
mdlWindow_nativeWindowHandleGet	get the native window identifier to the application.
mdlWindow_globalToContentRect	determine the content rectangle of a window.
mdlWindow_setInputFocus	attempt to set the input focus to a window.
mdlWindow_isView	determine whether the specified window is a MicroStation view window.
mdlWindow_isPopUp	query whether a window is a pop-up.
mdlWindow_setFunction	designate one of the user functions (listed below) to be called during MicroStation's window updating processes.
mdlWindow_getInputFocus	find a window which has input focus.
mdlWindow_setInputFocus	attempt to set the input focus to windowP.

The following table lists graphic user functions that can be designated by mdlWindow_setFunction:

User Function	Called by MicroStation when
userWindow_modifyEvents	any window shows, hides, resizes, restacks or changes screens.
userWindow_scrollEvents	any window's contents are scrolled.

Example

See gphtest.mc and rasticon.mc.

mdlWindow_cursorTurnOff

```
void mdlWindow_cursorTurnOff();
```

Description The `mdlWindow_cursorTurnOff` function allows you to turn off the cursor. This must be done before you begin to draw in a window. You never need to turn the cursor back on, since MicroStation will do this the next time it undergoes its input loop.

Returns The `mdlWindow_cursorTurnOff` function is of type `void`. It returns no value.

mdlWindow_flush

```
#include <mdl.h>

void mdlWindow_flush
(
MSWindow      *windowP /* => window to flush */
);
```

Description On some platforms, graphic output is buffered, rather than going to the screen immediately. To ensure that the screen is displaying all graphics that were drawn with the `mdlWindow` calls, the `mdlWindow_flush` function is used. To flush buffers for all windows, the *windowP* argument can be `NULL`.

Returns The `mdlWindow_flush` function is of type `void`. It returns no value.

mdlWindow_titleGet, mdlWindow_titleSet

```
#include <mdl.h>

void mdlWindow_titleGet
(
char          *titleP,          /* <= title of window */
int           numChars,         /* => size of title output buffer */
MSWindow      *windowP         /* => window to get title for */
);

void mdlWindow_titleSet
(
MSWindow      *windowP,         /* => window to set title for */
char          *titleP           /* => title to set */
);
```

Description `mdlWindow_titleGet` retrieves the window's title specified by *windowP* into *titleP*. `mdlWindow_titleSet` sets the title to *titleP*. When retrieving the title, you must specify the number of characters in your string in *numChars*.

Returns The `mdlWindow_titleGet` and `mdlWindow_titleSet` functions are of type `void`. They return no values.

mdlWindow_sink, mdlWindow_float

```
#include <mdl.h>

void mdlWindow_sink
(
MSWindow    *windowP /* => window to sink */
);

void mdlWindow_float
(
MSWindow    *windowP /* => window to float */
);
```

Description The mdlWindow_sink function lowers a window's priority from its base to the lowest possible priority. Each MicroStation window has an associated base priority. All dialog boxes are grouped into the same priority, and their base priority is higher than the base priority of graphic and tutorial windows. The mdlWindow_float function returns to its base priority. The window is specified with *windowP*.

Returns mdlWindow_sink and mdlWindow_float are of type void. They return no values.

See Also mdlWindow_toFront, mdlWindow_toBack.

mdlWindow_toBack, mdlWindow_toFront

```
#include <mdl.h>

void mdlWindow_toBack
(
MSWindow    *windowP /* => window to move to back */
);

void mdlWindow_toFront
(
MSWindow    *windowP /* => window to move to front */
);
```

Description Each MicroStation window has an associated priority. All dialog boxes are grouped into the same priority, and their priority is higher than the priority of graphics and tutorial windows. The mdlWindow_toFront function moves the window specified by *windowP* to the front of its priority grouping, and the mdlWindow_toBack function moves the window to the back of its priority grouping. To change a window's priority, use mdlWindow_sink and mdlWindow_float.

Returns The mdlWindow_toBack and mdlWindow_toFront functions are of type void. They return no values.

See Also mdlWindow_sink, mdlWindow_float.

mdlWindow_resize, mdlWindow_extentSet

```

#include <mdl.h>
#include <msdefs.h>
void mdlWindow_resize
(
MSWindow      *windowP,      /* => window to resize */
int           cornerNum,     /* => corner to move */
Point2d       *newPosP       /* => new position for corner */
);

void mdlWindow_extentSet
(
MSWindow      *windowP,      /* => window to resize */
int           newWidth,      /* => new window width */
int           newHeight      /* => new window height */
);

```

Description The `mdlWindow_resize` and `mdlWindow_extentSet` functions change the window size specified in *windowP*. `mdlWindow_extentSet` is the simpler of the two routines. It lets you specify a new height and width for the window in *newWidth* and *newHeight*, respectively. You can resize the window by moving its lower right corner. The `mdlWindow_resize` function is more flexible, but more difficult to use. With it, any corner can be moved, or the entire window can be moved without a size change. Specify the corner by setting *cornerNum* to `CORNER_UPPERLEFT`, `CORNER_LOWERLEFT`, `CORNER_LOWERRIGHT`, `CORNER_UPPERRIGHT` or `CORNER_ALL`. The new position of the specified corner of the window's content rectangle (in global coordinates) is specified in *newPosP*. If *cornerNum* is set to `CORNER_ALL`, *newPosP* is the new position of the upper left corner.

Returns The `mdlWindow_resize` and `mdlWindow_extentSet` functions are of type `void`. They return no values.

See Also `mdlWindow_contentRectGetGlobal`.

mdlWindow_hide, mdlWindow_show, mdlWindow_close

```

#include <mdl.h>

boolean mdlWindow_hide
(
MSWindow      *windowP,      /* => window to hide */
int           exiting,       /* => TRUE only if exiting uStation */
int           dontFocusOut   /* => bypass the focus out */
);

void mdlWindow_show
(
MSWindow      *windowP,      /* => window to show */
boolean       noInitRefresh  /* => don't do first refresh event */
);

```

```
void mdlWindow_close
(
MSWindow    *windowP,      /* => window to close */
int          exiting,      /* => TRUE only if exiting uStation */
int          dontFocusOut  /* => bypass the focus out */
);
```

Description The `mdlWindow_hide` function temporarily hides *windowP*. The `mdlWindow_show` function brings it back into view. The `mdlWindow_close` function hides the window and deallocates it, invalidating *windowP* for any further `mdlWindow_...` operations.

exiting and *dontFocusOut* should be set to FALSE.

noInitRefresh for `mdlWindow_show` inhibits the first refresh message normally sent in a window when it is first displayed. Generally, this argument should be set to FALSE.

Returns `mdlWindow_hide` returns TRUE if it fails for some reason and FALSE if it succeeds. It fails if the window pointer passed in is invalid or the window refused to relinquish focus and *dontFocusOut* is FALSE.

`mdlWindow_show` and `mdlWindow_close` are of type `void`. They return no values.

mdlWindow_updateAllWindows

```
void mdlWindow_updateAllWindows
(
int          systemOnly
);
```

Description `mdlWindow_updateAllWindows` forces all open views and dialogs to be redrawn.

systemOnly is intended for system use and should usually be set to 0.

Returns `mdlWindow_updateAllWindows` is of type `void`; it returns no value.

See Also `mdlView_updateSingle`, `mdlView_updateMulti`, `mdlView_updateMultiExtended`.

mdlWindow_changeScreen

```
#include <mdl.h>
#include <msdefs.h>

boolean mdlWindow_changeScreen/* <= FALSE if successful */
(
MSWindow    *windowP,      /* => window to resize */
int          oldScreen,     /* => old screen number */
int          newScreen,     /* => new screen number */
Point2d     *newGlobalOriginP, /* => new global origin */
Point2d     *newContentExtentP, /* => new content extent */
int          leaveAtCurrentPriority /* => new screen number */
);
```

Description The `mdlWindow_changeScreen` function moves the window specified by *windowP* from the screen indicated by *oldScreen* to the location specified by *newGlobalOriginP* (in global coordinates (0, 0) is top left hand corner of screen) on screen *newScreen*. The screen numbers used are indices into the `graphConfig` display descriptor table.

newContentExtentP defines the extents to be used in the dialog box when it is placed on the new screen. If this parameter is `NULL`, the same extents as are defined for the dialog box on the old screen are used.

leaveAtCurrentPriority is not currently used.

See Also `mdlWindow_contentRectGetGlobal`, `mdlWindow_resize`, `mdlWindow_extentSet`.

mdlWindow_maximize, mdlWindow_minimize, mdlWindow_restore

```
boolean mdlWindow_maximize
(
MSWindow      *windowP          /* => window to maximize */
);
boolean mdlWindow_minimize
(
MSWindow      *windowP          /* => window to minimize */
);
boolean mdlWindow_restore
(
MSWindow      *windowP          /* => min or maximized window to restore */
);
```

Description The `mdlWindow_maximize` function increases the size of the indicated window to its largest possible size. Similarly, `mdlWindow_minimize` decreases the size of the indicated window to its smallest possible size. If a window has been minimized or maximized, `mdlWindow_restore` returns the window to the size and position it occupied before the size-changing operation.



Only windows that are resizable can be minimized or maximized, and only windows that have been minimized or maximized (with no intervening size changes) can be restored.

The *windowP* parameter designates the window that is to be minimized, maximized or restored.

Returns `mdlWindow_maximize`, `mdlWindow_minimize` and `mdlWindow_restore` return `FALSE` on success and `TRUE` if it fails. Failure will occur if *windowP* is not a valid, sizable window.

See Also `mdlWindow_resize`, `mdlWindow_extentSet`.

mdlWindow_windowEventsProcessAll

```
void mdlWindow_windowEventsProcessAll();
```

Description When windows are resized, moved, sunk, floated, pushed to the back, or pulled to the front, MicroStation queues refresh events to accomplish the necessary screen changes. These changes are generally not processed until MicroStation returns to its main input loop. If your program will do further processing and wants immediate action on the screen, it should call mdlWindow_windowEventsProcessAll.

Returns The mdlWindow_windowEventsProcessAll function is of type void and returns no value.

mdlWindow_getFirst, mdlWindow_getNext

```
MSWindow *mdlWindow_getFirst(void);
MSWindow *mdlWindow_getNext
(
    MSWindow    *current/* => from call to *_getFirst or prev *_getNext */
);
```

Description The mdlWindow_getFirst and mdlWindow_getNext functions are used together to “walk through” all MicroStation windows. The program first calls mdlWindow_getFirst to get the first window in the list, and then repeatedly calls mdlWindow_getNext to get all the other windows. The *current* parameter must be the return value from mdlWindow_getFirst or mdlWindow_getNext.

A loop that processes all windows can be coded:

```
MSWindow *current;

for (current=mdlWindow_getFirst(); current;
     current=mdlWindow_getNext(current))
{
    .../* process window here */
}
```

Returns mdlWindow_getFirst and mdlWindow_getNext return a pointer to an MSWindow. When there are no more windows, a NULL pointer is returned.

mdlWindow_getLast, mdlWindow_getPrevious

```
MSWindow *mdlWindow_getLast(void);
MSWindow *mdlWindow_getPrevious
(
    MSWindow    *current /* current window */
);
```

Description mdlWindow_getLast and mdlWindow_getPrevious are used together to traverse MicroStation’s linked list of windows. The mdlWindow_getLast routine will return the most recently created window. The window returned from it can then be used

as the argument to `mdlWindow_getPrevious` to continue traversing the windows. Except in special circumstances the `mdlWindow_getFirst` and `mdlWindow_getNext` functions are generally used to traverse the window list.

Returns `mdlWindow_getLast` and `mdlWindow_getPrevious` return a pointer to an `MSWindow`. When there are no more windows, `NULL` is returned.

See Also `mdlWindow_getFirst`, `mdlWindow_getNext`.

mdlWindow_isVisible

```
#include <mdl.h>

boolean mdlWindow_isVisible
(
    MSWindow      *windowP      /* => window to test for visibility */
);
```

Description `mdlWindow_isVisible` tests to see whether a window is visible. In this context, visibility means that the window can be drawn on. On a PC with one screen and swap capability, a window on the invisible virtual screen cannot be drawn to and `mdlWindow_isVisible` will return `FALSE`. On a single screen Intergraph workstation, however, such a window can be drawn on, and `mdlWindow_isVisible` will return `TRUE`.

Returns The `mdlWindow_isVisible` function returns `TRUE` or `FALSE`, depending on the window's visibility status.

mdlWindow_isDialogBox

```
boolean mdlWindow_isDialogBox
(
    MSWindow      *windowP      /* => window to test */
);
```

Description The `mdlWindow_isDialogBox` function tests whether or not the designated window is a dialog box.

windowP is the window to test.

Returns `mdlWindow_isDialogBox` returns `TRUE` if *windowP* is a dialog box and `FALSE` if it is not.

mdlWindow_isDisplayed

```
boolean mdlWindow_isDisplayed
(
    MSWindow      *windowP      /* => window to test */
);
```

Description The mdlWindow_isDisplayed function tests whether or not the designated window is being displayed. A window is not displayed if it has been created but never shown or if it was shown but then hidden without being destroyed.

windowP is the window to test.

Returns mdlWindow_isDisplayed returns TRUE if *windowP* is being displayed and FALSE if it is not.

mdlWindow_displayDescrGet

```
#include <mdl.h>
#include <global.h>
MSDisplayDescr *mdlWindow_displayDescrGet
(
MSWindow      *windowP /* => window to get display descriptor for */
);
```

Description MicroStation maintains a structure (called the **display descriptor**) for each screen in the system. The mdlWindow_displayDescrGet function returns a pointer to this structure for the screen that *windowP* is on. From the display descriptor, which is defined in global.h as MSDisplayDescr, information about the screen resolution and the number of colors the screen supports can be obtained.

Returns The mdlWindow_displayDescrGet function returns a pointer to the display descriptor for the screen that the specified window is on.

mdlWindow_screenNumGet

```
#include <mdl.h>

int mdlWindow_screenNumGet
(
MSWindow      *windowP /* => window to get screen number for */
);
```

Description The mdlWindow_screenNumGet function finds the screen that *windowP* is on.

Returns The mdlWindow_screenNumGet function returns the screen number.

See Also mdlWindow_displayDescrGet.

mdlWindow_viewWindowGet

```
#include <mdl.h>

MSWindow *mdlWindow_viewWindowGet
(
int      viewIndex /* => view to get window pointer for */
);
```

Description The `mdlWindow_viewWindowGet` function retrieves a window pointer for one of MicroStation's views. *viewindex*, of the value 0 to `MAX_VIEWS - 1`, inclusive, specifies the MicroStation view. If the specified view is out of range, the function returns `NULL`. Otherwise, it returns a pointer to the window structure (which may or may not be visible) for the view.

Returns The `mdlWindow_viewWindowGet` function returns a pointer to the window of the given view. It returns `NULL` if *viewIndex* is out of range.

mdlWindow_pointToLocal, mdlWindow_pointToGlobal

```
void mdlWindow_pointToLocal
(
    Point2d      *localPtP,      /* <= point in local coordinates */
    MSWindow     *windowP,       /* => window to get coords for */
    Point2d      *globalPtP     /* => point in global coordinates */
);

void mdlWindow_pointToGlobal
(
    Point2d      *globalPtP,     /* <= point in global coordinates */
    MSWindow     *windowP,       /* => window to get coords for */
    Point2d      *localPtP      /* => point in local coordinates */
);
```

Description `mdlWindow_pointToLocal` converts a point in global coordinates to a point in the local coordinates for the window specified in *windowP*. The input point in global coordinates is specified by *globalPtP*. The output point in local coordinates is returned in *localPtP*. When *globalPtP* is `NULL`, `mdlWindow_pointToLocal` converts the output argument in place, using *localPtP* as both the input and output. `mdlWindow_pointToGlobal` is the inverse function. It also treats the first argument (*globalPtP* in this case) as both input and output if the input argument (*localPtP*) is `NULL`.

Returns `mdlWindow_pointToLocal` and `mdlWindow_pointToGlobal` are of type `void`.

mdlWindow_contentRectGetLocal, mdlWindow_contentRectGetGlobal, mdlWindow_globalRectGetLocal, mdlWindow_globalRectGetGlobal

```
#include <mdl.h>

void mdlWindow_contentRectGetLocal
(
    BSIRect      *rectP,        /* <= content
in local coords */
    MSWindow     *windowP      /* => window to get content rect for */
);

void mdlWindow_contentRectGetGlobal
(
```



```
BSIRect      *rectP,    /* <= content rect in global coords */
MSWindow     *windowP  /* => window to get content rect for */
);
void mdlWindow_globalRectGetLocal
(
BSIRect      *rectP,    /* <= global rect in local coords */
MSWindow     *windowP  /* => window to get global rect for */
);
void mdlWindow_globalRectGetGlobal
(
BSIRect      *rectP,    /* <= global rect in global coords */
MSWindow     *windowP  /* => window to get global rect for */
);
```

Description In MicroStation, each window has an associated content and global rectangle. The content rectangle is the window portion on which the application draws. The global rectangle includes the window's content rectangle and borders and title rectangle. The `mdlWindow_contentRectGetLocal` and `mdlWindow_contentRectGetGlobal` functions get the content rectangle for *windowP*, and `mdlWindow_globalRectGetLocal` and `mdlWindow_globalRectGetGlobal` are used to get the global rectangle for *windowP*. The `mdlWindow_...RectGetLocal` routines return the rectangle in local coordinates, while the `mdlWindow_...RectGetGlobal` routines return the same information in global coordinates.

Returns All four functions are of type `void`. They return no value.

mdlWindow_nativeWindowHandleGet

```
int mdlWindow_nativeWindowHandleGet
(
void          *nativeHandleP,    /* <= address of window handle */
MSWindow      *window           /* => window to get handle for */
int           type              /* => type of handle (NT only) */
);
```

Description `mdlWindow_nativeWindowHandleGet` returns the native window identifier to the application. This value is never used as an argument to any MDL built-in function, but some DLMs will be able to use it.



Any code that makes use of this function will generally not be easily ported from one MicroStation platform to another.

nativeHandleP points to memory in the application program that the function will fill in with the native window identifier. The type of handle returned by this function is determined by type as follows:

Platform	Value of <i>type</i>	Type of handle returned
MSDOS	0	int
Environ V	0	int
Macintosh	0	WindowPtr
X Window	HANDLETYPE_WINDOW	Window
	HANDLETYPE_DISPLAY	Display *
	HANDLETYPE_XVISUALINFO	XVisualInfo
Windows NT	HANDLETYPE_WINDOW	HWND
	HANDLETYPE_HDC	HDC (onscreen window)
	HANDLETYPE_BSHDC	HDC (for backstore)
	HANDLETYPE_BSHBITMAP	HBITMAP (for backstore)

If `mdlWindow_nativeWindowHandleGet` returns NULL for `HANDLETYPE_BSHDC` or `HANDLETYPE_BSHBITMAP`, it means that the window has no backing store.

Returns `mdlWindow_nativeWindowHandleGet` returns `SUCCESS` unless the *window* parameter is invalid, in which case it returns a nonzero value.

mdlWindow_globalToContentRect

```
void mdlWindow_globalToContentRect
(
    BSIRect      *contentRectP,      /* <= content rectangle */
    BSIRect      *globalRectP,      /* => global rectangle */
    int          screen,            /* => screen window goes on */
    GuiWAttribute *attributesP      /* => window attributes */
);
```

Description `mdlWindow_globalToContentRect` is used to determine what the content rectangle of a window will be when the global rectangle is known. Using this function, an application can determine how it wants to move or resize the window's content rectangle to cause it to fill a particular portion of the screen. This function is not generally used to obtain the current content rectangle of an existing window, which can be more conveniently determined by using `mdlWindow_contentRectGetGlobal`.

contentRectP points to a `BSIRectangle` structure that is used to hold the output, which is returned in global coordinates.

globalRectP points to a global rectangle that is used as an input.

screen indicates the screen that the window is on. It should be either 0 or 1.

attributesP points to a `GuiWAttribute` structure that defines window attributes for the window. The size difference between the global and content rectangles for a window is determined by whether or not the window is resizable, among other things.

Returns `mdlWindow_globalToContentRect` is of type `void`. It does not return a value.

See Also `mdlWindow_contentRectGetGlobal`.

mdlWindow_setInputFocus

```
boolean mdlWindow_setInputFocus
(
MSWindow    *windowP,          /* => window to receive input focus */
boolean     notClick,          /* => set to TRUE */
boolean     bypassFocusOut     /* => bypass focus out problems */
);
```

Description `mdlWindow_setInputFocus` attempts to set the input focus to *windowP*. The attempt may fail for any of the following reasons:

1. The window specified is not focusable.
2. The window that currently has the focus refuses to relinquish the focus.
3. The window specified is not visible.

notClick should always be set to `TRUE`, because the call will not be in response to a user button event.

bypassFocusOut should only be set to `TRUE` if the focus is to be changed to the specified window regardless of whether the current focus window wants to relinquish the focus. Dialog boxes, for example, refuse to relinquish the focus if the item focus is set to a text item, and the input currently in the text item is invalid. In general, MDL programs should set this argument to `FALSE`, to make sure they are cooperative with other applications.

Returns `mdlWindow_setInputFocus` returns `FALSE` if successful, `TRUE` if the system cannot set the focus to the specified window.

mdlWindow_isView

```
boolean mdlWindow_isView
(
int          *viewIndexP,      /* <= view index */
MSWindow    *windowP          /* => window to test */
);
```

Description `mdlWindow_isView` is used to determine whether the window specified by *windowP* is a MicroStation view window. If the window is a view, and *viewIndexP*

is not `NULL`, then integer pointed to by *viewIndexP* is set to the view index (0 through 7).

Returns `mdlWindow_isView` returns `TRUE` if the window tested is a view, and `FALSE` otherwise.

mdlWindow_isPopUp

```
int mdlWindow_isPopUp
(
    MWindow      *windowP /* => window to query */
);
```

Description `mdlWindow_isPopUp` queries whether the window indicated by *windowP* is a pop-up.

Returns `mdlWindow_isPopUp` returns `TRUE` if the window is a pop-up, or `FALSE` if it is not.

mdlWindow_setFunction

```
#include <userfnc.h>

MdlFunctionP mdlWindow_setFunction
(
    int          type,          /* => type of window function */
    MdlFunctionP functionP      /* => function to be called */
);
```

Description `mdlWindow_setFunction` is called to designate a function to be called whenever one of MicroStation's windows is modified or scrolled. A common usage is to detect when a window is made smaller, since in cases where the contents do not need to be resized there is no update event and it is consequently impossible to catch such window size changes.

The *type* parameter sets which type of `mdlWindow_...` notification function is being set by the call to `mdlWindow_setFunction`. This parameter can be either `WINDOW_MODIFYEVENTS` (window is shown on the screen or hidden, size changes, or stacking order changes) or `WINDOW_SCROLLLEVENTS` (window's contents are scrolled).

The *functionP* parameter designates the function that should be called when the specified window operation occurs. If *functionP* is `NULL`, the specified window event types are no longer processed by the application.

Returns `mdlWindow_setFunction` returns a pointer to the user function (of the same type) that was previously set using `mdlWindow_setFunction`. If *type* is invalid, `mdlWindow_setFunction` returns -1.

See Also `userWindow_modifyEvents`, `userWindow_scrollEvents`.

mdlWindow_getInputFocus

```
#include <mdl.h>
#include <mwindow.fdf>

MSWindow *mdlWindow_getInputFocus(void);
```

Description The `mdlWindow_getInputFocus` function is used to find a window which has input focus.



This function was implemented in MicroStation 95.

Returns `mdlWindow_getInputFocus` returns a pointer to the window in focus, or `NULL` if no window has input focus.

See Also `mdlWindow_setInputFocus`.

mdlWindow_setInputFocus

```
#include <mdl.h>
#include <mwindow.fdf>

boolean mdlWindow_setInputFocus
(
    MSWindow    *windowP,      /* => window to receive input focus */
    boolean     notClick,      /* => set to TRUE */
    boolean     bypassFocusOut /* => bypass focus out problems */
);
```

Description The `mdlWindow_setInputFocus` function attempts to set the input focus to *windowP*. The attempt may fail for any of the following reasons:

- The window specified is not focusable.
- The window that currently has the focus refuses to relinquish the focus.
- The window specified is not visible.

notClick should always be set to `TRUE`, because the call will not be in response to a user button event.

bypassFocusOut should only be set to `TRUE` if the focus is to be changed to the specified window regardless of whether the current focus window wants to relinquish the focus. Dialog boxes, for example, refuse to relinquish the focus if the item focus is set to a text item, and the input currently in the text item is invalid. In general, MDL programs should set this argument to `FALSE`, to make sure they are cooperative with other applications.



This function existed prior to MicroStation 95, but was included among new functions because of its close relationship with `mdlWindow_getInputFocus`.

Returns `mdlWindow_setInputFocus` returns `FALSE` if successful, `TRUE` if the system cannot set the focus to the specified window.

See Also `mdlWindow_getInputFocus`.

userWindow_modifyEvents

```
#include <userfnc.h>
#include <mdl.h>

void userWindow_modifyEvents
(
MSWindow      *window,      /* => window that event applies to */
int           eventType,    /* => type of event */
BSIRect       *oldGlobalP   /* => old global rectangle */
);
```

Description If an MDL application designates it as a window events function using `mdlWindow_setFunction` with the `WINDOW_MODIFYEVENTS` type parameter, *userWindow_modifyEvents* will be called whenever any window shows, hides, resizes, restacks, or changes screens. The application programmer determines the function name; *userWindow_modifyEvents* is used merely as an example. When MDL calls the user function, the arguments are used as follows:

window is the MicroStation window that was affected by the change.

eventType will be set to `WINDOW_SHOWEVENT`, `WINDOW_HIDEEVENT`, `WINDOW_MOVEEVENT`, `WINDOW_ORDEREVENT` or `WINDOW_CHANGESCREENEVENT`.

oldGlobalP is the address of the global rectangle of the window before the change took place. The current global rectangle of the window can be found by calling `mdlWindow_globalRectGetGlobal`.



When an application sets up a window event function, it gets calls for all changes for all windows. This is probably more information than the average application needs. Also, the ordering of events is somewhat system dependent. It is recommended that every other means of obtaining the information from MicroStation be tried before resorting to a window events function.

Returns The return value from *userWindow_modifyEvents* is ignored.

See Also `mdlWindow_setFunction`, `userWindow_scrollEvents`.

userWindow_scrollEvents

```
#include <userfnc.h>
#include <mdl.h>
int userWindow_scrollEvents
(
MSWindow    *window,      /* => window that event applies to */
int          eventType,    /* => type of event */
BSIRect      *scrollRectP, /* => area of window to scroll */
Point2d      *distanceP    /* => distance to scroll */
);
```

Description If an MDL application designates it as a window events function using `mdlWindow_setFunction` with the `WINDOW_SCROLL_EVENTS` type parameter, *userWindow_scrollEvents* will be called before and after any window's contents are scrolled. The application programmer determines the function name; *userWindow_scrollEvents* is used merely as an example. When MDL calls the user function, the arguments are used as follows:

window is the MicroStation window that was affected by the change.

eventType will be set to `WINDOW_SCROLLPRE` or `WINDOW_SCROLLPOST`.

scrollRectP is the portion of the window to be scrolled by distance *distanceP*.



When an application sets up a window event function, it gets calls for all changes for all windows. This is probably more information than the average application needs. Also, the ordering of events is somewhat system dependent. It is recommended that every other means of obtaining the information from MicroStation be tried before resorting to a window events function.

Returns For *eventType* of `WINDOW_SCROLLPRE`, a non-zero return value will inhibit the scroll operation from being passed to the window manager. For *eventType* of `WINDOW_SCROLLPOST`, zero should be returned.

See Also `mdlWindow_setFunction`, `userWindow_modifyEvents`.

Window Drawing Functions

Usually, MDL applications draw to dialog box windows, typically in response to requests from the dialog box manager. Usually, a generic item is created in the dialog box item list, and a user hook function is associated with the item. In response to update messages, the MDL application uses the graphic built-in functions to draw the item to its specifications. A pointer to the `DialogBox` is passed to the hook function as part of the update message.

When you draw to MicroStation windows, **local** coordinates are specified. The local coordinate system is (0, 0) in the upper left area of the window's content portion. Coordinates increase moving across and down the window. The application does not need to clip the graphic to the window's content or avoid drawing in covering windows. MicroStation's window manager handles those functions.

In some cases, MDL applications may need to draw to view windows. This is also supported. A built-in function returns an `MSWindow` pointer given a view index.

The following table lists the window drawing functions:

Function	Used to
<code>mdlWindow_lineStyleSet</code>	set color, style, weight for subsequent draws.
<code>mdlWindow_lineStyleSetCD</code>	set a window's default line style using color descriptor.
<code>mdlWindow_pointDraw</code>	draw a point in a window.
<code>mdlWindow_lineDraw</code>	draw a line in a window.
<code>mdlWindow_lineStringDraw</code>	draw a multivertex line string in a window.
<code>mdlWindow_shapeFill</code>	draw a filled shape in a window.
<code>mdlWindow_textDraw</code>	draw text in a window.
<code>mdlWindow_textDrawCD</code>	draw raster text using color descriptors.
<code>mdlWindow_arcDraw</code>	draw an arc in a window.
<code>mdlWindow_ellipseDraw</code>	draw an ellipse in a window.
<code>mdlWindow_ellipseFill</code>	draw a filled ellipse in a window.
<code>mdlWindow_rectDraw</code>	draw a rectangle in a window.
<code>mdlWindow_rectClear</code>	fill a rectangle in the background color.
<code>mdlWindow_rectFill</code>	draw a filled rectangle in a window.
<code>mdlWindow_rectFillByRGB</code>	fill a rectangle with RGB color.
<code>mdlWindow_rectInvert</code>	XOR the pixels within a specified rectangle.
<code>mdlWindow_iconDraw</code>	draw an icon resource in a window.
<code>mdlWindow_rasterDataDraw</code>	draw arbitrary raster data in a window.
<code>mdlWindow_rgbDataDraw</code>	draw RGB raster data in a window.
<code>mdlWindow_colorIndexGet</code>	get color index from element color number.
<code>mdlWindow_fixedColorIndexGet</code>	get color index for one of MicroStation's fixed colors.
<code>mdlWindow_isTrueColorCapable</code>	determine whether the specified window has true color capability.
<code>mdlWindow_backgroundColorSet</code>	change the background color of a window to a specified color.

Function	Used to
mdlWindow_backgroundCDGet	get the background color of a window.
mdlWindow_systemColorGet	get a pointer to the system color descriptor.
mdlWindow_transparentRasterDataDraw	draw mapped image with transparent background.
mdlWindow_transparentRgbDataDraw	set an application's database linkage interests.

Example

See grphptest.mc and rasticon.mc.

mdlWindow_lineStyleSet

```
#include <mdl.h>

void mdlWindow_lineStyleSet
(
MSWindow    *windowP,      /* => window to set attributes for */
int          pattern,      /* => line pattern */
int          color,        /* => line color */
int          mode,         /* => drawing mode */
int          weight        /* => line weight */
);
```

Description mdlWindow_lineStyleSet sets the characteristics of the graphics that are subsequently drawn. The *windowP* argument designates the window that you intend to draw to. On some platforms (such as the PC), the graphic attributes are shared for all windows on a given screen and are not maintained for each window individually. Therefore, as a general rule, set the graphic attributes when you draw on the screen.

pattern specifies an on/off pattern for the line to be drawn. Possible values are 0 through 7, resulting in solid, dotted, medium-dashed, long-dashed, dot-dashed, short-dashed, dash double-dot and long dash-short dash lines, respectively.

color specifies a color index for the graphics. The method for obtaining the appropriate color index depends on whether you want a known color or you will match a color in the design file's active color table. In the former case, use mdlWindow_fixedColorIndexGet; in the latter case, use mdlWindow_colorIndexGet.

mode determines whether the lines will be drawn on the screen (mode 0) or combined with the current screen contents using the XOR (exclusive OR) operation (mode 1).

weight specifies the line thickness. Possible values are 0 to 31. Sometimes two adjacent line thicknesses will display the same on the screen because

of the limited resolution available on some displays. (For example 0 and 1 may set only one pixel).

Returns `mdlWindow_lineStyleSet` is of type `void`. It returns no value.

See Also `mdlWindow_lineStyleSetCD`, `mdlWindow_fixedColorIndexGet`, `mdlWindow_colorIndexGet`.

mdlWindow_lineStyleSetCD

```
void mdlWindow_lineStyleSetCD
(
MSWindow      *windowP,      /* => window to set attributes for */
int            pattern,       /* => line pattern */
BSIColorDescr *colorP,       /* => line color descr. */
int            mode,          /* => drawing mode */
int            lweight        /* => line weight */
);
```

Description The `mdlWindow_lineStyleSetCD` function sets the line drawing style for the window specified by *windowP*.

pattern specifies an on/off pattern to be drawn. Possible values are 0 through 7, resulting in dotted, medium-dashed, long-dashed, dot-dashed, short-dashed, dash double-dot and long dash-short dash lines, respectively.

colorP points to a color descriptor.

mode may be 0, indicating lines are to be drawn on the screen (mode 0) or combined with the current screen contents using the exclusive OR (XOR) operation (mode 1).

lweight specifies the line thickness. Possible values are 0 through 31. Sometimes two adjacent line thicknesses will display the same on the screen because of the limited resolution available on some displays. (For example, 0 and 1 may both set one pixel).

Returns `mdlWindow_lineStyleSetCD` is of type `void`; it returns no value.

See Also `mdlWindow_lineStyleSet`.

mdlWindow_pointDraw, mdlWindow_lineDraw, mdlWindow_lineStringDraw, mdlWindow_shapeFill

```
#include <mdl.h>

void mdlWindow_pointDraw
(
MSWindow      *windowP,      /* => window to draw to */
int            xPos,          /* => point position */
int            yPos,
BSIRect        *clipRectP     /* => overriding clip rectangle */
);
```

```

);

void mdlWindow_lineDraw
(
MSWindow      *windowP,      /* => window to draw to */
int           startX,        /* => starting position */
int           startY,
int           endX,          /* => ending position */
int           endY,
BSIRect       *clipRectP     /* => overriding clip rectangle */
);

void mdlWindow_lineStringDraw
(
MSWindow      *windowP,      /* => window to draw to */
Point2d       *pointP,       /* => point array */
int           numPoints,      /* => # of points in input array */
BSIRect       *clipRectP     /* => overriding clip rectangle */
);

void mdlWindow_shapeFill
(
MSWindow      *windowP,      /* => window to draw to */
Point2d       *pointP,       /* => shape point array */
int           numPoints,      /* => # of points in input array */
BSIRect       *clipRectP     /* => overriding clip rectangle */
);

```

Description The `mdlWindow_pointDraw`, `mdlWindow_lineDraw` and `mdlWindow_lineStringDraw` functions draw points, lines, and line strings (polylines) to a designated window. The `mdlWindow_shapeFill` function fills a shape. The color, pattern, mode, and weight are as set by the most recent call to `mdlWindow_lineStyleSet`. The coordinates for the point, *xPos* and *yPos*, and for the line endpoints, *startX*, *startY*, *endX* and *endY*, are given in local coordinates, as are the vertices in the point array *pointP* for line strings and shapes. For `mdlWindow_lineStringDraw` and `mdlWindow_shapeFill`, the number of vertices in *pointP* is given in *numPoints*.

clipRectP specifies a clipping rectangle that overrides the default clipping. If this argument is `NULL`, the default clipping (to the window's content rectangle) is applied.

Returns All four functions are of type `void`. They return no values.

See Also `mdlWindow_lineStyleSet`.

mdlWindow_textDraw

```
#include <mdl.h>
#include <msdefs.h>
boolean mdlWindow_textDraw
(
  MSWindow      *windowP,      /* => window to draw to */
  int            fontIndex,     /* => font selector */
  Point2d       *pointP,       /* => position of upper left */
  char          *string,        /* => string to display */
  int            fgColor,       /* => color of text */
  int            bgColor,       /* => color of background */
  BSIRect       *clipRectP,     /* => overriding clip rectangle */
  boolean        halfTone       /* => dimmed text flag */
);
```

Description The `mdlWindow_textDraw` function displays text in a window. The window is specified in *windowP*.

fontIndex is set to `FONT_INDEX_SYSTEM`, `FONT_INDEX_BORDER`, `FONT_INDEX_DIALOG` or `FONT_INDEX_BOLD`. This argument designates which of the four predefined system fonts to use. You can configure these fonts using the `MS_SYSFONTS` environment variable. `FONT_INDEX_SYSTEM` is used in the command window. It is also used to open the screen. `FONT_INDEX_BORDER` is used in window borders. `FONT_INDEX_DIALOG` is the main dialog box font, and `FONT_INDEX_BOLD` is used in the rare instances when bold text is required in a dialog box.

pointP designates the upper left corner of the text, which is specified by the character array *string*. The foreground color index is *fgColor*, and the background color index is *bgColor*. See `mdlWindow_lineStyleSet` for information about obtaining the desired color indexes.

A clipping rectangle can be specified with *clipRectP*, or the value `NULL` causes the routine to use the window's content rectangle. If *halfTone* is `TRUE`, the text is drawn half-toned.

Returns The `mdlWindow_textDraw` returns `FALSE` if it is successful and `TRUE` if there was an error. Errors result if *windowP* does not point to a valid window or if *fontIndex* is out of range.

See Also `mdlWindow_textDrawCD`, `mdlWindow_colorIndexGet`, `mdlWindow_fixedColorIndexGet`.

mdlWindow_textDrawCD

```
boolean mdlWindow_textDrawCD      /* 5.0 */
(
MSWindow      *windowP,          /* => window to draw text within */
int            fontIndex,         /* => raster font */
Point2d        *ptP,             /* => point in window (local coords) */
char           *stringP,          /* => string to put out */
BSIColorDescr  *fgColorP,         /* => ptr to foreground color descr */
BSIColorDescr  *bgColorP,         /* => ptr to background color descr */
BSIRect        *clipRectP,        /* => rectangle to clip text to */
int            halfTone           /* => TRUE if draw in halftone */
);
```

Description mdlDialog_textDrawCD is used to draw text in a dialog box with specific foreground and background colors.

windowP is the window to draw in.

fontIndex indicates which raster font to draw in, for example, FONT_INDEX_DIALOG, FONT_INDEX_BOLD or FONT_INDEX_FIXED.

ptP indicates the position for the text.

stringP is the text string to draw.

fgColorP specifies the foreground color for the text. *bgColorP* specifies the background color for the text. If *fgColor* is NULL, mdlWindow_textDrawCD uses the default dialog item foreground color. If *bgColor* is NULL, the default dialog item background color is used.

clipRectP is a clipping rectangle (any drawing outside this rectangle is suppressed).

halfTone indicates whether the text should be drawn in halftone.

Returns mdlWindow_textDrawCD returns SUCCESS, or a nonzero value if there is an error.

See Also mdlWindow_textDraw.

mdlWindow_arcDraw, mdlWindow_ellipseDraw, mdlWindow_ellipseFill

```
#include <mdl.h>

void mdlWindow_arcDraw
(
MSWindow      *windowP,          /* => window to draw to */
Dpoint2d      *originP,          /* => center point of arc */
double        primary,           /* => primary axis */
double        secondary,         /* => secondary axis */
double        start,             /* => starting angle */
double        sweep,             /* => sweep angle */
double        rotation,          /* => rotation of entire arc */
BSIRect        *clipRectP        /* => overriding clip rectangle */
);
```

```

);

void mdlWindow_ellipseDraw
(
MSWindow      *windowP,      /* => window to draw to */
Dpoint2d      *originP,      /* => center point of ellipse */
double        primary,       /* => primary axis */
double        secondary,     /* => secondary axis */
double        rotation,      /* => rotation of entire ellipse */
BSIRect       *clipRectP     /* => overriding clip rectangle */
);

void mdlWindow_ellipseFill
(
MSWindow      *windowP,      /* => window to draw to */
Dpoint2d      *originP,      /* => center point of ellipse */
double        primary,       /* => primary axis */
double        secondary,     /* => secondary axis */
double        rotation,      /* => rotation of entire ellipse */
BSIRect       *clipRectP     /* => overriding clip rectangle */
);

```

Description The `mdlWindow_arcDraw` function draws an arc (partial ellipse) in the window designated by *windowP*. The `mdlWindow_ellipseDraw` and `mdlWindow_ellipseFill` functions draw or fill an ellipse. The origin of the arc or ellipse is given in a double-precision 2D point, *originP*. The primary axis (the axis at zero degrees in the arc/ellipse coordinate system) is passed in *primary*, and the secondary axis (the axis at 90 degrees) is passed in *secondary*. You can produce a circle or circular arc by setting *primary* equal to *secondary*.

For arcs, the starting angle (in radians) and sweep angle are passed in *start* and *sweep*, respectively. The entire arc or ellipse can be rotated by *rotation* degrees and can optionally be clipped by *clipRectP*. If *clipRectP* is NULL, the arc is clipped to the window's content rectangle. The graphic attributes are determined by the most recent call to `mdlWindow_lineStyleSet`.

Returns The `mdlWindow_arcDraw`, `mdlWindow_ellipseDraw` and `mdlWindow_ellipseFill` functions are of type `void`. They return no values.

See Also `mdlWindow_lineStyleSet`.

mdlWindow_rectDraw, mdlWindow_rectClear

```
#include <mdl.h>
void mdlWindow_rectDraw
(
    MSWindow    *windowP,        /* => window to draw to */
    BSIRect      *rectP,          /* => rectangle to draw */
    BSIRect      *clipRectP       /* => overriding clip rectangle */
);
void mdlWindow_rectClear
(
    MSWindow    *windowP,        /* => window to draw to */
    BSIRect      *rectP,          /* => rectangle to clear */
    BSIRect      *clipRectP       /* => overriding clip rectangle */
);
```

Description The mdlWindow_rectDraw function draws an outline of a rectangle in a window using the graphic attributes set by mdlWindow_lineStyleSet, and the mdlWindow_rectClear function clears the rectangle in the window's background color. The affected window is passed in *rectP*. A clipping rectangle can be passed to each function in *clipRectP*. If *clipRectP* is NULL, the rectangle is clipped to the window's content rectangle.

Returns The mdlWindow_rectDraw and mdlWindow_rectClear functions are of type void. They return no values.

See Also mdlWindow_rectFill, mdlWindow_rectFillByRGB, mdlWindow_lineStyleSet, mdlWindow_colorIndexGet, mdlWindow_fixedColorIndexGet.

mdlWindow_rectFill, mdlWindow_rectFillByRGB

```
void mdlWindow_rectFill
(
    MSWindow    *windowP,        /* => window to draw to */
    BSIRect      *rectP,          /* => rectangle to fill */
    int          colorIndex,      /* => color to fill with */
    BSIRect      *clipRectP       /* => overriding clip rectangle */
);
void mdlWindow_rectFillByRGB
(
    MSWindow    *windowP,        /* => window to draw to */
    BSIRect      *rectP,          /* => rectangle to fill */
    RGBColorDef *colorP,          /* => color */
    int          ditherMode,      /* => (0=Pattern, 1=Error Diffusn) */
    BSIRect      *clipRectP       /* => overriding clip rectangle */
);
```

Description `mdlWindow_rectFill` fills a rectangle in a given color. `mdlWindow_rectFillByRGB` fills a rectangle with an RGB color.

windowP is the window to draw to.

rectP is the rectangle to be filled.

colorIndex is the color index that will be used by `mdlWindow_rectFill` to fill the rectangle. *colorP* is the RGB color that will be used by `mdlWindow_rectFillByRGB` to fill the rectangle.

If the graphics display does not support true color, the color will be dithered as specified by *ditherMode*. A *ditherMode* of 0 (`DITHERMODE_Pattern`) will produce a patterned dithering that is generally faster, but less accurate than other dithering methods. A *ditherMode* of one (`DITHERMODE_ErrorDiffusion`) will use an error diffusion filter (Floyd-Steinberg) to produce a more accurate dithering at the expense of increased processing time.

A clipping rectangle can be passed to each function in *clipRectP*. If *clipRectP* is `NULL`, the rectangle is clipped to the window's content rectangle.

Returns `mdlWindow_rectFill` and `mdlWindow_rectFillByRGB` are of type `void`; they return no value.

See Also `mdlWindow_rgbDataDraw`.

mdlWindow_rectInvert

```
void mdlWindow_rectInvert
(
MSWindow      *windowP,      /* => window containing rectangle */
BSIRect       *rectP         /* => rectangle within window */
);
```

Description The `mdlWindow_rectInvert` function XORs the pixels within the rectangle specified by *rectP* in the window specified by *windowP*. It makes use of the color set by the most recent call to `mdlWindow_lineStyleSet`, and the call to `mdlWindow_lineStyleSet` should specify 1 (XOR) for the *mode* parameter. The *pattern* and *weight* parameters to `mdlWindow_lineStyleSet` are ignored by `mdlWindow_rectInvert`.

The color used for the XOR operation is actually the color you specify XOR'd with the background color for the window. Thus, if you know you want the majority of pixels in the inverted rectangle to end up with a particular color index after the operation, use that desired color index in the call to `mdlWindow_lineStyleSet`.

Returns `mdlWindow_rectInvert` is of type `void`. It does not return a value.

See Also `mdlWindow_lineStyleSet`.

mdlWindow_iconDraw

```
#include <mdl.h>
#include <dlogitem.h>

void mdlWindow_iconDraw
(
    MSWindow    *windowP,      /* => window to draw to */
    IconRsc     *iconRP,      /* => icon to draw */
    Point2d     *originP,     /* => upper left area of icon */
    int         onColor,       /* => color of 1 bits */
    int         offColor,      /* => color of 0 bits */
    int         style,         /* => icon drawing style */
    BSIRect     *clipRectP    /* => overriding clip rectangle */
);
```

Description The `mdlWindow_iconDraw` function draws an icon in the window specified by *windowP*. An icon is a binary bitmap data element consisting of a width, height, and actual data. Icons are stored as resources in a resource file and retrieved using the `mdlResource_load` function.

One MDL example program, `rasticon.mc`, creates icon resources. *iconRP* points to the icon resource.

onColor is the color index that draws the on (1) bits in the icon data, and *offColor* is the color index that draws the off (0) bits.

style lets icons be drawn in several different half-tone styles.

`ICON_STYLE_NORMAL` draws the icon exactly as it is stored.

`ICON_STYLE_LGREY` covers every fourth dot of the icon with a black dot,

`ICON_STYLE_DGREY` covers every third dot with a black dot, and

`ICON_STYLE_DISABLED` makes the icon half-tone.

clipRectP clips the icon. If *clipRectP* is `NULL`, the icon is clipped to the window's content rectangle.

Returns The `mdlWindow_iconDraw` function is of type `void`. It returns no value.

See Also `mdlWindow_colorIndexGet`, `mdlWindow_fixedColorIndexGet`.

mdlWindow_rasterDataDraw

```

#include <mdl.h>
void mdlWindow_rasterDataDraw
(
MSWindow      *windowP,      /* => window to draw to */
BSIRect       *rectP,        /* => rectangle to put data in */
byte          *srcDataP,     /* => raster data */
int           srcPitch,      /* => # of bytes in each input row */
int           displayMode,   /* => how to display the data */
int           fgColor,       /* => color of non-zero bytes */
int           bgColor,       /* => color of zero bytes */
BSIRect       *clipRectP     /* => overriding clip rectangle */
);

```

Description `mdlWindow_rasterDataDraw` draws arbitrary raster data in the window specified in *windowP*. The output rectangle for the data is specified in *rectP*, and the actual data, one byte per pixel starting at the upper left and going to the right and down, is stored in an array pointed to by *srcDataP*. If the source pitch or (width) of each row of data differs from the width of the rectangle, it is specified in *srcPitch*. Otherwise, *srcPitch* can be set to zero.

If *displayMode* is `RASTMODE_BYTE_MONO`, every byte that does not match *bgColor* is drawn on the screen as a pixel with color index *fgColor*.

If *displayMode* is `RASTMODE_BYTE_FIXEDCOLOR`, *fgColor* and *bgColor* are ignored. The byte values pointed to by *srcDataP* must be one of the following codes: `BLACK_INDEX`, `BLUE_INDEX`, `GREEN_INDEX`, `CYAN_INDEX`, `RED_INDEX`, `MAGENTA_INDEX`, `YELLOW_INDEX`, `WHITE_INDEX`, `LGREY_INDEX`, `DGREY_INDEX`, `MGREY_INDEX`.

If *displayMode* is `RASTMODE_BYTE_V4_COLOR`, *fgColor* and *bgColor* are ignored. The byte values pointed to by *srcDataP* will be treated as byte indices into the hardware color palette. To obtain byte indices into the hardware color palette, call `mdlWindow_colorIndexGet`, `mdlColor_convertRGBtoIndex` or `mdlWindow_fixedColorIndexGet` in conjunction with `mdlColorDescr_setDrawIndex`. Also, `mdlColor_matchColorMap` can be used to obtain an array of hardware color palette indices corresponding to an array of RGB values.

The raster data can be clipped to a rectangle specified by *clipRectP*. If *clipRectP* is `NULL`, the window's content rectangle is used.

Returns The `mdlWindow_rasterDataDraw` function is of type `void`. It returns no value.

See Also `mdlWindow_colorIndexGet`, `mdlWindow_fixedColorIndexGet`, `mdlColor_convertRGBtoIndex`, `mdlColor_matchColorMap`, `mdlColorDescr_getDrawIndex`.

mdlWindow_rgbDataDraw

```
int mdlWindow_rgbDataDraw
(
MSWindow    *gwP,          /* => window to copy to */
BSIRect     *rectP,        /* => portion of window to update */
int          pitch,        /* => source pitch */
byte        *rgbBufferP,   /* => RGB data */
int          ditherMode,   /* => (0=Pattern, 1=Error Diffusion) */
int          format,       /* => (0=RRRGGBBBB, 1=RGB, 2=RGBA) */
BSIRect     *clipRectP     /* => overriding clip rectangle */
);
```

Description mdlWindow_rgbDataDraw draws a block of RGB data to the window specified by *gwP*. The output rectangle for the data is specified in *rectP* and the actual data is specified by *rgbBufferP* with the top row first. If the source pitch (or row width) of the input data differs from the width of the rectangle, it is specified in *srcPitch*, otherwise *srcPitch* can be set to zero.

The format of the rows of RGB data is specified by *format*. The possible values and meanings are as follows:

<i>format</i> value	Meaning	4 pixel row example
0 (TRUECOLOR_Seperate)	Data stored separately	RRRRGGGGBBBB
1 (TRUECOLOR_RGB)	Data interleaved	RGBRGBRGBRGB
2 (TRUECOLOR_RGBA)	Data interleaved w/ Alpha (Alpha data is actually ignored)	RGBARGBARGBARGBA

If the graphics display does not support true color, the data will be dithered as specified by *ditherMode*. A *ditherMode* of 0 (DITHERMODE_Pattern) will produce a patterned dithering that is generally faster, but less accurate than other dithering methods. A *ditherMode* of one (DITHERMODE_ErrorDiffusion) will use an error diffusion filter (Floyd-Steinberg) to produce a more accurate dithering at the expense of increased processing time.

Returns mdlWindow_rgbDataDraw returns SUCCESS if the data is successfully drawn, and an appropriate error code otherwise (see mdlerrs.h).

See Also mdlWindow_rectFillByRGB, mdlWindow_rasterDataDraw.

mdlWindow_colorIndexGet

```
#include <mdl.h>
int mdlWindow_colorIndexGet
(
MSWindow      *windowP,      /* => window to get index for */
int           colorNumber,    /* => element color number */
int           fileNumber     /* => design file number */
);
```

Description `mdlWindow_colorIndexGet` gets the index of a color that matches the color in which a MicroStation graphic element is drawn. When MicroStation starts, it downloads a balanced color table (with the maximum size allowed by the number of colors supported by the graphic hardware) to the graphic hardware. It then processes the master file and reference file color tables to find each element's color index that most closely matches one of the balanced colors. `mdlWindow_colorIndexGet` returns those color indices.

colorNumber is the position (0-255) in the master file or reference file color table.

fileNumber is 0 for the master file and 1-255 (the attachment number) for reference files.

windowP specifies the window and subsequently the graphic screen for which you need the color index.

Returns The `mdlWindow_colorIndexGet` function returns a pointer to a color descriptor from which the color index may be obtained. In most instances, the color descriptor returned by this function may be used "as is." That is, any `mdlWindow_...` functions that take a color as an argument can use the color descriptor returned by this function. If the color being obtained by this function is going to be stored in a raster byte map, such as one that might be passed in to `mdlWindow_rasterDataDraw`, then it must first be dereferenced using the `mdlColorDescr_getDrawIndex` function.

See Also `mdlWindow_fixedColorIndexGet`, `mdlWindow_rasterDataDraw`, `mdlColorDescr_getDrawIndex`.

mdlWindow_fixedColorIndexGet

```
#include <mdl.h>
#include <msdefs.h>

int mdlWindow_fixedColorIndexGet
(
MSWindow      *windowP,      /* => window to get index for */
int           menuColor      /* => color number */
);
```

Description `mdlWindow_fixedColorIndexGet` returns the graphic hardware color index for the fixed colors available to MDL applications. *menucolor* specifies the color needed.

Possible values for *menuColor* are BLACK_INDEX, BLUE_INDEX, GREEN_INDEX, CYAN_INDEX, RED_INDEX, MAGENTA_INDEX, YELLOW_INDEX, WHITE_INDEX, LGREY_INDEX, DGREY_INDEX and MGREY_INDEX. On a monochrome screen, all values except BLACK_INDEX return 1.

Returns The `mdlWindow_fixedColorIndexGet` function returns a pointer to a color descriptor from which the color index may be obtained. In most instances, the color descriptor returned by this function may be used “as is.” That is, any `mdlWindow_...` functions that take a color as an argument can use the color descriptor returned by this function. If the color being obtained by this function is going to be stored in a raster byte map, such as one that might be passed in to `mdlWindow_rasterDataDraw`, then it must first be dereferenced using the `mdlColorDescr_getDrawIndex` function.

See Also `mdlWindow_colorIndexGet`, `mdlWindow_rasterDataDraw`, `mdlColorDescr_getDrawIndex`.

mdlWindow_isTrueColorCapable

```
boolean mdlWindow_isTrueColorCapable
(
MSWindow      *windowP  /* => window to test */
);
```

Description The `mdlWindow_isTrueColorCapable` function is used to determine whether the specified window has true color capability. If a window has true color capability, graphics drawn into the window can be drawn in any color, rather than only a color that is defined in MicroStation’s color palette.

Returns `mdlWindow_isTrueColorCapable` returns TRUE if the graphics hardware that the window is on supports true color graphics, and FALSE otherwise.

mdlWindow_backgroundColorSet

```
boolean mdlWindow_backgroundColorSet
(
MSWindow      *windowP,          /* => window whose bkcolor to set */
BSIColorDescr *bgColorP         /* => */
);
```

Description `mdlWindow_backgroundColorSet` changes the background color of the window specified by *windowP* to the color described by *bgColorP*. If *bgColorP* has the value BSIBGCOLOR_DIALOG, the default background color is used.

Returns `mdlWindow_backgroundColorSet` returns SUCCESS, or a nonzero value if *windowP* is NULL.

See Also `mdlWindow_backgroundCDGet`.

mdlWindow_backgroundCDGet

```

BSIColorDescr *mdlWindow_backgroundCDGet
(
  MSWindow      *windowP,      /* => window whose bkcolor to get */
  int           viewOrDialog    /* => if windowP=NULL, get vw or dbox? */
);

```

Description mdlWindow_backgroundCDGet is used to get a pointer to the color descriptor for the background color of the window indicated by *windowP*. If *windowP* is NULL, a pointer to the default background color descriptor for either views or dialogs is returned, depending on the value of *viewOrDialog*. BSIBGCOLOR_VIEW causes the default view background to be returned, and BSIBGCOLOR_DIALOG causes the default dialog background to be returned.

Returns mdlWindow_backgroundCDGet always returns a pointer to a color descriptor.

See Also mdlWindow_backgroundColorSet.

mdlWindow_systemColorGet

```

BSIColorDescr *mdlWindow_systemColorGet
(
  int           systemColorIndex
);

```

Description mdlWindow_systemColorGet returns a pointer to the system color descriptor specified by *systemColorIndex*, which may have values such as SYSCOLOR_MOTIF_DFOREGROUND, SYSCOLOR_WINDOWS_WINTXT and so on. These values are defined in msdefs.h.

Returns mdlWindow_systemColorGet returns a pointer to a system color descriptor, or NULL if *systemColorIndex* is out of range.

mdlWindow_transparentRasterDataDraw

```
#include <image.h>
#include <mdl.h>
#include <basetype.h>
#include <msimgd1m.fdf>
int mdlWindow_transparentRasterDataDraw
(
    MSWindow*windowP,      /* => window to draw to */
    BSIRect *rectP,        /* => drawing rectangle */
    byte  *inP,            /* => input byte map */
    int    srcPitch,        /* => source pitch */
    int    monoChrome,      /* => (unused) TRUE if data is monochrome */
    int    foreground,      /* => (unused) foreground color for monochrome */
    int    background,      /* => (unused) background color for monochrome */
    BSIRect *clipRectP,     /* => clipping rectangle (or NULL) */
    int    transparent      /* => transparent index */
);
```

Description The `mdlWindow_transparentRasterDataDraw` draws a block of raster data to the window identified by *windowP*, and overwrites the window's background only when the raster data does not match the specified transparent color.

Source data is a single byte per pixel, mapped to the screen palette for the specified window.

Each byte of source data is mapped to a single output pixel, starting at the upper left corner and going to the right and downward.

windowP identifies the window on which to draw the image data.

rectP identifies the output rectangle to be drawn. Values of *rectP* are specified in window coordinates.

inP identifies that source image data buffer, which contains a single byte per pixel. The value of each byte is index into the palette for the screen on which *windowP* is written.

srcPitch identifies the width of each input row. If the width of each source row is greater than the width to be displayed as specified in *rectP*, then the row width is specified in *srcPitch*. If *srcPitch* is zero, the width is computed from *rectP*. *srcPitch* should be zero, or greater than or equal to the width identified by *rectP*.

monoChrome, *foreground*, *background* are currently unused.

clipRectP identifies a clipping rectangle in window coordinates. If *clipRectP* is NULL, the window's content rectangle is used.

transparent is the hardware draw index used to specify the transparent color. This value is normally obtained from `mdlColorDescr_setDrawIndex`. Values in the source data that match the least significant byte of transparent will be skipped when image data is drawn to the window.



This function was implemented in MicroStation 95.

Returns mdlWindow_transparentRasterDataDraw returns SUCCESS.

See Also mdlWindow_rasterDataDraw, mdlWindow_transparentRgbDataDraw, mdlColorDescr_getDrawIndex.

mdlWindow_transparentRgbDataDraw

```
#include <image.h>
#include <mdl.h>
#include <basetype.h>
#include <mdlerrs.h>
#include <msdefs.h>
#include <msimgdlm.fdf>

int mdlWindow_transparentRgbDataDraw
(
    MSWindow      *windowP,      /* => window to draw to */
    BSIRect       *rectP,        /* => portion of window to update */
    int           srcPitch,      /* => source pitch (or zero to use rectP) */
    byte          *rgbBufferP,   /* => RGB data */
    int           format,        /* => (0=RRRGGBBBB, 1=RGB, 2=RGBA) */
    BSIRect       *clipRectP,    /* => potentially overriding clipdescr */
    RGBColorDef   *transparentRgbP /* => transparent RGB */
);
```

Description The mdlWindow_transparentRgbDataDraw function draws a block of RGB data to the window identified by *windowP*, and overwrites the window's background only when the RGB data does not match the specified transparent color.

Source data contain the red, blue and green color intensities in one of several possible formats. If the display supports true color, data will be written with these values. If the display does not support true color, colors will be dithered.

Each RGB source data value is mapped to a single output pixel, starting at the upper left corner and going to the right and downward.

windowP identifies the window on which to draw the image data.

rectP identifies the output rectangle to be drawn. Values of *rectP* are specified in window coordinates.

srcPitch identifies the width of each input row. If the width of each source row is greater than the width to be displayed as specified in *rectP*, then the row width is specified in *srcPitch*. If *srcPitch* is zero, the width is computed from *rectP*. *srcPitch* should be zero, or greater than or equal to the width identified by *rectP*.

rgbBufferP identifies that source image data buffer, which contains a the red, green and blue color intensities for each pixel.

format identifies the format of the RGB buffer. The possible values and meanings are as follow:

format value	Meaning	4 pixel row example
0 (TRUECOLOR_Seperate) 5 (IMAGEFORMAT_RGBSperate)	Data stored separately	RRRRGGGGBBBB
1 (TRUECOLOR_RGB) 6 (IMAGEFORMAT_RGB)	Data interleaved	RGBRGBRGBRGB
2 (TRUECOLOR_RGBA) 7 (IMAGEFORMAT_RGBA)	Data interleaved w/Alpha (Alpha data is actually ignored)	RGBARBGBARBGBA

clipRectP identifies a clipping rectangle in window coordinates. If *clipRectP* is NULL, the window's content rectangle is used.

transparentRgbP identifies the RGB value used to specify the transparent color. RGB values in the source data that match this color will be skipped when image data is drawn to the window.



This function was implemented in MicroStation 95.

Returns mdlWindow_transparentRgbDataDraw returns SUCCESS if no error is detected. MDLERR_NOMATCH is returned if an invalid format is detected.

See Also mdlWindow_rgbDataDraw, mdlWindow_transparentRasterDataDraw.

Window Docking Functions

The following table lists the window docking functions:

Function	Used to
mdlWindow_dockWindow	dock a window.
mdlWindow_getDocked	get the docked region of a window.
mdlWindow_isDockable	check if a window is dockable.
mdlWindow_organizeApplicationArea	update the positions of all docked windows.
mdlWindow_undockWindow	undock a window.

mdlWindow_dockWindow

```
#include <msdefs.h>
#include <mswindow.fdf>

StatusInt mdlWindow_dockWindow
(
    MSWindow    *windowP,          /* => window to dock */

```

```
int    dockPosition, /* => region in which to dock window */
int    dockPriority  /* => priority of window within region */
);
```

Description The `mdlWindow_dockWindow` function docks a window. The window must be capable of being docked. `mdlWindow_organizeApplicationArea` must be called after any dock/undock operation to organize the layout of the dock regions and, if necessary, reposition windows.

windowP is the window to dock.

dockPosition is a region number. `DOCK_TOP`, `DOCK_BOTTOM`, `DOCK_LEFT`, and `DOCK_RIGHT` are valid values for *dockPosition*.

dockPriority is the relative priority of a window within a given region. It is typically set based on one of the following values: `DOCKPRIORITY_Basetop`, `DOCKPRIORITY_Basebottom`, `DOCKPRIORITY_Baseleft` or `DOCKPRIORITY_Baseright`. A lower number decreases the window's priority and moves the window down and to the right in the selected dock region.



This function was implemented in MicroStation 95.

Returns `mdlWindow_dockWindow` returns `SUCCESS` if *windowP* is successfully docked.

See Also `mdlWindow_undockWindow`, `mdlWindow_organizeApplicationArea`, `mdlWindow_getDocked`, `mdlWindow_isDockable`.

mdlWindow_getDocked

```
#include <msdefs.h>
#include <mwindow.fdf>

int mdlWindow_getDocked
(
MSWindow    *windowP /* => window to check for docking region */
);
```

Description The `mdlWindow_getDocked` function is used to determine a window's current region number.

windowP is the window to test.



This function was implemented in MicroStation 95.

Returns For docked windows, `mdlWindow_getDocked` returns `DOCK_TOP`, `DOCK_BOTTOM`, `DOCK_LEFT`, or `DOCK_RIGHT` depending on the windows docked region number. If the window is not docked or there is an error, `DOCK_NOTDOCKED` is returned.

See Also `mdlWindow_dockWindow`, `mdlWindow_undockWindow`, `mdlWindow_isDockable`.

mdlWindow_isDockable

```
#include <mwindow.fdf>
#include <mdl.h>

BoolInt mdlWindow_isDockable
(
MSWindow    *windowP  /* => window to test */
);
```

Description The mdlWindow_isDockable function tests whether or not the designated window is dockable.

windowP is the window to test.



This function was implemented in MicroStation 95.

Returns mdlWindow_isDockable returns TRUE if *windowP* is a dockable window. Otherwise, FALSE is returned.

See Also mdlWindow_dockWindow, mdlWindow_undockWindow, mdlWindow_getDocked.

mdlWindow_organizeApplicationArea

```
#include <mwindow.fdf>

void mdlWindow_organizeApplicationArea(void);
```

Description The mdlWindow_organizeApplicationArea function repositions all docked windows. It must be called after a window (or group of windows) is either docked or undocked.



This function was implemented in MicroStation 95.

Returns mdlWindow_organizeApplicationArea is void. It returns no value.

See Also mdlWindow_dockWindow, mdlWindow_undockWindow.

mdlWindow_undockWindow

```
#include <mdl.h>
#include <basetype.h>
#include <mwindow.fdf>

StatusInt mdlWindow_undockWindow
(
MSWindow    *windowP,          /* => window to undock */
Point2d     *newPosP          /* => position of window after undocking */
);
```

Description The mdlWindow_undockWindow function undocks a window. mdlWindow_undockWindow has no affect on a window that is not currently docked. mdlWindow_organizeApplicationArea must be called after any dock/undock

operation to organize the layout of the dock regions and, if necessary, reposition windows.

windowP is the window to undock.

newPosP is the position, in global coordinates, of where to move the upper left corner of the window's content rectangle after it is undocked.



This function was implemented in MicroStation 95.

Returns mdlWindow_undockWindow returns SUCCESS if *windowP* is successfully undocked.

See Also mdlWindow_dockWindow, mdlWindow_organizeApplicationArea, mdlWindow_getDocked, mdlWindow_isDockable.

Resource Management Functions

The resource management functions are used to create, manipulate and destroy resources and resource files. For background information on how resources are used, see the "Developing Applications" chapter of the Programmer's Guide.

The following table lists resource management functions:

Function	Used to
mdlResource_openFile	open a resource file.
mdlResource_closeFile	close a resource file.
mdlResource_createFile	create a resource file.
mdlResource_createFileForPlatform	create a resource file optimized for a specific platform.
mdlResource_load	load resource.
mdlResource_loadByAlias	load resource qualified by an alias name.
mdlResource_loadFromStringList	load a string from a StringList resource.
mdlResource_directLoad	give resource loading control to the MDL application.
mdlResource_free	remove a resource from memory.
mdlResource_write	write an updated resource to its original position in the resource file.
mdlResource_resize	adjust the size of a resource.
mdlResource_add	add a new resource to a resource file according to a unique resource ID.

Function	Used to
mdlResource_addByAlias	add a new resource to a resource file according to a unique resource ID and alias name.
mdlResource_directAdd	add a new resource using fwrites.
mdlResource_directAddComplete	inform the resource manager that a new resource was added with mdlResource_directAdd.
mdlResource_delete	delete a resource with a given resource ID from a resource file.
mdlResource_deleteByAlias	delete a resource with a given resource ID and alias name from a resource file.
mdlResource_query	query the resource manager for information about an individual resource.
mdlResource_queryClass	query the resource manager for information on a <i>resourceclass</i> .
mdlResource_queryClassByAlias	query the resource manager for information on a <i>resourceclass</i> , but only for resources with the specified alias name.
mdlResource_queryFile	query the resource manager for information on a resource file.
mdlResource_queryFileHandle	query the resource manager for information on an open resource file.
mdlResource_getAppLoadFile	get the name of the resource file from which a given task was loaded.
mdlResource_getRscIdByAlias	use a resource's alias name to acquire its ID.
mdlResource_changeAlias	change a resource's alias name.

Example

See resmover.mc.

Binary Portable Resource Files

Starting with MicroStation Version 5, resource files are binary portable across all platforms. When Resource Manager functions are used to add resources to a resource file, there are some steps that application developers must take to ensure their portability. The following discussion describes how the MicroStation Resource Manager achieves binary portability and what developers need do to support it.

Machine format

The format of resources within a resource file can match the characteristics of any platform (PC, Clipper, SPARC, etc.). However, resource formats cannot be mixed in a given file. When the format of a file matches the format of the platform where it is currently being used, the file is said to be “optimized” for that platform. When a file is optimized for a platform, its resources can be read and written without the need for conversion. When the formats differ, a slight overhead is incurred for each resource file access as the Resource Manager must convert the resources.

Which format you should use depends on the needs of the application, but the format should usually reflect the platform where the file will be used the most.

Data definitions

The Resource Manager uses data definitions to convert resources. Within each resource file, there will be one data definition for each unique `resourceclass`. Each data definition acts as a “road map” for converting the resources of its associated `resourceclass`.

If the target resource file is optimized for the current platform, the data definitions are never used. Otherwise, they are used whenever resources are loaded from or added to a resource file.

The resource manager looks for a data definition in the target resource file first. If it cannot find one there, it begins searching all the files currently opened by the calling task. As a last resort, it looks in the resource files opened by MicroStation. If a data definition still cannot be found, the requested action (load or add) will fail.

Data definitions are generated automatically by the Resource Compiler (rcomp). A data definition is generated for each `resourceclass` that has at least one resource instance defined during the compilation.

Generating data definitions for dynamically created resources

The data definitions for resources added at runtime must already exist (somewhere) at the time the new resources are added to the file. Therefore, your application’s build process will need to generate all data definitions that your application might ever need and rlib them into your application’s `.ma` file.

The Resource Compiler has an option (`-r`) that instructs it to generate a data definition for each `resourceclass` statement it encounters regardless of whether an actual resource instance appears. Use this option to generate a `.rsc` file from a `.r` file that simply includes header files with the appropriate resource classes defined. Merge the resulting `.rsc` file into your `.ma` file using `rlib`. Then in your application code, make sure you

open your application's .ma file so the data definitions can be found by the Resource Manager.

Resources with Alias Names

First, and this is how they were originally intended, alias names were meant as a symbolic alias to the resource identifier (ID). Like a resource ID, each alias had to be unique within a resourceclass. Four functions operate on this principle:

mdlResource_add, mdlResource_directAdd, mdlResource_getRscIdByAlias and mdlResource_changeAlias.

Due to a deliberate "looseness" in the way alias names were implemented in the resource compiler (rcomp), a second and perhaps more popular way of using alias names has evolved. This second use is now supported in the resource manager as well. The alias name is sometimes used as an additional level of identification beyond the resource identifier. This allows multiple resources with the same resourceclass and resource ID to exist in the same file, as long as they have differing alias names. Or conversely, multiple resources with the same resourceclass and alias name may exist in the same file, as long as their resource IDs differ. However, two resources, from the same file, within the same resourceclass, with identical IDs and alias names, is still disallowed. Several resource manager functions operate under this second scheme, that is, they take the resource ID and alias name together when testing for uniqueness within a resourceclass in a single file. These functions are: mdlResource_loadByAlias, mdlResource_addByAlias, mdlResource_deleteByAlias and mdlResource_queryClassByAlias.

mdlResource_openFile

```
#include <mdl.h>
#include <rscdefs.h>
#include <mdlerrs.h>

int mdlResource_openFile /* <= SUCCESS */
(
    RscFileHandle    *rfHandle,    /* <= Resource File Handle */
    char             *fileName,    /* => Name of File to open */
    ULong            fileAttrMask /* => file attributes bit mask */
);
```

Description The mdlResource_openFile function opens a resource file, making its contents available to the application using subsequent mdlResource_... function calls.

The resource manager stores the resource file handle corresponding to the successfully opened file in *rfHandle*. The resource file handle is a parameter needed for most mdlResource_... functions.

fileName specifies the target file to be opened. If the resource manager cannot find the file specified by *fileName*, it will search for the file in the path specified by the MS_RSRCPATH environment variable. If *fileName* is NULL, the resource manager will default to the calling application's original load file.

fileAttrMask specifies file attributes to be assigned to the file such as the read/write capability. Each file attribute is specified as a bit mask. Several may be logically ORed together if more than one file attribute needs to be assigned. The valid file attribute masks, defined in rscdefs.h, are:

Mask value	Description
RSC_READ	Open file for read access, allow other writers.
RSC_READDENYWRITE	Open file for read access, deny writers.
RSC_READWRITE	Open file with read/write access.



Do not confuse the MS_RSRCPATH environment variable with the MS_RSRC environment variable used to specify the location for the MicroStation resource file, ustation.rsc.

Returns The mdlResource_openFile function returns SUCCESS if the file was opened successfully. Otherwise, it returns the following error defined in mdlerrs.h:

Value	Description
MDLERR_RSCFILENOTFOUND	<i>fileName</i> could not be found.

See Also mdlResource_createFile, mdlResource_closeFile.

mdlResource_closeFile

```
#include <mdl.h>
#include <rscdefs.h>
#include <mdlerrs.h>

void mdlResource_closeFile
(
  RscFileHandle    rfHandle    /* => Resource File handle */
);
```

Description The mdlResource_closeFile function closes a resource file. All resources currently in memory that were loaded from the file being closed are automatically freed.



Do not call mdlResource_free for a resource that has already been freed by virtue of a call to mdlResource_closeFile.

rfHandle is the resource file handle corresponding to the file to close.

rfHandle is originally obtained from the mdlResource_openFile function.

Returns The mdlResource_closeFile function is of type void. It returns no value.

See Also mdlResource_openFile.

mdlResource_createFile, mdlResource_createFileForPlatform

```
#include <mdl.h>
#include <rscdefs.h>
#include <mdlerrs.h>

int mdlResource_createFile
(
    char    *fileName,      /* => Name of file to create */
    char    *ident,         /* => ASCII string describing file */
    long    version         /* => User-assigned version number */
);

int mdlResource_createFileForPlatform
(
    char    *fileName,      /* => Name of file to create */
    char    *ident,         /* => ASCII string describing file */
    long    version,        /* => User-assigned version number */
    int     platformID      /* => optimize the file for this platform */
);
```

Description mdlResource_createFile and mdlResource_createFileForPlatform are used to generate a new resource file. The resulting file must be opened with mdlResource_openFile before new resources may be added to it.

fileName is the name of the new resource file to be created.

ident is an identity string containing up to 63 characters that is stored in the new resource file header.

version is an application-defined 32-bit integer stored in the file header.

The mdlResource_createFile function generates a resource file optimized for (i.e., in the format of) the current platform.

To create and optimize a resource file for a specific platform, the mdlResource_createFileForPlatform function is called with the additional *platformID* parameter. The mdlResource_createFileForPlatform function can also be used to generate a resource file that is compatible with version 4.x of MicroStation (for the current platform only).

platformID is a 32 bit identifier used to identify one of the platforms supported by MicroStation. The valid platform identifiers are defined in `basedefs.h`. Here are a few of the IDs:

ID	Description
PLATFORM_CLIPPER	Binary portable, optimized for Intergraph Workstation.
PLATFORM_HP_X	Binary portable, optimized for HP700.
PLATFORM_MAC_MACOS	Binary portable, optimized for Macintosh.
PLATFORM_PC_BSIWINMGR	Binary portable, optimized for PC.
PLATFORM_PC_WINNT	Binary portable, optimized for Windows NT on the PC.
PLATFORM_SPARC_X	Binary portable, optimized for SPARCstation.
PLATFORM_VAX_VMS_X	Binary portable, optimized for VAXstation.
PLATFORM_4X	Non-binary portable, compatible with 4.x MicroStation running on the current platform.

Returns The `mdlResource_createFile` and `mdlResource_createFileForPlatform` functions return `SUCCESS` if the file was created. Otherwise, they return one of the following errors defined in `mdlerrs.h`:

Value	Description
MDLERR_RSCFILEERROR	An undefined error was encountered.
MDLERR_RSCINSFMEM	Memory is insufficient to create the file header.
MDLERR_INVALIDPLATFORMID	Invalid platform identifier.

See Also `mdlResource_openFile`.

mdlResource_load, mdlResource_loadByAlias

```
#include <mdl.h>
#include <rsdefs.h>
#include <mdlerrs.h>

void *mdlResource_load
(
    RscFileHandle    rfHandle,        /* => Resource File Handle */
    unsigned long    resourceclass,   /* => resourceclass identifier */
    unsigned long    resourceID       /* => Resource identifier */
);

void *mdlResource_loadByAlias
(
    RscFileHandle    rfHandle,        /* => Resource File Handle */
    unsigned long    resourceclass,   /* => resourceclass identifier */
    unsigned long    resourceID,      /* => Resource identifier */
    char             *alias           /* => alias of resource */
);
```

);

Description The `mdlResource_load` and `mdlResource_loadByAlias` functions are used to load resources from an opened resource file.

rfHandle is the resource file handle corresponding to a file that was previously opened with the `mdlResource_openFile` function. If *rfHandle* equals zero, the resource manager will attempt to satisfy the request in one of two ways. First, it will search for the specified resource in all resource files currently opened by the calling application. If that search fails, it will attempt to find the resource in the system resource files currently opened by MicroStation. In either case, the files are searched in last opened, first searched sequence.

If the file specified by *rfHandle* is optimized for a different platform from the current platform, the resource manager will automatically convert the resource to the current platform format. It does this by finding a data definition resource and then using it to convert the target resource to native machine format. The data definition resource can come from any file opened by the current application, even from a file optimized for a different file format than the target resource.

For example, let's say the current platform is the Macintosh, the target resource file is optimized for the PC, and a data definition corresponding to the target resource's resourceclass has been found in a Clipper format resource file. The data definition will be translated from describing Clipper format to describe PC format, then the target resource is loaded and translated from PC to Macintosh format.

resourceclass identifies the class of the resource being loaded.

resourceID specifies the resource to load.

`mdlResource_loadByAlias` allows for an additional level of qualification by using the alias name of the resource. In this case, a resource with an ID of *resourceID* and an alias name of *alias* is located and loaded.

Returns The `mdlResource_load` and `mdlResource_loadByAlias` functions return a pointer to the loaded resource if the resource was found in the file. Otherwise, `NULL` is returned and `mdlErrno` will be set to one of the following error codes:

Value	Description
<code>MDLERR_RSCFILENOTFOUND</code>	An undefined error was encountered.
<code>MDLERR_INSMEMORY</code>	Insufficient memory to load the resource.
<code>MDLERR_RSCTYPEINVALID</code>	Invalid resource type for the file specified.
<code>MDLERR_RSCNOTFOUND</code>	Cannot find the resource in the file specified.

Value	Description
MDLERR_RSCFILEERROR	An undefined error was encountered while trying to read the resource from file.
MDLERR_DATADEFNOTFOUND	Could not find a data definition with which to convert the resource to memory format.



The resource pointer returned by `mdlResource_load` or `mdlResource_loadByAlias` is valid only as long as the resource file from which it came stays open and the resource is not freed. Once a resource file is closed, all of its associated resources in memory are immediately released.

See Also `mdlResource_free`.

mdlResource_loadFromStringList

```
#include <mdl.h>
#include <rsdefs.h>
#include <mdlerrs.h>

int mdlResource_loadFromStringList
(
    char            *string,          /* <= Where to store the string */
    RscFileHandle   rfHandle,        /* => Resource File Handle */
    unsigned long   stringListID,    /* => ID of target stringlist */
    unsigned long   stringNum        /* => ID of string in stringlist */
);
```

Description `mdlResource_loadFromStringList` copies the string identified by *stringListID* and *stringNum* into the memory area pointed to by *string*.

If necessary, the appropriate string list is loaded from the file indicated by *rfHandle*.

string points to a memory area where the target string will be copied. It must be large enough to accommodate the target string.

rfHandle is the resource file handle corresponding to a file that was opened using the `mdlResource_openFile` function. If *rfHandle* equals zero, the resource manager will attempt to satisfy the request as it does in `mdlResource_load`.

stringListID is the resource ID of the string list containing the target string.

stringNum specifies the string to load.

Returns The mdlResource_loadFromStringList function returns SUCCESS if the string was loaded. Otherwise, it returns one of the following errors defined in mdlerrs.h:

Value	Description
MDLERR_RSCFILENOTFOUND	The resource file could not be found.
MDLERR_RSCNOTFOUND	<i>stringListID</i> could not be found.

See Also “Message lists” in the “Resources” section of the “Developing Applications” chapter.

mdlResource_directLoad

```
#include <mdl.h>
#include <rsdefs.h>
#include <mdlerrs.h>

int mdlResource_directLoad
(
    RscFileHandle    rfHandle,        /* => Resource File Handle */
    unsigned long    resourceclass, /* => resourceclass identifier */
    unsigned long    resourceID,     /* => Resource identifier */
    RscDirectAccess  *direct         /* <= directLoad destination */
);
```

Description mdlResource_directLoad gives resource loading control to the MDL application.

Like mdlResource_load, its parameter list includes *rfHandle*, *resourceclass*, and *resourceID*. But, while mdlResource_load loads the resource in memory for the application, mdlResource_directLoad obtains a pointer to a structure that contains a file pointer and the resource’s size and position in the file. The MDL application can then load the resource using the standard fread function.



Since the application assumes control for loading the resource, the resource manager cannot convert the resource to memory format (if necessary) as it does in the mdlResource_load function. It is the application’s responsibility to check the parent resource file’s format to see if the file format differs from the current platform’s memory format. See mdlResource_queryFileHandle (*queryID* RSC_QRY_PLATFORM) for details.

Sometimes mdlResource_directLoad is preferred over mdlResource_load. For example, resource data loaded using mdlResource_load remains in memory only as long as the resource file remains open. A direct load, however, lets the application load a resource into the appropriate memory. Thus, the application does not need to keep the resource file open while accessing the resource data. A second reason for using a direct load concerns the size of resources. Since mdlResource_load obtains a resource in its entirety, memory could be wasted if only part of a very large

resource was needed. `mdlResource_directLoad` eliminates this problem by allowing the application to load in sections of a resource.

rfHandle is the resource file handle corresponding to a file that was opened using the `mdlResource_openFile` function. If *rfHandle* equals zero, the resource manager will attempt to satisfy the request as it does with `mdlResource_load`.

resourceclass identifies the type of resource being loaded, and *resourceID* specifies the resource to load.

direct points to the following structure (defined in the MDL system header file `rscdefs.h`) to be filled in by `mdlResource_directLoad`:

```
typedef struct rscdirectaccess
{
    RscFileHandle rfHandle;      /* resource file handle */
    FILE *fileP;                /* file ptr returned by fopen */
    unsigned long filePos;       /* position of rsc. in the file */
    unsigned long rscSize;       /* size of the resource */
} RscDirectAccess;
```



The `rfHandle` structure member will not differ from *rfHandle* unless the function parameter was 0. In this case, *rfHandle* indicates the first file where the resource manager was able to find the target resource. Also, the *fileP* member is not guaranteed to be valid after the calling application has given up program control and another application (or MicroStation) has made a resource manager function call.

Returns `mdlResource_directLoad` returns `SUCCESS` if the `RscDirectAccess` information was obtained. Otherwise, it returns one of these errors defined in `mdlerrs.h`:

Value	Description
<code>MDLERR_RSCFILENOTFOUND</code>	The resource file could not be found.
<code>MDLERR_RSCNOTFOUND</code>	<i>resourceID</i> could not be found.

See Also `mdlResource_load`, `mdlResource_directAdd`, `mdlResource_queryFileHandle`.

mdlResource_free

```
#include <mdl.h>
#include <rscdefs.h>
#include <mdlerrs.h>

int mdlResource_free
(
    void      *resource      /* => Pointer to resource to be freed */
);
```

Description The mdlResource_free function removes a resource from memory. The resource must have been loaded with the mdlResource_load function.

resource points to a resource.



Do not call mdlResource_free for a resource that has already been freed by a call to mdlResource_closeFile.

Returns The mdlResource_free function returns SUCCESS if the resource was removed successfully. Otherwise, it returns the following error defined in mdlerrs.h:

Value	Description
MDLERR_RSCADDRINVALID	<i>resource</i> is an invalid resource pointer.

See Also mdlResource_load, mdlResource_closeFile.

mdlResource_write

```
#include <mdl.h>
#include <rscdefs.h>
#include <mdlerrs.h>

int mdlResource_write
(
void    *resource    /* => Pointer to resource to be written */
);
```

Description The mdlResource_write function writes an updated resource (obtained with mdlResource_load) to its original position in the resource file from which it was loaded. The size of the write is determined by the size of the resource.

resource points to a resource obtained in memory through mdlResource_load.

Returns The mdlResource_write function returns SUCCESS if the write was successful. Otherwise, it returns one of the following errors defined in mdlerrs.h:

Value	Description
MDLERR_RSCWRITEVIOLATION	The resource file was opened as RSC_READONLY.
MDLERR_RSCADDRINVALID	<i>resource</i> is an invalid resource pointer.
MDLERR_RSCFILEERROR	An unrecoverable file error occurred.
MDLERR_DATADEFNOTFOUND	Could not find a data definition with which to convert the resource to file format.

See Also mdlResource_load.

mdlResource_resize

```
#include <mdl.h>
#include <rsdefs.h>
#include <mdlerrs.h>

void *mdlResource_resize /* <= Resized resource or NULL */
(
void                *resource,      /* => Pointer to rsc to be resized */
unsigned long       newSize         /* => New size of the resource */
);
```

Description The `mdlResource_resize` function adjusts the size of a resource in a file. No application except the calling application can load the resource. If *newSize* is larger than the previous size of the resource, the existing resource is padded with zero bytes until its size reaches *newSize* bytes. On the other hand, if *newSize* is smaller than the original size of the resource, bytes in the resource above *newSize* are truncated. If the attempt to resize the resource is successful, a pointer to the newly allocated resource is returned and the previous pointer is no longer valid.

resource points to a resource that was obtained in memory through the `mdlResource_load` function.

newSize is the new size to be assigned to the resource.

Returns The `mdlResource_resize` function returns a pointer to the resized resource. If an error occurs, `NULL` is returned and the global variable `mdlErrno` is set to the specific cause of the error. The following are possible values for `mdlErrno`:

Value	Description
MDLERR_RSCWRITEVIOLATION	The resource file was opened as <code>RSC_READONLY</code> .
MDLERR_RSCINUSE	The resource is being used by more than one application.
MDLERR_RSCADDRINVALID	The <i>resource</i> pointer is invalid.
MDLERR_RSCINSFMEM	Memory is insufficient to process the request.

mdlResource_add, mdlResource_addByAlias

```
#include <mdl.h>
#include <rsdefs.h>
#include <mdlerrs.h>

int mdlResource_add /* <= SUCCESS or error code */
(
RscFileHandle      rfHandle,      /* => Resource File Handle */
unsigned long       resourceclass, /* => resourceclass identifier */
unsigned long       resourceID,    /* => Resource identifier */
void               *resource,     /* => Pointer to would-be resource */
int                size,          /* => Size of would-be resource */
char               *alias         /* => Optional alias name or NULL */
);
```



```
int mdlResource_addByAlias /* <= SUCCESS or error code */
(
    RscFileHandle    rfHandle,      /* => Resource File Handle */
    unsigned long     resourceclass, /* => resourceclass identifier */
    unsigned long     resourceID,    /* => Resource identifier */
    void              *resource,     /* => Pointer to would-be resource */
    int               size,          /* => Size of would-be resource */
    char              *alias         /* => Optional alias name or NULL */
);
```

Description The `mdlResource_add` function is used to add a new resource to a resource file.

rfHandle is the resource file handle for the file where the new resource will be written. *rfHandle* is obtained from the `mdlResource_openFile` function.

resourceclass specifies the new resource's type or class.

resourceID uniquely identifies a resource within its class. If a resource of the specified *resourceclass* and *resourceID* already exists, the request to add the new resource is rejected.

resource points to a block of memory to be added to the file as a resource.

size is the size of the block of memory pointed to by *resource*.

alias is the alias name of the new resource. If no name is needed, *alias* is NULL. If a resource of the specified *resourceclass* and *alias* already exists, the request to add the new resource is rejected.

The `mdlResource_addByAlias` function is identical to the `mdlResource_add` function except the combination of *resourceID* and *alias* are used to ensure uniqueness. Put another way, `mdlResource_add` will fail if either *resourceID* or *alias* is not unique within the *resourceclass*. But with `mdlResource_addByAlias`, the combination of *resourceID* and *alias* are checked as a pair.



If the file specified by *rfHandle* is version 5 (or later) resource file format, then the resource manager must be able to find a data definition corresponding to the resource type being added. This is still true even if the target resource file is optimized for the current platform. The resource manager will be successful in finding a data definition for the target resource if a corresponding data definition can be found in one of the following:

- The target resource file
- Another file currently open by the calling application, or
- A MicroStation system resource file

Returns `mdlResource_add` returns `SUCCESS` if the new resource was added to the resource file. Otherwise, it returns one of the following errors defined in `mdlerrs.h`:

Value	Description
<code>MDLERR_RSCWRITEVIOLATION</code>	The resource file was opened as <code>RSC_READONLY</code> .
<code>MDLERR_RSCALREADYEXISTS</code>	<i>resourceID</i> or <i>alias</i> already exists in the file. In the case of <code>mdlResource_addByAlias</code> , <i>resourceID</i> and <i>alias</i> already exist as a pair on a resource.
<code>MDLERR_RSCFILEERROR</code>	Error encountered while reading the resource file header.
<code>MDLERR_RSCWRITEERROR</code>	Error encountered while writing to the file.
<code>MDLERR_RSCINSFMEM</code>	Memory is insufficient to process the request.
<code>MDLERR_DATADEFNOTFOUND</code>	Could not find a data definition corresponding to the resourceclass.

See Also `mdlResource_directAdd`.

mdlResource_directAdd

```
#include <mdl.h>
#include <rsdefs.h>
#include <mdlerrs.h>

int mdlResource_directAdd
(
    RscFileHandle    rfHandle,           /* => Resource File handle */
    unsigned long    resourceclass,      /* => resourceclass identifier */
    unsigned long    resourceID,         /* => Resource identifier */
    char             *alias,             /* => Optional alias name */
    RscDirectAccess  *direct             /* <= directAdd destination */
);
```

Description The `mdlResource_directAdd` and `mdlResource_directAddComplete` functions give the MDL programmer full control of writing resources to a resource file. Another routine for adding new resources, `mdlResource_add`, requires the resource (memory area) to exist in its entirety at the time it is added to the file. After calling `mdlResource_directAdd`, however, the programmer can write the new resource to the file piece by piece as necessary using the `fwrite` C function. The size of the resulting resource is equal to the sum of the sizes of the `fwrites` and is computed when the programmer calls `mdlResource_directAddComplete`.



Since the application assumes control for creating the resource, the resource manager cannot convert the resource from memory to file format (if necessary) as it does in the `mdlResource_add` function. It is the application's responsibility to check the parent resource file's format to see if the file format differs from the current platform's memory format. See `mdlResource_queryFileHandle` (*queryID* `RSC_QRY_PLATFORM`) for details.

The resource manager verifies that the specified *resourceID* and *alias* of the given *resourceclass* do not already exist in the file specified by *rfHandle*. If the proposed new resource is unique, the resource manager will return the information needed to write directly to the resource file in the structure pointed to by *direct*. The application then uses this information to write the resource to the file using one or more `fwrite` calls. When the application has completed its last `fwrite` call, it calls the `mdlResource_directAddComplete` function to inform the resource manager that the new resource has been added. The resource manager then updates the resource file's header information to reflect the new resource addition.



No intervening resource management function calls should be made between the `mdlResource_directAdd` and `mdlResource_directAddComplete`. Furthermore, no other MDL application or MicroStation should be allowed to access the resource file where the resource is being added until after the `mdlResource_directAddComplete` function call has been made. This means that the MDL application should make both function calls, `mdlResource_directAdd` and `mdlResource_directAddComplete` before returning to MicroStation. For example, the calls would not be made from separate event handlers.

rfHandle is the resource file handle of the file where the new resource will be written. *rfHandle* is obtained from the `mdlResource_openFile` function. This parameter must be specified. (Zero is invalid.)

resourceclass specifies the type or class of the new resource.

resourceID uniquely identifies a resource within its class. If a resource of the specified *resourceclass* and *resourceID* already exists, the request to add the new resource is rejected.

alias is the alias name of the new resource. If no name is needed, *alias* is NULL. If a resource of the specified *resourceclass* and *alias* already exists, the request to add the new resource is rejected.

direct points to the following structure (defined in the MDL system header file `mdlerrs.h`) to be filled in by `mdlResource_directAdd`:

```
typedef struct rscdirectaccess
{
    RscFileHandle rfHandle;      /* resource file handle */
    FILE *fileP;                /* file ptr returned by fopen */
    unsigned long filePos;      /* where to add the rsc */
    unsigned long rscSize;      /* does not apply */
} RscDirectAccess;
```

Returns The `mdlResource_directAdd` function returns `SUCCESS` if the application can safely begin writing the new resource to the resource file. Otherwise, it returns one of the following errors defined in `mdlerrs.h`:

Value	Description
<code>MDLERR_RSCWRITEVIOLATION</code>	The resource file was opened as <code>RSC_READONLY</code> .
<code>MDLERR_RSCALREADYEXISTS</code>	<i>resourceID</i> or <i>alias</i> already exists in the file.
<code>MDLERR_RSCFILE_ERROR</code>	The resource file header cannot be read.
<code>MDLERR_RSCINSFMEM</code>	Memory is insufficient to process the request.
<code>MDLERR_RSCERROR</code>	<code>NULL</code> pointer for <i>direct</i> is invalid.
<code>MDLERR_RSCDIRECTADDPEND</code>	A direct add is in progress.

See Also `mdlResource_directAddComplete`, `mdlResource_directLoad`, `mdlResource_add`.

mdlResource_directAddComplete

```
#include <mdl.h>
#include <rsdefs.h>
#include <mdlerrs.h>

int mdlResource_directAddComplete(void);
```

Description The application uses `mdlResource_directAddComplete` to inform the resource manager that a new resource was added with `mdlResource_directAdd`.

Returns The `mdlResource_directAddComplete` function returns `SUCCESS` if the new resource was added to the resource file. Otherwise, it returns the following error defined in `mdlerrs.h`:

Value	Description
<code>MDLERR_RSCERROR</code>	A direct add is not in progress.

See Also `mdlResource_directAdd`.

mdlResource_delete, mdlResource_deleteByAlias

```
#include <mdl.h>
#include <rsdefs.h>
#include <mdlerrs.h>
int mdlResource_delete
(
    RscFileHandle    rfHandle,          /* => Resource File Handle */
    unsigned long    resourceclass,     /* => resourceclass identifier */
    unsigned long    resourceID        /* => resource to delete */
);
int mdlResource_deleteByAlias
(
    RscFileHandle    rfHandle,          /* => Resource File Handle */
    unsigned long    resourceclass,     /* => resourceclass identifier */
    unsigned long    resourceID,       /* => resource to delete */
    char             *alias            /* => qualify with this alias */
);
```

Description The `mdlResource_delete` function deletes a resource with a given resource ID from a resource file. The `mdlResource_deleteByAlias` function deletes a resource with a given resource ID and alias name from a resource file. The calling application must have opened a file containing the resource using the `mdlResource_openFile` function. If the target resource is currently loaded by any application (including the calling application), the request is rejected. In this case, the resource must be freed using `mdlResource_free` before a subsequent delete request can be honored.

rfHandle is the resource file handle of the file where the resource will be deleted. *rfHandle* is obtained from the `mdlResource_openFile` function. Passing zero for *rfHandle* is not allowed.

resourceclass identifies the type of resource being deleted.

resourceID identifies the specific resource within the *resourceclass* to delete.

alias, only used with `mdlResource_deleteByAlias`, is used along with *resourceID* to uniquely identify the resource to be deleted.

Returns The `mdlResource_delete` and `mdlResource_deleteByAlias` functions return `SUCCESS` if the delete was successful. Otherwise, they return one of the following errors defined in `mdlerrs.h`:

Value	Description
<code>MDLERR_RSCTYPEINVALID</code>	Invalid resource type for the specified file.
<code>MDLERR_RSCWRITEVIOLATION</code>	The resource file was opened as <code>RSC_READONLY</code> .
<code>MDLERR_RSCINUSE</code>	The specified resource is loaded and in use; it cannot be deleted.
<code>MDLERR_RSCNOTFOUND</code>	The specified resource could not be found in the resource file.

mdlResource_query

```
#include <mdl.h>
#include <rsdefs.h>
#include <mdlerrs.h>

int mdlResource_query
(
    void          *reply,          /* <= Where to store the query reply */
    void          *resource,       /* => Resource being queried */
    int           queryID         /* => Query ID */
);
```

Description The `mdlResource_query` function queries the resource manager for information regarding a resource.

reply points to the location where the resource manager must place its answer to the request. The type of data that it points to depends on the query.

resource points to a resource loaded with the `mdlResource_load` function.

Returns *queryID* identifies the specific query request type. Valid request types are as follows:

Value	Description
RSC_QRY_SIZE	“What is the size of the resource as it appears in memory?” <i>reply</i> must point to an unsigned long.
RSC_QRY_FILESIZE	“What is the size of the resource as it appears in its parent file?” <i>reply</i> must point to an unsigned long.
RSC_QRY_TYPE	“What is the resourceclass of the resource?” <i>reply</i> must point to an unsigned long.
RSC_QRY_ID	“What is the resource ID of the resource?” <i>reply</i> must point to an unsigned long.
RSC_QRY_FILEHANDLE	“What is the handle of the resource’s parent file?” <i>reply</i> must point to a variable of type <code>RscFileHandle</code> .
RSC_QRY_ALIAS	“What is the alias name of the resource?” <i>reply</i> must point to a character pointer. Do not modify the alias using this pointer. Instead, use <code>mdlResource_changeAlias</code> .
RSC_QRY_FILENAME	“What is the file name of the resource’s parent file?” <i>reply</i> must point to a character pointer. Do not modify the file name using this pointer.
RSC_QRY_PLATFORM	“What is the format (platform ID) of the resource’s parent file?” <i>reply</i> must point to an integer.

Returns The mdlResource_query function returns SUCCESS if the query was successful. Otherwise, it returns one of the following errors defined in mdlerrs.h:

Value	Description
MDLERR_RSCADDRINVALID	<i>resource</i> is an invalid resource pointer.
MDLERR_RSCQRYIDINVALID	<i>queryID</i> is invalid.

See Also mdlResource_queryClass, mdlResource_queryFileHandle, mdlResource_queryFile.

mdlResource_queryClass, mdlResource_queryClassByAlias

```
#include <mdl.h>
#include <rscdefs.h>
#include <mdlerrs.h>

int mdlResource_queryClass
(
    void                *reply,          /* <= Where to store the reply */
    RscFileHandle       rfHandle,        /* => Resource File Handle */
    unsigned long        resourceclass,   /* => resourceclass identifier */
    int                 queryID,          /* => Query ID */
    void                *optarg          /* => Optional argument */
);

int mdlResource_queryClassByAlias
(
    void                *reply,          /* <= Where to store the reply */
    RscFileHandle       rfHandle,        /* => Resource File Handle */
    unsigned long        resourceclass,   /* => resourceclass identifier */
    int                 queryID,          /* => Query ID */
    void                *optarg,         /* => Optional argument */
    char                *alias           /* => Qualify query with this alias */
);
```

Description The mdlResource_queryClass function queries the resource manager for information regarding a *resourceclass*. The mdlResource_queryClassByAlias function has the same purpose except it allows the caller to limit the scope of the query to only reflect resources with a specific alias name, and it only honors the RSC_QRY_COUNT and RSC_QRY_ID query types.

reply points to the location where the resource manager must place its answer to the request. The type of data that it points to depends on the type of the query.

rfHandle is the resource file handle of the file containing the *resourceclass* being queried. *rfHandle* is obtained from the mdlResource_openFile function. If *rfHandle* equals zero, all resource files currently opened by the application are checked.

resourceclass identifies the type of resource being queried.

queryID identifies the specific query request type. Valid request types are as follows:

Value	Description
RSC_QRY_COUNT	Store the number of resources of this class in the integer pointed to by <i>reply</i> .
RSC_QRY_ID	Store the <i>resourceID</i> of the <i>n</i> th resource of the <i>resourceclass</i> (“n” is provided by * <i>optarg</i>) in the unsigned long variable pointed to by <i>reply</i> . For example, if <i>optarg</i> equals 5, the application is asking to receive the <i>resourceID</i> of the class’s fifth resource. <i>optarg</i> is zero-based.
RSC_QRY_ID_IN_USE	Set <i>reply</i> to TRUE if the resource ID pointed to by <i>optarg</i> is currently in use in the <i>resourceclass</i> . Otherwise set <i>reply</i> to FALSE.
RSC_QRY_ID_HIGHEST	Store the highest in-use <i>resourceID</i> of the <i>resourceclass</i> in the unsigned long variable pointed to by <i>reply</i> .
RSC_QRY_ID_LOWEST	Store the lowest in-use <i>resourceID</i> of the <i>resourceclass</i> in the unsigned long variable pointed to by <i>reply</i> .
RSC_QRY_ALIAS	Same as RSC_QRY_ID, but the <i>aliasname</i> is returned instead of the <i>resourceID</i> .
RSC_QRY_AVAIL_ID	Store the next unused <i>resourceID</i> of the <i>resourceclass</i> (greater than or equal to zero) in the unsigned long variable pointed to by <i>reply</i> .
RSC_QRY_AVAIL_ID_ABOVE	Store the next unused <i>resourceID</i> of the <i>resourceclass</i> , greater than * <i>optarg</i> , in the unsigned long variable pointed to by <i>reply</i> .
RSC_QRY_AVAIL_ID_BELOW	Store the next unused <i>resourceID</i> of the <i>resourceclass</i> , less than * <i>optarg</i> , in the unsigned long variable pointed to by <i>reply</i> .
RSC_QRY_FILEHANDLE	After locating the <i>n</i> th resource of the <i>resourceclass</i> (where “n” is a zero-based index pointed to by <i>optarg</i>), store the RscFileHandle of the file where the resource was found at the location pointed to by <i>reply</i> .
RSC_QRY_FILENAME	After locating the <i>n</i> th resource of the <i>resourceclass</i> (where “n” is a zero-based index pointed to by <i>optarg</i>), store the ASCII filename of the file where the resource was found in the string pointed to by <i>reply</i> .

optarg points to an optional argument whose value depends on the specific *queryID* as described above.

Returns The mdlResource_queryClass function returns SUCCESS if the query was successful. Otherwise, it returns one of the following errors defined in mdlerrs.h:

Value	Description
MDLERR_RSCERROR	An undefined error was encountered.
MDLERR_RSCQRYIDINVALID	<i>queryID</i> is invalid.

See Also mdlResource_query, mdlResource_queryFileHandle, mdlResource_queryFile.

mdlResource_queryFile

```
#include <mdl.h>
#include <rsdefs.h>
#include <mdlerrs.h>

int mdlResource_queryFile
(
    void    *reply,          /* <= Where to store the query reply */
    char    *fileName,       /* => Name of file to be queried */
    int     queryID          /* => Query ID */
);
```

Description The mdlResource_queryFile function queries the resource manager for information regarding a resource file. The resource manager will call mdlResource_openFile in order to gain access to the information. Afterwards, the file will be closed.



If the file to be queried is already open, use the mdlResource_queryFileHandle function to avoid the overhead incurred by mdlResource_queryFile.

reply points to the location where the resource manager must place its answer to the request. The type of data that it points to depends on the type of the query.

fileName points to a NULL-terminated ASCII filename.

queryID identifies the specific query request type. Valid request types are as follows:

Value	Description
RSC_QRY_NUMTYPES	Store the number of <i>resourceclasses</i> in the file in the unsigned long variable pointed to by <i>reply</i> .
RSC_QRY_CLASS	Store a list of the <i>resourceclasses</i> in the file in <i>reply</i> . <i>reply</i> points to an array of unsigned long values allocated by the caller. The caller can determine the number of elements to malloc in the array by calling this function with <i>queryID</i> equal to RSC_QRY_NUMTYPES.
RSC_QRY_VERSION	Store the version of the resource file in <i>reply</i> .
RSC_QRY_IDENT	Copy the <i>ident</i> string of the resource file to the character array pointed to by <i>reply</i> .

Returns mdlResource_queryFile returns SUCCESS if the query was successful. Otherwise, it returns one of the following errors defined in mdlerrs.h:

Value	Description
MDLERR_RSCINSFMEM	Memory is insufficient to read the file's header.
MDLERR_RSCFILEERROR	The resource file cannot be read.
MDLERR_RSCQRYIDINVALID	<i>queryID</i> is invalid.

See Also mdlResource_queryFileHandle, mdlResource_query, mdlResource_queryClass.

mdlResource_queryFileHandle

```
#include <mdl.h>
#include <rsdefs.h>
#include <mdlerrs.h>

int mdlResource_queryFileHandle
(
    void            *reply,           /* <= Where to store the query reply */
    int             replySize,        /* <= Size of reply area */
    RscFileHandle   rfHandle,         /* => Resource File Handle */
    int             queryID,          /* => Query ID */
    void            *optarg,          /* => Optional argument */
);
```

Description The mdlResource_queryFileHandle function queries the resource manager for information regarding an open resource file (or files).

reply points to the location where the resource manager must place its answer to the request. The type of data that it points to depends on the type of the query.

replySize indicates the size of the area pointed to by *reply* (in bytes).

rfHandle is a handle to the open resource file to be queried. *rfHandle* is ignored if *queryID* equals RSC_QRY_COUNT or RSC_QRY_FILEHANDLE.

queryID identifies the specific query request type. Valid request types are as follows:

Value	Description
RSC_QRY_NUMTYPES	Store the number of <i>resourceclasses</i> in the file in the unsigned long variable pointed to by <i>reply</i> .
RSC_QRY_CLASS	Store a list of the <i>resourceclasses</i> in the file in <i>reply</i> . <i>reply</i> points to an array of unsigned long values allocated by the caller. The caller can determine the number of elements to malloc in the array by calling this function with <i>queryID</i> equal to RSC_QRY_NUMTYPES.
RSC_QRY_VERSION	Store the version of the resource file in <i>reply</i> .
RSC_QRY_IDENT	Copy the <i>ident</i> string of the resource file to the character array pointed to by <i>reply</i> .
RSC_QRY_FILENAME	Store the open file's name in the character array pointed to by <i>reply</i> for a maximum of <i>replySize</i> bytes.
RSC_QRY_FILEATTRS	Store the file attribute mask for the file (as it was specified to the mdlResource_openFile function) in the 32-bit storage location pointed to by <i>reply</i> .
RSC_QRY_PLATFORM	Store the platformID of the open file in the unsigned long variable pointed to by <i>reply</i> .
RSC_QRY_OWNER	Store the MDL descriptor pointer of the task that originally opened the resource file in the 32-bit storage location pointed to by <i>reply</i> .

Value	Description
RSC_QRY_COUNT	Return the number of resource file handles owned by an application in the integer pointed to by <i>reply</i> . The number returned will also include files inherited from MicroStation. The target application's MDL descriptor is provided in <i>optArg</i> . If <i>optArg</i> is <code>NULL</code> , the number of handles owned by MicroStation is returned.
RSC_QRY_FILEHANDLE	Store all the resource file handles owned by an application in the array of resource file handles pointed to by <i>reply</i> . The number of handles returned will be dictated by the <i>replySize</i> argument. <i>replySize</i> should equal (number of handles * <code>sizeof(RscFileHandle)</code>). The target application's MDL descriptor is provided in <i>optArg</i> . If <i>optArg</i> is <code>NULL</code> , the handles owned by MicroStation are returned. To get the number of handles owned by the target task, call this function with a <i>queryID</i> of <code>RSC_QRY_COUNT</code> .

optArg is only used when *queryID* equals `RSC_QRY_COUNT` or `RSC_QRY_FILEHANDLE`. In these cases, *optArg* is the MDL descriptor of the task whose open files are being queried. Use the functions `mdlSystem_findMdlDesc` or `mdlSystem_getCurrMdlDesc` to obtain the MDL descriptor for a given task.

Returns `mdlResource_queryFileHandle` returns `SUCCESS` if the query was successful. Otherwise, it returns one of the following errors defined in `mdlerrs.h`:

Value	Description
<code>MDLERR_RSCINSFMEM</code>	Memory is insufficient to read the file's header.
<code>MDLERR_RSCFILEERROR</code>	The resource file cannot be read.
<code>MDLERR_RSCQRYIDINVALID</code>	<i>queryID</i> is invalid.

See Also `mdlResource_queryFile`, `mdlResource_query`, `mdlResource_queryClass`, `mdlSystem_findMdlDesc`, `mdlSystem_getCurrMdlDesc`.

mdlResource_getApplLoadFile

```
#include <mdl.h>
#include <mdlerrs.h>

char *mdlResource_getApplLoadFile
(
    void      *mdlDesc      /* => Task MDL descriptor */
);
```

Description mdlResource_getAppLoadFile is used to get the name of the resource file from which a given application was loaded. The application is identified by its MDL descriptor.

mdlDesc is an MDL descriptor for a task obtained from mdlSystem_findMdlDesc or mdlSystem_getCurrMdlDesc.

Returns mdlResource_getAppLoadFile returns a pointer to the file name if the MDL descriptor was valid. Otherwise, it returns NULL.

See Also mdlSystem_findMdlDesc, mdlSystem_getCurrMdlDesc.

mdlResource_getRscIdByAlias

```
#include <mdl.h>
#include <rsdefs.h>
#include <mdlerrs.h>

int mdlResource_getRscIdByAlias
(
    unsigned long    *resourceID,    /* <= Where to store the rsc ID */
    RscFileHandle    rfHandle,       /* => Resource File handle */
    unsigned long    resourceclass,  /* => resourceclass identifier */
    char             *alias          /* => Alias name of resource */
);
```

Description The mdlResource_getRscIdByAlias function obtains a resource's identification number from its ASCII alias name. A resource's alias name can be a maximum of 16 characters and is assigned using the mdlResource_add, mdlResource_directAdd or mdlResource_changeAlias function.

rfHandle is the resource file handle of the file that will be checked for *resourceclass* and *alias*. If *rfHandle* equals zero, the resource manager will check all resource files currently opened by the application.

resourceclass identifies the resource class that will be searched to find *alias*.

alias points to the ASCII alias name associated with the resource whose identification number will be obtained. All resources of class *resourceclass* will be searched to find the matching alias. An alias name can be a maximum of 16 characters.

resourceID points to an unsigned long where the ID of the resource with the matching alias name will be stored.

Returns `mdlResource_getRscIdByAlias` returns `SUCCESS` if the function was successful. Otherwise, it returns one of the following errors defined in `mdlerrs.h`:

Value	Description
<code>MDLERR_RSCERROR</code>	<code>NULL</code> is not a valid <i>resourceID</i> pointer.
<code>MDLERR_RSCNOTFOUND</code>	<i>alias</i> could not be found in the resource file(s).

See Also `mdlResource_changeAlias`.

mdlResource_changeAlias

```
#include <mdl.h>
#include <rsdefs.h>
#include <mdlerrs.h>

int mdlResource_changeAlias
(
    RscFileHandle rfHandle,          /* => Resource File Handle */
    unsigned long resourceclass,     /* => resourceclass identifier */
    unsigned long resourceID,        /* => resource identifier */
    char          *alias             /* => New alias name to be assigned */
);
```

Description `mdlResource_changeAlias` lets an application assign a new alias name to a resource.

rfHandle is the resource file handle of the file containing the resource whose alias name will be changed. If *rfHandle* equals zero, the resource manager will check all resource files currently opened by the application.

resourceclass identifies the class of the resource whose alias name will be changed.

resourceID identifies the resource whose alias name will be changed.

alias points to the new ASCII alias name to be assigned to the resource.

Returns The `mdlResource_changeAlias` function returns `SUCCESS` if the query was successful. Otherwise, it returns one of the following errors defined in `mdlerrs.h`:

Value	Description
<code>MDLERR_RSCWRITEVIOLATION</code>	The resource file was opened as <code>RSC_READONLY</code> .
<code>MDLERR_RSCNOTFOUND</code>	<i>resourceID</i> could not be found in the file(s).

Parse Functions

Parse functions control how MicroStation translates key-ins into commands. When a key-in is entered, MicroStation parses the key-in and creates a `CMDNUM` queue element. The `CMDNUM` queue element starts a command. The command can be an MDL command, MicroCSL command or a MicroStation command.

An MDL application can affect MicroStation's parsing by providing an application parse table, MDL parse user function, or both.

A key-in starts an MDL command if one of the following conditions is true:

- The key-in is an MDL `COMMAND <command name>` command.
- MicroStation parses the key-in using the application's command tables.
- The MDL parse user function creates a `CMDNUM` queue element that starts a command.

When the `CMDNUM` queue element is created from an MDL parse user function, the application's user functions are called at the beginning of the parsing process. If a user function creates a queue element, MicroStation queues it. If the queue element specifies an MDL application and function, MDL receives the queue element and calls the appropriate function. See "Input Handling Functions" for more information.

If you need to replace a MicroStation command, use an input command filter (`userInput_commandFilter`). Since commands are often queued by command number, you cannot completely replace a command by replacing it at the parsing stage.

The following table lists parse built-in functions. Most MDL applications use only `mdlParse_loadCommandTable`. You can disregard the other functions in this section while you are learning MDL.

The following table lists parse functions. Most MDL applications use only `mdlParse_loadCommandTable`. You can disregard the other functions in this section while you are learning MDL.

Function	Used to
<code>mdlParse_loadCommandTable</code>	load an application parse table tree and include it in the list of standard parse tables.
<code>mdlParse_unloadTable</code>	unload an application parse table.
<code>mdlParse_loadKeywordTable</code>	load a parse table tree but do not include it in the list of parse tables MicroStation uses.
<code>mdlParse_keyWord</code>	parse a string using a specified parse table.
<code>mdlParse_loadCommandTableByNumber</code>	load a parse table tree starting with a resource ID other than 1.

Function	Used to
<code>mdlParse_changeTargetTask</code>	modify the target task ID associated with a command table.
<code>mdlParse_getTableName</code>	return a pointer to the task ID associated with a specified table.
<code>mdlParse_findTableByName</code>	return a pointer to the table descriptor from a pointer to a table name.
<code>mdlParse_reconstructFullKeyin</code>	construct a command corresponding to a command number and task ID.
<code>mdlParse_setFunction</code>	specify a parse user function.

The following table lists the parse user function. This is a user-supplied function that MDL calls when a certain event occurs within MicroStation. The application programmer defines the user function name. (The name below is for illustration only). This function is designated to MDL through function pointer arguments to `mdlParse_setFunction`.

Function	MicroStation calls when
<code>userParse_handleString</code>	MicroStation parses a command key-in.

Example

See `parse.mc` and `chngtxt.mc` examples.

`mdlParse_loadCommandTable`, `mdlParse_unloadTable`

```
void *mdlParse_loadCommandTable
(
char    *pTableName          /* => resource filename or NULL */
);

void mdlParse_unloadTable
(
char    *pTableDescriptor    /* => pointer to table descriptor */
);
```

Description The `mdlParse_loadCommandTable` function loads an application command table from the resource file specified by *pTableName*.

The directories specified by the `MS_MDL` MicroStation environment variable and `MS_EXE` are searched for the filename. If *pTableName* is `NULL`, MicroStation loads the command table from the same resource file that the application was loaded from. MicroStation starts loading the application parse table and continues loading all subtables needed to parse any

command. It starts from the main table, using resource ID 1. Ultimately, mdlParse_loadCommandTable loads the entire tree in memory.

The command table is unloaded automatically when the MDL application is unloaded. The mdlParse_unloadTable function unloads a parse table. This function is not needed unless the application needs to disable the commands or load another table. The input to mdlParse_unloadTable is a table descriptor returned by mdlParse_loadCommandTable.

Returns The mdlParse_loadCommandTable function returns NULL and stores an error code in mdlErrno if the table is not loaded successfully. The values for the errors are defined in mdlerrs.h. Otherwise, it returns a pointer to a parse table descriptor. The pointer can be used as a parameter to other parse functions.

The mdlParse_unloadTable function is of type void. It returns no value.

See Also mdlParse_loadCommandTableByNumber, userParse_handleString, the “Compiling an Application Command Table” section in the “Developing Applications” chapter of the MDL Programmer’s Guide.

mdlParse_loadKeywordTable

```
int mdlParse_loadKeywordTable
(
    char    *pTableName,    /* => resource filename or NULL */
    int     tableNumber     /* => resource ID of start of parse tree */
);
```

Description The mdlParse_loadKeywordTable function loads an entire parse tree from the resource file specified by *pTableName*. If *pTableName* is NULL, it loads the table from the same resource file that the application was loaded from.

tableNumber is the resource ID of the parse tree root.

Tables loaded with mdlParse_loadKeywordTable are not put in the list of tables MicroStation uses when parsing commands. MicroStation uses these tables when parsing a keyword with mdlParse_keyWord.

Returns mdlParse_loadKeywordTable returns NULL and stores an error code in mdlErrno if the table is not loaded successfully. Otherwise, it returns a pointer to a parse table descriptor. The pointer can be used as a parameter to other parse functions.

mdlParse_keyWord

```
int mdlParse_keyWord
(
    char    *tableDescP,    /* => pointer to table descriptor */
    int     tableNumber,    /* => resource ID of start of parse tree */
    char    *keywordP      /* => keyword to be parsed */
);
```

Description The `mdlParse_keyWord` function parses the string that *keywordP* points to using the parse table specified by *tableDescP* and *tableNumber*.

tableDescP is a table descriptor obtained from `mdlParse_loadCommandTable`, `mdlParse_loadKeywordTable` or `mdlParse_loadCommandTableByNumber`.

tableNumber is the resource ID of the table to be used for parsing.

keywordP points to the word to be parsed.

`mdlParse_keyWord` converts the string to upper case and then compares it to all keywords specified in the table.

Returns If the keyword is a substring of exactly one of the keywords, the `mdlParse_keyWord` function returns the command number of the entry it matches. If the keyword is not a keyword substring, `mdlParse_keyWord` returns `MDLERR_NOMATCH`. If the keyword is a substring of more than one keyword, `mdlParse_keyWord` returns `MDLERR_AMBIGUOUS`. Since `MDLERR_NOMATCH` and `MDLERR_AMBIGUOUS` are negative, they cannot be confused with valid return values.

mdlParse_loadCommandTableByNumber

```
void *mdlParse_loadCommandTableByNumber
(
    char    *pTableName,          /* => resource filename or NULL */
    int     tableNumber           /* => resource ID */
);
```

Description The `mdlParse_loadCommandTableByNumber` function loads an application command table from the resource file specified by *pTableName*. Unlike `mdlParse_loadCommandTable`, this function does not assume that the resource ID of the parse table root is 1. The resource ID of the parse table root is specified by *tableNumber*.

Returns The `mdlParse_loadCommandTableByNumber` function returns `NULL` and stores an error code in `mdlErrno` if the table is not loaded successfully. The error codes are defined in `mdlerrs.h`. Otherwise, it returns a pointer to a parse table descriptor. The pointer can be used as a parameter to other parse functions.

See Also `mdlParse_loadCommandTable`.

mdlParse_changeTargetTask, mdlParse_getTableName, mdlParse_findTableByName

```
int mdlParse_changeTargetTask
(
    char    *pTableDescriptor,    /* => pointer table descriptor */
    char    *pTableName           /* => pointer to application name */
);

char *mdlParse_getTableName
(
```

```
char    *pTableDescriptor    /* => pointer to table descriptor */
);

char *mdlParse_findTableByName
(
char    *pTableName          /* => pointer to application name */
);
```

Description A target task ID is associated with every command table. When a task loads a command table, MDL sets the table's target task ID to the MDL task's task ID.

The `mdlParse_changeTargetTask` function modifies the target task ID associated with a command table. This function is useful if one task needs to load a table for another task.

The `mdlParse_getTableName` function returns a pointer to the task ID associated with a specified table.

The `mdlParse_findTableByName` function returns a pointer to the table descriptor from a pointer to a task ID. The table descriptor can be used as an argument to `mdlParse_changeTargetTask` or `mdlParse_unloadTable`. It has the same value as that returned by `mdlParse_loadCommandTable`.

Returns The `mdlParse_findTableByName` function returns a pointer to a table descriptor if the table is found. Otherwise, it returns `NULL`.

The `mdlParse_changeTargetTask` function returns zero if it is successful. It returns a non-zero value if the target task ID is already assigned to a command table.

The `mdlParse_getTableName` function returns a pointer to the table's target task ID.

mdlParse_reconstructFullKeyin

```
void mdlParse_reconstructFullKeyin
(
char    *commandStringP,      /* <= Reconstructed keyin */
char    *taskIdP,             /* => owning application */
ULong   commandNumber,        /* => number of command */
char    *unparsedP            /* => unparsed portion of command */
);
```

Description The `mdlParse_reconstructFullKeyin` function constructs a keyin that would invoke the specified command.

commandStringP points to the buffer to receive the generated command keyin. The buffer should be at least 256 characters long.

taskIdP specifies the application associated with the command. It specifies the application that loads the command table to be used to parse the command.

unparsedP points to the unparsed portion of the command. It points to the portion that is passed directly to the function that processes the command.

The following code fragment illustrates how the parameters correspond to the field in the `Inputq_command` structure. In the function call, assume *iqeP* points to an `inputq` element and *iqeP->hdr.u.cmdtype* is `CMDNUM`.

```
mdlParse_reconstructFullKeyin(resultBuffer, iqeP->u.cmd.taskId,
                             iqeP->u.cmd.command, iqeP->u.cmd.unparsedP);
```

Returns `mdlParse_reconstructFullKeyin` is of type `void`. It does not return anything.

mdlParse_setFunction

```
MdlFunctionP mdlParse_setFunction
(
    int          type,          /* => PARSE_HANDLE_STRING */
    MdlFunctionP function      /* => addr of function in MDL program */
);
```

Description The `mdlParse_setFunction` function specifies a parse user function.

Currently, `PARSE_HANDLE_STRING` is the only value supported for *type*.

function must be a valid MDL function pointer or `NULL`.

Returns `mdlParse_setFunction` returns a pointer to the user function (of the same type) that was previously set using `mdlParse_setFunction`. If *type* is invalid, `mdlParse_setFunction` returns `-1`.

See Also `userParse_handleString`.

userParse_handleString

```
#include <msinputq.h>
#include <userfnc.h>

int userParse_handleString
(
    Inputq_element *queueElementP, /* <=> receives new queue element */
    char          *stringP        /* => string to be parsed */
);
```

Description The `userParse_handleString` function parses input strings and creates MicroStation queue elements.

This function is designated to MicroStation when the user function name is specified in a call to `mdlParse_setFunction`. The `userParse_handleString` function name is arbitrary and is insignificant to MicroStation. You can change the name.

stringP points to the string being parsed.

If the `userParse_handleString` function successfully parses the string, it must create a `CMDNUM` queue element in the buffer pointed to by `queueElementP`.

Returns The `userParse_handleString` function returns -1 if it cannot parse the string, -2 if the command is ambiguous, and 0 if the parse is successful.

See Also `mdlParse_setFunction`, `mdlParse_loadCommandTable`.

Input Handling Functions

This section describes functions used to send and receive queue elements. Although an application can receive user input by receiving queue elements, applications should use the state functions and dialog boxes to gather user input.

A frequent parameter to these functions is a pointer to a task ID. If the queue element is intended for MicroStation, specify the built-in variable `ustnTaskId` as the task ID. (`ustnTaskId` points to MicroStation's task ID.) If the queue element is intended for a specific task, specify that task's task ID. If the queue element will be processed like user input, specify `NULL` as the task ID. Typically, `NULL` is the recommended value.

Functions that queue elements accept a position parameter. Generally, elements should be queued at the head of the queue. If the queue contains elements when the `mdlInput_send...` function is called and the task has no knowledge of the queue elements, the new element should be processed before the previously queued elements.

When an MDL program queues an element, the element is not processed until the MDL program returns control to MicroStation.

The following table lists input functions:

Function	Used to
<code>mdlInput_requeueLastInput</code>	send the last queue element received by the sending task for MicroStation to process next.
<code>mdlInput_sendCommand</code>	create a <code>CMDNUM</code> queue element and insert it in the MicroStation command queue.
<code>mdlInput_sendKeyin</code>	create a <code>KEYIN</code> queue element and insert it in the MicroStation command queue.
<code>mdlInput_sendKeyinFrom</code>	specifies the source from which to create a <code>KEYIN</code> queue element.
<code>mdlInput_sendKeyinExtended</code>	create a <code>KEYIN</code> queue element, but allows the user to decide whether it is retained in history.

Function	Used to
<code>mdlInput_sendKeystroke</code>	create a <code>RAWKEYSTROKE</code> queue element and insert it in the MicroStation command queue.
<code>mdlInput_sendReset</code>	create a <code>RESET</code> queue element and insert it in the MicroStation command queue.
<code>mdlInput_sendMessage</code>	insert the specified element in the queue.
<code>mdlInput_sendUORPoint</code>	create and queue a queue element for a precision data point.
<code>mdlInput_getMessage</code>	cause MDL to copy the most recently received queue element into a specified area.
<code>mdlInput_waitForMessage</code>	suspend the MDL task until another message is received.
<code>mdlInput_sendResume</code>	create a <code>NULL</code> element for the calling task in the queue.
<code>mdlInput_startCommand</code>	request an active command status for the current task.
<code>mdlInput_endCommand</code>	relinquish active command status.
<code>mdlInput_pause</code>	stop MicroStation completely until a keystroke or button is entered.
<code>mdlInput_commandState</code>	determine if the currently executing command is a view or primitive command.
<code>mdlInput_getTabletType</code>	determine whether a mouse or tablet is being used.
<code>mdlInput_disableCommandClass</code>	disable command classes.
<code>mdlInput_enableCommandClass</code>	enable command classes.
<code>mdlInput_longjmp</code>	do a long jump in an MDL application that uses <code>mdlInput_waitForMessage</code> .
<code>mdlInput_setjmp</code>	uses <code>mdlInput_waitForMessage</code> .
<code>mdlInput_setFunction</code>	designate user functions to be associated with different aspects of MicroStation's queue element processing.
<code>mdlInput_setMonitorFunction</code>	designate a user function to inspect queue elements before they are sent to the input processor.

The following table lists input user functions. These are user-supplied functions that MDL will call when certain events occur in MicroStation. The programmer determines

the user function names. (The names in the table are merely illustrations). Function pointer arguments to MDL routines designate the functions to MDL.

Function	MicroStation calls when
userInput_preprocessKeyin	MicroStation queues a key-in queue element.
userInput_monitor	MicroStation processes a new queue element.
userInput_receive	MDL receives a queue element for the task.
userInput_commandFilter	MicroStation processes a new command queue element.

Example

See input.mc.

Input processors and sequencing

The main input loop for MicroStation collects input, places it in a queue, and then calls input processors to process queue elements. MicroStation reads elements from the queue and dispatches input processors until the queue is empty or a preprocessor instructs MicroStation to gather more input. Then MicroStation resumes gathering input.

Examples of input processors are user commands, MicroCSL applications, and MicroStation's main input processor. MDL applications can be organized as input processors or as extensions of MicroStation's main input processor. Applications should be organized as extensions of MicroStation's main input processor. See "State Control Functions" for information on organizing applications.

In most cases, applications should interact with MicroStation by calling MicroStation built-in functions. For example, to create an element, an application calls an element creation built-in function rather than sending the appropriate PLACE key-in followed by data points to MicroStation.

Interacting with MicroStation by sending queue elements is called **sequencing** MicroStation. Sequencing MicroStation does not work well for performing complicated tasks. Applications that use the state control functions to collect input cannot sequence MicroStation commands. Sequencing MicroStation causes MicroStation to use the state control functions, destroying the command state set up by the MDL application. Applications that use the state control functions to collect input must interact with MicroStation by calling built-in functions.

An MDL application should not be organized as an input processor if one of the following conditions exists:

- The application's commands appear to be MicroStation commands.

- The application co-exists with other MDL applications.

Applications organized as input processors cannot work with selection sets. Also, applications organized as input processors are likely to be affected by changes to MicroStation's functionality. Only simple macros should be organized as input processors.

Although the functions described in this section should not be used in an application's main logic, these functions are still valuable in other instances. For example, dialog manager user hooks use input functions to queue commands.

MicroStation's main dispatching loop

```
do {
    Gather input;

    Create queue elements; This process includes expanding menus
    and parsing key-ins. At this point, user functions
    for parsing (userParse_handleString) and preprocessing
    key-ins (userInput_preprocessKeyin) are called.

    while (there are elements in the queue)
    {
        Call the user functions (userInput_monitor) to
        monitor queue elements;

        if (queue element is for a command)
            call user functions (userInput_commandFilter) to
            validate commands. These can reject or modify
            the commands.

        Process common queue elements; A number of queue elements
        such as tentative points are not useful to the command
        processors. These elements are processed here.

        while (queue element is not completely processed)
        {
            Determine the intended processor; the algorithm
            for determining the intended processor is
            described below;
            Give the queue element to the intended processor;
        }
    }
} while (no exit command);
```

The intended processor is determined as follows. The criteria are listed in the order in which it is applied.

- If the queue element was not queued by a user command and a user command has an outstanding get, the intended processor is the user command interpreter.

- If the queue element header *taskId* field is for a specific task, that task is the intended processor.
- A task that has active command status is the intended processor, (see `mdlInput_startCommand`).
- If the queue element is a `CMDNUM` element and the *cmd.taskId* field specifies a task, the task it specifies is the intended processor.
- Otherwise, MicroStation's main input processor is the intended processor.

MicroStation queue elements

```
typedef struct
{
    short cmdtype;      /* type of input following */
    short bytes;        /* bytes in queue element */
    short source;       /* source of input */
    short uc_fno_value; /* value to put in tcb->uc_fno */
    int sourcepid;      /* source pid for queue element */
    char taskId[16];    /* destination task */
} Inputq_header;
```

The type of input is contained in *iqel->hdr.cmdtype*. Possible values are defined in `msinputq.h`. Only the following types are used by a task program:

```
#define DATAPNT    240      /* user hit data point button */
#define RESET     243      /* user hit reset button */
#define KEYIN      501      /* normal key-in was entered */
#define UNASSIGNEDCB 502    /* cursor button is unassigned */
#define TUTKEYIN   506      /* user typed in tutorial field */
#define NULLCMDELEM 509     /* NULLCMD queue element */
#define CMDNUM     550      /* input was parsed to command */
#define APPLICATION_EVENT 990 /* reserved for tasks */
```

Other values are possible, but MicroStation uses them internally only.

The total number of bytes in the input queue element is contained in *iqel->hdr.bytes*. The input source (such as keyboard or menus) is identified in *iqel->hdr.source*. Possible values are defined in `msinputq.h`, but are unlikely to be useful to the programmer.

The process ID for the task that generated the input is contained in *iqel->hdr.sourcepid*. This ID is zero if MicroStation generated the request.

The string *iqel->hdr.taskId* is set to a zero-length string if the queue element is not addressed to a specific task. Otherwise, it contains the task ID of the task to get the queue element. The FNO value for user command GET statements is stored in *iqel->hdr.uc_fno_value*.

The other relevant structures composing the input queue element are defined as follows:

```
typedef struct
{
    short    len;           /* length of string */
    short    type;          /* 0 = key-in, 1 = command key-in */
    char     keyin[1];      /* actual string */
} Inputq_keyin;

typedef struct
{
    short    view;          /* view that data point is in */
    short    phys_screen;   /* physical screen point is in */
    Uspoint3d cmdpoint;     /* point in input device space */
    short    region;        /* tablet region */
    Spoint2d scrnpoint;     /* point in screen space */
    Point3d  rawUors;       /* point in design file coords */
    Point3d  uors;          /* point adjusted for locks */
    short    buttonTrans;   /* type of transition */
    short    qualifierMask; /* qualifying keys when hit */
} Inputq_pnt;

typedef struct
{
    Dpoint3d dpRawUors;     /* point in design file coords */
    Dpoint3d dpUors;        /* point adjusted for locks */
} Inputq_dpnt;

typedef struct
{
    Inputq_pnt pnt;         /* point structure */
    short    precision;     /* TRUE = precision input */
    short    viewflags[MAX_VIEWS]; /* views point is near */
    byte     dpData[sizeof(Inputq_dPnt)]; /* double precision data */
} Inputq_datapnt;

typedef struct
{
    int      class;         /* command class */
    int      immediate;     /* immediate mode */
    long     command;       /* unique command ID */
    char     taskId[16];    /* destination child task */
    char     unparsed[1];   /* unparsed portion (if any) */
} Inputq_command;

typedef struct
{
    short    reset;         /* place holder */
} Inputq_reset;
```

```
typedef struct
{
    Inputq_pnt pnt;           /* position data */
    short      cursbutn;      /* cursor button number */
} Inputq_unassignedcb;

typedef struct
{
    short      nullcmd;       /* place holder */
} Inputq_nullcmd;

typedef struct
{
    short      len;           /* length of string */
    char       keyin[1];      /* actual string */
} Inputq_tutkeyin;

typedef struct
{
    Inputq_header hdr; /* common information */
    union
    {
        Inputq_keyin keyin;           /* keyed in (not command) */
        Inputq_datapnt data;          /* data point */
        Inputq_command cmd;           /* parsed command */
        Inputq_reset reset;           /* reset */
        Inputq_unassignedcb cursbutn; /* unassigned button */
        Inputq_tutkeyin tutkeyin;     /* tutorial key-in */
        Inputq_nullcmd nullcmd;        /* NULL command element */
        char fill[480];                /* padding */
    } u;
} Inputq_element;
```

The data specific to each input type is contained in a union of structures. Since some structures have a variable length (which the last member declares as an array of dimension 1), a fill member of the union is as large as the largest possible variable-length union member.

Tasks should determine which member of the union *iqel->u* to examine based on the value of *iqel->hdr.cmdtype*. This value matches the value returned by the `mdlInput_waitForMessage` function.

When *iqel->hdr.cmdtype* is set to `DATAPNT`, *iqel->u.data.pnt.uors* contains the X, Y and possibly Z values for the data point's position in the design file. Similarly, *iqel->u.data.pnt.cmdpoint* and *iqel->u.data.pnt.scrnpnt* contain the X and Y values for the position in input device coordinates and screen coordinates. *iqel->u.data.pnt.view* is the view in which the point was entered (starting at 0 for an array index in C). *iqel->u.data.pnt.phys_screen* is set to 0 if the view was on the right screen and 1 if the view was on the left screen.

Iqel->u.data.pnt.region is 0 if the data point was entered from the digitizing tablet's screen tracking portion (or was entered with a mouse) and non-zero if it was entered outside this region (in the tablet's digitizing section). *Iqel->u.data.precision* is 0 for points entered from the tablet cursor and 1 for precision key-ins or data points caused by a user command PNT statement. This statement specifies the data point position.

Iqel->u.data.viewflags shows the view(s) that would be affected by a viewing command. For example, *Iqel->u.data.viewflags* [0] is non-zero if view 1 (*iqel->u.data.pnt.view* = 0) is affected.

When *iqel->hdr.cmdtype* is set to RESET, *iqel->u.reset* contains no useful information. It is declared only for consistency with other input types.

When *iqel->hdr.cmdtype* is set to KEYIN, *iqel->u.keyin.len* is set to the string length, which is contained in *iqel->u.keyin.keyin*. The string is always NULL-terminated, so the length is actually redundant information. If the key-in must be treated as a command, *iqel->u.keyin.type* is set to 1. Otherwise, *iqel->u.keyin.type* is set to 0, and the key-in can be treated as a command, query response, or input to the PLACE TEXT command.

When *iqel->hdr.cmdtype* is set to UNASSIGNEDCB, the relevant information is in *iqel->u.cursbutn*. Information about the position and view where the button was pressed is the same as information about data points. The button number is in *iqel->u.cursbutn.cursbutn*.

When *iqel->hdr.cmdtype* is set to TUTKEYIN, the information consists of the string length, NULL-terminated string (in *iqel->u.tutkeyin.len* and *iqel->u.tutkeyin.keyin*), and field number. The held number is stored in *iqel->hdr.uc_fno_value*.

When *iqel->hdr.cmdtype* is set to NULLCMDELEM, *iqel->u.nullcmd* contains no additional information. This structure is declared only for consistency.

When *iqel->hdr.cmdtype* is set to CMDNUM, the 32-bit command number is stored in *iqel->u.cmd.command*, and the command class is stored in *iqel->u.cmd.class*. The command class is one of 64 possible values. The first 48 classes are reserved for MicroStation, and applications can use the remainder. The currently defined command classes are defined in the cmdclass.h include file. *Iqel->u.cmd.immediate* is a Boolean number indicating whether the command has an immediate mode. This number has no current use.

The task that the command belongs to is stored in *iqel->u.cmd.taskId*. This task is set to *ustnTaskId* (an MDL built-in variable) or to a zero-length string for a MicroStation task. The final member, *iqel->u.cmd.unparsed*, is a variable-length, NULL-terminated string. This string contains the input portion following the successfully parsed command portion.

The task determines the action for input data. However, the task should honor the NULLCMDELEM request type by returning to a neutral state. If the task has active command status, it must use the mdlInput_endCommand function.

mdlInput_requeueLastInput

```
void mdlInput_requeueLastInput
(
    int    position      /* => queue position */
);
```

Description The mdlInput_requeueLastInput function sends the last queue element received by the sending task to MicroStation.

The element is queued at the position indicated by *position*. Generally, *position* 0 is used to place the element at the head of the queue. A value of INPUTQ_EOQ for *position* tells mdlInput_requeueLastInput to place the queue element at the end of the queue.

The mdlInput_requeueLastInput function sets the task ID in the queue element to the string contained in the built-in variable, *usrnTaskId*.

Only MicroStation's input processor can receive the queue element. However, the user functions to monitor the queue and validate commands receive the queued element before MicroStation's input processor does.

Returns mdlInput_requeueLastInput is of type void. It returns no value.

See Also userInput_monitor.

mdlInput_sendCommand

```
#include <cmdclass.h>
void mdlInput_sendCommand
(
    long    commandNumber, /* => command number */
    char    *cmdParamP,    /* => string passed to command */
    int     position,      /* => queue position */
    char    *taskIdP,      /* => task to receive message */
    int     class          /* => standard command classes */
);
```

Description The mdlInput_sendCommand function creates a CMDNUM queue element with the command number designated by *commandNumber*.

cmdParamP points to a string to be passed to the command. mdlInput_sendCommand copies the string into the queue element. The function that processes the command receives a pointer to the copy. The string that *cmdParamP* points to can be in a temporary location. If there is no string to pass to the command, *cmdParamP* can be NULL. It can also point to a zero-length string.

position specifies the queue position. Position 0 places the element at the head of the queue. A value of `INPUTQ_EOQ` for *position* tells MicroStation to place the queue element at the end of the queue.

taskIdP specifies the task to execute the command. The task ID is copied to the taskID field of the input queue element's *cmd* field. If a MicroStation command is being requested, specify the built-in variable, `ustnTaskId`, as the task ID. If the queue element is intended for a specific task, specify the task ID. If *taskIdP* is `NULL`, the command is interpreted as a MicroStation command.

class should be a standard command class from `cmdclass.h`. See "Compiling an Application Command Table" in the "Developing Applications" chapter for more information on command classes.

MicroStation command numbers are defined in the `cmdlist.h` header file.

Returns The `mdlInput_sendCommand` function is of type `void`. It returns no value.

See Also `mdlInput_sendKeyin`, `mdlInput_sendReset`, `mdlInput_sendMessage`, `mdlInput_sendUORPoint`.

mdlInput_sendKeyin

```
void mdlInput_sendKeyin
(
    char    *stringP,      /* => key-in string */
    int     literal,       /* => 1 means inhibit parsing */
    int     position,      /* => queue position */
    char    *taskIdP      /* => task to receive message */
);
```

Description The `mdlInput_sendKeyin` function creates a `KEYIN` queue element with the string designated by *stringP*.

If *literal* is non-zero, `mdlInput_sendKeyin` queues a `KEYIN` element without first trying to parse the string.

If *literal* is zero, `mdlInput_sendKeyin` tries to parse the string before queuing it. If the string can be parsed, `mdlInput_sendKeyin` places a `CMDNUM` queue element in the queue. Otherwise, it places a `KEYIN` queue element in the queue.

position specifies the queue position. Position 0 places the element at the head of the queue. A value of `INPUTQ_EOQ` for *position* tells MicroStation to place the queue element at the end of the queue.



position is ignored and the key-in is always placed at the beginning of the queue if *literal* is `FALSE` (parsing is enabled).

taskIdP specifies the task to receive the queue element. If the queue element is intended for MicroStation, specify the built-in variable,

ustnTaskId, as the task ID. If the queue element is intended for a specific task, specify the task ID. If the queue element will be processed like user input, pass NULL for *taskIdP*.

If MicroStation successfully parses the string and creates a command queue element, it ignores the *taskIdP* parameter.

Returns mdlInput_sendKeyin is of type void. It returns no value.

See Also mdlInput_sendKeystroke, mdlInput_sendMessage, mdlInput_sendCommand.

mdlInput_sendKeyinFrom

```
void mdlInput_sendKeyinFrom
(
    char    *stringP,        /* => key-in string */
    int     literal,         /* => 1 means inhibit parsing */
    int     position,        /* => queue position */
    char    *taskIdP,        /* => task to receive message */
    int     from             /* => keyin source (for queue element) */
);
```

Description mdlInput_sendKeyinFrom creates a KEYIN queue element with the string designated by *stringP*. It differs from mdlInput_sendKeyin in that a source of the key-in can be specified.

If *literal* is non-zero, mdlInput_sendKeyinFrom queues a KEYIN element without first trying to parse the string.

If *literal* is zero, mdlInput_sendKeyinFrom tries to parse the string before queuing it. If the string can be parsed, mdlInput_sendKeyinFrom places a CMDNUM queue element in the queue. Otherwise, it places a KEYIN queue element in the queue.

position specifies the queue position. Position 0 places the element at the head of the queue. A value of INPUTQ_EOQ for *position* tells MicroStation to place the queue element at the end of the queue.



position is ignored and the key-in is always placed at the beginning of the queue if *literal* is FALSE (parsing is enabled).

taskIdP specifies the task to receive the queue element. If the queue element is intended for MicroStation, specify the built-in variable, ustnTaskId, as the task ID. If the queue element is intended for a specific task, specify the task ID. If the queue element will be processed like user input, pass NULL for *taskIdP*.

If MicroStation successfully parses the string and creates a command queue element, it ignores the *taskIdP* parameter.

from can be any of the following found in `msinputq.h`: `FROM_KEYBOARD`, `FROM_CMDFILE`, `FROM_TUTORIAL`, `FROM_APP`, `FROM_UCM`, `FROM_STARTUP`, `FROM_DIALOG`, `FROM_PROCESS`, `FROM_MDL`, `FROM_PRDFPI`, `FROM_OPER_SYSTEM`.

Returns The `mdlInput_sendKeyinFrom` function is of type `void`. It returns no value.

See Also `mdlInput_sendKeystroke`, `mdlInput_sendMessage`, `mdlInput_sendCommand`.

mdlInput_sendKeyinExtended

```
void mdlInput_sendKeyinFrom
(
  char    *stringP,      /* => key-in string */
  int     literal,       /* => 1 means inhibit parsing */
  int     position,      /* => queue position */
  char    *taskIdP,      /* => task to receive message */
  int     from,          /* => keyin source (for queue element) */
  int     saveKeyin,     /* => whether to save keyin in history */
  int     journalKeyin   /* => whether to CAD input journal the keyin */
);
```

Description `mdlInput_sendKeyinFrom` creates a `KEYIN` queue element with the string designated by *stringP*. It differs from `mdlInput_sendKeyin` in that a source of the key-in can be specified.

If *literal* is non-zero, `mdlInput_sendKeyinFrom` queues a `KEYIN` element without first trying to parse the string.

If *literal* is zero, `mdlInput_sendKeyinFrom` tries to parse the string before queuing it. If the string can be parsed, `mdlInput_sendKeyinFrom` places a `CMDNUM` queue element in the queue. Otherwise, it places a `KEYIN` queue element in the queue.

position specifies the queue position. Position 0 places the element at the head of the queue. A value of `INPUTQ_EOQ` for *position* tells MicroStation to place the queue element at the end of the queue.



position is ignored and the key-in is always placed at the beginning of the queue if *literal* is `FALSE` (parsing is enabled).

taskIdP specifies the task to receive the queue element. If the queue element is intended for MicroStation, specify the built-in variable, `ustnTaskId`, as the task ID. If the queue element is intended for a specific task, specify the task ID. If the queue element will be processed like user input, pass `NULL` for *taskIdP*.

If MicroStation successfully parses the string and creates a command queue element, it ignores the *taskIdP* parameter.

from can be any of the following found in `msinputq.h`: `FROM_KEYBOARD`, `FROM_CMDFILE`, `FROM_TUTORIAL`, `FROM_APP`, `FROM_UCM`, `FROM_STARTUP`, `FROM_DIALOG`, `FROM_PROCESS`, `FROM_MDL`, `FROM_PRDFPI`, `FROM_OPER_SYSTEM`.

saveKeyin specifies whether the keyin is saved in history. *saveKeyin* equals 1, indicates that the keyin is saved. *saveKeyin* equals 0, indicates that the keyin is not saved.

journalKeyin specifies whether the keyin is CAD input journaled.

journalKeyin equals 1 indicates that the keyin is CAD input journaled.

journalKeyin equals 0 indicates that the keyin is not CAD input journaled.

Returns `mdlInput_sendKeyinExtended` is of type `void`. It returns no value.

See Also `mdlInput_sendKeystroke`, `mdlInput_sendMessage`, `mdlInput_sendCommand`.

mdlInput_sendKeystroke

```
#include <keys.h>
void mdlInput_sendKeystroke
(
    int      keystroke,          /* => keystroke to send */
    int      qualifierMask,      /* => modifier keys */
    int      position,           /* => location in input queue */
    char     *taskIdP            /* => task to receive keystroke */
);
```

Description `mdlInput_sendKeystroke` places a keystroke into MicroStation's input queue.

keystroke specifies a keystroke. It is acceptable to specify any codes that can be generated from the keyboard. Both printable characters and control characters such as the return key can be specified. See the include file `keys.h` for `#define` statements for the control characters.

qualifierMask is used to specify the modifier keys such as shift, control, or alt. An upper case character is normally generated just by specifying the upper case character for the *keystroke* parameter. It is not necessary to specify the shift *qualifierMask* to generate an upper case character. The standard qualifier masks `SHIFTKEY`, `CTRLKEY` and `ALTKEY` are defined in `keys.h`.

position specifies the location in the input queue for the keystroke queue element. Position 0 places the queue element at the head of the queue. A value of `INPUTQ_EOQ` for *position* tells MicroStationMicroStation to place the queue element at the end of the queue.

taskIdP specifies the task to receive the queue element. If the queue element is intended for MicroStation, specify the built-in variable `ustnTaskId` as the task ID. If the queue element is intended for a specific task, specify the task ID. If the queue element is to be processed like user input, pass `NULL` for the task ID.

Returns `mdlInput_sendKeystroke` is of type `void`. It returns no value.

See Also `mdlInput_sendKeyin`.

mdlInput_sendReset

```
void mdlInput_sendReset
(
    int      position,      /* => queue position */
    char     *taskIdP       /* => task to receive message */
);
```

Description `mdlInput_sendReset` creates a RESET queue element and inserts it in the queue. This function has the same effect as an operator entering a Reset.

position specifies the queue position. Position 0 places the element at the head of the queue. A value of `INPUTQ_EQQ` for *position* tells MicroStation to place the queue element at the end of the queue.

taskIdP specifies the task to receive the queue element. If the queue element is intended for MicroStation, specify the built-in variable, `ustnTaskId` as the task ID. If the queue element is intended for a specific task, specify the task ID. If the queue element will be processed like user input, pass `NULL` for *taskIdP*.

Returns The `mdlInput_sendReset` function is of type `void`. It returns no value.

See Also `mdlInput_sendMessage`.

mdlInput_sendMessage

```
#include <msinputq.h>

void mdlInput_sendMessage
(
    Inputq_element *queueElementP, /* => queue element */
    int            position         /* => queue position */
);
```

Description `mdlInput_sendMessage` queues the element specified by *queueElementP*. It is placed at the head of the queue. This is the most powerful function for queuing elements, since the task completely controls the queue element.

Before `mdlInput_sendMessage` is called, the following queue header fields must be filled in: *cmdtype*, *bytes*, *source* and *taskId*. See “MicroStation Queue Elements” at the beginning of this section for more information on these fields.

In many cases, you can obtain a queue element with `mdlInput_getMessage`, make minor modifications to the queue element, set the *hdr.taskId* field to the string specified by the built-in variable `ustnTaskId`, and then send the message with `mdlInput_sendMessage`.

position specifies the queue position. *Position 0* places the element at the head of the queue. A value of `INPUTQ_EOQ` for *position* tells MicroStation to place the queue element at the end of the queue.

Returns The `mdlInput_sendMessage` function is of type `void`. It returns no value.

mdlInput_sendUORPoint

```
void mdlInput_sendUORPoint
(
Dpoint3d    *pointP,        /* => data point to be placed */
int          viewNumber,    /* => view number */
int          position,      /* => location in input queue */
char         *taskIdP       /* => task to receive point */
);
```

Description The `mdlInput_sendUORPoint` function creates a data point queue element and inserts it at the head of MicroStation's input queue. It applies the current transformation to the point specified by *pointP*. It then converts this point to an integer point and stores it in the queue element.

viewNumber specifies the view number and is stored in the queue element. Since the value is zero-based, the view numbers are 0 through 7. Typically, a command uses either the data point or the view number. For example, the data point is important for an element creation command. The view number is important for an update command.

MicroStation commands require the *viewNumber* value or the value specified by *pointP* (but MicroStation does not require both values). When MicroStation needs a UOR point, it ignores the *viewNumber* value. When MicroStation needs a view specified, it ignores the *pointP* value.

position specifies the location in the input queue for the keystroke queue element. *Position 0* places the queue element at the head of the queue. A value of `INPUTQ_EOQ` for *position* tells MicroStation to place the queue element at the end of the queue.

taskIdP specifies the task to receive the queue element. If the queue element is intended for MicroStation, specify the built-in variable `ustnTaskId` as the task ID. If the queue element is intended for a specific task, specify the task ID. If the queue element is to be processed like user input, pass `NULL` for *taskIdP*.

Returns `mdlInput_sendUORPoint` is of type `void`. It returns no value.

See Also `mdlInput_sendMessage`.

mdlInput_getMessage

```
#include <msinputq.h>

void mdlInput_getMessage
```

```
(
Inputq_element    *queueElementP /* <=> buffer to receive message */
);
```

Description mdlInput_getMessage causes MDL to copy the most recently received queue element into the area specified by *queueElementP*.

Returns The mdlInput_getMessage function is of type void. It returns no value.

See Also mdlInput_waitForMessage, mdlInput_sendMessage.

mdlInput_waitForMessage

```
int mdlInput_waitForMessage(void);
```

Description mdlInput_waitForMessage suspends the MDL task until it receives another message. It cannot be called from a user function. For example, it cannot be called from a dialog box hook, a dialog item hook, a state function hook, a view hook or an update hook.

A function cannot call mdlInput_waitForMessage if the function was called as the result of a function pointer provided to MicroStation. If this rule is violated, MicroStation aborts the task.



An application that calls this function cannot use the common stack. Therefore, the `-csoff` option must be specified to `mlink` when linking the MDL application. If the application is not linked using this option and it calls mdlInput_waitForMessage, MicroStation will abort the application and will display the error message “Invalid use of suspend.”

Returns The mdlInput_waitForMessage function returns the type of message received. The most common values are DATAPNT, RESET, KEYIN, UNASSIGNEDCB, TUTKEYIN, NULLCMDELEM and CMDNUM.

See Also mdlInput_getMessage.

mdlInput_sendResume

```
void mdlInput_sendResume
(
int    position    /* => queue position */
);
```

Description mdlInput_sendResume creates a NULL queue element with the *taskId* field set to the task ID of the task that calls mdlInput_sendResume. The task that calls the mdlInput_sendResume function will receive the resulting NULL queue element

regardless of whether the task has active command status. This function along with `mdlInput_waitForMessage` can be used to pause and resume.

position specifies the queue position. *Position* 0 places the element at the head of the queue. A value of `INPUTQ_EOQ` for *position* tells MicroStation to place the queue element at the end of the queue.

Returns The `mdlInput_sendMessage` function is of type `void`. It returns no value.

See Also `mdlInput_waitForMessage`.

mdlInput_startCommand

```
int mdlInput_startCommand(void);
```

Description The `mdlInput_startCommand` function requests active command status for the current task. If the current task acquires active command status, this task receives most queue elements. See “MicroStation’s main dispatching loop.”

Returns `mdlInput_startCommand` returns `TRUE` if active command status is granted.

See Also `mdlInput_endCommand`.

mdlInput_endCommand

```
void mdlInput_endCommand(void);
```

Description The `mdlInput_endCommand` function relinquishes “active command” status.

Returns The `mdlInput_endCommand` function is of type `void`. It returns no value.

See Also `mdlInput_startCommand`.

mdlInput_pause

```
void mdlInput_pause(void);
```

Description The `mdlInput_pause` function stops MicroStation completely until a keystroke or button is entered.

Returns The `mdlInput_pause` function is of type `void`. It returns no value.

See Also `mdlDialog_openAlert`, `mdlSystem_getChar`.

mdlInput_commandState

```
#include <msinputq.h>
```

```
int mdlInput_commandState(void);
```

Description The `mdlInput_commandState` function determines whether the executing command is view or primitive.

Returns The `mdlInput_commandState` function returns `PRIMITIVE_COMMAND` or `VIEW_COMMAND`.

mdlInput_getTabletType

```
int mdlInput_getTabletType(void);
```

Description The `mdlInput_getTabletType` function determines whether a mouse or tablet is being used.

Returns The `mdlInput_getTabletType` function returns 0 if a tablet is being used. It returns 1 if a mouse is being used.

mdlInput_disableCommandClass

```
#include <cmdclass.h>

void mdlInput_disableCommandClass
(
    long    classMask1,    /* => bit mask */
    long    classMask2    /* => bit mask */
);
```

Description The `mdlInput_disableCommandClass` function disables command classes. The possible class numbers are 1 through 64. The low-order bit of *classMask1* corresponds to class 1. The high-order bit of *classMask1* corresponds to class 32. The low-order bit of *classMask2* corresponds to class 33. The high-order bit of *classMask2* corresponds to class 64.

The possible values for *classNumber* are given in `cmdclass.h`.

Returns The `mdlInput_disableCommandClass` function is of type `void`. It returns no value.

See Also `mdlInput_enableCommandClass`, `userInput_commandFilter`.

mdlInput_enableCommandClass

```
#include <cmdclass.h>

void mdlInput_enableCommandClass
(
    long    classMask1,    /* => bit mask */
    long    classMask2    /* => bit mask */
);
```

Description The `mdlInput_enableCommandClass` function enables command classes. The possible class numbers are 1 through 64. The low-order bit of *classMask1* corresponds to class 1. The high-order bit of *classMask1* corresponds to class 32. The low-order bit of *classMask2* corresponds to class 33. The high-order bit of *classMask2* corresponds to class 64.

The possible values for *classNumber* are given in `cmdclass.h`.

The command classes are all enabled by default, so the `mdlInput_enableCommandClass` function is not needed unless the `mdlInput_disableCommandClass` function is used.

Calling mdlInput_enableCommandClass does not affect command classes disabled by other tasks.

Returns The mdlInput_enableCommandClass function is of type void. It returns no value.

See Also mdlInput_disableCommandClass.

mdlInput_longjmp, mdlInput_setjmp

```
#include <setjmp.h>
void mdlInput_longjmp
(
    jmp_buf      jmpbuf,    /* => initialized by mdlInput_setjmp */
    int          arg        /* => passed to setjmp */
);

int mdlInput_setjmp
(
    jmp_buf      jmpbuf      /* <=> buffer to be initialized */
);
```

Description mdlInput_setjmp and mdlInput_longjmp must be used if the contents of the jump buffer must be saved across calls to the mdlInput_waitForMessage function. Most programs should not use the mdlInput_waitForMessage function, and very few of these should use the mdlInput_setjmp and the mdlInput_longjmp functions. Most programs that need long jump functionality should use the standard C functions setjmp and longjmp.

The parameters for mdlInput_setjmp are the same as for setjmp, and the parameters for mdlInput_longjmp are the same as for longjmp.

The mdlInput_setjmp function saves information in *jmpbuf* to allow mdlInput_longjmp to jump back to the return from mdlInput_setjmp.

The mdlInput_longjmp function causes MicroStation to use the information in *jmpbuf* to return to the caller of mdlInput_setjmp. The value passed in *jmpbuf* is used as the mdlInput_setjmp return value.

The mdlInput_setjmp function is typically used in an if statement as follows:

```
if (setjmp (jmpbuf) == 0)
    /* setjmp returned from call */
else
    /* setjmp returned as result of longjmp */
```

Returns The mdlInput_setjmp function returns 0 when it is returning from a call to mdlInput_setjmp. It returns the value provided by mdlInput_longjmp when returning from mdlInput_longjmp call.

The mdlInput_longjmp does not return. It causes a return from the mdlInput_setjmp call that caused the jump buffer to be initialized.

mdlInput_setFunction

```
#include <userfnc.h>
MdlFunctionP mdlInput_setFunction
(
    int          type,      /* => INPUT_KEYIN_PREPROCESS */
    MdlFunctionP function /* => addr. of function in MDL program */
);
```

Description mdlInput_setFunction designates user functions to be associated with different aspects of MicroStation's queue element processing.

type can be INPUT_COMMAND_FILTER, INPUT_MESSAGE_RECEIVED or INPUT_KEYIN_PREPROCESS.

functionP must be a valid MDL function pointer or NULL.

Returns mdlInput_setFunction returns a pointer to the user function (of the same type) that was previously set using mdlInput_setFunction. If *type* is invalid, mdlInput_setFunction returns -1.

See Also userInput_commandFilter, userInput_receive, userInput_preprocessKeyin, mdlInput_setMonitorFunction.

mdlInput_setMonitorFunction

```
#include <userfnc.h>
MdlFunctionP mdlInput_setMonitorFunction
(
    int          type,      /* => MONITOR_ALL, etc. */
    MdlFunctionP function /* => addr of function in MDL program */
);
```

Description mdlInput_setMonitorFunction designates user functions to inspect queue elements before they are sent to the input processor.

type can be MONITOR_NOT_FROM_APP, MONITOR_ALL or MONITOR_FROM_APP.

functionP must be a valid MDL function pointer or NULL.

Returns mdlInput_setMonitorFunction returns a pointer to the user function (of the same type) that was previously set using mdlInput_setMonitorFunction.

See Also userInput_monitor, mdlInput_setFunction.

userInput_preprocessKeyin

```
#include <userfnc.h>
int userInput_preprocessKeyin
(
    char    *stringP      /* <=> pointer to string */
);
```


Description A preprocess key-in user function is designated to MicroStation when the user function name is specified in a call to `mdlInput_setFunction` with `INPUT_KEYIN_PREPROCESS` for the *type* parameter. *userInput_preprocessKeyin* does not need to be the function name. The actual function name is insignificant to MicroStation.

MicroStation calls all *userInput_preprocessKeyin* functions for each key-in. This call is MicroStation's first step in processing the key-in before creating queue elements. The *userInput_preprocessKeyin* function can tell MicroStation to accept or discard the key-in. If it tells MicroStation to accept the key-in, it can modify the string. MicroStation then uses the modified string for the remainder of the processing. MicroStation skips the preprocessing step if the first character in the string is the backquote (grave accent mark) “`”, or if the MicroStation SET PARSEALL OFF command was executed.

Returns A *userInput_preprocessKeyin* function returns `INPUT_ACCEPT` to accept the key-in string and `INPUT_REJECT` to reject it.

Example See the example in the `mdl/examples/calculat` directory.

See Also *userParse_handleString*, `mdlInput_setFunction`.

userInput_monitor

```
#include <userfnc.h>
#include <msinputq.h>

int userInput_monitor
(
    Inputq_element    *queueElementP    /* <=> pointer to queue element */
);
```

Description An input monitor user function is designated to MicroStation when the user function name is specified in a call to the `mdlInput_setMonitorFunction` function. The actual function name is insignificant to MicroStation.

userInput_monitor views every queue element that the command processors will process. The *userInput_monitor* function can manipulate the queue element.

When the monitor function sees the queue element, the element is in a buffer large enough to hold any legal queue element. Therefore, the queue element can be transformed into any legal queue element.

userInput_monitor is called after MicroStation has completed its preprocessing. MicroStation's preprocessing consists of turning RESET commands into RESET queue elements and turning precision key-in commands into DATAPNT queue elements.

Returns A *userInput_monitor* function must return `INPUT_ACCEPT` to accept the queue element or `INPUT_REJECT` to reject it. If the function returns `INPUT_ACCEPT`, MicroStation continues processing the queue element.

See Also `mdlInput_setMonitorFunction`, `userInput_commandFilter`.

userInput_receive

```
#include <msinputq.h>

void userInput_receive
(
  Inputq_element    *queueElementP    /* => pointer to queue element */
);
```

Description A receive user function is designated to MicroStation when the user function name is specified in a call to `mdlInput_setFunction` and `INPUT_MESSAGE_RECEIVED` is specified as the *type* parameter. *userInput_receive* does not need to be the function name. The actual function name is insignificant to MicroStation.

MicroStation calls the receive user function when a queue element other than a command queue element is directed to the task.

Receive user functions are generally used when MDL tasks communicate with each other using `APPLICATION_EVENT` queue elements.

Returns MDL ignores the *userInput_receive* return value.

See Also `mdlInput_sendMessage`, `mdlInput_setFunction`.

userInput_commandFilter

```
#include <userfnc.h>
#include <msinputq.h>

int userInput_commandFilter
(
  Inputq_element    *queueElementP    /* <=> ptr to queue element */
);
```

Description A command filter user function is designated to MicroStation when the user function name is specified in a call to `mdlInput_setFunction` and `INPUT_COMMAND_FILTER` is specified as the *type* parameter. *userInput_commandFilter* does not need to be the function name. The actual function name is insignificant to MicroStation.

MicroStation calls all *userInput_commandFilter* functions for each command queue element before dispatching the command queue element to the proper command processor. The *userInput_commandFilter* function disables commands and modifies command queue elements.

If the command specified in the queue element belongs to a class that `mdlInput_disableCommandClass` has disabled, the queue element is not passed to the `userInput_commandFilter` function.

If a `userInput_commandFilter` function indicates that the queue element was modified, MicroStation restarts the validation process. Therefore, the `userInput_commandFilter` function will be called again to see the modified queue element.

Returns `INPUT_COMMAND_ACCEPT` indicates that the queue element was not changed, and `INPUT_COMMAND_CHANGED` indicates that the queue element was changed. `INPUT_COMMAND_REJECT` indicates that the command was rejected.

See Also `mdlInput_setFunction`, `mdlInput_enableCommandClass`, `mdlInput_disableCommandClass`.

Functions That Display Messages (Output)

Output functions display messages for the MicroStation user. Some functions display messages in all fields in the command window. The messages display in the MicroStation command window.

The following table lists output functions:

Function	Used to
<code>mdlOutput_errorU</code> <code>mdlOutput_promptU</code> <code>mdlOutput_commandU</code> <code>mdlOutput_keyinU</code> <code>mdlOutput_messageU</code> <code>mdlOutput_statusU</code> <code>mdlOutput_flyoverMsgU</code>	unconditionally display the message in the Command Window field that corresponds to a given message name.
<code>mdlOutput_error</code> <code>mdlOutput_prompt</code> <code>mdlOutput_command</code> <code>mdlOutput_keyin</code> <code>mdlOutput_message</code> <code>mdlOutput_status</code> <code>mdlOutput_flyoverMsg</code>	display the message in the Command Window field that corresponds to a given message name only after checking the TCB variable.
<code>mdlOutput_printf</code>	display a message in a Command Window field.
<code>mdlOutput_vprintf</code>	The message is formatted according to a specified format string.

Function	Used to
mdlOutput_rscPrintf mdlOutput_rscvPrintf	display a message in a Command Window field. The message is formatted according to a string taken from a resource file.
mdlOutput_rscfPrintf mdlOutput_rscvfPrintf	write a string to a file. The string is formatted according to a string taken from a resource file.
mdlOutput_rscsPrintf mdlOutput_rscvsPrintf	generate a string into a specified buffer. The string is formatted according to a string taken from a resource file.

Two fields in the `tcb` variable or in the structure referenced by the global data variable, `tcb`, can inhibit normal display of messages in the command window. These fields are `tcb->control.inh_err` for inhibiting all non-fatal error messages and `tcb->control.inh_msg` for inhibiting command, status and prompt messages. These flags do not affect output functions with names ending in 'U'. These functions display the message, regardless of the flag value. Other functions display the messages only if the appropriate flag is zero. User command applications often use these flags to disable MicroStation messages so the applications can display their own prompts.

MDL commands that behave as if they are part of MicroStation use output functions that do not end with 'U.'

Example

See `output.mc`.

mdlOutput_errorU, mdlOutput_promptU, mdlOutput_commandU, mdlOutput_keyinU, mdlOutput_messageU, mdlOutput_statusU

```
void mdlOutput_errorU
(
char    *pMessage      /* => points to the message */
);

void mdlOutput_promptU
(
char    *pMessage      /* => points to the message */
);

void mdlOutput_commandU
(
char    *pMessage      /* => points to the message */
);

void mdlOutput_keyinU
(
```

```

char    *pMessage      /* => points to the message */
);

void mdlOutput_messageU
(
char    *pMessage      /* => points to the message */
);

void mdlOutput_statusU
(
char    *pMessage      /* => points to the message */
);

```

Description *mdlOutput_errorU, mdlOutput_promptU, mdlOutput_commandU, mdlOutput_keyinU, mdlOutput_messageU and mdlOutput_statusU* display the message designated by *pMessage* of the appropriate field in the command window.

In the status area, the command window fields correspond to status area fields as follows:

Command Window Field	Corresponding Status Area Field
command field	in the left portion of status area.
prompt field	in left portion of status area, to the right of the command string, (i.e., command> prompt).
error field	overlays the command> prompt in the left portion of the status area.
msg field	in the right portion of the status area.
status field	in the right portion of the status area, to the right of the snap mode indicator.



The flyover msg field DOES NOT EXIST in the command window. In the key-in area, it overlays the command prompt error field. This is the field where tool descriptions are displayed. It is erased automatically if a tool description is displayed, a prompt or error is displayed, or the user clicks in the status bar.

For all of these functions, the appropriate message displays regardless of the values in *tcb->control.inh_err* and *tcb->control.inh_msg*.

Returns These functions are all of type `void`. They return no values.

See Also *mdlOutput_error, mdlOutput_prompt, mdlOutput_command, mdlOutput_keyin, mdlOutput_message, mdlOutput_status.*

mdlOutput_error, mdlOutput_prompt, mdlOutput_command, mdlOutput_keyin, mdlOutput_message, mdlOutput_status

```

void mdlOutput_error
(
char    *pMessage      /* => pointer to the message */
);
void mdlOutput_prompt
(
char    *pMessage      /* => pointer to the message */
);
void mdlOutput_command
(
char    *pMessage      /* => pointer to the message */
);
void mdlOutput_keyin
(
char    *pMessage      /* => pointer to the message */
);
void mdlOutput_message
(
char    *pMessage      /* => pointer to the message */
);
void mdlOutput_status
(
char    *pMessage      /* => pointer to the message */
);

```

Description The mdlOutput_error, mdlOutput_prompt, mdlOutput_command, mdlOutput_keyin, mdlOutput_message and mdlOutput_status functions display the message designated by *pMessage* in the appropriate field in the command window.

A non-zero value for *tcb->control.inh_err* prevents mdlOutput_error from displaying messages.

A non-zero value for *tcb->control.inh_msg* prevents all other functions from displaying messages.

Returns These functions are all of type void. They return no values.

See Also mdlOutput_errorU, mdlOutput_promptU, mdlOutput_commandU, mdlOutput_keyinU, mdlOutput_messageU, mdlOutput_statusU.

mdlOutput_printf

```
#include <mdl.h>
void mdlOutput_printf
(
    int      fieldID,          /* => MSG_KEYIN, MSG_ERROR, MSG_PROMPT */
    char     *pFormat          /* => pointer to format string */
    ...
);
```

Description The mdlOutput_printf function displays a message in the command window field designated by *fieldID*. Valid values are MSG_MESSAGE, MSG_ERROR, MSG_PROMPT, MSG_STATUS, MSG_COMMAND and MSG_KEYIN.

The message is assembled according to the format specified by *pFormat* and using the values supplied by the variable argument list. The format string is a standard printf format string.

Returns The mdlOutput_printf function is of type void. It returns no value.

See Also mdlOutput_vprintf, mdlOutput_rscPrintf, mdlOutput_rscvPrintf.

mdlOutput_vprintf

```
#include <mdl.h>
#include <varargs.h>
void mdlOutput_vprintf
(
    int      fieldID,          /* => MSG_KEYIN, MSG_ERROR, MSG_PROMPT */
    char     *pFormat,         /* => pointer to format string */
    va_list  varArgList        /* => variable argument list */
);
```

Description The mdlOutput_vprintf function displays a message in the command window field designated by *fieldID*. Valid values are MSG_MESSAGE, MSG_ERROR, MSG_PROMPT, MSG_STATUS, MSG_COMMAND and MSG_KEYIN.

The message is assembled according to the format specified by *pFormat* using the values specified by *varArgList*. See the documentation of printf for a list of supported format specifiers.



varArgList represents a variable argument list started by the calling function. mdlOutput_vprintf is typically used by functions that are called with variable argument lists. These functions can pass the variable argument list to mdlOutput_vprintf.



The string resulting from interpreting the format string can be of unlimited length; it is no longer limited.

Returns The mdlOutput_vprintf function is of type void. It returns no value.

See Also `mdlOutput_printf`, `mdlOutput_rscPrintf`, `mdlOutput_rscvPrintf`.

mdlOutput_rscPrintf, mdlOutput_rscvPrintf

```
#include <mdl.h>
#include <varargs.h>
#include <rscdefs.h>

void mdlOutput_rscPrintf
(
    int                fieldID,                /* => MSG_MESSAGE, etc. */
    RscFileHandle      resourceFileHandle, /* => handle or NULL */
    long               resourceId,            /* => ID of string list */
    int                stringNumber,          /* => number of string */
    ...
);

void mdlOutput_rscvPrintf
(
    int                fieldID,                /* => MSG_MESSAGE, etc. */
    RscFileHandle      resourceFileHandle, /* => handle or NULL */
    long               resourceId,            /* => ID of string list */
    int                stringNumber,          /* => number of string */
    va_list            varArgList
);
```

Description The `mdlOutput_rscPrintf` and `mdlOutput_rscvPrintf` functions display a message in the command window field designated by *fieldID*. The message is assembled according to the format specified by the string in a `MessageList` resource. For `mdlOutput_rscPrintf`, the values substituted into the string are passed as arguments.

For `mdlOutput_rscvPrintf`, the values are specified by *varArgList*. *varArgList* represents a variable argument list started by the calling function. `mdlOutput_rscvPrintf` is typically used by functions that are called with variable argument lists. These functions can pass the variable argument list to `mdlOutput_rscvPrintf`.

The format string is retrieved from the resource file. *resourceFileHandle* specifies the resource file containing the `MessageList`.

resourceId specifies the message list containing the string.

stringNumber is the number of the string in the message list.

The string resulting from interpreting the format string must have less than 600 characters.

Returns The `mdlOutput_rscPrintf` and `mdlOutput_rscvPrintf` functions are of type `void`. They return no values.

See Also `mdlOutput_vprintf`, `mdlOutput_printf`.

**mdlOutput_rscfPrintf, mdlOutput_rscsPrintf, mdlOutput_rscvfPrintf,
mdlOutput_rscvsPrintf**

```

#include <msoutput.fdf>

void int mdlOutput_rscfPrintf
(
FILE                *fp,                /* => file pointer */
RscFileHandle       resourceFileHandle, /* => source file */
long                resourceId,         /* => source resource */
int                 stringNumber,       /* => source string */
...
);

void int mdlOutput_rscsPrintf
(
char                *bufferP,           /* => receiving buffer */
RscFileHandle       resourceFileHandle, /* => source file */
long                resourceId,         /* => source resource */
int                 stringNumber,       /* => source string */
...
);

void int mdlOutput_rscvfPrintf
(
FILE                *fp,                /* => file pointer */
RscFileHandle       resourceFileHandle, /* => source file */
long                resourceId,         /* => source resource */
int                 stringNumber,       /* => source string */
va_list             varg                /* => arg. list to pass on */
);

void int mdlOutput_rscvsPrintf
(
char                *bufferP,           /* => receiving buffer */
RscFileHandle       resourceFileHandle, /* => source file */
long                resourceId,         /* => source resource */
int                 stringNumber,       /* => source string */
va_list             varg                /* => arg. list to pass on */
);

```

Description The mdlOutput_rscfPrintf, mdlOutput_rscsPrintf, mdlOutput_rscvfPrintf and mdlOutput_rscvsPrintf functions round out the mdlOutput_rscPrintf functions the same way fprintf, sprintfv, fprintf and vsprintf round out the printf class of functions. The functions consist of a core of functionality that generates an output string based on a format string and a variable number of arguments. As with the mdlOutput_rscPrintf function, the *resourceFileHandle*, *resourceId* and *stringNumber* parameters identify a printf style format string.

For mdlOutput_rscfPrintf and mdlOutput_rscvfPrintf, the generated string is written to the file identified by the *fp* parameter.

For `mdlOutput_rscsPrintf` and `mdlOutput_rscvsPrintf`, the generated string is copied to the buffer provided by the *bufferP* parameter.

For the `mdlOutput_rscfPrintf` and `mdlOutput_rscsPrintf` functions, the variable argument list is appended to the parameter list after the *stringNumber* parameter.

For the `mdlOutput_rscvfPrintf` and `mdlOutput_rscvsPrintf` functions, a variable argument list is passed along in the *varg* parameter. This is a standard method used when one function must receive a variable argument list and pass it on to another function.

Returns These functions return an integer specifying the number of bytes in the generated string.

See Also `mdlOutput_rscPrintf`, `mdlOutput_rscvPrintf`.

C Expression Handling Functions

The C Expression Handling functions evaluate C expressions generated at runtime. This set of functions is powerful for user interfaces.

An expression evaluated by the C expression functions can contain variable and function references. For the names to be available, they must be published using `mdlCExpression_symbolPublish`.

Symbols are managed in **symbol sets**. Before symbols can be published, a symbol set must be allocated. When an MDL application calls `mdlCExpression_symbolPublish` to publish a symbol, it must specify a symbol set. The symbol becomes a member of that set. When a symbol set is freed, the definitions of all published symbols are removed. Associated with each symbol set is a **visibility flag**. Functions which evaluate expressions require a visibility flag as a parameter. The visibility flags are bit masks. When an `mdlCExpression_...` function searches for the symbol definition, it examines the visibility mask of each symbol set to determine the symbol sets to include in the search. To compare the visibility flags, it does a bitwise AND. It does not search the symbol set if the bitwise AND produces zero.

When an `mdlCExpression_...` function searches the symbol sets, it starts the search with symbol sets that were created by the application that called the `mdlCExpression_...` function. Different applications can publish symbols with the same name.

When a symbol is published, it is necessary to specify the type. If the type definition is large, the type definition is generally contained in a resource file. To generate a type definition in a resource file, use the `rsctype` utility. The input to `rsctype` is a list of structure and union definitions. The output is a resource manager source file that must be compiled with `rcomp`. See “Generating Resource Files from C Type Defs” in the “Developing Applications” chapter for more information.

The C expression functions divide integers differently than C does. If the result of integer division produces a floating point result, C converts it to 0 by truncating. The `mdlCExpression_...` functions treat the result as a floating point value. For example, in C, the result of $1/4$ is 0. With the `mdlCExpression_...` functions, the result of $1/4$ is 0.25.

The following table lists C expression handling functions:

Function	Used to
<code>mdlCExpression_symbolPublish</code>	make a symbol available to be used by <code>mdlCExpression_getValue</code> and <code>mdlCExpression_setValue</code> .
<code>mdlCExpression_initializeSet</code>	initialize a symbol set.
<code>mdlCExpression_freeSet</code>	free a symbol set.
<code>mdlCExpression_typeFromRsc</code>	make a memory-resident type definition based on a type definition in a resource file.
<code>mdlCExpression_typePointer</code>	make a pointer type definition.
<code>mdlCExpression_typeArray</code>	make an array type definition.
<code>mdlCExpression_getValue</code>	evaluate a C expression and return the value returned by <code>mdlCExpression_getValue</code> and <code>mdlCExpression_setValue</code> .
<code>mdlCExpression_setValue</code>	evaluate a C expression and set the value returned by <code>mdlCExpression_getValue</code> and <code>mdlCExpression_setValue</code> .
<code>mdlCExpression_generateMessage</code>	generate an error message based on the last call to <code>mdlCExpression_getValue</code> and <code>mdlCExpression_setValue</code> .
<code>mdlCExpression_isArray</code>	determine if a type definition specifies an array.
<code>mdlCExpression_isStructUnion</code>	determine if a type definition specifies a structure or union.
<code>mdlCExpression_isCharPointer</code>	determine if a type definition specifies a character pointer.

Example

See `cexpr.mc` and `calculat.mc`.

mdlCEXpression_symbolPublish

```
#include <cexpr.h>
int mdlCEXpression_symbolPublish
(
    void    *setP,          /* => Symbol set descriptor */
    char    *nameP,         /* => Symbol to publish */
    int     class,          /* => SYMBOL_CLASS_FUNCTION, etc. */
    CType   *typeP,         /* => Type of data */
    void    *dataP          /* => Location of data or function */
);
```

Description The `mdlCEXpression_symbolPublish` function publishes a symbol so `mdlCEXpression_getValue` can use it.

setP specifies the symbol set to use. The symbol set must be previously initialized with `mdlCEXpression_initializeSet`.

nameP specifies the name to be published.

class can be `SYMBOL_CLASS_FUNCTION`, `SYMBOL_CLASS_STRUCT` or `SYMBOL_CLASS_VAR`.

If `SYMBOL_CLASS_FUNCTION` is used, *dataP* must be the address of an MDL function. It cannot be the address of a built-in function.

If `SYMBOL_CLASS_STRUCT` is used, *typeP* must point to the definition of a structure or union. `SYMBOL_CLASS_STRUCT` makes a structure or union available in casts in expressions. *dataP* is ignored for `SYMBOL_CLASS_STRUCT`.

If `SYMBOL_CLASS_VAR` is used, *typeP* defines the type of the variable. *dataP* must define the location of the variable.

typeP can point to a type definition obtained from `mdlCEXpression_typeFromRsc`, `mdlCEXpression_typePointer` or `mdlCEXpression_typeArray`. It can also point to one of the built-in variables that specify types: `longType`, `shortType`, `charType`, `ulongType`, `ucharType`, `doubleType` and `voidType`.

Returns The `mdlCEXpression_symbolPublish` function returns `SUCCESS` if no errors are encountered. Otherwise, it returns a non-zero value and stores more specific information in `mdlErrno`. To generate an error message based on the result of `mdlCEXpression_symbolPublish`, use `mdlCEXpression_generateMessage`.

mdlCExpression_initializeSet

```
#include <mdl.h>
void *mdlCExpression_initializeSet
(
    unsigned long    visibilityFlag,    /* => visibility mask */
    int              sizeHashTable,    /* => usually 0 */
    int              global             /* => TRUE or FALSE */
);
```

Description The mdlCExpression_initializeSet function initializes a symbol set.

visibilityFlag is a bitmask. If the symbols in a symbol set are used for dialog boxes, specify VISIBILITY_DIALOG_BOX. If the symbols in a symbol set are used for the calculator/preprocessor, specify VISIBILITY_CALCULATOR. If the symbols in a symbol set are used for both dialog boxes and the preprocessor, specify (VISIBILITY_DIALOG_BOX | VISIBILITY_CALCULATOR).

sizeHashTable specifies the number of buckets used by a symbol set. Typically, this number is 0 to specify the default size.

To prevent other applications from accessing the symbol set, specify FALSE for *global*; otherwise, specify TRUE.

global can be TRUE or FALSE for a symbol set being created for symbols being made available to the Dialog Manager. *global* should be TRUE for a symbol set being created for symbols being made available to the calculator/preprocessor.

Returns The mdlCExpression_initializeSet function returns a pointer to the symbol set. If NULL is returned, more specific error information is provided in mdlErrno.

mdlCExpression_freeSet

```
void mdlCExpression_freeSet
(
    void *setP      /* => pointer to a symbol set */
);
```

Description The mdlCExpression_freeSet function frees a symbol set. All memory used by the symbol set is freed. The symbols published in the symbol set are no longer available for C expression handling.

When an MDL application is unloaded, all symbol sets belonging to that application are freed.

Returns The mdlCExpression_freeSet function is of type void. It returns no value.

mdlCEXpression_typeFromRsc

```
#include <rsdefs.h>
void *mdlCEXpression_typeFromRsc
(
    void          *setP,      /* => visibility set */
    char          *name,      /* => name of the structure or union */
    RscFileHandle rfHandle /* => resource file handle */
);
```

Description The `mdlCEXpression_typeFromRsc` function creates an in-memory definition of a structure or union based on the definition contained in a resource file. The definition in memory does not contain information on the members of the structure. It contains only enough information for MicroStation to find the members quickly.

setP specifies the symbol set that the type will be associated with. When this symbol set is freed, the type definition is also freed.

rfHandle specifies the resource file handle for the resource file that contains the definition. If *rfHandle* is 0, `mdlCEXpression_typeFromRsc` searches the resource files the application has open. The resource file must remain open the entire time the type could be accessed.

Returns The `mdlCEXpression_typeFromRsc` function returns a pointer to the type definition. If an error occurs, it returns NULL; more information is available in `mdlErrno`.

See Also The `rsctype` utility section in the “Developing Applications” chapter.

mdlCEXpression_typePointer

```
#include <cexpr.h>
CType *mdlCEXpression_typePointer
(
    void          *setP,      /* => visibility set */
    CType         *madeofP /* => type of what it points to */
);
```

Description The `mdlCEXpression_typePointer` function creates a definition of a pointer type.

setP specifies the symbol set the type will be associated with. When this symbol set is freed, the type definition is also freed.

madeofP specifies a type created by `mdlCEXpression_typeFromRsc`, `mdlCEXpression_typePointer`, `mdlCEXpression_typeArray` or a built-in type that the new type points to. This type can be created with one of the `mdlCEXpression_type...` functions, or one of the built-in types.

Returns The `mdlCEXpression_typePointer` function returns a pointer to the type definition. If an error occurs, it returns NULL; more information is available in `mdlErrno`.

See Also `mdlCEXpression_typeFromRsc`, `mdlCEXpression_typeArray`.

mdlCExpression_typeArray

```
#include <cexpr.h>

CType *mdlCExpression_typeArray
(
    void      *setP,          /* => visibility set */
    CType     *madeofP,       /* => type of what it points to */
    int       count           /* => count of entries in array */
);
```

Description The mdlCExpression_typeArray function creates a definition of an array type.

setP specifies the symbol set the type will be associated with. When this symbol set is freed, the type definition is also freed.

madeofP specifies the type of the array elements. This type can be created with one of the mdlCExpression_type... functions or one of the built-in types.

count provides a count of the number of elements in the array.

Returns The mdlCExpression_typeArray function returns a pointer to the type definition. If an error occurs, it returns NULL; more information is available in mdlErrno.

See Also mdlCExpression_typeFromRsc, mdlCExpression_typePointer.

mdlCExpression_getValue

```
#include <cexpr.h>

int mdlCExpression_getValue
(
    CExprValue *valueP,          /* <=> value of expression */
    CExprResult *resultP,        /* <=> result of expression */
    char       *expressionP,     /* => expr. to be evaluated */
    unsigned long visibilityFlag /* => symbol set mask */
);
```

Description The mdlCExpression_getValue function evaluates the expression specified by *expressionP*.

visibilityFlag determines the symbol sets used in evaluating the expression. When searching for symbols, MicroStation examines each symbol set performing a bitwise AND of *visibilityFlag* and the symbol set's visibility flag. If the result is not zero, MicroStation looks for a definition of the symbol in that symbol set.

Most applications that use mdlCExpression_getValue use the result returned in *valueP*.

valueP->type is CEXPR_TYPE_POINTER, CEXPR_TYPE_DOUBLE or CEXPR_TYPE_LONG. If the result of the operation is smaller than a long, the

result is promoted to a long.

valueP->*val* contains the value resulting from evaluating the expression.

resultP contains a detailed description of the expression result.

resultP->*class* can be CL_VALUE, CL_LVALUE or CL_ERROR. If it is CL_ERROR, the result cannot be used. If it is CL_LVALUE, *resultP*->*value* does not contain the actual value. It contains a pointer to where the value is kept. If *resultP*->*class* is CL_VALUE, *resultP*->*value* does contain the actual value. If *resultP*->*class* is CL_LVALUE, *resultP* can be used as a parameter to mdlCEXpression_setValue.

Returns The mdlCEXpression_getValue function returns SUCCESS if no errors are encountered. Otherwise, it returns a non-zero value and stores more specific information in mdlErrno. To generate an error message based on the result of mdlCEXpression_getValue and mdlCEXpression_setValue, use mdlCEXpression_generateMessage.

See Also mdlCEXpression_setValue.

mdlCEXpression_setValue

```
#include <cexpr.h>

int mdlCEXpression_setValue
(
    CExprValue *valueP,           /* <=> expression value */
    CExprResult *resultP,        /* => previous result or NULL */
    char *expressionP,          /* => expression or NULL */
    unsigned long visibilityFlag /* => symbol set mask */
);
```

Description The mdlCEXpression_setValue function stores the value specified by the structure *valueP* points to. The location to receive the value is specified as *resultP* if *resultP* is not NULL, or by *expressionP* if *resultP* is NULL. Generally, if *resultP* is not NULL, it is the result of a call to mdlCEXpression_getValue.

visibilityFlag determines the symbol sets used in evaluating the expression. When searching for symbols, MicroStation examines each symbol set performing a bitwise AND of *visibilityFlag* and the symbol set's visibility flag. If the result is not zero, MicroStation looks for a symbol definition in that symbol set.

valueP->*type* is CEXPR_TYPE_POINTER, CEXPR_TYPE_DOUBLE or CEXPR_TYPE_LONG. If the result of the operation is smaller than a long, the result is promoted to a long.

resultP->*val* contains the value resulting from evaluating the expression.

Returns The mdlCEXpression_setValue function returns SUCCESS if no errors are encountered. Otherwise, it returns a non-zero value and stores more specific information in mdlErrno. To generate an error message based on the result of

mdlCExpression_getValue and mdlCExpression_setValue, use
mdlCExpression_generateMessage.

See Also mdlCExpression_getValue.

mdlCExpression_generateMessage

```
void mdlCExpression_generateMessage
(
    char    *messageBuffer,    /* => buffer to receive message */
    int     errorNumber        /* => value taken from mdlErrno */
);
```

Description The mdlCExpression_generateMessage function generates an error message based on the value passed in *errorNumber*. *errorNumber* should contain a value taken from mdlErrno after one of the mdlCExpression_... functions returns an error.

Returns The mdlCExpression_generateMessage function is of type void. It returns no value.

mdlCExpression_isArray

```
int mdlCExpression_isArray
(
    CType   *typeP            /* => points to a type definition */
);
```

Description The mdlCExpression_isArray function determines whether the type specified by *typeP* is an array.

Returns The mdlCExpression_isArray returns TRUE if the type is an array type. It returns FALSE if the type does not specify an array.

mdlCExpression_isStructUnion

```
#include <cexpr.h>

int mdlCExpression_isStructUnion
(
    CType   *typeP            /* => points to a type definition */
);
```

Description The mdlCExpression_isStructUnion function determines whether the type specified by *typeP* is a structure or union.

Returns mdlCExpression_isStructUnion returns TRUE if the type is a structure or union type. It returns FALSE if the type does not specify a structure or union.

mdlCExpression_isCharPointer

```
#include <cexpr.h>
int mdlCExpression_isCharPointer
(
  CType  *typeP    /* => points to a type definition */
);
```

Description The `mdlCExpression_isCharPointer` function determines if the type specified by *typeP* is a character pointer.

Returns The `mdlCExpression_isCharPointer` returns `TRUE` if the type is a character pointer type. It returns `TRUE` regardless of whether it points to signed or unsigned characters and whether the type is an array type. It returns `FALSE` if the type does not specify a character pointer.

3

Dialog Box Manager

This section contains the functionality necessary to create dialog boxes for MDL applications.

This chapter will discuss:

- Menu bar item functions
- Text pull-down menu functions
- Option pull-down menu functions
- Option button item functions
- Icon functions
- List box item functions
- Text item functions
- Scroll bar item functions
- Push Button Item functions
- Tab Page List and Tab Page functions
- Toggle Button Item functions
- Level Map Item functions
- Color Picker Item functions
- Completion Bar Item functions
- Dialog box general functions
- Dialog box item functions
- Dialog box drawing functions
- Rectangle drawing functions
- Dialog box font functions
- Queuing functions
- Track bar window functions

- Busy bar window functions
- Command number functions
- Miscellaneous dialog box functions
- Tab Page List and Tab Page functions
- Combo Box Item functions
- Spin Box Item functions

Menu Bar Item Functions

The menu bar item functions manipulate the pull-down menus in menu bars. Menus can be added or deleted. Menu items can be deleted from individual menus. The menu title and state (enabled or disabled) can be changed using the functions in this section.

See the “Menu bar item” section of the “Standard Dialog Items” chapter for more information on using menu bar items.

The following table lists the menu bar functions:

Function	Used to
<code>mdlDialog_menuBarActivate</code>	make a menu bar active in the Command Window.
<code>mdlDialog_menuBarRegister</code>	add an item to applications pull-down menu.
<code>mdlDialog_menuBarAddAppMenu</code>	add the applications pull-down to a menu bar.
<code>mdlDialog_menuBarAddCmdWinMenu</code>	add a menu to the Command Window.
<code>mdlDialog_menuBarAttachMenu</code>	attach a pull-down menu to a menu bar.
<code>mdlDialog_menuBarDeleteCmdWinMenu</code>	remove a pull-down menu from the Command Window menu bar
<code>mdlDialog_menuBarDetachMenu</code>	remove a pull-down menu from a menu bar.
<code>mdlDialog_menuBarFind</code>	find a dialog's menu bar.
<code>mdlDialog_menuBarFindAppMenu</code>	get pointer to applications pull-down menu.
<code>mdlDialog_menuBarGetCmdWinP</code>	get pointer to Command Window menu bar.
<code>mdlDialog_menuBarSetDefault</code>	set default menu for Command Window.
<code>mdlDialog_menuBarUnloadApp</code>	remove menu bar from applications menu.
<code>mdlDialog_menuBarFindItem [ditemlb.ml]</code>	find a pull-down menu item by its search ID.
<code>mdlDialog_menuBarGetItem [ditemlb.ml]</code>	get a pull-down menu item by its position within a pull-down menu.
<code>mdlDialog_menuBarFindMenu [ditemlb.ml]</code>	find a pull-down menu by its type and ID.

Function	Used to
mdlDialog_menuBarGetMenu [ditemlb.ml]	get a pull-down menu by its position within a menu bar item.
mdlDialog_menuBarGetNMenus [ditemlb.ml]	get the number of pull-down menus contained by a menu bar item.
mdlDialog_menuBarGetNItems [ditemlb.ml]	get the number of items contained by a pull-down menu.
mdlDialog_menuBarGetSelection [ditemlb.ml]	get information about the last selected pull-down menu item.
mdlDialog_menuBarDeleteAllMenus [ditemlb.ml]	delete all pull-down menus from a menu bar item.
mdlDialog_menuBarDeleteMenu [ditemlb.ml]	delete a pull-down menu from a menu bar item.
mdlDialog_menuBarInsertMenu mdlDialog_menuBarInsMenu	insert a pull-down menu into a menu bar item.
mdlDialog_menuBarDeleteItem	delete a pull-down menu item.
mdlDialog_menuBarDeleteAllItems	delete all the items from a pull-down menu.
mdlDialog_menuBarMenuGetTitle	get the title of a pull-down menu.
mdlDialog_menuBarMenuSetTitle	set the title of a pull-down menu.
mdlDialog_menuBarMenuGetEnabled	get the state (enabled or disabled) of a pull-down menu.
mdlDialog_menuBarMenuSetEnabled	set the state (enabled or disabled) of a pull-down menu.

mdlDialog_menuBarActivate

```

boolean mdlDialog_menuBarActivate /* <= TRUE if error */
(
    DItem_PulldownMenuItem      *menuItemP,      /* => item to activate */
    char      *taskIdP,          /* => or use task id ptr */
    long      menuBarId,         /* => and id of menu bar */
    boolean override,            /* => FALSE=ignore next 2 args */
    boolean disableMicroStation, /* => disable uStation menu bar */
    boolean setCmdWindowTitle /* => set title to appName when activated? */
);

```

Description mdlDialog_menuBarActivate makes a menu bar active in the command window.

menuItemP indicates the menu item within the Applications Menu that corresponds to the menu to make active. If *menuItemP* is not available, *taskIdP* and *menuBarId* may be used instead. Pass NULL in *taskId* to use the current task. Pass zero in *menuBarId* to match the first item found.

override tells whether to ignore the remaining two arguments: *disableMicroStation*, which indicates whether to disable the MicroStation menu bar, and *setCmdWindowTitle*, which indicates whether to set the title of the Command Window to the application name.

Returns mdlDialog_menuBarActivate returns SUCCESS, or a non-zero value on error.

mdlDialog_menuBarRegister

```
boolean mdlDialog_menuBarRegister /* <= TRUE if error */
(
  DItem_PulldownMenuItem      *newMenuItemP, /* <= created menu item */
  long   menuBarId,            /* => new menu bar to load */
  char   *appName,             /* => application name */
  boolean disableMicroStation, /* => disable uStation menus */
  boolean setCmdWindowTitle    /* => set title to appName when active? */
);
```

Description mdlDialog_menuBarRegister creates a menu bar item from resource information. The newly created menu appears as a selection on the applications menu.

menuItemP points to the newly created menu bar item.

menuBarId tells which menu bar to create.

appName gives the full name of the application owning the menu. The name is inserted into the applications menu.

disableMicroStation indicates whether the MicroStation menu bar should be disabled when this menu is active, and *setCmdWindowTitle* indicates whether the title of the Command Window should be set to *appName*.

Returns mdlDialog_menuBarRegister returns SUCCESS, or a non-zero value if there is an error.

mdlDialog_menuBarAddAppMenu

```
DItem_PulldownMenu *mdlDialog_menuBarAddAppMenu /* <= NULL=error */
(
  RawItemHdr *menuBarP
);
```

Description The mdlDialog_menuBarAddAppMenu function adds the Applications menu to the Command Window menu bar, or to another menu bar if specified.

menuBarP points to the raw item header of the menu bar to which to add the Applications menu. If *menuBarP* is NULL, it is added to the Command Window menu bar.

Returns mdlDialog_menuBarAddAppMenu returns a pointer to the Applications pull-down menu, or NULL if there is an error.

mdlDialog_menuBarAddCmdWinMenu

```
boolean mdlDialog_menuBarAddCmdWinMenu /* <= TRUE if error */
(
  Ditem_PulldownMenuItem *mbarItemP, /* <= inserted item (never NULL) */
  long mbarId, /* => id of menu to load */
  boolean insertAlphabetically /* => load in alpha order */
);
```

Description The mdlDialog_menuBarAddCmdWinMenu function adds a pull-down menu to the Command Window menu bar.

mbarItemP points to the pull-down menu once it is added.

mbarId identifies which pull-down menu to add.

insertAlphabetically indicates whether the new pull-down menu should be inserted in alphabetical order among the existing menus. If not, it is added to the end.

Returns mdlDialog_menuBarAddCmdWinMenu returns SUCCESS or a non-zero value if there is an error.

mdlDialog_menuBarAttachMenu

```
Ditem_PulldownMenu *mdlDialog_menuBarAttachMenu /* <= NULL=error */
(
  RawItemHdr *mbarP, /* => ptr to a menu bar */
  Ditem_PulldownMenu *menuP, /* => menu to attach */
  Ditem_PulldownMenu *beforeMenuP /* => NULL means append */
);
```

Description mdlDialog_menuBarAttachMenu adds a pull-down menu to any menu bar.

mbarP points to the raw item header of the menu bar to which to add *menuP*.

If *beforeMenuP* is specified, the new pull-down menu *menuP* is added to the menu bar in the position just before *beforeMenuP*; otherwise it is added to the end.

Returns mdlDialog_menuBarAttachMenu returns a pointer to the added pull-down menu item, or NULL if there is an error.

mdlDialog_menuBarDeleteCmdWinMenu

```
boolean mdlDialog_menuBarDeleteCmdWinMenu /* <= TRUE if error */
(
  Ditem_PulldownMenuItem *mbarItemP /* <= item to delete (never NULL) */
);
```

Description `mdlDialog_menuBarDeleteCmdWinMenu` is used to remove a pull-down menu from the Command Window.

mbarItemP points to the pull-down menu item to be removed from the Command Window menu.

Returns `mdlDialog_menuBarDeleteCmdWinMenu` returns `SUCCESS`, or a non-zero value on error.

mdlDialog_menuBarDetachMenu

```
boolean mdlDialog_menuBarDetachMenu/* <= TRUE if error */
(
  DItem_PulldownMenu*menuP, /* => non-NULL=ignore last 2 args */
  RawItemHdr          *mbarP, /* => ptr to a menu bar */
  ULONG               menuType, /* => type of menu to delete */
  long                menuId,   /* => resourceId of menu to delete */
  boolean             redraw    /* => redraw menubar */
);
```

Description `mdlDialog_menuBarDetachMenu` removes a pull-down menu from a menu bar.

menuP points to the pull-down menu item to remove. If it is `NULL`, the *menuType* and *menuId* parameters are used to identify the pull-down menu item instead.

mbarP points to the raw item header of the menu bar from which to delete the pull-down.

redraw indicates whether the menu bar should be redrawn after the item is removed.

Returns `mdlDialog_menuBarDetachMenu` returns `SUCCESS`, or a non-zero value on error.

mdlDialog_menuBarFind

```
DialogItem *mdlDialog_menuBarFind/* <= menu bar item ptr */
(
  DialogBox    *dbP /* => dialog box that contains menu bar */
);
```

Description `mdlDialog_menuBarFind` returns a pointer to the menu bar of the specified dialog box, if any.

dbP is a pointer to the dialog box in question.

Returns `mdlDialog_menuBarFind` returns a pointer to the dialog's menu bar if there is one, or else `NULL`.

mdlDialog_menuBarFindAppMenu

```
DItem_PulldownMenu *mdlDialog_menuBarFindAppMenu /* <= NULL=error */
(
  RawItemHdr **menuBarPP /* <= ptr to cmd window menu bar */
);
```

Description mdlDialog_menuBarFindAppMenu is used to get the pointer to the applications menu.

menuBarPP points to an MDL variable that will point to raw item header of the Command Window menu bar after the function is called.

Returns mdlDialog_menuBarFindAppMenu returns a pointer to the applications pull-down menu, or NULL if there is an error.

mdlDialog_menuBarGetCmdWinP

```
RawItemHdr *mdlDialog_menuBarGetCmdWinP(void);
/* <= Cmd win menu bar */
```

Description mdlDialog_menuBarGetCmdWinP is used to get a pointer to the raw item header of the Command Window menu bar.

Returns mdlDialog_menuBarGetCmdWinP returns NULL on error.

mdlDialog_menuBarSetDefault

```
boolean mdlDialog_menuBarSetDefault/* <= TRUE if error */
(
  DItem_PulldownMenuItem *menuItemP, /* => ctrling item to make def. */
  char *taskIdP, /* => task id ptr, NULL = current task */
  long menuBarId /* => menu bar id; 0 = match any found */
);
```

Description mdlDialog_menuBarSetDefault sets MicroStation's default Command Window menu bar from among the choices on the applications menu.

menuItemP is a pointer to the pull-down menu item on the applications menu that controls the menu to be made the new default. If *menuItemP* is NULL, *taskIdP* and *menuBarId* identify the desired menu bar instead.

Returns mdlDialog_menuBarSetDefault returns SUCCESS, or a non-zero value on error.

mdlDialog_menuBarUnloadApp

```
boolean mdlDialog_menuBarUnloadApp /* <= TRUE if error */
(
  DItem_PulldownMenuItem *menuItemP, /* => mbar w/ item to delete */
  char *taskIdP, /* => task id ptr, NULL = current task */
  long menuBarId /* => menu bar id; 0 = match any found */
);
```

Description `mdlDialog_menuBarUnloadApp` removes an applications menu item and the associated menu bar.

menuItemP points to the item to delete from the applications pull-down menu. If it is `NULL`, *taskIdP* and *menuBarId* may identify the item.

Returns `mdlDialog_menuBarUnloadApp` returns `SUCCESS`, or a non-zero value on error.

mdlDialog_menuBarFindItem [ditemlb.ml]

```
#include <dlogitem.h>

boolean mdlDialog_menuBarFindItem /* <= TRUE if not found */
(
    Ditem_PulldownMenuItem *menuItemP,      /* <= found menuItem */
    Ditem_PulldownMenu      **menuPP,       /* <=> NULL=don't want ptr */
    RawItemHdr *mbarP,                    /* => ptr to a menu bar */
    ULong      menuType,                  /* => *menuPP not NULL:ignored */
    long       menuId,                    /* => *menuPP not NULL:ignored */
    long       searchId                    /* => ID of item to find */
);
```

Description The `mdlDialog_menuBarFindItem` function finds, within a specified pull-down menu, the menu item whose search ID is *searchId*. All of the functions that manipulate menu items use a pointer to `Ditem_PulldownMenuItem` to indicate a particular menu item.

When the number of items in a menu is changing dynamically, obtaining a pointer to a menu item by searching for its search ID is more useful than getting the *n*th menu item. A menu item that is the seventh item could become the fifth item if earlier items are deleted by calling `mdlDialog_menuBarDeleteItem`.

If the menu item is found, the location specified by the *menuItemP* parameter will be filled with information identifying the menu item. If the menu item is not found, the location will be filled with `NULL`s. The *menuItemP* parameter cannot be `NULL`.

The *mbarP* parameter specifies the menu bar item to search.

The menu to search can be indicated in two ways. If a pointer to a menu was obtained through a call to `mdlDialog_menuBarFindMenu` or a previous call to `mdlDialog_menuBarFindItem` or `mdlDialog_menuBarGetItem`, the *menuPP* parameter can be set so that it points to the pointer. In this case, only the menu specified by *menuPP* will be searched for *searchId*. The *mbarP* parameter is unused and can be `NULL`.

If *menuPP* is a pointer to `NULL` or a `NULL` pointer, the menu bar item will first be searched for a menu whose type is *menuType* and ID is *menuId*. If *menuPP* was not a `NULL` pointer, the location pointed at by *menuPP* will be set to be a pointer to the menu that was found, or `NULL` if the menu was not found. The *mbarP* parameter must point to a menu bar item.

The following code fragment shows how to get pointers to MicroStation's Open/Close menu and the View 8 menu item:

```
{
    DialogBox *cmdWindDb;
    DialogItem *menuBarDiP;
    RawItemHdr *menuBarP;
    Ditem_PulldownMenuItem mbarItem;
    Ditem_PulldownMenu *mbarMenuP;

    /* first get a pointer to MicroStation's Command Window */
    cmdWindDb=mdlDialog_find(DIALOGID_CommandWindow, NULL);
    if (!cmdWindDb)
        return;

    /* now find the menu bar item */
    menuBarDiP=mdlDialog_itemGetByTypeAndId(cmdWindDb,
                                             RTYPE_MenuBar, MENUBARID_Main, 0);

    if (!menuBarDiP)
        return;
    menuBarP=menuBarDiP->rawItemP;

    mbarMenuP=NULL;
    if (mdlDialog_menuBarFindItem(&mbarItem, &menuBarP, menuBarP,
                                   RTYPE_PulldownMenu, PULLDOWNMENUID_SubOpenClose,
                                   MENUSEARCHID_ViewOpenCloseSub_8))

        return;

    /* mbarItem now contains information on the view 8 menu item */
    ...
}
```

Returns mdlDialog_menuBarFindItem returns TRUE if an error occurred or the menu item was not found.

See Also mdlDialog_menuBarGetItem [ditemlb.ml].

mdlDialog_menuBarGetItem [ditemlb.ml]

```
#include <dlogitem.h>

boolean mdlDialog_menuBarGetItem /* <= TRUE if error */
(
    Ditem_PulldownMenuItem *menuItemP,    /* <= NULL if not in range */
    Ditem_PulldownMenu **menuPP,         /* <=> send NULL don't want ptr */
    RawItemHdr *mbarP,                   /* => ptr to a menu bar */
    ULong menuType,                       /* => *menuPP not NULL:ignored */
    long menuId,                          /* => *menuPP not NULL:ignored */
    int subItemIndex /* => index of item to get */
);
```

Description The `mdlDialog_menuBarGetItem` function retrieves a pointer to the *subItemIndex* menu item of a specified pull-down menu. `mdlDialog_menuBarGetItem` should only be used if the number of items in the specified pull-down menu is not changing dynamically. Otherwise, `mdlDialog_menuBarFindItem [ditemlb.ml]` should be used instead.

subItemIndex specifies which menu item to retrieve a pointer to. *subItemIndex* must be greater than or equal to 0 and less than the number of items in the specified pull-down menu. Use the `mdlDialog_menuBarGetNItems [ditemlb.ml]` function to determine the number of items in a pull-down menu.

If *subItemIndex* is in range, the location specified by the *menuItemP* parameter will be filled with information identifying the appropriate menu item. If the *subItemIndex* is not in range, the location will be set to `NULL`. This *menuItemP* parameter can not be `NULL`. All of the functions that manipulate menu items use a pointer to `DItem_PulldownMenuItem` to indicate a particular menu item.

The *mbarP* parameter specifies a menu bar item.

The pull-down menu can be specified in two ways. If a pointer to a menu was obtained through a call to `mdlDialog_menuBarFindMenu [ditemlb.ml]` or a previous call to `mdlDialog_menuBarGetItem` or `mdlDialog_menuBarFindItem [ditemlb.ml]`, the *menuPP* parameter can be set so that it points to the pointer. The *mbarP* parameter is unused and can be `NULL` in this case.

If *menuPP* is a pointer to `NULL` or a `NULL` pointer, the menu bar item will first be searched for a menu whose type is *menuType* and ID is *menuId*. If *menuPP* was not a `NULL` pointer, the location pointed at by *menuPP* will be set to be a pointer to the menu that was found, or `NULL` if the menu was not found. The *mbarP* parameter must point to a menu bar item.

Returns The `mdlDialog_menuBarGetItem` function returns `TRUE` if an error occurred or *subItemIndex* was out of range.

See Also `mdlDialog_menuBarFindItem [ditemlb.ml]`, `mdlDialog_menuBarGetNItems [ditemlb.ml]`.

mdlDialog_menuBarFindMenu [ditemlb.ml]

```
#include <dlogitem.h>

DItem_PulldownMenu *mdlDialog_menuBarFindMenu /* <= NULL if error */
(
  RawItemHdr *mbarP,          /* => pointer to a menu bar */
  ULONG      menuType,        /* => type of menu to find */
  long       menuId           /* => resource ID of menu to find */
);
```

Description The mdlDialog_menuBarFindMenu function returns a pointer to the pull-down menu whose type is *menuType* and ID is *menuId* that is contained in the menu bar item *mbarP*. All of the functions that manipulate menus use a pointer to DItem_PulldownMenu to indicate a particular pull-down menu.

Returns The mdlDialog_menuBarFindMenu function returns NULL if an error occurs or the menu is not found.

See Also mdlDialog_menuBarGetMenu [ditemlb.ml].

mdlDialog_menuBarGetMenu [ditemlb.ml]

```
#include <dlogitem.h>

DItem_PulldownMenu *mdlDialog_menuBarGetMenu    /* <= NULL if error */
(
  RawItemHdr *mbarP,          /* => ptr to a menu bar */
  int         menuIndex       /* => index of menu to get */
);
```

Description The mdlDialog_menuBarGetMenu function returns a pointer to the pull-down menu at the *menuIndex* position within the menu bar item *mbarP*. All of the functions that manipulate menus use a pointer to DItem_PulldownMenu to indicate a particular pull-down menu.

The *menuIndex* parameter specifies which pull-down menu to retrieve a pointer to. *menuIndex* must be greater than or equal to 0 and less than the number of menus in the specified menu bar item. Use the mdlDialog_menuBarGetNMenus [ditemlb.ml] function to determine the number of menus in a menu bar item.

Returns The mdlDialog_menuBarGetMenu function returns NULL if an error occurs. This means that *mbarP* does not point at a menu bar item or *menuIndex* is out of range.

See Also mdlDialog_menuBarFindMenu [ditemlb.ml].

mdlDialog_menuBarGetNMenus [ditemlb.ml]

```
#include <dlogitem.h>

int mdlDialog_menuBarGetNMenus /* <= # of menus in menu bar */
(
  RawItemHdr *mbarP,          /* => ptr to a menu bar */
);
```

Description The mdlDialog_menuBarGetNMenus function returns the number of pull-down menus contained within the menu bar item specified by *mbarP*.

Returns The mdlDialog_menuBarGetNMenus function returns the number of pull-down menus in the specified menu bar item. It returns -1 if *mbarP* does not point at a menu bar item.

mdlDialog_menuBarGetNItems [ditemlb.ml]

```
#include <dlogitem.h>

int mdlDialog_menuBarGetNItems /* <= # of items in menu */
(
    Ditem_PulldownMenu**menuPP,      /* <=> send NULL don't want ptr */
    RawItemHdr      *mbarP,          /* => ptr to a menu bar */
    ULong           menuType,        /* => type of menu */
    long            menuId           /* => resource ID of menu */
);
```

Description The `mdlDialog_menuBarGetNItems` function returns the number of menu items contained within a specified pull-down menu.

The *mbarP* parameter specifies a menu bar item.

The pull-down menu can be specified in two ways. If a pointer to a menu was obtained through a call to `mdlDialog_menuBarFindMenu [ditemlb.ml]` or a previous call to `mdlDialog_menuBarGetItem [ditemlb.ml]` or `mdlDialog_menuBarFindItem [ditemlb.ml]` the *menuPP* parameter can be set so that it points to the pointer. The *mbarP* parameter is unused and can be `NULL` in this case.

If *menuPP* is a pointer to `NULL` or a `NULL` pointer, the menu bar item will first be searched for a menu whose type is *menuType* and ID is *menuId*. If *menuPP* was not a `NULL` pointer, the location pointed at by *menuPP* will be set to be a pointer to the menu that was found, or `NULL` if the menu was not found. The *mbarP* parameter must point to a menu bar item.

Returns `mdlDialog_menuBarGetNItems` returns the number of menu items in the specified menu. It returns -1 if an error occurs or the specified menu is not found.

See Also `mdlDialog_menuBarFindMenu [ditemlb.ml]`.

mdlDialog_menuBarGetSelection [ditemlb.ml]

```
#include <dlogitem.h>

boolean mdlDialog_menuBarGetSelection /* <= TRUE if error */
(
    ULong      *menuTypeP,      /* <= resourceType of menu selected */
    long       *menuIdP,        /* <= resource ID of menu selected */
    int        *subItemIndexP, /* <= item index of item selected */
    long       *searchIdP,      /* <= search ID of item selected */
    RawItemHdr *mbarP           /* => ptr to a menu bar */
);
```

Description The `mdlDialog_menuBarGetSelection` function retrieves information about the last menu item that was selected by the user from the menu bar item specified by *mbarP*.

The resource type and resource ID of the last pull-down menu the user selected an item from are copied to the locations specified by the *menuTypeP* and *menuIdP* parameters. Either of these parameters can be `NULL`, which indicates that the caller doesn't need the corresponding value.

The index and search ID of the last user selected pull-down menu item are copied to the locations specified by the *subItemIndexP* and *searchIdP* parameters. Either of these parameters can be `NULL`, which indicates that the caller doesn't need the corresponding value.

The *mbarP* parameter specifies a menu bar item.

This function is usually called as part of a item hook function attached to the menu bar item. In this way one hook function can handle the processing of multiple pull-down menus and pull-down menu items. Usually the `DITEM_MESSAGE_BUTTON` message is trapped with `buttonTrans` equal to `BUTTONTRANS_UP`. This message is sent when the data button is released inside of a pull-down menu item. Different actions can be taken based on the menu and item within the menu that was just selected.

Returns `mdlDialog_menuBarGetSelection` returns `TRUE` if an error occurs. This usually means that *mbarP* is not a pointer to a menu bar item.

See Also `mdlDialog_menuBarFindItem [ditemlb.ml]`, `mdlDialog_menuBarGetNItems [ditemlb.ml]`.

mdlDialog_menuBarDeleteAllMenus [ditemlb.ml]

```
#include <dlogitem.h>

boolean mdlDialog_menuBarDeleteAllMenus    /* <= TRUE if error */
(
    RawItemHdr *mbarP    /* => ptr to a menu bar */
);
```

Description The `mdlDialog_menuBarDeleteAllMenus` function removes all pull-down menus from the menu bar item specified by *mbarP*.

Returns `mdlDialog_menuBarDeleteAllMenus` returns `TRUE` if an error occurs. This means that *mbarP* does not point at a menu bar item.

See Also `mdlDialog_menuBarDeleteMenu [ditemlb.ml]`.

mdlDialog_menuBarDeleteMenu [ditemlb.ml]

```
#include <dlogitem.h>
boolean mdlDialog_menuBarDeleteMenu /* <= TRUE if error */
(
  Ditem_PulldownMenu*menuP, /* => if !NULL ignore last 2 args */
  RawItemHdr *mbarP,        /* => ptr to a menu bar */
  ULONG      menuType,       /* => type of menu to delete */
  long       menuId          /* => resource ID of menu to delete */
);
```

Description The `mdlDialog_menuBarDeleteMenu` function deletes the specified pull-down menu from the menu bar item specified by *mbarP*.

The pull-down menu to be deleted can be specified in two ways. If a pointer to a pull-down menu was obtained through a call to `mdlDialog_menuBarFindMenu [ditemlb.ml]`, `mdlDialog_menuBarGetItem [ditemlb.ml]` or `mdlDialog_menuBarFindItem [ditemlb.ml]`, the *menuP* parameter can be set to the pointer.

If *menuP* is `NULL`, the menu bar item will first be searched for a menu whose type is *menuType* and ID is *menuId*. If the menu is found, it will be deleted.

The *mbarP* parameter must point to a menu bar item. It cannot be `NULL`.

Returns `mdlDialog_menuBarDeleteMenu` returns `TRUE` if an error occurs or the specified menu is not found.

See Also `mdlDialog_menuBarDeleteAllMenus`, `mdlDialog_menuBarFindMenu`, `mdlDialog_menuBarFindItem`, `mdlDialog_menuBarGetItem`.

**mdlDialog_menuBarInsertMenu [ditemlb.ml] (obsolete),
mdlDialog_menuBarInsMenu [ditemlib.ml]**

```
#include <dlogitem.h>
boolean mdlDialog_menuBarInsertMenu /* <= TRUE if error */
(
    RawItemHdr    *mbarP,           /* => ptr to a menu bar */
    ULONG         menuType,         /* => type of menu to insert
    long          menuId,           /* => rsc ID of menu to insert */
    DItem_PulldownMenu *beforeMenuP /* => NULL means append */
);
DItem_PulldownMenu *mdlDialog_menuBarInsMenu /* <= NULL if error */
(
    RawItemHdr    *mbarP,           /* => ptr to a menu bar */
    ULONG         menuType,         /* => type of menu to insert
    long          menuId,           /* => rsc ID of menu to insert */
    DItem_PulldownMenu *beforeMenuP /* => NULL means append */
);
```

Description mdlDialog_menuBarInsMenu inserts the pull-down menu whose resource type is *menuType* and ID is *menuId* into the menu bar item specified by *mbarP*. This function provides the same functionality as the now obsolete mdlDialog_menuBarInsertMenu except that mdlDialog_menuBarInsMenu returns a pointer to the newly inserted pull-down menu item.

mbarP must point to a menu bar item. It cannot be NULL.

beforeMenuP indicates which pull-down menu to insert the new menu before. A pointer to a pull-down menu can be obtained by calling mdlDialog_menuBarFindMenu, mdlDialog_menuBarFindItem, mdlDialog_menuBarGetItem or mdlDialog_menuBarGetNItems. If *beforeMenuP* is NULL, the pull-down menu will be appended at the end of the menu bar item's list of pull-down menus.

Returns mdlDialog_menuBarInsMenu and mdlDialog_menuBarInsertMenu return NULL if an error occurs. This means that *mbarP* does not point to a menu bar item, the pull-down menu is not found, or *beforeMenuP* does not point to a pull-down menu. If successful, mdlDialog_menuBarInsMenu returns a pointer the inserted pull-down menu item.

See Also mdlDialog_menuBarFindMenu [ditemlb.ml], mdlDialog_menuBarFindItem [ditemlb.ml], mdlDialog_menuBarGetItem [ditemlb.ml], mdlDialog_menuBarGetNItems [ditemlb.ml].

mdlDialog_menuBarDeleteItem, mdlDialog_menuBarDeleteAllItems [ditemlb.ml]

```
#include <dlogitem.h>
boolean mdlDialog_menuBarDeleteItem /* <= TRUE if error */
(
  DItem_PulldownMenuItem *menuItemP          /* => menu item to delete */
);
boolean mdlDialog_menuBarDeleteAllItems /* <= TRUE if error */
(
  DItem_PulldownMenu      *menuP              /* => menu to remove from */
);
```

Description The `mdlDialog_menuBarDeleteItem` function deletes the pull-down menu item specified by *menuItemP*. A pointer to a menu item is obtained by calling `mdlDialog_menuBarFindItem [ditemlb.ml]` or `mdlDialog_menuBarGetItem [ditemlb.ml]`.

The `mdlDialog_menuBarDeleteAllItems` function deletes all the pull-down menu items from the pull-down menu specified by *menuP*. A pointer to a pull-down menu can be obtained by calling `mdlDialog_menuBarFindMenu [ditemlb.ml]`, `mdlDialog_menuBarFindItem [ditemlb.ml]`, `mdlDialog_menuBarGetItem [ditemlb.ml]` or `mdlDialog_menuBarGetNItems [ditemlb.ml]`.

Returns The `mdlDialog_menuBarDeleteItem` function returns `TRUE` if an error occurs. This means that *menuItemP* does not point to a pull-down menu item.

The `mdlDialog_menuBarDeleteAllItems` function returns `TRUE` if an error occurs. This means that *menuP* does not point to a pull-down menu.

See Also `mdlDialog_menuBarFindMenu [ditemlb.ml]`, `mdlDialog_menuBarFindItem [ditemlb.ml]`, `mdlDialog_menuBarGetItem [ditemlb.ml]`, `mdlDialog_menuBarGetNItems [ditemlb.ml]`.

mdlDialog_menuBarMenuSetTitle, mdlDialog_menuBarMenuSetTitle [ditemlb.ml]

```
#include <dlogitem.h>
boolean mdlDialog_menuBarMenuSetTitle /* <= TRUE if error */
(
  char                **titlePP,          /* <= ptr to title */
  DItem_PulldownMenu  *menuP              /* => menu to get title of */
);
void mdlDialog_menuBarMenuSetTitle /* <= TRUE if error */
(
  DItem_PulldownMenu  *menuP,             /* => menu to set title of */
  char                *titleP             /* => new title */
);
```

Description The `mdlDialog_menuBarMenuGetTitle` function retrieves a pointer to the title of the pull-down menu specified by *menuP*. The location pointed to by *titlePP* is set to point to the pull-down menu's title.

The `mdlDialog_menuBarMenuSetTitle` function sets the title of the pull-down menu specified by *menuP* to be the string pointed to by *titleP*.

A pointer to a pull-down menu can be obtained by calling

```
mdlDialog_menuBarFindMenu [ditemlb.ml],
mdlDialog_menuBarFindItem [ditemlb.ml], mdlDialog_menuBarGetItem
[ditemlb.ml] or mdlDialog_menuBarGetNItems [ditemlb.ml].
```

Returns The `mdlDialog_menuBarMenuGetTitle` function returns `TRUE` if an error occurs. This means that *menuP* does not point to a pull-down menu.

The `mdlDialog_menuBarMenuSetTitle` function is of type `void`. It returns no value.

See Also `mdlDialog_menuBarFindMenu [ditemlb.ml]`, `mdlDialog_menuBarFindItem [ditemlb.ml]`, `mdlDialog_menuBarGetItem [ditemlb.ml]`, `mdlDialog_menuBarGetNItems [ditemlb.ml]`.

mdlDialog_menuBarMenuGetEnabled, mdlDialog_menuBarMenuSetEnabled [ditemlb.ml]

```
#include <dlogitem.h>

boolean mdlDialog_menuBarMenuGetEnabled    /* <= TRUE if error */
(
    boolean          *enabledP,           /* <= enabled state */
    Ditem_PulldownMenu *menuP             /* => menu to get state of */
);

boolean mdlDialog_menuBarMenuSetEnabled    /* <= TRUE if error */
(
    Ditem_PulldownMenu *menuP,           /* => menu to set enab. state */
    boolean            enabled           /* => new enabled state */
);
```

Description The `mdlDialog_menuBarMenuGetEnabled` function retrieves the enabled state (enabled or disabled) of the pull-down menu specified by *menuP*. The location pointed to by *enableP* is set to the menu's enabled state.

`mdlDialog_menuBarMenuSetEnabled` sets the enabled state (enabled or disabled) of the pull-down menu specified by *menuP* to the value specified by *enabled*.

If *enabled* is `TRUE`, the pull-down menu title is drawn with normal bold text and the user can select items from the menu. If *enabled* is `FALSE`, the pull-down menu title is drawn with dimmed text and the user cannot select any items from the menu.



The pull-down menu is not redrawn when `mdlDialog_menuBarMenuSetEnabled` is called. Instead, it is redrawn either when the user causes an update to the pull-down menu entry (clicks on it) or the application calls `mdlDialog_itemDraw` for the menu bar item. It is recommended that the programmer call `mdlDialog_itemDraw` only after all necessary pull-downs have been enabled or disabled.

A pointer to a pull-down menu can be obtained by calling
`mdlDialog_menuBarFindMenu [ditemlb.ml],`
`mdlDialog_menuBarFindItem [ditemlb.ml], mdlDialog_menuBarGetItem`
`[ditemlb.ml]` or `mdlDialog_menuBarGetNItems [ditemlb.ml]`.

Returns The `mdlDialog_menuBarMenuGetEnabled` and `mdlDialog_menuBarMenuSetEnabled` functions return `TRUE` if an error occurs. This means that *menuP* does not point to a pull-down menu.

See Also `mdlDialog_menuBarFindMenu [ditemlb.ml], mdlDialog_menuBarFindItem`
`[ditemlb.ml], mdlDialog_menuBarGetItem [ditemlb.ml],`
`mdlDialog_menuBarGetNItems [ditemlb.ml].`

Text Pull-down Menu Functions

The text pull-down menu functions manipulate the contents of text pull-down menus. Text pull-down menu items can be added to a text pull-down menu. The text pull-down menu item's label, mark and state (enabled or disabled) can be changed using the functions in this section.

See the "Text Pull-down Menu" section of the "Standard Dialog Items" chapter for more information on using text pull-down menus.

The following table lists the text pull-down menu functions:

Function	Used to
<code>mdlDialog_textPDMItemSetLabel</code>	set a text pull-down menu item's label.
<code>mdlDialog_textPDMItemSetEnabled</code>	set a text pull-down menu item's enabled state (enabled or disabled).
<code>mdlDialog_textPDMItemSetMark</code>	set a text pull-down menu item's mark.
<code>mdlDialog_textPDMItemGetInfo</code>	get information about a text pull-down menu item.
<code>mdlDialog_textPDMItemSetInfo</code>	set information within a text pull-down menu item.

Function	Used to
mdlDialog_textPDMItemInsert mdlDialog_textPDMItemIns	insert a text pull-down menu item into a text pull-down menu.
mdlDialog_menuBarGetCmdWinP	get a pointer to the RawItemHdr of the main menu.

mdlDialog_textPDMItemSetLabel [ditemlib.ml], mdlDialog_textPDMItemSetEnabled [ditemlib.ml], mdlDialog_textPDMItemSetMark [ditemlib.ml]

```
#include <dlogitem.h>

boolean mdlDialog_textPDMItemSetLabel /* <= TRUE if error */
(
  Ditem_PulldownMenuItem *menuItemP,      /* => item to set */
  char *labelP                             /* => new label */
);

boolean mdlDialog_textPDMItemSetEnabled /* <= TRUE if error */
(
  Ditem_PulldownMenuItem *menuItemP,      /* => menu item to set */
  boolean enabled                         /* => TRUE if enabled */
);

boolean mdlDialog_textPDMItemSetMark /* <= TRUE if error */
(
  Ditem_PulldownMenuItem *menuItemP,      /* => item to set */
  int markType                           /* => MARK_TOGGLE_OUT... */
);
```

Description The mdlDialog_textPDMItemSetLabel function sets the label of the text pull-down menu item specified by *menuItemP* to the string pointed to by *labelP*.

The mdlDialog_textPDMItemSetEnabled function sets the enabled state (enabled or disabled) of the text pull-down menu item specified by *menuItemP* to the value specified by *enabled*. If *enabled* is TRUE, the text pull-down menu item is drawn with normal bold text and the user can select the item. If *enabled* is FALSE, the pull-down menu item is drawn with dimmed text and the user cannot select the item.

The mdlDialog_textPDMItemSetMark function sets the mark of the text pull-down menu item specified by *menuItemP* to the value specified by *markType*. *markType* can be one of the following constants:

MARK_TOGGLE_IN, MARK_TOGGLE_OUT, MARK_RADIO_IN, MARK_RADIO_OUT or MARK_RIGHT_ARROW.

A pointer to a text pull-down menu item can be obtained by calling mdlDialog_menuBarFindItem [ditemlib.ml] or mdlDialog_menuBarGetItem [ditemlib.ml].

These functions are usually called as part of an item hook function attached to a menu bar item. The `DITEM_MESSAGE_BUTTON` message should be trapped with `buttonTrans` equal to `BUTTONTRANS_DOWN`. This message is sent when the data button is pressed inside a menu bar item and before any pull-down menus are displayed. Trapping this message allows the state, label, or mark of all text pull-down menu items to be correctly set before the menus are displayed.

Returns The `mdlDialog_textPDMItemSetLabel`, `mdlDialog_textPDMItemSetEnabled` and `mdlDialog_textPDMItemSetMark` functions return `TRUE` if an error occurs. This means that *menuItemP* does not point to a text pull-down menu item.

See Also `mdlDialog_menuBarFindItem [ditemlb.ml]`, `mdlDialog_menuBarGetItem [ditemlb.ml]`.

mdlDialog_textPDMItemGetInfo [ditemlib.ml], mdlDialog_textPDMItemSetInfo [ditemlib.ml]

```
#include <dlogitem.h>

boolean mdlDialog_textPDMItemGetInfo /* <= TRUE if error */
(
    TextPDM_Item          *textPDMInfoP, /* <= text menu item info */
    Ditem_PulldownMenuItem *menuItemP    /* => item to get info */
);

boolean mdlDialog_textPDMItemSetInfo /* <= TRUE if error */
(
    Ditem_PulldownMenuItem *menuItemP,    /* => item to set info in */
    TextPDM_Item          *textPDMInfoP, /* => text menu item info */
    TextPDM_ItemModify    *textPDMModifyP /* => modify info */
);
```

Description The `mdlDialog_textPDMItemGetInfo` function gets information about the text pull-down menu item specified by *menuItemP* in the `TextPDM_Item` structure pointed to by *textPDMInfoP*.

The `mdlDialog_textPDMItemSetInfo` function sets information in the text pull-down menu item specified by *menuItemP* using the `TextPDM_Item` structure pointed to by *textPDMInfoP*, and the `TextPDM_ItemModify` variable pointed to by *textPDMModifyP*.

The `TextPDM_Item` structure (declared in `dlogitem.h`) contains fields analogous to those found in the `Ditem_PulldownMenuItemRsc` structure. See the “Text Pull-down Menu” section of the “Standard Dialog Items” chapter for more information on the `Ditem_PulldownMenuItemRsc` structure.

The `TextPDM_ItemModify` structure (also declared in `dlogitem.h`) contains one bitfield for each field of a `TextPDM_Item` structure. Setting a bitfield in the `TextPDM_ItemModify` structure to `TRUE` indicates the corresponding

field in the `TextPDM_Item` should be used when setting information in a text pull-down menu item with the `mdlDialog_textPDMItemSetInfo` function. If a bitfield is `FALSE`, the corresponding field in the text pull-down menu item is unchanged.

A pointer to a text pull-down menu item can be obtained by calling `mdlDialog_menuBarFindItem [ditemlib.ml]` or `mdlDialog_menuBarGetItem [ditemlib.ml]`.

Returns The `mdlDialog_textPDMItemGetInfo` and `mdlDialog_textPDMItemSetInfo` functions return `TRUE` if an error occurs. This means that *menuItemP* does not point to a text pull-down menu item.

See Also `mdlDialog_menuBarFindItem [ditemlib.ml]`, `mdlDialog_menuBarGetItem [ditemlib.ml]`.

mdlDialog_textPDMItemInsert [ditemlib.ml], mdlDialog_textPDMItemIns [ditemlib.ml]

```
#include <dlogitem.h>

boolean mdlDialog_textPDMItemInsert /* <= TRUE if error */
(
    DItem_PulldownMenu      *menuP,           /* => menu to insert into */
    DItem_PulldownMenuItem *menuItemP,       /* => item to ins before */
    TextPDM_Item            *textPDMInfoP,    /* => text menu item info */
    TextPDM_ItemModify      *textPDMModifyP   /* => modify info */
);

boolean mdlDialog_textPDMItemIns /* <= TRUE if error */
(
    DItem_PulldownMenuItem *newMenuItemP,     /* <= new menu item */
    DItem_PulldownMenu      *menuP,           /* => menu to insert into */
    DItem_PulldownMenuItem *menuItemP,       /* => item to ins before */
    TextPDM_Item            *textPDMInfoP,    /* => text menu item info */
    TextPDM_ItemModify      *textPDMModifyP   /* => modify info */
);
```

Description `mdlDialog_textPDMItemInsert` and `mdlDialog_textPDMItemIns` insert the text pull-down menu item specified by *textPDMInfoP* and the `TextPDM_ItemModify` variable pointed to by *textPDMModifyP* into a text pull-down menu.

newMenuItemP in `mdlDialog_textPDMItemIns` (which makes `mdlDialog_textPDMItemInsert` obsolete) specifies a non-NULL pointer to a memory area allocated by the application program into which the newly created `PulldownMenuItem` information can be copied upon successful insertion of the item into the menu. This information is useful if further operations are to be performed on the newly created item.

menuItemP specifies the text pull-down menu item before which the new menu item is to be inserted. If *menuItemP* is `NULL`, the menu item is

appended to the end of the menu specified by *menuP*, which must not also be NULL. If *menuItemP* is not NULL, *menuP* can be set to NULL, since the text pull-down menu to insert into can be determined from a valid pointer to a text pull-down menu item.

The `TextPDM_Item` structure (declared in `dlogitem.h`) contains fields analogous to those found in the `Ditem_PulldownMenuItemRsc` structure. See the “Text Pull-down Menu” section of the “Standard Dialog Items” chapter for more information on the `Ditem_PulldownMenuItemRsc` structure.

The `TextPDM_ItemModify` structure (also declared in `dlogitem.h`) contains one bitfield for each field of a `TextPDM_Item` structure. Setting a bitfield in the `TextPDM_ItemModify` structure to TRUE indicates the corresponding field in the `TextPDM_Item` should be used when creating the new text pull-down menu item. If a bitfield is FALSE, an appropriate default value will be used for the corresponding field.

A pointer to a text pull-down menu item can be obtained by calling `mdlDialog_menuBarFindItem [ditemlb.ml]` or `mdlDialog_menuBarGetItem [ditemlb.ml]`.

Returns The `mdlDialog_textPDMItemInsert` function returns TRUE if an error occurs.

Related Strcuts

```
typedef struct textpdm_item /* text pulldown menu item info */
{
    Ditem_PulldownMenu submenu; /* Resource info for sub-menu */
    byte    enabled;           /* True if item enabled */
    byte    mark;              /* Item mark value */
    ULONG    helpInfo;         /* help info */
    ULONG    helpType;
    char    *helpTaskIdP;
    long    pulldownItemHookId; /* Item hook data */
    long    pulldownSearchId;
    ULONG    commandNumber;     /* Command assoc. with item */
    char    *commandTaskIdP;
    char    *unparsedP;
    char    *labelP;            /* Item text label */
    ULONG    accelerator;
    int      mnemonic;
    int      mneIndex;
} TextPDM_Item;

typedef struct textpdm_itemmodify
{
    /* Set each bit for corresponding item */
    ULONG    submenu:1;        /* item in TextPDM_Item struct inittd */
    ULONG    enabled:1;
```



```

    ULONG    mark:1;
    ULONG    helpInfo:1;
    ULONG    helpType:1;
    ULONG    helpTaskIdP:1;
    ULONG    pulldownItemHookId:1;
    ULONG    pulldownSearchId:1;
    ULONG    commandNumber:1;
    ULONG    commandTaskIdP:1;
    ULONG    unparsedP:1;
    ULONG    labelP:1;
    ULONG    acceleratorP:1;
    ULONG    mnemonic:1;
    ULONG    mneIndex:1;
} TextPDM_ItemModify;

```

See Also mdlDialog_menuBarFindItem [ditemlb.ml], mdlDialog_menuBarGetItem [ditemlb.ml].

mdlDialog_menuBarGetCmdWinP

```

#include <ditemlib.fdf>

RawItemHdr *mdlDialog_menuBarGetCmdWinP /* <= Cmd win menu bar */
(
    /* <= SUCCESS or ERROR */
    void
);

```

Description mdlDialog_menuBarGetCmdWinP is used to get a pointer to the raw item header of the main menu bar.



This function was implemented in MicroStation 95.

Returns mdlDialog_menuBarGetCmdWinP returns NULL on ERROR.

Option Pull-down Menu Functions

Option menu functions manipulate the contents of option menus. Items can be added to an option menu. The option menu item's state (enabled or disabled) or any other information can be changed using the functions in this section.

See the "Option Pull-down Menu" section of the "Standard Dialog Items" chapter for more information on using option menus.

The following table lists the option menu functions:

Function	Used to
<code>mdlDialog_optionPDMItemSetEnabled [ditemlb.ml]</code>	set the enabled state (enabled or disabled) of an option menu item.
<code>mdlDialog_optionPDMItemGetInfo</code>	get information about an option menu item.
<code>mdlDialog_optionPDMItemSetInfo</code>	set information about an option menu item.
<code>mdlDialog_optionPDMItemInsert [ditemlb.ml]</code>	insert a new item into an option menu.

mdlDialog_optionPDMItemSetEnabled [ditemlb.ml]

```
#include <dlogitem.h>

boolean mdlDialog_optionPDMItemSetEnabled /* <= TRUE if error */
(
    DItem_PulldownMenuItem *menuItemP,    /* => menu item to set */
    boolean                  enabled        /* => TRUE if enabled */
);
```

Description The `mdlDialog_optionPDMItemSetEnabled` function sets the enabled state (enabled or disabled) of the option pull-down menu item specified by *menuItemP* to the value specified by *enabled*. If *enabled* is `TRUE`, the option pull-down menu item is drawn with normal bold text (if the menu is not using icons) and the user can select the item. If *enabled* is `FALSE`, the pull-down menu item is drawn with dimmed text and the user cannot select the item.

A pointer to an option pull-down menu item can be obtained by calling `mdlDialog_menuBarFindItem [ditemlb.ml]` or `mdlDialog_menuBarGetItem [ditemlb.ml]`.



This function is usually called as part of an item hook function attached to a menu bar item. The `DITEM_MESSAGE_BUTTON` message should be trapped with `buttonTrans` equal to `BUTTONTRANS_DOWN`. This message is sent when the data button is pressed inside a menu bar item and before any pull-down menus are displayed. Trapping this message allows the state of all option pull-down menu items to be correctly set before the menus are displayed.

Returns The `mdlDialog_optionPDMItemSetEnabled` function returns `TRUE` if an error occurs. This means that *menuItemP* does not point to an option pull-down menu item.

See Also `mdlDialog_menuBarFindItem [ditemlb.ml]`, `mdlDialog_menuBarGetItem [ditemlb.ml]`.

mdlDialog_optionPDMItemGetInfo, mdlDialog_optionPDMItemSetInfo [ditemlb.ml]

```
#include <dlogitem.h>
boolean mdlDialog_optionPDMItemGetInfo /* <= TRUE if error */
(
char    *labelP,           /* <= NULL = don't want label */
ULong   *iconTypeP,       /* <= NULL = don't want type */
long    *iconIdP,         /* <= NULL = don't want ID */
int     *commandSourceP,  /* <= NULL = don't want cmd source */
ULong   *valueP,          /* <= NULL = don't want value */
ULong   *maskP,           /* <= NULL = don't want mask */
int     *enabledP,        /* <= NULL = don't want enab. state */
void    **userDataPP,     /* <= NULL = don't want userDataP */
DItem_PulldownMenuItem *menuItemP /* => item to get info on */
);

boolean mdlDialog_optionPDMItemSetInfo /* <= TRUE if error */
(
char    *labelP,           /* => NULL = not setting label */
ULong   *iconTypeP,       /* => NULL = not setting type */
long    *iconIdP,         /* => NULL = not setting ID */
int     *commandSourceP,  /* => NULL = not setting cmd source */
ULong   *valueP,          /* => NULL = not setting value */
ULong   *maskP,           /* => NULL = not setting mask */
int     *enabledP,        /* => NULL = not setting enab. state */
void    **userDataPP,     /* => NULL = not setting userDataP */
DItem_PulldownMenuItem *menuItemP /* => item to set */
);
```

Description The mdlDialog_optionPDMItemGetInfo function gets information about the option pull-down menu item specified by *menuItemP*.

The mdlDialog_optionPDMItemSetInfo function sets information in the option pull-down menu item specified by *menuItemP*.

The parameters other than *menuItemP* and *userDataPP* are analogous to those found in the DItem_PulldownOptionItemRsc structure. See the “Option Pull-down Menu” and “Option Button Item” sections of the “Standard Dialog Items” chapter for more information on the DItem_PulldownOptionItemRsc structure.

User defined data can be attached to any option pull-down menu item. *userDataPP* is used to attach or retrieve a pointer to the user defined data area.

When using mdlDialog_optionPDMItemGetInfo the caller can specify NULL for any parameter (other than *menuItemP*), which indicates that the corresponding value is not required.

When using `mdlDialog_optionPDMItemSetInfo` the caller can specify `NULL` for any parameter (other than *menuItemP*), which indicates that the corresponding value is not being set and should remain unchanged.

A pointer to an option pull-down menu item can be obtained by calling `mdlDialog_menuBarFindItem` or `mdlDialog_menuBarGetItem`.

Returns `mdlDialog_optionPDMItemGetInfo` and `mdlDialog_optionPDMItemSetInfo` return `TRUE` if an error occurs. This means that *menuItemP* does not point to an option pull-down menu item.

See Also `mdlDialog_menuBarFindItem [ditemlb.ml]`, `mdlDialog_menuBarGetItem [ditemlb.ml]`.

mdlDialog_optionPDMItemInsert [ditemlb.ml]

```
#include <dlogitem.h>

boolean mdlDialog_optionPDMItemInsert /* <= TRUE if error */
(
    char          *labelP,           /* => NULL = no label */
    ULONG         *iconTypeP,        /* => NULL = no icon */
    long          *iconIdP,          /* => NULL = no icon */
    int           *commandSourceP,    /* => NULL = no cmd source */
    ULONG         *valueP,           /* => NULL = value of 0 */
    ULONG         *maskP,            /* => NULL = NOMASK */
    int           *enabledP,          /* => NULL = enabled */
    void          *userDataP,         /* => NULL = no userData */
    DItem_PulldownMenu *menuP,       /* => menu to insert into */
    DItem_PulldownMenuItem *menuItemP /* => mitem to insert bef, NULL=apnd */
);
```

Description The `mdlDialog_optionPDMItemInsert` function inserts an option pull-down menu item into an option pull-down menu.

The *menuItemP* parameter specifies the option pull-down menu item before which the new menu item is to be inserted. If *menuItemP* is `NULL`, the menu item is appended to the end of the menu specified by *menuP*, which must not also be `NULL`. If *menuItemP* is not `NULL`, *menuP* can be set to `NULL`, since the option pull-down menu to insert into can be determined from a valid pointer to an option pull-down menu item.

User defined data can be attached to any option pull-down menu item. The *userDataP* parameter is used to attach a pointer to user defined data to an option pull-down menu item.

The parameters other than *menuItemP*, *menuP*, and *userDataP* point to variables that are analogous to those found in the `DItem_PulldownOptionItemRsc` structure. See the “Option Pull-down Menu” and “Option Button Item” sections of the “Standard Dialog Items” chapter for more information on the `DItem_PulldownOptionItemRsc`

structure. Specifying `NULL` for a parameter either means that the parameter does not apply or a default value should be used. See the summary argument comments for more information.

A pointer to an option pull-down menu item can be obtained by calling `mdlDialog_menuBarFindItem [ditemlb.ml]` or `mdlDialog_menuBarGetItem [ditemlb.ml]`.

Returns The `mdlDialog_optionPDMItemInsert` function returns `TRUE` if an error occurs.

See Also `mdlDialog_menuBarFindItem [ditemlb.ml]`, `mdlDialog_menuBarGetItem [ditemlb.ml]`.

Option Button Item Functions

The option button item functions manipulate the contents of option button items. Option button sub-items can be added to an option button item. The option button sub-item's state (enabled or disabled) or any other information can be changed using the function's in this section.

See the "Option Button Item" section of the "Standard Dialog Items" chapter for more information on using option button items.

The following table lists the option button item functions:

Function	Used to
<code>mdlDialog_optionButtonGetNItems</code>	get the number of sub-items contained within an option button item.
<code>mdlDialog_optionButtonSetEnabled</code>	set the enabled state (enabled or disabled) of an option button sub-item.
<code>mdlDialog_optionButtonSetExtent</code>	allow an option button to be resized by causing the dialog manager to recompute the extents of the option button item.
<code>mdlDialog_optionButtonGetItemInfo</code>	get information about an option button item.
<code>mdlDialog_optionButtonSetItemInfo</code>	set information in an option button item.
<code>mdlDialog_optionButtonGetSubInfo</code>	get information about an option button sub-item.
<code>mdlDialog_optionButtonSetSubInfo</code>	set information in an option button sub-item.
<code>mdlDialog_optionButtonInsertItem</code>	insert a sub-item into an option button item.
<code>mdlDialog_optionButtonInsSubItem</code>	add a selection to an option button.

Function	Used to
<code>mdlDialog_optionButtonDeleteItem</code>	delete a sub-item from an option button item.
<code>mdlDialog_optionButtonDeleteAll</code>	delete all the sub-items from an option button item.

mdlDialog_optionButtonGetNItems

```
#include <dlogitem.h>

int mdlDialog_optionButtonGetNItems /* <= # of option btn items */
(
    RawItemHdr *oButP /* => option button */
);
```

Description The `mdlDialog_optionButtonGetNItems` function returns the number of sub-items contained in the option button item specified by *oButP*.

Returns `mdlDialog_optionButtonGetNItems` returns the number of sub-items in the specified option button item. It returns -1 if *oButP* does not point to an option button item.

See Also `mdlDialog_optionButtonSetEnabled`.

mdlDialog_optionButtonSetEnabled

```
#include <dlogitem.h>

boolean mdlDialog_optionButtonSetEnabled /* <= TRUE if error */
(
    RawItemHdr *oButP, /* => option button to set */
    int subItemIndex, /* => index of subitem to set */
    boolean enabled /* => TRUE if enabled */
);
```

Description The `mdlDialog_optionButtonSetEnabled` function sets the enabled state (enabled or disabled) of a sub-item of the option button item specified by *oButP* to the value specified by *enabled*.

subItemIndex indicates which sub-item's enabled state to set. It must be greater than or equal to zero and less than the number of sub-items in the option button. The `mdlDialog_optionButtonGetNItems` function can be used to get the number of sub-items contained by an option button item.

If *enabled* is `TRUE`, the option button sub-item is drawn with normal bold text (if the option button is not using icons) and the user can select the sub-item. If *enabled* is `FALSE`, the option button sub-item is drawn with dimmed text and the user cannot select the sub-item.

Returns The `mdlDialog_optionButtonSetEnabled` function returns `TRUE` if an error occurs. This means that *oButP* does not point to an option button item.

See Also mdlDialog_optionButtonGetNItems.

mdlDialog_optionButtonSetExtent

```
boolean mdlDialog_optionButtonSetExtent    /* <= TRUE if error */
(
    RawItemHdr  *oButP    /* => option button to set */
);
```

Description The mdlDialog_optionButtonSetExtent function causes the Dialog Manager to recompute the extents of the option button item pointed to by *oButP*. This function allows the option button to be resized by MicroStation after the labels of the option button subitems have been changed.

oButP is a pointer to the option button's raw item header.

Returns mdlDialog_optionButtonSetExtent returns TRUE upon successful resizing of the item and FALSE if an error occurred.

See Also mdlDialog_optionButtonSetSubInfo, mdlDialog_optionButtonSetItemInfo, mdlDialog_optionButtonInsSubItem, mdlDialog_optionButtonInsertItem.

mdlDialog_optionButtonGetItemInfo, mdlDialog_optionButtonSetItemInfo (obsolete)

```
#include <dlogitem.h>

boolean mdlDialog_optionButtonGetItemInfo /* <= TRUE if error */
(
    char    *labelP,          /* <= set NULL for no label */
    ULong   *iconTypeP,       /* <= set NULL for no type */
    long    *iconIdP,         /* <= set NULL for no ID */
    int     *commandSourceP,  /* <= set NULL for no cmd source */
    ULong   *valueP,          /* <= set NULL for no value */
    ULong   *maskP,           /* <= set NULL for no mask */
    int     *enabledP,        /* <= set NULL for no enabled state */
    void    **userDataPP,     /* <= set NULL for no userData */
    RawItemHdr *oButP,        /* => option button */
    int     subItemIndex      /* => subitem to get info on */
);

boolean mdlDialog_optionButtonSetItemInfo /* <= TRUE if error */
(
```

```

char    *labelP,           /* => NULL = not setting label */
ULong   *iconTypeP,        /* => NULL = not setting type */
long    *iconIdP,          /* => NULL = not setting ID */
int     *commandSourceP,   /* => NULL = not setting cmd source */
ULong   *valueP,           /* => NULL = not setting value */
ULong   *maskP,            /* => NULL = not setting mask */
int     *enabledP,         /* => NULL = not setting enabled state */
void    **userDataPP,      /* => NULL = not setting userDataP */
RawItemHdr *oButP,         /* => option button */
int     subItemIndex       /* => subitem to set info on */
);

```

Description `mdlDialog_optionButtonGetItemInfo` gets information about an option button sub-item, and `mdlDialog_optionButtonSetItemInfo` sets information in an option button sub-item. These functions are basically obsolete; you should usually use `mdlDialog_optionButtonGetSubInfo` and `mdlDialog_optionButtonSetSubInfo` instead.

The *oButP* parameter specifies a particular option button item.

The *subItemIndex* parameter indicates which sub-item to get information about or set information in. It must be greater than or equal to zero, and less than the number of sub-items in the option button. The `mdlDialog_optionButtonGetNItems` function can be used to get the number of sub-items contained by an option button item.

User defined data can be attached to any option button sub-item. The *userDataPP* parameter is used to attach or retrieve a pointer to the user defined data area.

The parameters other than *oButP*, *subItemIndex*, and *userDataPP* are analogous to those found in the `DItem_OptionButtonItemRsc` structure. See the “Option Button Item” section of the “Standard Dialog Items” chapter for more information on the `DItem_OptionButtonItemRsc` structure.

When using `mdlDialog_optionButtonGetItemInfo` the caller can specify `NULL` for any parameter (other than *oButP* and *subItemIndex*), which indicates that the corresponding value is not required.

When using `mdlDialog_optionButtonSetItemInfo` the caller can specify `NULL` for any parameter (other than *oButP* and *subItemIndex*), which indicates that the corresponding value is not being set and should remain unchanged.

Returns The `mdlDialog_optionButtonGetItemInfo` and `mdlDialog_optionButtonSetItemInfo` functions return `TRUE` if an error occurs. This means that *oButP* does not point to an option button item.

See Also `mdlDialog_optionButtonGetNItems`, `mdlDialog_optionButtonGetSubInfo`, `mdlDialog_optionButtonSetSubInfo`.

mdlDialog_optionButtonGetSubInfo, mdlDialog_optionButtonSetSubInfo

```

#include <dlogitem.h>

boolean mdlDialog_optionButtonGetSubInfo /* <= TRUE if error */
(
    char    *labelP,           /* <= set NULL for no label */
    ULong   *iconTypeP,        /* <= set NULL for no type */
    long    *iconIdP,          /* <= set NULL for no ID */
    int     *commandNumberP,   /* <= set NULL for no cmd number */
    int     *commandSourceP,   /* <= set NULL for no cmd source */
    ULong   *valueP,           /* <= set NULL for no value */
    ULong   *maskP,            /* <= set NULL for no mask */
    int     *enabledP,         /* <= set NULL for no enabled state */
    void    **userDataPP,      /* <= set NULL for no userDataP */
    RawItemHdr *oButP,         /* => option button */
    int     subItemIndex       /* => subitem to get info on */
);

boolean mdlDialog_optionButtonSetSubInfo /* <= TRUE if error */
(
    char    *labelP,           /* => NULL = not setting label */
    ULong   *iconTypeP,        /* => NULL = not setting type */
    long    *iconIdP,          /* => NULL = not setting ID */
    int     *commandNumberP,   /* => NULL = not setting cmd number */
    int     *commandSourceP,   /* => NULL = not setting cmd source */
    ULong   *valueP,           /* => NULL = not setting value */
    ULong   *maskP,            /* => NULL = not setting mask */
    int     *enabledP,         /* => NULL = not setting enabled state */
    void    **userDataPP,      /* => NULL = not setting userDataP */
    RawItemHdr *oButP,         /* => option button */
    int     subItemIndex       /* => subitem to set info on */
);

```

Description The `mdlDialog_optionButtonGetSubInfo` function gets information about an option button sub-item.

The `mdlDialog_optionButtonSetSubInfo` function sets information in an option button sub-item.

The *oButP* parameter specifies a particular option button item.

The *subItemIndex* parameter indicates which sub-item to get information about or set information in. It must be greater than or equal to zero, and less than the number of sub-items in the option button. The `mdlDialog_optionButtonGetNItems` function can be used to get the number of sub-items contained by an option button item.

User defined data can be attached to any option button sub-item. The *userDataPP* parameter is used to attach or retrieve a pointer to the user defined data area.

The parameters other than *oButP*, *subItemIndex*, and *userDataPP* are analogous to those found in the `DItem_OptionButtonItemRsc` structure. See the “Option Button Item” section of the “Standard Dialog Items” chapter for more information on the `DItem_OptionButtonItemRsc` structure.

When using `mdlDialog_optionButtonGetSubInfo` the caller can specify `NULL` for any parameter (other than *oButP* and *subItemIndex*), which indicates that the corresponding value is not required.

When using `mdlDialog_optionButtonSetSubInfo` the caller can specify `NULL` for any parameter (other than *oButP* and *subItemIndex*), which indicates that the corresponding value is not being set and should remain unchanged.



These functions replace `mdlDialog_optionButtonGetItemInfo` and `mdlDialog_optionButtonSetItemInfo`. The functionality is identical except for the addition of the `commandNumberP` field and better error-checking.

Returns The `mdlDialog_optionButtonGetSubInfo` and `mdlDialog_optionButtonSetSubInfo` functions return `TRUE` if an error occurs. This means that *oButP* does not point to an option button item.

See Also `mdlDialog_optionButtonGetNItems`.

`mdlDialog_optionButtonInsertItem` (obsolete), `mdlDialog_optionButtonInsSubItem`

```
#include <dlogitem.h>

boolean mdlDialog_optionButtonInsertItem /* <= TRUE if error */
(
    char    *labelP,           /* => NULL = no label */
    ULong   *iconTypeP,        /* => NULL = no icon */
    long     *iconIdP,         /* => NULL = no icon */
    int      *commandSourceP,  /* => NULL = not setting cmd source */
    ULong    *valueP,          /* => NULL = value of 0 */
    ULong    *maskP,           /* => NULL = NOMASK */
    int      *enabledP,        /* => NULL = enabled */
    void     *userDataP,       /* => NULL = no userData */
    RawItemHdr *oButP,         /* => option button */
    int      subItemIndex      /* => insert BEFORE this item, -1=append */
);
```

```

boolean mdlDialog_optionButtonInsSubItem  /* <= TRUE if error */
(
char    *labelP,                          /* => NULL = no label */
ULong   *iconTypeP,                       /* => NULL = no icon */
long    *iconIdP,                         /* => NULL = no icon */
ULong   *commandNumberP,                  /* => NULL = not setting cmd number */
int     *commandSourceP,                  /* => NULL = not setting cmd source */
ULong   *valueP,                          /* => NULL = value of 0 */
ULong   *maskP,                           /* => NULL = NOMASK */
int     *enabledP,                        /* => NULL = enabled */
void    *userDataP,                       /* => NULL = no userData */
RawItemHdr *oButP,                        /* => option button */
int     subItemIndex                      /* => insert BEFORE this item, -1=append */
);

```

Description `mdlDialog_optionButtonInsertItem` and `mdlDialog_optionButtonInsSubItem` insert an option button sub-item into the option button item specified by *oButP*. `mdlDialog_optionButtonInsSubItem` differs from the earlier `mdlDialog_optionButtonInsertItem` only in that it also allows the command number to be set.

The *subItemIndex* parameter indicates the sub-item before which the new sub-item is inserted. If *subItemIndex* is -1, the new sub-item will be appended after the last the option button sub-item. The `mdlDialog_optionButtonGetNItems` function can be used to get the number of sub-items contained by an option button item.

User defined data can be attached to any option button sub-item. *userDataP* is used to attach a pointer to user defined data to an option button sub-item.

The parameters other than *oButP*, *subItemIndex*, and *userDataP* are analogous to those found in the `DItem_OptionButtonItemRsc` structure. See the “Option Button Item” section of the “Standard Dialog Items” chapter for more information on the `DItem_OptionButtonItemRsc` structure. Specifying `NULL` for a parameter either means that the parameter does not apply or a default value should be used. See the summary argument comments for more information.

Returns `mdlDialog_optionButtonInsertItem` and `mdlDialog_optionButtonInsSubItem` return `SUCCESS` or a non-zero value to indicate an error.

See Also `mdlDialog_optionButtonGetNItems`.

mdlDialog_optionButtonDeleteItem, mdlDialog_optionButtonDeleteAll

```
#include <dlogitem.h>
boolean mdlDialog_optionButtonDeleteItem /* <= TRUE if error */
(
  RawItemHdr *oButP,          /* => option button */
  int         subItemIndex    /* => index of subitem to delete */
);
void mdlDialog_optionButtonDeleteAll
(
  RawItemHdr *oButP          /* => option button */
);
```

Description The `mdlDialog_optionButtonDeleteItem` function deletes an option button sub-item from the option button specified by *oButP*.

The *subItemIndex* parameter specifies the sub-item to delete. It must be greater than or equal to zero and less than the number of sub-items in the option button. The `mdlDialog_optionButtonGetNItems` function can be used to get the number of sub-items contained by an option button item.

The `mdlDialog_optionButtonDeleteAll` function deletes all the sub-items from the option button specified by *oButP*.

Returns `mdlDialog_optionButtonDeleteItem` returns `TRUE` if an error occurs. This means that *oButP* does not point to an option button item, or *subItemIndex* is not in range.

The `mdlDialog_optionButtonDeleteAll` function is of type `void`. It returns no value.

See Also `mdlDialog_optionButtonGetNItems`.

Icon and Icon Frame Functions

The functions in this section are used to manipulate icon frames, icon palettes, and the icons in them. The following table lists icon and icon frame functions:

Function	Used to
<code>mdlDialog_icFrameGetItemInfo</code>	get info on item in an icon command frame.
<code>mdlDialog_icFrameSetItemInfo</code>	set info on an item in an icon command frame.
<code>mdlDialog_icFrameGetNItems</code>	get number of items in an icon command frame.
<code>mdlDialog_icFrameGetNSubItems</code>	get number of sub-items of an icon palette in an icon command frame.
<code>mdlDialog_icFrameSelectIcon</code>	set the selection state of an icon in an icon command frame.
<code>mdlDialog_selectIconsById</code>	set selection state of an icon given its ID and owner.
<code>mdlDialog_selectIconsByIdNoMsg</code>	set selection state of an icon given its ID and owner and do not display the command message.
<code>mdlDialog_selectIconsByCmd</code> <code>mdlDialog_selectIconsByCmdNoMsg</code>	set selection state of an icon given command number and owner.
<code>mdlDialog_icPaletteSelectIcon</code>	change the selection status of an icon in an icon command palette.
<code>mdlDialog_icPaletteSetItemInfo</code>	set the icon command resource of an icon in an icon command palette.
<code>mdlDialog_icPaletteGetItemInfo</code>	get the icon command resource ID of an icon in an icon command palette.
<code>mdlDialog_icPaletteGetNItems</code>	query the number of icons contained in an icon command palette.

mdlDialog_icFrameGetItemInfo, mdlDialog_icFrameSetItemInfo

```
boolean mdlDialog_icFrameSetItemInfo /* <= TRUE if error */
(
  RawItemHdr *icFrameP,      /* => ptr to icon cmd frame item */
  RscFileHandle rFileH,      /* => NULL=use opened rsrc files */
  int subItemIndex,          /* => index of subitem to set */
  ULONG subItemType,         /* => subitem's new type */
  long subItemId             /* => subitem's new id */
);
boolean mdlDialog_icFrameGetItemInfo /* <= TRUE if error */
(
  ULONG *subItemTypeP, /* <= set NULL if don't want type */
  long *subItemIdP,    /* <= set NULL if don't want id */
  RawItemHdr *icFrameP, /* => ptr to icon cmd frame item */
  int subItemIndex /* => subitem to get info on */
);
```

Description mdlDialog_icFrameSetItemInfo and mdlDialog_icFrameGetItemInfo set and return information on an item in an icon command frame.

icFrameP points to the raw item header of the icon command frame containing the item to get or set information about. *subItemIndex* indicates which item.

rFileH points to the resource file to use, and is usually NULL indicating any open resource file.

subItemTypeP and *subItemType* indicate the type of the item.

subItemIdP and *subItemId* indicate the ID of the item.

Returns mdlDialog_icFrameGetItemInfo and mdlDialog_icFrameSetItemInfo return SUCCESS, or a non-zero value on error.

See Also mdlDialog_icFrameGetNItems, mdlDialog_icFrameGetNSubItems.

mdlDialog_icFrameGetNItems

```
boolean mdlDialog_icFrameGetNItems /* <= -1 if error */
(
  RawItemHdr *icFrameP /* => ptr to icon cmd frame item */
);
```

Description mdlDialog_icFrameGetNItems gets the number of items present in an icon command frame.

icFrameP points to the raw item header of the icon command frame in question.

Returns mdlDialog_icFrameGetNItems returns -1 on error.

See Also mdlDialog_icFrameGetNSubItems.

mdlDialog_icFrameGetNSubItems

```
int mdlDialog_icFrameGetNSubItems /* <= -1 if error; 4.0.3 */
(
    RawItemHdr *icFrameP,      /* => icon cmd frame item */
    int        subItemIndex
);
```

Description mdlDialog_icFrameGetNSubItems gets the number of sub-items of an icon palette in an icon command frame.

icFrameP points to the raw item header of the icon command frame in question.

subItemIndex is the index of the icon palette whose sub-items should be counted.

Returns mdlDialog_icFrameGetNSubItems returns -1 on error.

See Also mdlDialog_icFrameGetNItems.

mdlDialog_icFrameSelectIcon

```
boolean mdlDialog_icFrameSelectIcon /* <= TRUE if error; 4.0.3 */
(
    RawItemHdr *icFrameP,      /* => ptr to icon cmd frame item */
    int        iconIndex,      /* => index of icon to select */
    int        subIconIndex,   /* => index of icon in subpalette */
    int        selectType,     /* => ICONCMD_SELECTTYPE_xxx */
    boolean    deselectOthers, /* => normally TRUE */
    boolean    updateToolSettings /* => TRUE = also set TS dialog box */
);
```

Description mdlDialog_icFrameSelectIcon is used to set the selection state of an icon in an icon command frame.

icFrameP points to the raw item header of the icon command frame, and *iconIndex* and *subItemIndex* indicate which icon to select.

selectType gives the desired selection state of the icon. It may be ICONCMD_SELECTTYPE_OFF, ICONCMD_SELECTTYPE_SINGLESLOT or ICONCMD_SELECTTYPE_LOCKED.

deselectOthers indicates whether all other icons should be set to ICONCMD_SELECTTYPE_OFF, and should usually be TRUE.

updateToolSettings indicates whether the Tool Settings dialog should be updated to show the settings associated with the icon whose selection state is being updated.

Returns mdlDialog_icFrameSelectIcon returns SUCCESS, or a non-zero value on error.

See Also mdlDialog_icPaletteSelectIcon.

mdlDialog_selectIconsById, mdlDialog_selectIconsByIdNoMsg

```
void mdlDialog_selectIconsById
(
    long    iconId,           /* => id of icon to select */
    void    *ownerMD,        /* => NULL or ownerMD of icon */
    int     selectType,       /* => ICONCMD_SELECTTYPE_xxx */
    boolean deselectOthers,   /* => should be TRUE */
    boolean updateToolSettings /* => TRUE=update TS dialog box */
);

void mdlDialog_selectIconsByIdNoMsg
(
    long    iconId,           /* => id of icon to select */
    void    *ownerMD,        /* => NULL or ownerMD of icon */
    int     selectType,       /* => ICONCMD_SELECTTYPE_xxx */
    boolean deselectOthers,   /* => should be TRUE */
    boolean updateToolSettings /* => TRUE=update TS dialog box */
);
```

Description *mdlDialog_selectIconsById* and *mdlDialog_selectIconsByIdNoMsg* are used to set the selection state of an icon given its ID and a pointer to the MDL descriptor of its owner. *mdlDialog_selectIconsByIdNoMsg* does not display the command message in the Command Window.

iconId is the ID of the icon whose selection state is to be modified.

ownerMD points to the MDL descriptor of the icon's owner.

selectType gives the desired selection state of the icon. It may be `ICONCMD_SELECTTYPE_OFF`, `ICONCMD_SELECTTYPE_SINGLESHOT` or `ICONCMD_SELECTTYPE_LOCKED`.

deselectOthers indicates whether all other icons should be set to `ICONCMD_SELECTTYPE_OFF`, and should usually be `TRUE`.

updateToolSettings indicates whether the Tool Settings dialog should be updated to show the settings associated with the icon whose selection state is being updated.

Returns *mdlDialog_selectIconsById* and *mdlDialog_selectIconsByIdNoMsg* are of type `void`.

See Also *mdlDialog_selectIconsByCmd*, *mdlDialog_selectIconsByCmdNoMsg*.

mdlDialog_selectIconsByCmd, mdlDialog_selectIconsByCmdNoMsg

```
void mdlDialog_selectIconsByCmd
(
DialogBox    *initiatingDbP, /* => NULL or initiating dialog */
ULong        commandNumber, /* => 0 = use stateData.commandNumber */
char         *taskId,        /* => NULL = use stateData.procNumber */
int          selectType      /* => NORMALLY -1, = use last selctType */
);

void mdlDialog_selectIconsByCmdNoMsg
(
DialogBox    *initiatingDbP, /* => NULL or initiating dialog */
ULong        commandNumber, /* => 0 = use stateData.commandNumber */
char         *taskId,        /* => NULL = use stateData.procNumber */
int          selectType      /* => NORMALLY -1, = use last selctType */
);
```

Description *mdlDialog_selectIconsByCmd* and *mdlDialog_selectIconsByCmdNoMsg* are used to set the selection state of an icon given its command number and the task ID of its owner. *mdlDialog_selectIconsByCmdNoMsg* does not display the command message in the Command Window.

initiatingDbP points to the initiating dialog box, or may be NULL.

commandNumber indicates which icon to set by specifying the command number. A value of 0 indicates that the number of the currently active command should be used.

taskId indicates the task owning the command number and icon to seek. If it is NULL, the currently active task is used.

selectType gives the desired selection state of the icon. It may be *ICONCMD_SELECTTYPE_OFF*, *ICONCMD_SELECTTYPE_SINGLESHOT* or *ICONCMD_SELECTTYPE_LOCKED*. If it is -1, the previous value of *selectType* is used.

Returns *mdlDialog_selectIconsByCmd* and *mdlDialog_selectIconsByCmdNoMsg* are of type void.

See Also *mdlDialog_selectIconsById, mdlDialog_selectIconsByIdNoMsg.*

mdlDialog_icPaletteSelectIcon

```
boolean mdlDialog_icPaletteSelectIcon /* <= TRUE if error */
(
RawItemHdr   *icPalP,          /* => ptr to icon cmd palette item */
int           iconIndex,        /* => index of icon to select */
int           selectType,       /* => ICONCMD_SELECTTYPE_xxx */
boolean       deselectOthers,   /* => normally TRUE */
boolean       updateToolSettings /* => TRUE = also set TS dialog box */
);
```

Description The `mdlDialog_icPaletteSelectIcon` function changes the selection status of the icon at index position *iconIndex* in the icon command palette described by the Raw Item Header pointed by *icPalP*.

selectType defines the type of selection being performed on the icon in the command palette. Allowable values for this field are:

Value	Description
ICONCMD_SELECTTYPE_OFF	Deselects the indicated icon. The icon is displayed in normal mode with the standard grey background.
ICONCMD_SELECTTYPE_SINGLESOT	Selects the indicated icon for single shot operation mode. The icon is displayed with a low density background highlight instead of the normal dark background highlight.
ICONCMD_SELECTTYPE_LOCKED	Selects the indicated icon. The icon is displayed in the normal dark background highlight mode.

deselectOthers is a boolean parameter indicating whether any icons in the command palette which are currently selected should be deselected. If TRUE, all other icons in the palette are deselected, otherwise they are not changed.

If *updateToolSettings* is TRUE, the Tool Settings window will be updated with the parameter fields associated with the icon selected or cleared if the icon is being deselected.

Returns `mdlDialog_icPaletteSelectIcon` returns TRUE if an error occurs and FALSE (SUCCESS) on successful completion.

See Also `mdlDialog_icFrameSelectIcon`.

mdlDialog_icPaletteSetItemInfo, mdlDialog_icPaletteGetItemInfo

```
boolean mdlDialog_icPaletteSetItemInfo /* <= TRUE if error */
(
  RawItemHdr  *icPalP,           /* => ptr to icon cmd palette item */
  RscFileHandlerFileH,          /* => NULL=use opened rsrc files */
  int          subItemIndex,     /* => index of subitem to set info */
  long         iconId            /* => subitem's new icon cmd id */
);

boolean mdlDialog_icPaletteGetItemInfo /* <= TRUE if error */
(
  long         *iconCmdIdP,      /* <= set NULL if don't want id */
  RawItemHdr  *icPalP,          /* => ptr to icon cmd palette item */
  int          subItemIndex     /* => index of subitem to get info on */
);
```

Description `mdlDialog_icPaletteSetItemInfo` sets the icon command resource of an icon in an icon command palette and `mdlDialog_icPaletteGetItemInfo` returns the icon command resource ID of an icon in an icon command palette.

icPalP is a pointer to the Raw Item Header of the icon command palette containing the icon indexed by the *subItemIndex* parameter.

iconId is the icon command resource ID in the resource file indicated by the *rFileH* parameter which the icon position in the command palette is to be set to. If *rFileH* is `NULL`, then the command searches through the application's resource list until it either finds the indicated resource or reaches the end of the list.

iconCmdIdP is a pointer to a long memory location into which the icon command resource ID of the icon in the icon palette is returned.

Returns These functions return `TRUE` if an error occurred, and `SUCCESS` if the function completed successfully.

See Also `mdlDialog_icPaletteGetNItems`.

mdlDialog_icPaletteGetNItems

```
boolean mdlDialog_icPaletteGetNItems /* <= -1 if error */
(
    RawItemHdr *icPalP /* => ptr to icon cmd palette item */
);
```

Description The `mdlDialog_icPaletteGetNItems` function returns the number of icons contained in the icon command palette described by the Raw Item Header pointed to by the *icPalP* parameter.

Returns `mdlDialog_icPaletteGetNItems` returns -1 if an error occurred or the number of icons in the command palette otherwise.

See Also `mdlDialog_icPaletteSetItemInfo`, `mdlDialog_icPaletteGetItemInfo`.

List Box Item Functions

The contents of a list box are implemented as a string list. The list box functions control what part of the string list is displayed in the list box, and manipulate the list box's current selection. For more information on manipulating the actual contents of a list box, see the "String List Manager" section.

See the "List box item" section of the "Standard Dialog Items" chapter for more information on using list box items.

The following table lists the list box functions:

Function	Used to
mdlDialog_listBoxDeleteAll	delete all columns from a list box.
mdlDialog_listBoxDeleteColumn	delete one column from a list box.
mdlDialog_listBoxGetColInfo	get attributes of a list box column.
mdlDialog_listBoxSetColInfo	set attributes of a list box column.
mdlDialog_listBoxSetInfo	set attributes of a list box.
mdlDialog_listBoxGetInfo	get attributes of a list box.
mdlDialog_listBoxGetLocationCursor	get the location of the selection cursor in a list box.
mdlDialog_listBoxSetLocationCursor	set the location of the selection cursor in a list box.
mdlDialog_listBoxGetNColumns	get number of columns in a list box.
mdlDialog_listBoxGetSelections	get location of all currently selected cells.
mdlDialog_listBoxSetSelections	set location of all currently selected cells.
mdlDialog_listBoxInsertColumn	insert a column into an existing list box.
mdlDialog_listBoxSetTopRowRedraw	set which row is the first displayed in a list box.
mdlDialog_listBoxSetStrListP	set the string list that a list box will manipulate.
mdlDialog_listBoxGetStrListP	get the string list that a list box is manipulating.
mdlDialog_listBoxNRowsChanged [ditemlb.ml]	inform a list box that the size of its string list has changed.
mdlDialog_listBoxGetDisplayRange	get the range of cells which are currently displayed in a list box.
mdlDialog_listBoxSetTopRow	set which string list row will be displayed in the first row of a list box.
mdlDialog_listBoxDrawContents [ditemlb.ml]	draw the contents of a list box.
mdlDialog_listBoxSelectCells	select cells in a list box.
mdlDialog_listBoxEnableCells	enable cells in a list box.
mdlDialog_listBoxGetSelectRange [ditemlb.ml]	determine the selection range.
mdlDialog_listBoxGetNextSelection [ditemlb.ml]	get the next selected cell.
mdlDialog_listBoxIsCellSelected	determine if a cell is selected.
mdlDialog_listBoxIsCellEnabled	determine if a cell is enabled.
mdlDialog_listBoxLastCellClicked [ditemlb.ml]	retrieve the last cell that was clicked in.

mdlDialog_listBoxDeleteAll

```
boolean mdlDialog_listBoxDeleteAll /* <= TRUE if error */
(
    RawItemHdr *listBoxP /* => listBox to delete all columns from */
);
```

Description mdlDialog_listBoxDeleteAll deletes all columns from a list box.

listBoxP points to the raw item header of the list box.

Returns mdlDialog_listBoxDeleteAll returns FALSE on success or TRUE if there is an error.

See Also mdlDialog_listBoxDeleteColumn.

mdlDialog_listBoxDeleteColumn

```
boolean mdlDialog_listBoxDeleteColumn /* <= TRUE if error */
(
    RawItemHdr *listBoxP, /* => listBox to delete column from */
    int columnIndex, /* => index of col to delete */
    boolean redraw /* => redraw listBox? */
);
```

Description mdlDialog_listBoxDeleteColumn deletes one column from a list box.

listBoxP points to the raw item header of the list box from which to delete a column.

columnIndex indicates which column to delete.

redraw indicates whether the listbox should be redrawn after the column is deleted.

Returns mdlDialog_listBoxDeleteColumn returns SUCCESS or TRUE if there is an error.

See Also mdlDialog_listBoxDeleteAll.

mdlDialog_listBoxGetColInfo, mdlDialog_listBoxSetColInfo

```
boolean mdlDialog_listBoxGetColInfo /* <= TRUE if error */
(
    long *widthP, /* <= NULL = don't want width (dcoords) */
    long *maxSizeP, /* <= NULL = don't want max size */
    ULong *attributesP, /* <= NULL = don't want attributes */
    char *headingP, /* <= NULL = don't want heading */
    RawItemHdr *listBoxP, /* => listBox to get info on */
    int columnIndex /* => index of col to get info on */
);

boolean mdlDialog_listBoxSetColInfo /* <= TRUE if error */
```

```
(
long      *widthP,          /* => NULL=not setting width (dcoords) */
long      *maxSizeP,        /* => NULL=not setting max size */
ULong     *attributesP,     /* => NULL=not setting attributes */
char      *headingP,        /* => NULL=not setting heading */
RawItemHdr *listBoxP,        /* => listBox to set info on */
int        columnIndex,     /* => index of col to set info on */
boolean    redraw           /* => redraw listBox? */
);
```

Description `mdlDialog_listBoxGetColInfo` and `mdlDialog_listBoxSetColInfo` are used to set and to retrieve several parameters associated with individual columns in list boxes. The first four variables passed to each function are pointers to variables representing four parameters of the specified list box column. If any of the four pointers passed is `NULL`, that parameter is not retrieved by `mdlDialog_listBoxGetColInfo` or set by `mdlDialog_listBoxSetColInfo`.

widthP points to an integer representing the width of the column in dialog coordinates.

maxSizeP points to an integer representing the maximum allowable length of a list box entry in the specified column.

attributesP points to an integer that contains the column's attributes. The bits that can be set include `ALIGN_LEFT`, `ALIGN_CENTER` and `ALIGN_RIGHT` constants defined in `dlogbox.h`.

headingP points to a character buffer that contains the column's title.

listBoxP points to the raw item header of the list box containing the column in question.

columnIndex is the index of the column in question.

redraw is a boolean flag indicating whether `mdlDialog_listBoxSetColInfo` should cause the list box to be redrawn after the column information is modified.

Returns `mdlDialog_listBoxGetColInfo` and `mdlDialog_listBoxSetColInfo` return `FALSE` on success or `TRUE` if there is an error.

See Also `mdlDialog_listBoxSetInfo`, `mdlDialog_listBoxGetInfo`.

mdlDialog_listBoxSetInfo, mdlDialog_listBoxGetInfo

```
boolean mdlDialog_listBoxSetInfo /* <= TRUE if error */
(
ULong     *attributesP, /* => NULL=don't set attributes */
ULong     *nRowsP,      /* => NULL=don't set # of displ. rows */
ULong     *sizeNumColumnP, /* => NULL=don't set size num column */
boolean    redraw,       /* => TRUE=redraw */
RawItemHdr *listBoxP     /* => list box to get info on */
);
```

```
boolean mdlDialog_listBoxGetInfo /* <= TRUE if error */
(
    ULong      *attribuesP, /* <= NULL=don't want attributes */
    ULong      *nRowsP,    /* <= NULL=don't want # displ. rows */
    ULong      *sizeNumColumnP, /* <= NULL=don't want size num column */
    RawItemHdr *listBoxP    /* => listBox to get info on */
);
```

Description `mdlDialog_listBoxSetInfo` and `mdlDialog_listBoxGetInfo` are used to set and retrieve several parameters associated with list boxes. The first three variables passed to each function are pointers to variables representing three parameters of the specified list box. If any of the three pointers passed is `NULL`, that parameter is not retrieved by `mdlDialog_listBoxGetInfo` or set by `mdlDialog_listBoxSetInfo`.

attribuesP points to an integer that contains the list box's attributes. The bits that can be set include `LISTATTR_LABEL_ON_SIDE`, `LISTATTR_RANGE_SELECTION`, and so on. The constants are defined in `dlogbox.h`.

nRowsP points to an integer representing the number of rows the list box can display at once.

sizeNumColumnP points to the integer allowing the list box handler to automatically number the rows. The maximum number of digits necessary to number the rows is specified. For example, if a list can have up to 99 rows, 2 should be specified. If the integer pointed to by *sizeNumColumnP* is 0, the rows are not numbered.

listBoxP points to the raw item header of the list box in question.

redraw is a boolean flag indicating whether `mdlDialog_listBoxSetInfo` should cause the list box to be redrawn after the list box information is modified.

Returns `mdlDialog_listBoxSetInfo` and `mdlDialog_listBoxGetInfo` return `FALSE` on success or `TRUE` if there is an error.

See Also `mdlDialog_listBoxGetColInfo`, `mdlDialog_listBoxSetColInfo`.

mdlDialog_listBoxGetLocationCursor

```
boolean mdlDialog_listBoxGetLocationCursor /* <= TRUE if error */
(
    int      *locationRowP, /* <= row index of location cursor */
    int      *locationColP, /* <= column index of location cursor */
    RawItemHdr *listBoxP    /* => listbox to get selections for */
);
```

Description `mdlDialog_listBoxGetLocationCursor` returns the location of the selection cursor within a listbox.

locationRowP points to an integer to receive the selection cursor's row number.

locationColumnP points to an integer to receive the selection cursor's column number.

listBoxP point to the raw item header of the list box in question.

Returns `mdlDialog_listBoxGetLocationCursor` returns FALSE on success or TRUE if there is an error.

See Also `mdlDialog_listBoxSetLocationCursor`.

mdlDialog_listBoxSetLocationCursor

```
boolean mdlDialog_listBoxSetLocationCursor /* <= TRUE if error */
(
  RawItemHdr *listBoxP,      /* => listbox to get selections for */
  int         locationRow,    /* => new row index of location cursor */
  int         locationCol     /* => new col. index of location cursor */
);
```

Description `mdlDialog_listBoxSetLocationCursor` sets the location of the selection cursor within a listbox.

listBoxP point to the raw item header of the list box in question.

locationRow is an integer indicating the selection cursor's row number.

locationColumn is an integer indicating the selection cursor's column number.

Returns `mdlDialog_listBoxSetLocationCursor` returns FALSE on success or TRUE if there is an error.

See Also `mdlDialog_listBoxGetLocationCursor`.

mdlDialog_listBoxGetNColumns

```
int mdlDialog_listBoxGetNColumns /* <= # of list box columns */
(
  RawItemHdr *listBoxP      /* => list box */
);
```

Description The `mdlDialog_listBoxGetNColumns` function reports the number of columns present in a list box.

listBoxP points to the raw item header of the list box to check.

Returns `mdlDialog_listBoxGetNColumns` returns the number of columns in the list box, or -1 on error.

mdlDialog_listBoxGetSelections

```
boolean mdlDialog_listBoxGetSelections /* <= TRUE if error */
(
    int          *nSelectionsP,      /* <= # of selected cells */
    Spoint2d     **selectionsPP,     /* <= selected cells array */
    RawItemHdr   *listBoxP           /* => listbox to get selections for */
);
```

Description The mdlDialog_listBoxGetSelections function returns a pointer to an array indicating which cells of the given listbox are currently selected. The array is a series of contiguous Spoint2d structures, each of which has two short integer fields called *x* and *y*. There is no need for the calling function to deallocate the array.

nSelectionsP points to an integer to receive the number of currently selected cells, which is also the size of the array.

selectionsPP is a pointer to the pointer variable that is to receive the location of the array. After this function is called, **selectionsPP* points to the beginning of the array.

listBoxP points to the raw item header of the list box in question.

Returns mdlDialog_listBoxGetSelections returns FALSE on success or TRUE if there is an error.

See Also mdlDialog_listBoxSetSelections.

mdlDialog_listBoxSetSelections

```
boolean mdlDialog_listBoxSetSelections /* <= TRUE if error */
(
    RawItemHdr   *listBoxP,          /* => listbox to get selections for */
    int          nSelections,        /* => # of selected cells */
    Spoint2d     *selectionsP,       /* => selected cells */
    boolean      deselectAllFirst,    /* => clear prev selection first? */
    boolean      redraw              /* => TRUE means redraw cells */
);
```

Description The mdlDialog_listBoxSetSelections function sets which cells in a list box are selected given a pointer to an array of cell coordinates. The array is a series of contiguous Spoint2d structures, each of which has two short integer fields called *x* and *y*. *nSelections* is an integer indicating the number of cells to select, which is also the size of the array.

listBoxP points to the raw item header of the list box in question.

nSelections indicates the number of selections in the array.

selectionsP points to the array of cell coordinates to select.

deselectAllFirst indicates whether all current selections should be dropped before the specified selections are made.

redraw indicates whether `mdlDialog_listBoxSetSelections` should cause the list box to be redrawn after the selections are made.

Returns `mdlDialog_listBoxSetSelections` returns FALSE on success or TRUE if there is an error.

See Also `mdlDialog_listBoxGetSelections`.

mdlDialog_listBoxInsertColumn

```
boolean mdlDialog_listBoxInsertColumn /* <= TRUE if error */
(
    long          *widthP,          /* => or NULL means 8*XC */
    long          *maxSizeP,        /* => or NULL means 10 */
    ULONG         *attributesP,     /* => or NULL means 0 */
    char          *headingP,        /* => set NULL if no heading */
    RawItemHdr    *listBoxP,        /* => listBox to insert column into */
    int           columnIndex,      /* => insert BEFORE this col, -1=append */
    boolean       redraw            /* => redraw listBox? */
);
```

Description The `mdlDialog_listBoxInsertColumn` function inserts a new column into an existing list box.

widthP points to an integer representing the width of the column in dialog coordinates. NULL means 8*XC.

maxSizeP points to an integer representing the maximum allowable size of a list box entry in the specified column. The maximum size is set to 10 if *maxSizeP* is NULL.

attributesP points to an integer that contains the column's attributes. The bits that can be set include `ALIGN_LEFT`, `ALIGN_CENTER` and `ALIGN_RIGHT` constants defined in `dlogbox.h`. NULL means 0.

headingP points to a character buffer that contains the column's title. The new column will have no heading if *headingP* is NULL.

listBoxP points to the raw item header of the list box containing the column in question.

columnIndex is the index of the column before which to insert. A value of -1 means make the new column the right most.

redraw is a boolean flag indicating whether `mdlDialog_listBoxInsertColumn` should cause the list box to be redrawn after the column information is modified.

Returns `mdlDialog_listBoxInsertColumn` returns FALSE on success or TRUE if there is an error.

mdlDialog_listBoxSetTopRowRedraw

```
boolean mdlDialog_listBoxSetTopRowRedraw /* <= TRUE if error */
(
    RawItemHdr *listBoxP,      /* => listbox to set */
    int         topRowIndex,    /* => new first displayed row (0 based) */
    boolean     redraw         /* => TRUE = redraw listBox contents */
);
```

Description The mdlDialog_listBoxSetTopRowRedraw function set which row number of a list box is displayed as the first row. Optionally, it can also cause the list box to be redrawn showing the new position.

listBoxP points to the raw item header of the list box in question.

topRowIndex is the index of the row to be displayed as the new top row.

redraw indicates whether the listbox should be redrawn.

Returns mdlDialog_listBoxSetTopRowRedraw returns FALSE on success or TRUE if there is an error.

mdlDialog_listBoxSetStrListP, mdlDialog_listBoxGetStrListP [ditemlb.ml]

```
#include <dlogitem.h>

boolean mdlDialog_listBoxSetStrListP /* <= TRUE if error */
(
    RawItemHdr *listBoxP,      /* => listbox to set strListP of */
    StringList *strListP,      /* => list's string list */
    int         nColumns       /* => # of columns in list */
);

StringList *mdlDialog_listBoxGetStrListP /* <= NULL if error */
(
    RawItemHdr *listBoxP      /* => listbox to get strListP of */
);
```

Description The mdlDialog_listBoxSetStrListP function sets *strListP* to be the string list that the list box item specified by *listBoxP* will manipulate. *nColumns* is the number of columns per row in the list box. It should be set to the same number of columns that were specified in the list box resource specification. A list box item must have an item hook function attached to it and mdlDialog_listBoxSetStrListP must be called upon receipt of the DITEM_MESSAGE_CREATE message. Otherwise the list box item won't know what to display as its contents.

To take full advantage of the list box manager features, the string list associated with a list box should have at least one information field. The list box manager uses the first information field (index 0) to store information regarding the state of that string.

The mdlDialog_listBoxGetStrListP function retrieves a pointer to the string list that the list box item specified by *listBoxP* is manipulating.

The following example code fragment shows the part of a list box item hook function that calls `mdlDialog_listBoxSetStrListP`, and also has an example of `mdlDialog_listBoxGetStrListP`:

```
Private void listBoxHook
(
DialogItemMessage *dimP
);
{
    dimP->msgUnderstood = TRUE;
    switch (dimP->messageType)
    {
        case DITEM_MESSAGE_CREATE:
        {
            StringList *strListP;

            strListP=mdlStringList_create(20, 1);

            /* Set up the members of the string list here */
            ...

            mdlDialog_listBoxSetStrListP(dimP->dialogItemP->rawItemP,
                                         strListP, 1);

            break;
        }

        case DITEM_MESSAGE_DESTROY:
        {
            *strListP;

            strListP=mdlDialog_listBoxGetStrListP(dimP->dialogItemP->rawItemP);
            mdlStringList_destroy (strListP);

            /* free up any other memory allocated for the item */
            ...

            break;
        }

        ...
        default:
        {
            dimP->msgUnderstood=FALSE;
            break;
        }
    }
}
```

Returns The `mdlDialog_listBoxSetStrListP` function returns `TRUE` if the list box's string list was improperly set. This usually means that either *listBoxP* or *strListP* is invalid.

The `mdlDialog_listBoxGetStrListP` function returns a pointer to the list box's string list, or `NULL` if an error occurred.

mdlDialog_listBoxNRowsChanged [ditemlb.ml]

```
#include <dlogitem.h>

boolean mdlDialog_listBoxNRowsChanged /* <= TRUE if error */
(
    RawItemHdr *listBoxP      /* => listbox to notify of # change */
);
```

Description The mdlDialog_listBoxNRowsChanged function informs the list box item specified by *listBoxP* that the number of rows in its string list has changed. An MDL application should call this function whenever the number of rows in the list box's string list changes because of calls to mdlStringList_insertMember or mdlStringList_deleteMember. mdlDialog_listBoxNRowsChanged corrects the size of the list box's scroll bar.

Returns The mdlDialog_listBoxNRowsChanged function returns TRUE if an error occurred. This usually means that *listBoxP* is not a pointer to a list box item.

See Also mdlStringList_insertMember, mdlStringList_deleteMember.

mdlDialog_listBoxGetDisplayRange, mdlDialog_listBoxSetTopRow [ditemlb.ml]

```
#include <dlogitem.h>

boolean mdlDialog_listBoxGetDisplayRange /* <= TRUE if error */
(
    int *topRowIndexP,      /* <= row # of first row displayed */
    int *bottomRowIndexP,   /* <= row # of last row displayed */
    int *leftColIndexP,     /* <= col # of leftmost col disp. */
    int *rightColIndexP,    /* <= col # of rightmost col disp. */
    RawItemHdr *listBoxP    /* => listbox to get range of */
);

boolean mdlDialog_listBoxSetTopRow /* <= TRUE if error */
(
    RawItemHdr *listBoxP,   /* => listbox to set 1st disp. row */
    int topRowIndex         /* => new first disp. row (0 based) */
);
```

Description The mdlDialog_listBoxGetDisplayRange function gets the range of the cells that are displayed in the list box specified by *listBoxP*. The function is used to determine what part of the list box's string list is currently being displayed. The first and last displayed rows in the list box are copied to the locations specified by *topRowIndexP* and *bottomRowIndexP*. The first and last displayed columns in the list box are copied to the locations specified by *leftColIndexP* and *rightColIndexP*. *topRowIndexP*, *bottomRowIndexP*, *leftColIndexP* or *rightColIndexP* can be NULL, indicating that the caller doesn't need the value.

For example, suppose that a list box is configured to display five rows at a time with one column, and that its string list contains twenty members. If the first row of the list box is displaying the sixth member of the string list

(whose row index is five), the integers pointed at by *topRowIndexP*, *bottomRowIndexP*, *leftColIndexP* and *rightColIndexP* will be set to 5, 9, 0 and 0 respectively.

The `mdlDialog_listBoxSetTopRow` function sets the first displayed row of a list box to be *topRowIndex*.

Row and column indexes start with 0.

Returns The `mdlDialog_listBoxGetDisplayRange` and `mdlDialog_listBoxSetTopRow` functions return `TRUE` if an error occurred. This usually means that *listBoxP* is not a pointer to a list box item.

mdlDialog_listBoxDrawContents [ditemlb.ml]

```
#include <dlogitem.h>

boolean mdlDialog_listBoxDrawContents /* <= TRUE if error */
(
  RawItemHdr *listBoxP,      /* => listbox to draw */
  int         relRowIndex,    /* => displayed row to draw, -1 = all */
  int         relColIndex     /* => displayed col. to draw, -1 = all */
);
```

Description The `mdlDialog_listBoxDrawContents` function draws the contents of the list box item specified by *listBoxP*.

relRowIndex specifies the relative row index (0 based) of the row to draw. Relative row 0 indicates that whatever is contained in the first row should be drawn, not that the first row of the list box's string list should be drawn. If *relRowIndex* is -1, all the displayed rows in the list box will be redrawn.

relColIndex specifies the relative column index (0 based) of the column to draw. Relative column 0 indicates that the first column in the list box should be drawn. If *relColIndex* is -1, all the displayed columns in the list box will be redrawn.

The entire list box contents is redrawn if -1 is specified for both *relRowIndex* and *relColIndex*.

Suppose for example, that the fifth member of a list box's string list is changed by a call to `mdlStringList_setMember`. The entire list box could be redrawn by calling `mdlDialog_itemDraw`, but that will cause more drawing than is necessary. Instead, `mdlDialog_listBoxGetDisplayRange` should be called to determine what part of the string list is currently being displayed. Only if the fifth member is in the display range should `mdlDialog_listBoxDrawContents` be called, with *relRowIndex* set to 4 (the fifth string list member) - *topRowIndex* (the first displayed string list row).

Returns The `mdlDialog_listBoxDrawContents` function returns `TRUE` if an error occurred. This usually means that *listBoxP* is not a pointer to a list box item.

See Also `mdlDialog_listBoxGetDisplayRange`.

mdlDialog_listBoxSelectCells, mdlDialog_listBoxEnableCells [ditemlb.ml]

```
#include <dlogitem.h>

boolean mdlDialog_listBoxSelectCells /* <= TRUE if error */
(
    RawItemHdr *listBoxP,      /* => listbox to select */
    int minRowIndex,          /* => min row of area to select */
    int maxRowIndex,          /* => max row of area to select */
    int minColIndex,          /* => min col of area to select */
    int maxColIndex,          /* => max col of area to select */
    boolean selectOn,          /* => FALSE = selection is turned off */
    boolean redraw             /* => TRUE = redraw cells */
);

boolean mdlDialog_listBoxEnableCells /* <= TRUE if error */
(
    RawItemHdr *listBoxP,      /* => listbox to enable */
    int minRowIndex,          /* => min row of area to enable */
    int maxRowIndex,          /* => max row of area to enable */
    int minColIndex,          /* => min col of area to enable */
    int maxColIndex,          /* => max col of area to enable */
    boolean enable,           /* => FALSE=disabled (greyed,dim,unselectable) */
    boolean redraw             /* => TRUE = redraw cells */
);
```

Description The `mdlDialog_listBoxSelectCells` function selects a range of cells in the list box specified by *listBoxP*. The *selectOn* parameter should be set to `TRUE` to select the range of cells, or `FALSE` to deselect the cells. Selected cells are displayed with a black background color.

The `mdlDialog_listBoxEnableCells` function enables or disables a range of cells in the list box specified by *listBoxP*. The *enable* parameter should be set to `TRUE` to enable the range of cells, or `FALSE` to disable the cells. An enabled cell is selectable; a disabled cell is dim and unselectable.

The *minRowIndex*, *maxRowIndex*, *minColIndex*, and *maxColIndex* parameters together specify the range of cells to affect. Row and column indexes start with 0.

The *redraw* parameter, when `TRUE`, specifies that the affected cells should be redrawn if they are currently visible inside of the list box.

Returns `mdlDialog_listBoxSelectCells` and `mdlDialog_listBoxEnableCells` return `TRUE` if an error occurred. This usually means that *listBoxP* is not a pointer to a list box item.

mdlDialog_listBoxGetSelectRange [ditemlb.ml]

```
#include <dlogitem.h>

boolean mdlDialog_listBoxGetSelectRange /* <= TRUE if error */
(
    int      *minRowIndexP,      /* <= min row of selection */
    int      *maxRowIndexP,      /* <= max row of selection */
    int      *minColIndexP,      /* <= min col of selection */
    int      *maxColIndexP,      /* <= max col of selection */
    RawItemHdr *listBoxP          /* => listbox to get selection of */
);
```

Description The `mdlDialog_listBoxGetSelectRange` function gets the range of the cells that are currently selected in the list box specified by *listBoxP*. The first and last selected rows in the list box are copied to the locations specified by the *minRowIndexP* and *maxRowIndexP* parameters. The first and last selected columns in the list box are copied to the locations specified by the *minColIndexP* and *maxColIndexP* parameters. Any of the parameters *minRowIndexP*, *maxRowIndexP*, *minColIndexP* or *maxColIndexP* can be `NULL`, which indicates that the caller doesn't need the value. Row and column indexes start with 0.

If the list box is set up to allow disjoint selections, not all of the cells in the returned range will necessarily be selected. This function only returns the bounds of the selected cells. Use `mdlDialog_listBoxIsCellSelected` to determine if a cell in the range is actually selected when disjoint selections are allowed.

Returns The `mdlDialog_listBoxGetSelectRange` function returns `TRUE` if an error occurred. This usually means that *listBoxP* is not a pointer to a list box item.

See Also `mdlDialog_listBoxGetNextSelection` [ditemlb.ml],
`mdlDialog_listBoxIsCellSelected`.

mdlDialog_listBoxGetNextSelection [ditemlb.ml]

```
#include <dlogitem.h>

boolean mdlDialog_listBoxGetNextSelection /* <= TRUE if error */
(
    boolean *foundP,             /* <= TRUE if selection was found */
    int      *rowIndexP,         /* <=> on input these args specify cell */
    int      *colIndexP,         /* to start search AFTER; ptrs to -1, -1=search
                                from beginning. Output is next selected cell */
    RawItemHdr *listBoxP         /* => listbox to get next selection of */
);
```

Description The `mdlDialog_listBoxGetNextSelection` function searches for the next selected cell of the list box specified by *listBoxP*.

The boolean variable pointed at by *foundP* is set to `TRUE` if a selected cell was found.

On input *rowIndexP* and *colIndexP* specify the cell to start the search after. Pointing at -1 with both of these parameters indicates that the search should start from the beginning of the list.

If a selected cell is found, its row index and column index are copied to the locations specified by *rowIndexP* and *colIndexP*.

Returns The `mdlDialog_listBoxGetNextSelection` function returns `TRUE` if an error occurred. This usually means that *listBoxP* is not a pointer to a list box item.

See Also `mdlDialog_listBoxGetSelectRange [ditemlb.ml]`.

mdlDialog_listBoxIsCellSelected, mdlDialog_listBoxIsCellEnabled [ditemlb.ml]

```
#include <dlogitem.h>

boolean mdlDialog_listBoxIsCellSelected /* <= TRUE if selected */
(
    RawItemHdr *listBoxP,      /* => listbox to test */
    int    rowIndex,          /* => row index of cell to test */
    int    colIndex          /* => column index of cell to test */
);

boolean mdlDialog_listBoxIsCellEnabled /* <= TRUE if enabled */
(
    RawItemHdr *listBoxP,      /* => listbox to test */
    int    rowIndex,          /* => row index of cell to test */
    int    colIndex          /* => column index of cell to test */
);
```

Description The `mdlDialog_listBoxIsCellSelected` function determines if a cell in the list box specified by *listBoxP* is selected. A selected cell is a cell that is displayed with a black background color instead of the normal light gray.

The `mdlDialog_listBoxIsCellEnabled` function determines if a cell in the list box specified by *listBoxP* is enabled. An enabled cell is selectable; a disabled cell is dim and unselectable.

The *rowIndex* and *colIndex* parameters indicate which cell to test. Row and column indexes start with 0.

Returns `mdlDialog_listBoxIsCellSelected` returns `TRUE` if the cell is selected. `mdlDialog_listBoxIsCellEnabled` returns `TRUE` if the cell is enabled. Both functions will return `FALSE` if *listBoxP* is not a pointer to a list box item.

See Also `mdlDialog_listBoxLastCellClicked [ditemlb.ml]`.

mdlDialog_listBoxLastCellClicked [ditemlb.ml]

```
#include <dlogitem.h>
boolean mdlDialog_listBoxLastCellClicked
(
    int      *rowIndexP,    /* <= row index of last cell clicked */
    int      *colIndexP,    /* <= column index of last cell clicked */
    RawItemHdr *listBoxP    /* => listbox to get last click cell of */
);
```

Description The `mdlDialog_listBoxLastCellClicked` function determines the cell in the list box specified by *listBoxP* in which the data button was last clicked. The row index and column index of this cell are copied to the locations specified by *rowIndexP* and *colIndexP*. Either *rowIndexP* or *colIndexP* can be `NULL`, which indicates that the caller doesn't need the value. Row and column indexes start with 0.

Returns The `mdlDialog_listBoxLastCellClicked` function returns `TRUE` if an error occurred. This usually means that *listBoxP* is not a pointer to a list box item.

See Also `mdlDialog_listBoxIsCellSelected`, `mdlDialog_listBoxIsCellEnabled`.

Text Item Functions

The text item functions manipulate text items. Currently, only the minimum and maximum range strings of a text item can be retrieved or modified.

See the “Text Item” section of the “Standard Dialog Items” chapter for more information on using text items.

The following table lists the text item functions:

Function	Used to
<code>mdlDialog_textGetRange</code>	get the range of a text item.
<code>mdlDialog_textSetRange</code>	set the range of a text item.
<code>mdlDialog_textGetInfo</code>	get info on a text item.
<code>mdlDialog_textSetInfo</code>	set info on a text item.
<code>mdlDialog_m1TextGetInfo</code>	get info on a multi-line text item.
<code>mdlDialog_m1TextSetInfo</code>	set info on a multi-line text item.
<code>mdlDialog_m1TextGetCursor</code>	get the current cursor position for a multi-line text element.
<code>mdlDialog_m1TextSetCursor</code>	set current cursor position in a multi-line text item.
<code>mdlDialog_m1TextGetLineCoords</code>	translate a byte offset into a line number and offset in a multi-line text item.

Function	Used to
mdlDialog_m1TextGetLineRange	get the byte indices of the start and end of a line in a multi-line text item.
mdlDialog_m1TextInsertString	insert a string at a specified offset.
mdlDialog_m1TextTopRowNumber	force a specified row to be the first row in the display area.

mdlDialog_textGetRange, mdlDialog_textSetRange

```
#include <dlogitem.h>

boolean mdlDialog_textGetRange
(
    char    *minimumP,      /* <= minimum value string */
    char    *maximumP,      /* <= maximum value string */
    RawItemHdr *textP       /* => text item */
);

boolean mdlDialog_textSetRange
(
    RawItemHdr *textP,      /* => text item */
    char    *minimumP,      /* => new minimum value string */
    char    *maximumP       /* => new maximum value string */
);
```

Description The mdlDialog_textGetRange function retrieves the minimum and maximum value strings from the text item specified by *textP*.

The mdlDialog_textSetRange function sets the minimum and maximum value strings for the text item specified by *textP*.

Either of the parameters *minimumP* or *maximumP* can be NULL. The corresponding value will not be retrieved or set.

Returns The mdlDialog_textGetRange and mdlDialog_textSetRange functions return TRUE if an error occurs. This means that *textP* does not point to a text item.

mdlDialog_textGetInfo, mdlDialog_textSetInfo

```
boolean mdlDialog_textGetInfo /* <= TRUE if error */
(
    ULong *commandNumberP, /* <= NULL = don't want cmd number */
    ULong *commandSourceP, /* <= NULL = don't want cmd source */
    int *maxSizeP,         /* <= NULL = don't want max size */
    char *formatToDisplayP, /* <= NULL = don't want format */
    char *formatToInternalP, /* <= NULL = don't want format */
    char *minimumP,        /* <= NULL = don't want minimum */
    char *maximumP,        /* <= NULL = don't want maximum */
    ULong *maskP,          /* <= NULL = don't want maskP */
    ...
);
```

```

USHort *attributesP,          /* <= NULL = don't want attributes */
long *labelAboveP,           /* <= NULL = don't want labelabove */
RawItemHdr *textP            /* => text item to get info on */
);

boolean mdlDialog_textSetInfo /* <= TRUE if error */
(
  ULong *commandNumberP,      /* => NULL = not setting cmd number */
  ULong *commandSourceP,      /* => NULL = not setting cmd source */
  int *maxSizeP,              /* => NULL = not setting maxSize */
  char *formatToDisplayP,     /* => NULL = not setting format */
  char *formatToInternalP,    /* => NULL = not setting format */
  char *minimumP,             /* => NULL = not setting minimum */
  char *maximumP,             /* => NULL = not setting maximum */
  ULong *maskP,               /* => NULL = not setting mask */
  UShort *attributesP,        /* => NULL = not setting attributes */
  long *labelAboveP,          /* => NULL = not setting labelabove */
  boolean redraw,             /* => TRUE means redraw */
  RawItemHdr *textP          /* => text item to get info on */
);

```

Description The `mdlDialog_textGetInfo` and `mdlDialog_textSetInfo` functions are used to set or obtain information on the attributes of a text item. See the discussion of the text item resource for further information on the information associated with text items. For all of the pointer parameters, `NULL` indicates to `mdlDialog_textGetInfo` that the field should not be modified, and indicates to `mdlDialog_textSetInfo` that the field is not desired.

commandNumberP points to a variable containing the command number associated with the text item.

commandSourceP points to a variable indicating the source of the command, typically `LCMD` indicating a command from the current MDL application, or `MCMD`, indicating a MicroStation command.

maxSizeP points to a variable indicating the maximum allowable size of the text string that can be contained by the item, up to 256 characters.

formatToDisplayP points to a `sprintf` format string to convert the value of the variable associated with the text item into a displayable string.

formatToInternalP points to a `scanf` format string to convert from the text item's string value to the format expected by the variable associated with the text item.

minimumP points to a string that contains the minimum value of text item.

maximumP points to a string that contains maximum value of text item.

maskP points to a variable indicating which bits of the associated variable may be set if the variable is an integer. Can be `NOMASK`, which is equivalent to `0xFFFFFFFF`.

attributesP points to a variable that specifies text item characteristics, such as TEXT_CONCAT, TEXT_NOCONCAT and TEXT_READONLY.

labelAboveP points to a variable containing the *labelAbove* field.

If *redraw* is TRUE, the item should be redrawn after the attributes are set.

textP points to the raw item header of the text item to get or set information on.

Returns mdlDialog_textGetInfo and mdlDialog_textSetInfo return TRUE if there is an error, or SUCCESS on success.

mdlDialog_m1TextGetInfo, mdlDialog_m1TextSetInfo [ditemlib.ml]

```
boolean mdlDialog_m1TextGetInfo /* <= TRUE if error */
(
  ULong   *attributesP,          /* <= NULL = don't want attributes */
  long    *displayRowsP,        /* <= NULL = don't want display rows */
  long    *labelAboveP,         /* <= NULL = don't want label above */
  RawItemHdr *m1TextP           /* => m1text to get info on */
);

boolean mdlDialog_m1TextSetInfo /* <= TRUE if error */
(
  ULong   *attributesP,          /* => NULL = not setting attributes */
  long    *displayRowsP,        /* => NULL = not setting display rows */
  long    *labelAboveP,         /* => NULL = not setting label above */
  boolean redraw,               /* => TRUE means redraw */
  RawItemHdr *m1TextP           /* => m1text to set info about */
);
```

Description The mdlDialog_m1TextGetInfo and mdlDialog_m1TextSetInfo functions are used to get and set information on scroll bar items. For each of the pointer variables below except *m1TextP*, NULL indicates to mdlDialog_m1TextSetInfo that the scroll bar information is not to be changed, and indicates to mdlDialog_m1TextGetInfo that the information is not desired.

attributesP points to a variable indicating the attributes of the multi-line text item. See the discussion of the multi-line text item resource for details.

displayRowsP points to a variable indicating the number of rows of text that may be displayed at one time.

labelAboveP points to a flag indicating whether the label is above the multi-line text item, or to the side.

If *redraw* is TRUE, the multi-line text item is redrawn after mdlDialog_m1TextSetInfo is finished.

m1TextP points to the raw item header of the multi-line text item in question.

Returns `mdlDialog_m1TextGetInfo` and `mdlDialog_m1TextSetInfo` return `SUCCESS` if successful, or a non-zero value on error.

See Also `mdlDialog_m1TextGetInfo`, `mdlDialog_m1TextSetInfo`, `mdlDialog_m1TextSetCursor`, `mdlDialog_m1TextGetCursor`, `mdlDialog_m1TextTopRowNumber`, `mdlDialog_m1TextGetLineRange`, `mdlDialog_m1TextInsertString`, `mdlDialog_m1TextGetLineCoords`.

mdlDialog_m1TextGetCursor

```
boolean mdlDialog_m1TextGetCursor
(
    int      *lineNumberP,
    ULONG    *startCharPosP,      /* => Cursor position or 1st highlight */
    RawItemHdr *rawM1TextP      /* => m1text to set info about */
);
```

Description The `mdlDialog_m1TextGetCursor` function retrieves the current cursor position for the multi-line text element specified by *rawM1TextP*.

lineNumberP points to a word that receives the line number for the current cursor position. This line number is a logical line number relative to the start of the text, not relative to the display area. *lineNumberP* may be `NULL`.

startCharPosP points to an unsigned long that receives the byte position corresponding to the current cursor position. If *lineNumberP* is `NULL`, byte index is relative to the start of the string. If *lineNumberP* is not `NULL`, byte index is relative to the start of the line that contains the cursor.

startCharPosP may be `NULL`.

rawM1TextP points to the raw item header for the multi-line text item.

Returns `mdlDialog_m1TextGetCursor` returns `SUCCESS` if it processed the call. It returns a non-zero value if *rawM1TextP* does not point to the raw item header of a multi-line text item.

See Also `mdlDialog_m1TextGetInfo`, `mdlDialog_m1TextSetInfo`, `mdlDialog_m1TextSetCursor`, `mdlDialog_m1TextGetCursor`, `mdlDialog_m1TextTopRowNumber`, `mdlDialog_m1TextGetLineRange`, `mdlDialog_m1TextInsertString`, `mdlDialog_m1TextGetLineCoords`.

mdlDialog_m1TextSetCursor

```
boolean mdlDialog_m1TextSetCursor /* <= TRUE if error */
(
    RawItemHdr *rawM1TextP,      /* => m1text to set info about */
    int        lineNumber,       /* => -1 if pos. relative to buf. start */
    ULONG      startCharPos,     /* => Cursor position for cursor */
    boolean     forceVisible     /* => TRUE, scroll if cur not in dispArea */
);
```

Description The mdlDialog_m1TextSetCursor function sets the cursor position in a multi-line text item.

rawM1TextP points to the raw item header for the multi-line text item.

lineNumber specifies the line to position the cursor on. This line number is relative to the start of the string, not to the start of the display area. If *lineNumber* is -1, it is ignored.

startCharPos specifies the byte index of the new cursor position. It is relative to the start of the line specified by *lineNumber*. If *lineNumber* is -1, it is relative to the start of the text.

If *forceVisible* is non-zero and the cursor is positioned to an area that is not displayed, then MicroStation scrolls the text so that the cursor position is visible.

Returns mdlDialog_m1TextSetCursor returns SUCCESS if it processed the call. It returns a non-zero value if *rawM1TextP* does not point to the raw item header of a multi-line text item.

See Also mdlDialog_m1TextGetInfo, mdlDialog_m1TextSetInfo, mdlDialog_m1TextSetCursor, mdlDialog_m1TextGetCursor, mdlDialog_m1TextTopRowNumber, mdlDialog_m1TextGetLineRange, mdlDialog_m1TextInsertString, mdlDialog_m1TextGetLineCoords.

mdlDialog_m1TextGetLineCoords

```
int mdlDialog_m1TextGetLineCoords /* <= TRUE if error */
(
    int          *lineP,          /* <= line number; may be NULL */
    int          *offsetP,        /* <= bytes from start of line; NULL OK */
    RawItemHdr   *rawM1TextP,    /* => pointer to raw item header */
    int          byteOffset      /* => index into string */
);
```

Description The mdlDialog_m1TextGetLineCoords function translates a byte offset into a line number and offset.

lineP points to an integer than receives the line number of the line that contains the character indexed by *byteOffset*. *lineP* may be NULL.

offsetP points to an integer that receive the byte position relative to the start of the line. *offsetP* may be NULL.

rawM1TextP points to the raw item header for the multi-line text item.

byteOffset is an index into the string associated with the multi-line text item.

Returns mdlDialog_m1TextGetLineCoords returns SUCCESS if it processed the call. It returns a non-zero value if *rawM1TextP* does not point to the raw item header of a multi-line text item.

See Also `mdlDialog_m1TextGetInfo`, `mdlDialog_m1TextSetInfo`,
`mdlDialog_m1TextSetCursor`, `mdlDialog_m1TextGetCursor`,
`mdlDialog_m1TextTopRowNumber`, `mdlDialog_m1TextGetLineRange`,
`mdlDialog_m1TextInsertString`, `mdlDialog_m1TextGetLineCoords`.

mdlDialog_m1TextGetLineRange

```
boolean mdlDialog_m1TextGetLineRange /* <= TRUE if error */
(
    ULong          *startCharPosP, /* => Cursor position or 1st highlight */
    ULong          *endCharPosP,   /* => Cursor position or end highlight */
    RawItemHdr     *rawM1TextP,    /* => m1text to set info about */
    int            lineNumber       /* => logical line number */
);
```

Description The `mdlDialog_m1TextGetLineRange` function retrieves the byte indices for the beginning and end of a line in the multi-line text item.

startCharPosP points to an integer that receives the byte index of the first character displayed on the specified line.

endCharPosP points to an integer that receives the byte index of the last character displayed on the specified line. It is assumed that both *startCharPosP* and *endCharPosP* are non-NULL.

If a line contains a newline and nothing else, then **startCharPosP* and **endCharPosP* will be equivalent.

rawM1TextP points to the raw item header for the multi-line text item.

lineNumber specifies a logical line number. Both **startCharPosP* and **endCharPosP* will be set to -1 if *lineNumber* specifies a line beyond the end of the data.

Returns `mdlDialog_m1TextGetLineRange` returns `SUCCESS` if it processed the call. It returns a non-zero value if *rawM1TextP* does not point the raw item header of a multi-line text item.

See Also `mdlDialog_m1TextGetInfo`, `mdlDialog_m1TextSetInfo`,
`mdlDialog_m1TextSetCursor`, `mdlDialog_m1TextGetCursor`,
`mdlDialog_m1TextTopRowNumber`, `mdlDialog_m1TextGetLineRange`,
`mdlDialog_m1TextInsertString`, `mdlDialog_m1TextGetLineCoords`.

mdlDialog_m1TextInsertString

```
boolean mdlDialog_m1TextInsertString /* <= TRUE if error */
(
  RawItemHdr *rawM1TextP, /* => item's raw item header */
  char *stringP, /* => string to insert */
  int charsToReplace, /* => bytes to replace */
  int offset /* => offset for string */
);
```

Description The `mdlDialog_m1TextInsertString` function inserts a string at the specified offset. It does not move the cursor or cause the text item to scroll.

`mdlDialog_m1TextInsertString` does display the string.

rawM1TextP points to the raw item header for the multi-line text item.

stringP points to the string to insert.

charsToReplace specifies the number of characters to replace. It may be 0.

offset is a byte offset that specifies where to insert *stringP*.

Returns `mdlDialog_m1TextInsertString` returns `SUCCESS` if it processed the call, and a non-zero value if *rawM1TextP* does not point to the raw item header of a multi-line text item.

See Also `mdlDialog_m1TextGetInfo`, `mdlDialog_m1TextSetInfo`,
`mdlDialog_m1TextSetCursor`, `mdlDialog_m1TextGetCursor`,
`mdlDialog_m1TextTopRowNumber`, `mdlDialog_m1TextGetLineRange`,
`mdlDialog_m1TextInsertString`, `mdlDialog_m1TextGetLineCoords`.

mdlDialog_m1TextTopRowNumber

```
int mdlDialog_m1TextTopRowNumber
(
  RawItemHdr *rawM1TextP, /* => m1text to set info about */
  int firstRow /* => logical row number */
);
```

Description The `mdlDialog_m1TextTopRowNumber` function forces the specified row to be the first row in the text item's display area.

rawM1TextP points to the raw item header for the multi-line text item.

firstRow is a row number relative to the start of text string being processed by the text item.

Returns `mdlDialog_m1TextTopRowNumber` returns `SUCCESS` if it processed the call, and a non-zero value if *rawM1TextP* does not point to the raw item header of a multi-line text item.

See Also `mdlDialog_m1TextGetInfo`, `mdlDialog_m1TextSetInfo`,
`mdlDialog_m1TextSetCursor`, `mdlDialog_m1TextGetCursor`,
`mdlDialog_m1TextTopRowNumber`, `mdlDialog_m1TextGetLineRange`,
`mdlDialog_m1TextInsertString`, `mdlDialog_m1TextGetLineCoords`.

Scroll Bar Item Functions

The scroll bar item functions manipulate the state of scroll bar items. See the “Scroll Bar Item” section of the “Standard Dialog Items” chapter for more information on using scroll bar items.

The following table lists the scroll bar item functions:

Function	Used to
<code>mdlDialog_scrollBarGetInfo</code>	get info on a scroll bar.
<code>mdlDialog_scrollBarSetInfo</code>	set info on a scroll bar.
<code>mdlDialog_scrollBarSetRange</code>	set the range of a scroll bar item.

mdlDialog_scrollBarGetInfo, mdlDialog_scrollBarSetInfo [ditemlib.mtl]

```
boolean mdlDialog_scrollBarGetInfo /* <= TRUE if error */
(
    long    *minValueP,          /* <= NULL=don't want min value */
    long    *maxValueP,          /* <= NULL=don't want max value */
    long    *incAmountP,         /* <= NULL=don't want inc amount */
    long    *pageIncAmountP,     /* <= NULL=don't want page inc amount */
    double  *sliderSizeP,        /* <= NULL =don't want slider size */
    boolean *isVerticalP,        /* <= NULL =don't want isVertical */
    RawItemHdr *sbarP           /* => scroll bar to set info on */
);

boolean mdlDialog_scrollBarSetInfo /* <= TRUE if error */
(
    long    *minValueP,          /* => NULL=not setting min value */
    long    *maxValueP,          /* => NULL=not setting max value */
    long    *incAmountP,         /* => NULL=not setting inc amount */
    long    *pageIncAmountP,     /* => NULL=not setting page inc amnt */
    double  *sliderSizeP,        /* => NULL=not setting slider size */
    boolean *isVerticalP,        /* => NULL=not setting isVertical */
    boolean redraw,              /* => TRUE=redraw */
    RawItemHdr *sbarP           /* => scroll bar to set info on */
);
```

Description The `mdlDialog_scrollBarGetInfo` and `mdlDialog_scrollBarSetInfo` functions are used to get and set information on scroll bar items. For each of the pointer variables below except *sbarP*, NULL indicates to `mdlDialog_scrollBarSetInfo` that the scroll bar information is not to be changed, and indicates to `mdlDialog_scrollBarGetInfo` that the information is not desired.

minValueP points to a variable indicating the minimum possible value of the variable associated with the scroll bar.

maxValueP points to a variable indicating the maximum possible value of the variable associated with the scroll bar.

incAmountP points to a variable indicating the amount by which to increase or decrease the associated variable when the user clicks on the arrows in the scroll bar to single-step.

pageIncAmountP points to a variable indicating the amount by which to increase or decrease the associated variable when the user clicks on the areas between the slider and the arrows in the scroll bar.

sliderSizeP points to a variable indicating the size of the slider to be displayed in the scroll bar. Values between 0 and 1 are valid. The slider is never smaller than a square inside the scroll bar.

If *isVerticalP* is TRUE, the scroll bar is displayed vertically. Otherwise, it is horizontal.

If *redraw* is TRUE, the scroll bar is redrawn after `mdlDialog_scrollBarSetInfo` is finished.

sbarP points to the raw item header of the scroll bar for which to get or set information.

Returns `mdlDialog_scrollBarGetInfo` and `mdlDialog_scrollBarSetInfo` return SUCCESS if successful, or non-zero if an error occurred.

mdlDialog_scrollBarSetRange [ditemlib.ml]

```
#include <dlogitem.h>

boolean mdlDialog_scrollBarSetRange /* <= TRUE if error */
(
    RawItemHdr  *sbarP,          /* => scroll bar item */
    double sliderSize,          /* => relative size of slider */
    int  minValue,              /* => min value of scroll bar */
    int  maxValue,              /* => max value of scroll bar */
    int  incAmount,             /* => arrow click scroll amount */
    int  pageIncAmount          /* => page scroll amount */
);
```

Description The `mdlDialog_scrollBarSetRange` function sets the range information associated with the scroll bar item specified by the *sbarP* parameter.

The parameters other than *sbarP* are analogous to those found in the `DItem_ScrollBarRsc` structure. See the “Scroll Bar Item” section of the “Standard Dialog Items” chapter for more information on the `DItem_ScrollBarRsc` structure.

Returns The `mdlDialog_scrollBarSetRange` function returns TRUE if an error occurs. This means that *sbarP* does not point to a scroll bar item.

See Also `mdlDialog_scrollArrowDraw`.

Push Button Item Functions

The push button item functions manipulate push button items. Push buttons can be activated, made the default push button of a dialog, and made cancel buttons. You can also both obtain and set push button information.

See the “Push Button Item” section of the “Standard Dialog Items” chapter for more information on using push button items.

The following table lists the push button item functions:

Function	Used to
<code>mdlDialog_pushButtonActivate</code> [ditemlib.ml]	activate a push button.
<code>mdlDialog_pushButtonSetDefault</code> [ditemlib.ml]	set a default push button.
<code>mdlDialog_pushButtonSetCancel</code> [ditemlib.ml]	set a Cancel push button.
<code>mdlDialog_pushButtonGetInfo</code> [ditemlib.ml]	obtain information about a push button.
<code>mdlDialog_pushButtonSetInfo</code> [ditemlib.ml]	set information for a push button.

mdlDialog_pushButtonActivate [ditemlib.ml]

```
#include <dlogitem.h>

boolean mdlDialog_pushButtonActivate /* <= TRUE if error */
(
  RawItemHdr *pButP /* => push button item to activate */
);
```

Description The `mdlDialog_pushButtonActivate` function activates the push button identified by *pButP*. Activating a push button simulates the user clicking the mouse while the mouse cursor is over the push button.

Returns `mdlDialog_pushButtonActivate` returns TRUE if an error occurs. This means that *pButP* does not point to a push button item.

mdlDialog_pushButtonSetDefault [ditemlib.ml]

```
#include <dlogitem.h>

boolean mdlDialog_pushButtonSetDefault
(
  RawItemHdr *pButP, /* => push button item to change */
  boolean isDefault /* => FALSE=button is no longer default */
);
```

Description The `mdlDialog_pushButtonSetDefault` function changes the default status of the push button identified by *pButP*. If *isDefault* is `TRUE`, the push button is made the default. If *isDefault* is `FALSE`, the push button has its default status revoked.



There can be only one default button, so no danger exists that more than one button might be activated by default. However, because this function does not reset the default flag on previous default buttons, it is possible for several buttons to have the default flag set, and thus be drawn in the dialog with default highlights. To avoid more than one button with default highlights in a dialog, use this function to clear the default flag on previous default buttons before you set another button to default.

Returns `mdlDialog_pushButtonSetDefault` returns `TRUE` if an error occurs. This means that *pButP* does not point to a push button item.

See Also `mdlDialog_pushButtonActivate` [ditemlib.ml], `mdlDialog_pushButtonSetCancel` [ditemlib.ml], `mdlDialog_pushButtonGetInfo` [ditemlib.ml], `mdlDialog_pushButtonSetInfo` [ditemlib.ml].

mdlDialog_pushButtonSetCancel [ditemlib.ml]

```
#include <dlogitem.h>

boolean mdlDialog_pushButtonSetCancel
(
  RawItemHdr *pButP,          /* => push button item to change */
  boolean      isCancel       /* => FALSE=button is no longer cancel */
);
```

Description The `mdlDialog_pushButtonSetCancel` function changes the cancel status of the push button identified by *pButP*. If *isCancel* is `TRUE`, the push button is made the Cancel button. If *isCancel* is `FALSE`, the push button has its cancel status revoked.



You should take care to ensure that only one button in each dialog has its cancel flag set. Disable any existing Cancel button(s) before assigning cancel status to a new button.

Returns `mdlDialog_pushButtonSetCancel` returns `TRUE` if an error occurs. This means that *pButP* does not point to a push button item.

See Also `mdlDialog_pushButtonSetDefault` [ditemlib.ml], `mdlDialog_pushButtonActivate` [ditemlib.ml], `mdlDialog_pushButtonGetInfo` [ditemlib.ml], `mdlDialog_pushButtonSetInfo` [ditemlib.ml].

mdlDialog_pushButtonGetInfo [ditemlib.ml]

```
#include <dlogitem.h>
boolean mdlDialog_pushButtonGetInfo
(
    ULong    *commandNumberP,    /* <= NULL=don't want cmd number */
    ULong    *commandSourceP,    /* <= NULL=don't want cmd source */
    char     **unparsedPP,       /* <= NULL=don't want ptr unparsed */
    int      *buttonTypeP,       /* <= NULL=don't want button type */
    RawItemHdr *pButP           /* => push button to get info on */
);
```

Description The `mdlDialog_pushButtonGetInfo` function obtains information about the push button specified by *pButP*.

commandNumberP points to a variable to receive the command that is activated when the button is pressed.

commandSourceP indicates the owning application. If zero, the owning application is MicroStation. If non-zero, the owning application is the application that built the dialog box.

unparsedPP points to a string to be passed to the command. The string is copied into the queue element, and the function that processes the command receives a pointer to the copy. The string that *unparsedPP* points to can be in a temporary location. If there is no string to pass to the command, *unparsedPP* can be NULL. It can also point to a zero-length string.

buttonTypeP returns the button type. Possible values are `DEFAULT_BUTTON`, `CANCEL_BUTTON`, or zero (meaning the button is of some other type).

Returns `mdlDialog_pushButtonGetInfo` returns TRUE if an error occurs. This means that *pButP* does not point to a push button item.

See Also `mdlDialog_pushButtonSetInfo` [ditemlib.ml], `mdlDialog_pushButtonActivate` [ditemlib.ml], `mdlDialog_pushButtonSetDefault` [ditemlib.ml], `mdlDialog_pushButtonSetCancel` [ditemlib.ml], `mdlInput_sendCommand`.

mdlDialog_pushButtonSetInfo [ditemlib.ml]

```
#include <dlogitem.h>
boolean mdlDialog_pushButtonSetInfo
(
    ULong    *commandNumberP,    /* => NULL=not setting cmd number */
    ULong    *commandSourceP,    /* => NULL=not setting cmd source */
    char     **unparsedPP,       /* => NULL=not setting ptr unparsed */
    int      *buttonTypeP,       /* => NULL=not setting button type */
    boolean redraw,              /* => TRUE=redraw */
    RawItemHdr *pButP           /* => push button to set info on */
);
```

Description The mdlDialog_pushButtonSetInfo function sets information about the push button specified by *pButP*.

commandNumberP points to a variable to receive the command to be activated when the button is pressed.

commandSourceP indicates the owning application. If zero, the owning application will be MicroStation. If non-zero, the owning application is the application that built the dialog box.

unparsedPP points to a string to be passed to the command. The string is copied into the queue element, and the function that processes the command receives a pointer to the copy. The string that *unparsedPP* points to can be in a temporary location. If there is no string to pass to the command, *unparsedPP* can be NULL. It can also point to a zero-length string.

buttonTypeP sets the button type. Possible values are DEFAULT_BUTTON, CANCEL_BUTTON, or zero (meaning the button is of some other type).

redraw, if set to TRUE, will redraw the button after the new information is set, so that the button's appearance will reflect any changes.

Returns mdlDialog_pushButtonSetInfo returns TRUE if an error occurs. This means that *pButP* does not point to a push button item.

See Also mdlDialog_pushButtonActivate [ditemlib.m]l,
mdlDialog_pushButtonSetDefault [ditemlib.m]l,
mdlDialog_pushButtonSetCancel [ditemlib.m]l, mdlDialog_pushButtonGetInfo [ditemlib.m]l, mdlInput_sendCommand.

Toggle Button Item Functions

The toggle button item functions manipulate the state of toggle button items. See the "Toggle Button Item" section of the "Standard Dialog Items" chapter for more information on using toggle button items.

The following table lists the toggle button item functions:

Function	Used to
mdlDialog_toggleButtonGetInfo	get info on a toggle button.
mdlDialog_toggleButtonSetInfo	set info on a toggle button.

mdlDialog_toggleButtonGetInfo, mdlDialog_toggleButtonSetInfo [ditemlib.mtl]

```
boolean mdlDialog_toggleButtonGetInfo /* <= TRUE if error */
(
  ULong *commandNumberP, /* <= NULL = don't want cmd number */
  ULong *commandSourceP, /* <= NULL = don't want cmd source */
  ULong *maskP,          /* <= NULL = don't want mask */
  int *invertP,          /* <= NULL = don't want button type */
  RawItemHdr *tButP      /* => toggleButton to get info on */
);
boolean mdlDialog_toggleButtonSetInfo /* <= TRUE if error */
(
  ULong *commandNumberP, /* => NULL = not setting cmd number */
  ULong *commandSourceP, /* => NULL = not setting cmd source */
  ULong *maskP,          /* => NULL = not setting mask */
  int *invertP,          /* => NULL = not setting button type */
  RawItemHdr *tButP      /* => toggleButton to set info on */
);
```

Description The `mdlDialog_toggleButtonGetInfo` and `mdlDialog_toggleButtonSetInfo` functions are used to get and set information on toggle button items. Each of the parameters except *tButP* may be NULL, indicating that the information is not to be retrieved or set.

commandNumberP points to a variable indicating the command number associated with the toggle button. *commandSourceP* points to a variable indicating the source of the command, typically MCMD for a MicroStation command or LCMD for an MDL application command.

maskP points to a variable containing the mask for setting bits in the variable associated with the toggle button. See the discussion of the toggle button item resource for details.

invertP indicates whether to invert the function of the *maskP* variable. Again, see the discussion of the toggle button item resource for details.

tButP points to the raw item header of the toggle button in question.

Returns `mdlDialog_toggleButtonGetInfo` and `mdlDialog_toggleButtonSetInfo` return SUCCESS if successful, or non-zero if an error occurred.

Level Map Item Functions

The level map item functions manipulate the state of level map items. See the “Level Map Item” section of the “Standard Dialog Items” chapter for more information on using level map items.

The following table lists the level map item functions:

Function	Used to
mdlDialog_levelMapGetInfo	get info on a level map.
mdlDialog_levelMapSetInfo	set info on a level map.

mdlDialog_levelMapGetInfo, mdlDialog_levelMapSetInfo [ditemlib.ml]

```
boolean mdlDialog_levelMapGetInfo /* <= TRUE if error*/
(
  char          *activeLevelAccessStrP, /* <= active level ac str */
  RawItemHdr    *lmapP                  /* => level map to get info on */
);

boolean mdlDialog_levelMapSetInfo /* <= TRUE if error*/
(
  char          *activeLevelAccessStrP, /* => active level ac str */
  RawItemHdr    *lmapP                  /* => level map to set info on */
);
```

Description mdlDialog_levelMapGetInfo and mdlDialog_levelMapSetInfo set and retrieve the active level access string for the level map specified by *lmapP*.

activeLevelAccessStrP points to a buffer into which to receive the active level access string, or to the value to which to set the access string.

lmapP points to the raw item header of the level map for which to get or set the access string.

Returns mdlDialog_levelMapGetInfo and mdlDialog_levelMapSetInfo return SUCCESS if they succeed, or a non-zero value if there is an error.

Color Picker Item Functions

The color picker item functions manipulate the state of color picker items. See the “Color Picker Item” section of the “Standard Dialog Items” chapter for more information on using color picker items.

The following table lists the color picker item functions:

Function	Used to
<code>mdlDialog_colorPickerGetInfo</code>	get info on a color picker.
<code>mdlDialog_colorPickerSetInfo</code>	set info on a color picker.

mdlDialog_colorPickerGetInfo, mdlDialog_colorPickerSetInfo [ditemlib.ml]

```
boolean mdlDialog_colorPickerGetInfo
(
    ULONG *commandNumberP,      /* <= NULL = don't want cmd number */
    ULONG *commandSourceP,      /* <= NULL = don't want cmd source */
    ULONG *maskP,               /* <= NULL = don't want mask */
    long *associatedTextIdP,     /* <= NULL = don't want text id */
    RawItemHdr *cPickerP        /* => color picker to get info on */
);

boolean mdlDialog_colorPickerSetInfo
(
    ULONG *commandNumberP,      /* => NULL = not setting cmd number */
    ULONG *commandSourceP,      /* => NULL = not setting cmd source */
    ULONG *maskP,               /* => NULL = not setting mask */
    long *associatedTextIdP,     /* => NULL = not setting text id */
    RawItemHdr *cPickerP        /* => color picker to get info on */
);
```

Description The `mdlDialog_colorPickerGetInfo` function is used to obtain internal information about a given color picker dialog item. The `mdlDialog_colorPickerSetInfo` function is used to set the internal attributes associated with a color picker dialog item.

commandNumberP is a pointer to an unsigned integer to get/set the command number associated with the color picker item. The command number associated with a color picker item is the command that gets placed at the end of the MicroStation input queue when a button is released within the item.

commandSourceP is a pointer to an unsigned integer to get/set the command source attribute of a color picker item. The command source attribute specifies the task that will execute the command identified by *commandNumberP*. The constant `LCMD` indicates that the task that owns (originally creates) the dialog should execute the command. The constant

MCMD indicates that MicroStation should be used to execute the command, in which case *commandNumberP* indicates a MicroStation command defined in cmdlist.h.

maskP is a pointer to an unsigned integer to get/set the state variable mask. This mask indicates which bits of the item's underlying state variable are to be used to store the color picker's current color index. Usually the mask is set to NOMASK (0xFFFFFFFF) which indicates that the entire variable will be used to determine the color index.

associatedTextIdP is a pointer to an unsigned integer to get/set the resource identifier of the text field associated with the color picker item.

cPickerP is the color picker item to have its internal information obtained in the case of mdlDialog_colorPickerGetInfo or set in the case of mdlDialog_colorPickerSetInfo.

Returns mdlDialog_colorPickerGetInfo and mdlDialog_colorPickerSetInfo return SUCCESS or a non-zero value on error.

Completion Bar Functions

The completion bar functions are used to manipulate the completion bar window. Any MDL application that performs lengthy tasks should use a completion bar to keep the user updated on the progress of those tasks.

The following table lists the completion bar functions:

Function	Used to
mdlDialog_openCompletionBar	open the completion bar window.
mdlDialog_closeCompletionBar	close the completion bar window.
mdlDialog_updateCompletionBar	update the completion bar window.
mdlDialog_displayCompletionBarMessage	display a message in the completion bar window.
mdlDialog_completionBarClose	close the completion bar window.
mdlDialog_completionBarDisplayMessage	display a message in the completion bar window.
mdlDialog_completionBarOpen	open the completion bar window.
mdlDialog_completionBarUpdate	update the completion bar window.

mdlDialog_openCompletionBar

```
#include <mdl.h>
#include <dlogbox.h>
DialogBox *mdlDialog_openCompletionBar /* <= NULL if fails */
(
    char    *messageText    /* => text message to be displayed */
);
```

Description mdlDialog_openCompletionBar opens the completion bar window, displays the text defined by the *messageText* parameter in the window, and returns a pointer to the window to the calling application. If the window is already opened, the window is re-initialized and a pointer to the existing window is returned.

The completion bar window displays a rectangle graphically depicting the percentage complete for the operation indicated by the text message passed in *messageText*. A numeric percent complete is displayed above the completion bar. A message box is displayed below the completion bar if status messages are sent to the completion bar window—if no messages are sent, no message box is displayed.

Returns The mdlDialog_openCompletionBar function returns a pointer to the opened completion bar window or NULL if the open failed.

See Also mdlDialog_closeCompletionBar, mdlDialog_updateCompletionBar, mdlDialog_displayCompletionBarMessage.

mdlDialog_closeCompletionBar

```
#include <mdl.h>
#include <dlogbox.h>

void mdlDialog_closeCompletionBar
(
    DialogBox    *dbP        /* => pointer to window */
);
```

Description mdlDialog_closeCompletionBar closes the completion bar window pointed to by the *dbP* parameter. If the message box below the completion bar was opened by sending messages to the completion bar window using the mdlDialog_displayCompletionBarMessage function, then the window is not closed and the user must close the window by pressing the “Close” button.

Returns The mdlDialog_closeCompletionBar function is of type void and returns no value.

See Also mdlDialog_openCompletionBar, mdlDialog_updateCompletionBar, mdlDialog_displayCompletionBarMessage.

mdlDialog_updateCompletionBar

```
#include <mdl.h>
#include <msdefs.h>
void mdlDialog_updateCompletionBar
(
    DialogBox    *dbP,           /* => dialog box ptr */
    char         *messageText,    /* => Message text */
    int          percentComplete /* => % complete on bar */
);
```

Description mdlDialog_updateCompletionBar updates the completion bar window to the display the message defined by the *messageText* parameter and the percent complete defined by the *percentComplete* parameter. The valid values for *percentComplete* are 0 through 100.

Returns The mdlDialog_updateCompletionBar function is of type `void` and returns no value.

See Also mdlDialog_openCompletionBar, mdlDialog_closeCompletionBar, mdlDialog_displayCompletionBarMessage.

mdlDialog_displayCompletionBarMessage

```
#include <mdl.h>
#include <msdefs.h>
void mdlDialog_displayCompletionBarMessage
(
    DialogBox    *dbP,           /* => dialog box ptr */
    char         *messageText    /* => Message text */
);
```

Description mdlDialog_displayCompletionBarMessage displays a text message defined by the *messageText* parameter in the message box of the completion bar window.

Returns The mdlDialog_displayCompletionBarMessage function is of type `void` and returns no value.

See Also mdlDialog_openCompletionBar, mdlDialog_closeCompletionBar, mdlDialog_updateCompletionBar.

mdlDialog_completionBarClose

```
#include <dlogbox.h>
#include <msdialog.fdf>
void mdlDialog_completionBarClose
(
    DialogBox    *dbP           /* => pointer to window */
);
```

Description `mdlDialog_completionBarClose` closes the completion bar window pointed to by the *dbP* parameter. If the message box below the completion bar was opened by sending messages to the completion bar window using the `mdlDialog_completionBarDisplayMessage` function, then the window is not closed and the user must close the window by pressing the “Close” button.



This function was implemented in MicroStation 95.

Returns The `mdlDialog_completionBarClose` function is of type `void` and returns no value.

See Also `mdlDialog_completionBarOpen`, `mdlDialog_completionBarUpdate`, `mdlDialog_completionBarDisplayMessage`, `mdlDialog_busyBarStopProcessing` [`ditemlib.ml`, `ditemlib.msl`], `mdlDialog_trackBarStopProcessing` [`ditemlib.ml`, `ditemlib.msl`].

mdlDialog_completionBarDisplayMessage

```
#include <dlogbox.h>
#include <msdialog.fdf>

void mdlDialog_completionBarDisplayMessage
(
    DialogBox    *dbP,           /* => dialog box ptr */
    char         *messageText    /* => Message text */
);
```

Description `mdlDialog_completionBarDisplayMessage` displays a text message defined by the *messageText* parameter in the message box of the completion bar window.



This function was implemented in MicroStation 95.

Returns The `mdlDialog_completionBarDisplayMessage` function is of type `void` and returns no value.

See Also `mdlDialog_completionBarOpen`, `mdlDialog_completionBarClose`, `mdlDialog_completionBarUpdate`, `mdlDialog_busyBarUpdateMessage` [`ditemlib.ml`, `ditemlib.msl`], `mdlDialog_trackBarUpdateDisplayInfo` [`ditemlib.ml`, `ditemlib.msl`], `mdlDialog_busyBarUpdateControlParms` [`ditemlib.ml`, `ditemlib.msl`], `mdlDialog_busyBarUpdateControlParms` [`ditemlib.ml`, `ditemlib.msl`].

mdlDialog_completionBarOpen

```
#include <dlogbox.h>
#include <msdialog.fdf>

DialogBox *mdlDialog_completionBarOpen /* <= NULL if fails */
(
    char         *messageText    /* => text message to be displayed */
);
```

Description mdlDialog_completionBarOpen opens the completion bar window, displays the text defined by the *messageText* parameter in the window, and returns a pointer to the window to the calling application. If the window is already opened, the window is re-initialized and a pointer to the existing window is returned.

The completion bar window displays a rectangle graphically depicting the percentage complete for the operation indicated by the text message passed in *messageText*. A numeric percent complete is displayed above the completion bar. A message box is displayed below the completion bar if status messages are sent to the completion bar window—if no messages are sent, no message box is displayed.



This function was implemented in MicroStation 95.

Returns The mdlDialog_completionBarOpen function returns a pointer to the opened completion bar window or NULL if the open failed.

See Also mdlDialog_completionBarClose, mdlDialog_completionBarUpdate, mdlDialog_completionBarDisplayMessage, mdlDialog_busyBarStartProcessing [ditemlib.m1, ditemlib.msl], mdlDialog_trackBarStartProcessing [ditemlib.m1, ditemlib.msl].

mdlDialog_completionBarUpdate

```
#include <dlogbox.h>
#include <msdialog.fdf>

void mdlDialog_completionBarUpdate
(
  DialogBox    *dbP,          /* => dialog box ptr */
  char         *messageText,  /* => Message text */
  int          percentComplete/* => percent complete on bar */
);
```

Description mdlDialog_completionBarUpdate updates the completion bar window to display the message defined by the *messageText* parameter and the percent complete defined by the *percentComplete* parameter. The valid values for *percentComplete* are 0 through 100.



This function was implemented in MicroStation 95.

Returns The mdlDialog_completionBarUpdate function is of type void and returns no value.

See Also mdlDialog_completionBarOpen, mdlDialog_completionBarClose, mdlDialog_completionBarDisplayMessage, mdlDialog_busyBarUpdateMessage [ditemlib.m1, ditemlib.msl], mdlDialog_trackBarUpdateDisplayInfo [ditemlib.m1, ditemlib.msl], mdlDialog_busyBarUpdateControlParms [ditemlib.m1, ditemlib.msl], mdlDialog_trackBarUpdateControlParms [ditemlib.m1, ditemlib.msl].

Dialog Box General Functions

The general dialog box functions manipulate dialogs as a whole. They allow the opening and closing of dialogs. User data pointers that are attached to specific dialog boxes can also be set and retrieved.

In addition, some of the functions needing to use run-time C expressions within resource files are described in this section.

The following table lists the general dialog box functions:

Function	Used to
<code>mdlDialog_open</code>	open a modeless dialog box.
<code>mdlDialog_openModal</code>	open a modal dialog box.
<code>mdlDialog_openPalette</code>	open a dialog box that only contains an icon command palette.
<code>mdlDialog_openAlert</code> <code>mdlDialog_openAlertById</code>	open an alert box to display a warning message.
<code>mdlDialog_openInfoBox</code>	open an info box to display information to the user.
<code>mdlDialog_openMessageBox</code> (Windows NT only)	open a modal box to display information and ask the user to make a selection.
<code>mdlDialog_autoOpen</code>	open all dialogs that were open when MicroStation was last shut down.
<code>mdlDialog_create</code>	open a modal or modeless dialog box without displaying it on the user's display.
<code>mdlDialog_show</code>	bring a dialog to the front.
<code>mdlDialog_find</code>	get a pointer to a particular dialog box.
<code>mdlDialog_findByTypeAndId</code>	find a loaded dialog by its type and identifier.
<code>mdlDialog_hasFocus</code>	given a dialog box pointer, return <code>TRUE</code> if the dialog box has the input focus.
<code>mdlDialog_ownerMDGet</code>	get the MDL descriptor of the task that owns a particular dialog box.
<code>mdlDialog_commandWindowGet</code>	return a pointer to the current command window.
<code>mdlDialog_parentIdGet</code>	get the parent ID of a dialog box.
<code>mdlDialog_parentIdSet</code>	set the parent ID of a dialog box.
<code>mdlDialog_userDataPtrGet</code>	get the user pointer associated with a dialog.
<code>mdlDialog_userDataPtrSet</code>	set the user pointer associated with a dialog.
<code>mdlDialog_hookDialogSendUserMsg</code>	send a user message to a dialog hook function.

Function	Used to
mdlDialog_hookItemSendUserMsg	send a user message to an item hook function.
mdlDialog_lastActionTypeSet	set the reason why a dialog box was closed.
mdlDialog_hookPublish	publish hook function addresses.
mdlDialog_publishBasicVariable	publish a basic C data type variable for use in run-time C expression strings.
mdlDialog_publishComplexVariable	publish a structure type variable for use in run-time C expression strings.
mdlDialog_publishBasicPtr	publish a pointer to a basic C data type variable for use in run-time C expression strings.
mdlDialog_publishComplexPtr	publish a pointer to a structure type variable for use in run-time C expression strings.
mdlDialog_publishStructure	publish a structure declaration for use in run-time C expression strings.
mdlDialog_synonymsSynch	synchronize all occurrences of a group of items.
mdlDialog_setFilterString	change the file filter string of the currently open file open or file create dialog.

mdlDialog_open

```
#include <dlogitem.h>

DialogBox *mdlDialog_open /* <= NULL if error */
(
  RscFileHandle    rFileH,          /* => NULL = search open rsc files */
  int              resourceId       /* => ID of dialog to open */
);
```

Description The mdlDialog_open function opens a modal or modeless dialog box. It returns immediately to its caller. Most modal dialogs should be opened with the mdlDialog_openModal function instead.

The *rFileH* parameter specifies the resource file to search for the dialog box resource. If NULL, all the calling application's open resource files and then MicroStation's open resource files will be searched.

The *resourceId* parameter specifies the resource ID of the dialog box resource that is used to create the dialog box. The dialog box will be modal if the DIALOGATTR_MODAL bit is set in the attributes field of the dialog box resource.

Returns The `mdlDialog_open` function returns a pointer to the opened dialog box. `NULL` is returned if the dialog box resource was not found or errors occurred while loading the items contained in the dialog box.

See Also `mdlDialog_openModal`, “Modeless and modal dialog boxes,” “DialogBoxRsc Structure”.

mdlDialog_openModal

```
#include <dlogitem.h>

boolean mdlDialog_openModal /* <= TRUE if error opening */
(
    int          *lastActionTypeP, /* <= usually ACTIONBUTTON_... */
    RscFileHandle rFileH,          /* => NULL = use opened */
    int          resourceId         /* => ID of dialog to open */
);
```

Description The `mdlDialog_openModal` function opens a modal dialog box. It does not return to its caller until the user closes the modal dialog box.

The *rFileH* parameter specifies the resource file to search for the dialog box resource. If `NULL`, all the calling application’s open resource files and then MicroStation’s open resource files will be searched.

The *resourceId* parameter specifies the resource ID of the dialog box resource that is used to create the dialog box. Be sure to make the dialog box modal by setting the `DIALOGATTR_MODAL` bit in the *attributes* field of the dialog box resource.

The only way that a user can close a modal dialog box is to activate either of the special push buttons which have IDs: `PUSHBUTTONID_OK` or `PUSHBUTTONID_Cancel`. One of these must be included in any modal dialog box. Remember that the labels of these standard push buttons can be overridden as part of the item list specification. This allows the actions associated with clicking “OK” or “Cancel” in a modal dialog box to be performed by buttons that are labeled differently.

The *lastActionTypeP* parameter points to a variable that is set to indicate which push button the user used to close the modal dialog box. It will be set to `ACTIONBUTTON_OK` or `ACTIONBUTTON_CANCEL`.

Returns The `mdlDialog_openModal` function returns `TRUE` if an error occurs. This means that the modal dialog box resource was not found, or errors occurred while loading the items contained in the modal dialog box.

See Also “Modeless and modal dialog boxes,” “DialogBoxRsc Structure,” “Push Button Items”.

mdlDialog_openPalette

```
#include <dlogitem.h>
DialogBox *mdlDialog_openPalette /* <= NULL if error */
(
    RscFileHandle rFileH,          /* => NULL = use opened resource files */
    void          *ownerMD,        /* => MDL task to be owner of palette */
    int           paletteId        /* => ID of palette to open */
);
```

Description The mdlDialog_openPalette function opens a modeless dialog box that only contains a single icon command palette item.

The *rFileH* parameter specifies the resource file to search for the icon command palette resource. If *NULL*, all the calling application's open resource files and then MicroStation's open resource files will be searched.

The *ownerMD* parameter should be set to *NULL*.

The *paletteId* parameter specifies the resource ID of the icon command palette resource that is contained in the dialog box. The dialog box itself is created on the fly and its size is based on the size of the icon command palette.

Normally tool palettes are created by specifying a dialog box resource that only has one item: an icon command frame. This icon command frame in turn contains icon commands and icon command palettes. The icon command palettes are then automatically made into separate dialog boxes when the users tear them off of the icon command frame. The mdlDialog_openPalette function allows the direct creation of icon command palette dialog boxes.

Returns The mdlDialog_openPalette function returns a pointer to the opened dialog box. *NULL* is returned if the icon command palette resource was not found or errors occurred while loading the items contained by icon commands within the palette.

See Also "Icon command palette," "Icon command," "Icon Command Frames".

mdlDialog_openAlert

```
#include <dlogitem.h>
int mdlDialog_openAlert /* <= usually ACTIONBUTTON_ */
(
    char          *stringP        /* => string to display in alert */
);
```

Description The mdlDialog_openAlert function opens a modal dialog box that contains two push buttons labeled OK and Cancel, and a multi-line label item that will be set to

the string pointed to by *stringP*. It does not return to its caller until the user closes the modal dialog box.



This function is the same as `mdlDialog_openInfoBox` except that it also contains a Cancel button.

Returns The `mdlDialog_openAlert` function returns either `ACTIONBUTTON_OK` or `ACTIONBUTTON_CANCEL` depending on whether the user activated the OK or Cancel push button to dismiss the dialog box.

See Also `mdlDialog_openAlertById`, `mdlDialog_openInfoBox`.

mdlDialog_openAlertById

```
int mdlDialog_openAlertById /* <= usually ACTIONBUTTON_ */
(
    long    dialogId,        /* => Dialog Id of alert box to create. */
    char    *stringP         /* => string to display in alert */
);
```

Description `mdlDialog_openAlertById` allows an application to open an alternate alert dialog box. The standard `mdlDialog_openAlert` function calls this routine with a dialog ID of `DIALOGID_StandardAlert`. An alternate alert dialog box must have, at a minimum, one multi-line text field (item index 0), an OK button and a Cancel button.

dialogId is the ID of an alternate alert dialog box.

stringP is the alert message to be displayed in the multi-line text field of the dialog.

Returns `mdlDialog_openAlertById` returns either `ACTIONBUTTON_OK` or `ACTIONBUTTON_CANCEL` depending on whether the user activated the OK or Cancel push button to dismiss the dialog.

See Also `mdlDialog_openAlert`, `mdlDialog_openInfoBox`.

mdlDialog_openInfoBox

```
#include <dlogitem.h>

int mdlDialog_openInfoBox /* <= usually ACTIONBUTTON_ */
(
    char    *stringP         /* => string to display in alert */
);
```

Description `mdlDialog_openInfoBox` opens a modal dialog box that contains a single push button labeled OK, and a multi-line label item that will be set to the string pointed

to by *stringP*. It does not return to its caller until the user closes the modal dialog box.



This function is the same as `mdlDialog_openAlert` except that it does not contain a Cancel button.

Returns The `mdlDialog_openInfoBox` function returns `ACTIONBUTTON_OK`.

See Also `mdlDialog_openAlert`, `mdlDialog_openAlertById`.

mdlDialog_openMessageBox (Windows NT only)

```
int mdlDialog_openMessageBox /* <= usually ACTIONBUTTON_ */
(
    long    dialogId,        /* => Dialog Id of message box to create. */
    char    *stringP,        /* => string to display in message area */
    long    whichIcon        /* => Id of symbol to be drawn in box*/
);
```

Description `mdlDialog_openMessageBox` opens a modal dialog box which contains one to three push buttons, a multi-line label item, and one of four icons. It does not return to its caller until the user closes the modal dialog box.

dialogId is the ID of the message box which will be displayed, and will be used to specify which push buttons are included in the dialog. The following values for *dialogId* are available for use with `mdlDialog_openMessageBox`:

Value	Description
DIALOGID_MsgBoxOK	The message box contains one button: OK.
DIALOGID_MsgBoxOKCancel	The message box contains two push buttons: OK and Cancel.
DIALOGID_MsgBoxYesNo	The message box contains two push buttons: Yes and No.
DIALOGID_MsgBoxYesNoCancel	The message box contains three push buttons: Yes, No and Cancel.

`mdlDialog_openMessageBox` can also be used to display Icons in the following, (pre-6.0) dialogs:

```
DIALOGID_StandardAlert
DIALOGID_MediumAlert
DIALOGID_LargeAlert
DIALOGID_StandardInfoBox
DIALOGID_MediumInfoBox
DIALOGID_LargeInfoBox
DIALOGID_YesNoCancelAlert
```

If, for example, you want to display a large information box, you can use:

```
mdlDialog_openMessageBox(DIALOGID_LargeInfoBox, stringP,  
MSGBOX_ICON_INFORMATION);
```

stringP is the multi line text string to be displayed as the alert message of the dialog.

whichIcon specifies the raster icon to be drawn in the dialog box to the left of the alert message. `mdlDialog_openMessageBox` automatically detects the current GUIMODE and displays icons appropriate to Windows or Motif interface styles.:

whichIcon	symbol	Description
MSGBOX_ICON_INFORMATION		Provides information about the results of commands. Offers no user choices; user acknowledges message by pressing OK button. This means that the information symbol will nearly always appear in the <code>MsgBoxOK</code> dialog, i.e., <code>mdlDialog_openMessageBox(DIALOGID_MsgBoxOK, stringP, MSGBOX_ICON_INFORMATION);</code>
MSGBOX_ICON_WARNING		Alerts user to an error condition or situation that requires user decision and input before proceeding, such as an impending action with potentially destructive, irreversible consequences. The Message can be a question (for example, "Save changes to mechrft.upf?").
MSGBOX_ICON_QUESTION		Alerts user to an error condition or situation that requires user decision and input before proceeding, such as an impending action with potentially destructive, irreversible consequences. The Message can be a question (for example, "Save changes to mechrft.upf?"). Some applications may find the question mark symbol more appropriate if the alert is in the form of a question. Note, however, that the question mark is also used as a help symbol and may therefore confuse users.
MSGBOX_ICON_CRITICAL		Informs user of a serious system-related or application-related problem that must be corrected before work can continue with the application.

The above table is also available in section 7.1.4 (Message Dialogs) of *The Windows Interface: An application design guide*.

Returns `mdlDialog_openMessageBox` returns -1 if it fails, and returns `ACTIONBUTTON_Ok`, `ACTIONBUTTON_Cancel`, etc. to indicate SUCCESS.

See Also `mdlDialog_openAlert`, `mdlDialog_openAlertById`, `mdlDialog_openInfoBox`.

mdlDialog_autoOpen

```
int mdlDialog_autoOpen /* <= SUCCESS or error */
(
    int      *numOpenedP,          /* <= number of dlogs/pals opened */
    ULong    dialogWindowType,     /* => auto open dialogs or palettes? */
    boolean  openDialogs          /* => TRUE=open, FALSE=just count */
);
```

Description The `mdlDialog_autoOpen` function is used to automatically open any dialogs or palettes belonging to the calling application that were present on the screen when MicroStation last shut down. If this function is used, it is generally called during the application's initialization.

numOpenedP is a pointer to an integer where the number of palettes or dialogs opened is returned. If *openDialogs* is `FALSE`, the value pointed to by *numOpenedP* will indicate the number of dialogs or palettes that would have been opened if *openDialogs* was `TRUE`.

dialogWindowType indicates whether palettes or dialogs should be redisplayed.

openDialogs indicates whether the dialogs or palettes should really be opened or whether a count should just be obtained of those that could be opened.

Returns `mdlDialog_autoOpen` returns `SUCCESS` if the palettes/dialogs could be opened, otherwise it returns `ERROR`.

mdlDialog_create

```
#include <dlogitem.h>

DialogBox *mdlDialog_create /* <= NULL if error */
(
    RscFileHandle rFileH,          /* => NULL = search open rsc files */
    void          *ownerMD,        /* => NULL = use current MDL descr. */
    ULong         dialogType,       /* => resource type of dialog box */
    long          dialogId,         /* => ID of dialog to open */
    boolean       noWarnResourceError /* => usually FALSE */
);
```

Description `mdlDialog_create` opens a modal or modeless dialog box without displaying it on the user's display. It returns immediately to its caller. Most dialogs should be opened with the `mdlDialog_open` and `mdlDialog_openModal` functions instead.

rFileH specifies the resource file to search for the dialog box resource. If `NULL`, all the calling application's open resource files and then MicroStation's open resource files will be searched.

ownerMD specifies the MDL descriptor of the application which is to own the dialog box. If `NULL`, the dialog box is owned by the calling application.

dialogtype and *dialogId* specifies the resource type and ID of the dialog box resource that is used to create the dialog box. The dialog box will be modal if the `DIALOGATTR_MODAL` bit is set in the `attributes` field of the dialog box resource.

noWarnResourceError indicates whether a warning message should be displayed if the resource could not be found. This parameter is usually `FALSE`.

Returns The `mdlDialog_create` function returns a pointer to the created dialog box. `NULL` is returned if the dialog box resource was not found or errors occurred while loading the items contained in the dialog box.

See Also `mdlDialog_open`, `mdlDialog_openModal`, “Modeless and modal dialog boxes,” “DialogBoxRsc Structure”.

mdlDialog_show

```
#include <mdl.h>

void mdlDialog_show
(
    DialogBox    *dialogBoxP    /* => dialog box to show */
);
```

Description `mdlDialog_show` brings the dialog box pointed to by *dialogBoxP* back into view. If the dialog box can accept input focus, focus is changed to the dialog box. If the dialog box is modal, the dialog box is raised to the top of the display and priority list and made the dominant window. If the dialog box described an icon command palette, the last selected command is selected in the palette.

Returns The `mdlDialog_show` function is of type `void` and returns no values.

mdlDialog_find

```
#include <dlogitem.h>

DialogBox *mdlDialog_find /* <= NULL if error */
(
    int      dialogId,      /* => ID of dialog to find */
    void     *ownerMD       /* => usually NULL, owner mdl task */
);
```

Description The `mdlDialog_find` function gets a pointer to the currently open dialog box whose resource ID is *dialogId*.

The *ownerMD* parameter should be set to `NULL`.

Returns The `mdlDialog_find` function returns a pointer to the found dialog box, or `NULL` if no dialog box with the specified resource ID is currently open.

See Also `mdlDialog_findByTypeAndId`.

mdlDialog_findByTypeAndId

```
#include <rtypes.h>
#include <dlogitem.h>

DialogBox *mdlDialog_findByTypeAndId /* <= dialog box found */
(
    ULong   dialogType,    /* => type of dialog to find */
    int     dialogId,      /* => id of dialog to find */
    void    *ownerMD       /* => usually NULL, owner mdl task of dialog */
);
```

Description mdlDialog_findByTypeAndId obtain a pointer to an open dialog box according to its resource class (dialog type) and ID.

dialogType specifies the type of dialog. This is usually RTYPE_DialogBox.

dialogId is the resource ID of the dialog box.

ownerMD is the MDL descriptor of the owner of the dialog box. If NULL is specified, then the calling application is assumed.

Returns mdlDialog_findByTypeAndId returns a pointer to the targeted dialog box if found, otherwise it returns NULL.

See Also mdlDialog_find.

mdlDialog_hasFocus

```
#include <dlogitem.h>

boolean mdlDialog_hasFocus /* <= TRUE if has focus */
(
    DialogBox *dbP         /* => dialogBox to check for focus */
);
```

Description mdlDialog_hasFocus is used to determine if a given dialog box currently has the input focus.

dbP is the dialog to check for the input focus.

Returns mdlDialog_hasFocus returns TRUE if the dialog pointed to by *dbP* has the input focus or else it returns FALSE.

mdlDialog_ownerMDGet

```
#include <dlogitem.h>

void *mdlDialog_ownerMDGet /* <= owner MDL descr */
(
    DialogBox *dbP         /* => dialogBox whose owner MD to get */
);
```

Description `mdlDialog_ownerMDGet` returns the MDL descriptor of the task that owns the given dialog box.

dbP points to a dialog box from which the owning MDL task will be determined.

Returns `mdlDialog_ownerMDGet` returns an MDL descriptor pointer or `NULL`.

mdlDialog_commandWindowGet

```
#include <dlogitem.h>

DialogBox *mdlDialog_commandWindowGet(void);
/* <= ptr to Command Window */
```

Description `mdlDialog_commandWindowGet` is used to obtain a pointer to the current MicroStation command window.

Returns `mdlDialog_commandWindowGet` returns a pointer to the MicroStation command window.

mdlDialog_parentIdGet, mdlDialog_parentIdSet

```
#include <dlogitem.h>

int mdlDialog_parentIdGet /* <= 0 if error */
(
    DialogBox    *db                /* => dialog whose parent ID to get */
);

int mdlDialog_parentIdSet /* <= TRUE if error */
(
    DialogBox    *db,               /* => dialog whose parent ID to get */
    long         parentId           /* => new parent ID */
);
```

Description The `mdlDialog_parentIdGet` function retrieves the resource ID of the parent of the dialog box specified by *db*.

The `mdlDialog_parentIdSet` function sets the parent resource ID of the dialog box specified by *db* to be *parentId*.

The parent of a dialog box is usually specified in a dialog box resource specification. The parent dialog box's dialog hook function, if any, will be sent a `DIALOG_MESSAGE_CHILDDESTROYED` message when the child dialog box is closed.

Returns The `mdlDialog_parentIdGet` function returns the resource ID of the dialog box's parent dialog box. 0 is returned if *db* is not a pointer to a dialog box.

The `mdlDialog_parentIdSet` function returns `TRUE` if an error occurs. This means that *db* is not a pointer to a dialog box.

mdlDialog_userDataPtrGet, mdlDialog_userDataPtrSet

```
#include <dlogitem.h>

void *mdlDialog_userDataPtrGet /* <= dialog's userDataPtr */
(
    DialogBox    *db          /* => dialogBox whose userDataPtr to get */
);

boolean mdlDialog_userDataPtrSet /* <= TRUE if error */
(
    DialogBox    *db,          /* => dialogBox whose userDataPtr to set */
    void         *userDataP    /* => new user data pointer */
);
```

Description The mdlDialog_userDataPtrGet function retrieves the user data pointer of the dialog box specified by *db*.

The mdlDialog_userDataPtrSet function sets the user data pointer of the dialog box specified by *db* to be *userDataP*. Since the user data pointer is usually set indirectly by a dialog hook function upon receipt of the `DIALOG_MESSAGE_CREATE` message, the mdlDialog_userDataPtrSet function is rarely needed.

The user data pointer of a dialog box can be used to store a pointer to anything the MDL programmer wishes. The dialog box manager does not use it for any reason.

Returns The mdlDialog_userDataPtrGet function returns the user data pointer of the specified dialog box.

The mdlDialog_userDataPtrSet function returns `TRUE` if an error occurs. This means that *db* is not a pointer to a dialog box.

See Also “Dialog Hook Functions”.

mdlDialog_hookDialogSendUserMsg, mdlDialog_hookItemSendUserMsg

```
#include <dlogitem.h>

boolean mdlDialog_hookDialogSendUserMsg /* <= TRUE if error */
(
    DialogBox    *db,          /* => where to send hook user msg */
    int          type,         /* => type of message (user defined) */
    void         *userDataP    /* => user defined ptr passed w/msg */
);

boolean mdlDialog_hookItemSendUserMsg /* <= TRUE if error */
(
    RawItemHdr   *rihP,        /* => item whose hook to send to */
    int          type,         /* => type of msg (user defined) */
    void         *userDataP    /* => user defined ptr passed w/msg */
);
```

Description The `mdlDialog_hookDialogSendUserMsg` function sends a user message to any dialog hook function that is attached to the dialog box specified by *db*.

The `mdlDialog_hookItemSendUserMsg` function sends a user message to any item hook function attached to the dialog box item specified by *rihP*.

The *type* and *userDataP* parameters are simply passed on to the hook function. They can be any value the MDL programmer wishes.

Returns The `mdlDialog_hookDialogSendUserMsg` and `mdlDialog_hookItemSendUserMsg` functions return `TRUE` if an error occurs. This means that *db* is not a pointer to a dialog box or *rihP* is not a pointer to a dialog item header.

See Also “Dialog Hook Functions,” “Item Hook Functions”.

mdlDialog_lastActionTypeSet

```
#include <dlogitem.h>

boolean mdlDialog_lastActionTypeSet
(
  DialogBox    *db,           /* => dialogBox whose lastAction to set */
  int          actionType     /* => usually ACTIONBUTTON_? */
);
```

Description The `mdlDialog_lastActionTypeSet` function sets the action type that is sent to a dialog hook function as part of the `DIALOG_MESSAGE_DESTROY` message when a dialog box is closed. It is used to indicate why the dialog box was closed.

The *db* parameter indicates which dialog box to set the last action type for.

The *actionType* parameter specifies the dialog box's last action type.

The `mdlDialog_lastActionTypeSet` function is called as part of the item hook function that is attached to the special OK and Cancel push buttons which have IDs `PUSHBUTTONID_OK` and `PUSHBUTTONID_Cancel`. This is how `mdlDialog_openModal` can return which push button was pressed. Most MDL dialog programmers will never need to call this function.

Returns The `mdlDialog_lastActionTypeSet` function returns `TRUE` if an error occurs. This usually means that *db* is not a pointer to a dialog box.

See Also `mdlDialog_openModal`, “Modeless and modal dialog boxes”, “Push Button Items”.

mdlDialog_hookPublish

```
#include <dlogitem.h>

boolean mdlDialog_hookPublish /* <= TRUE if error */
(
  int          nHooks,        /* => # of hooks in next array */
  DialogHookInfo *hooks       /* => array of item & dialog hooks */
);
```

Description The `mdlDialog_hookPublish` function associates hook function ID numbers with hook function addresses.

The *nHooks* parameter specifies the size of the *hooks* array. Each member of the *hooks* array consists of a hook function ID and an MDL function address.

The `mdlDialog_hookPublish` function is usually called near the beginning of an MDL application program. It must be called before any dialog boxes that use hook functions are opened.

For example, suppose the include file `ids.h` contains the following definitions:

```
#define HOOKITEMID_listHook 1    /* an item hook function ID */
#define HOOKDIALOGID_dialogHook 2 /* a dialog hook func. ID */
```

Then the following lines would appear in `example.mc`:

```
#include "ids.h"

void example_listHook(), example_dialogHook();
Private DialogHookInfo uHooks[]=
{
    {HOOKITEMID_listHook, example_listHook},
    {HOOKDIALOGID_dialogHook, example_dialogHook},
};

main(int argc, char *argv[])
{
    ...
    mdlDialog_hookPublish(sizeof(uHooks)/sizeof(DialogHookInfo),
                          uHooks);
    ...
}
```

Returns The `mdlDialog_hookPublish` function returns `TRUE` if an error occurs.

See Also “Hook function IDs”.

mdlDialog_publishBasicVariable, mdlDialog_publishComplexVariable, mdlDialog_publishBasicPtr, mdlDialog_publishComplexPtr, mdlDialog_publishStructure

```
#include <dlogitem.h>

boolean mdlDialog_publishBasicVariable /* <= TRUE if error */
(
    void    *setP,          /* => symbol set to publish variable in */
    void    *typeP,         /* => type of data */
    char    *variableName, /* => name of variable */
    void    *variableP      /* => address of variable */
);
```

```

boolean mdlDialog_publishComplexVariable /* <= TRUE if error */
(
    void      *setP,          /* => symbol set to publish variable in */
    char      *structName,    /* => structure "tag" name */
    char      *variableName,  /* => name of variable */
    void      *variableP      /* => address of variable */
);

boolean mdlDialog_publishBasicPtr /* <= TRUE if error */
(
    void      *setP,          /* => symbol set to publish ptr in */
    void      *typeP,         /* => type of data */
    char      *variableName,  /* => name of variable */
    void      *variableP      /* => address of variable */
);

boolean mdlDialog_publishComplexPtr /* <= TRUE if error */
(
    void      *setP,          /* => symbol set to publish ptr in */
    char      *structName,    /* => structure "tag" name */
    char      *variableName,  /* => name of variable */
    void      *variableP      /* => address of variable */
);

boolean mdlDialog_publishStructure /* <= TRUE if error */
(
    void      *setP,          /* => symbol set to publish structure in */
    char      *structName    /* => structure "tag" name */
);

```

Description The `mdlDialog_publishBasicVariable` function publishes a basic C data type variable for use in C expression strings.

The `mdlDialog_publishComplexVariable` function publishes a structure type variable for use in C expression strings.

The `mdlDialog_publishBasicPtr` function publishes a pointer to a basic C data type variable for use in C expression strings.

The `mdlDialog_publishComplexPtr` function publishes pointer to a structure type variable for use in C expression strings.

The `mdlDialog_publishStructure` function publishes a structure declaration for use in C expression strings.

The *setP* parameter specifies the symbol set to publish the variable, pointer, or structure into.

The *typeP* parameter points to a basic C data type specifier. It can be `&longType`, `&shortType`, `&charType`, `&ulongType`, `&ucharType`, `&doubleType` or `&voidType`.

The *variableName* parameter is a NULL terminated string that specifies the name by which the variable or pointer will be referenced in C expression strings.

The *variableP* parameter specifies the address of the variable or pointer.

The *structName* parameter is a NULL terminated string that contains the “tag” of a structure. Any structures referenced must also have type declaration resources present in an open resource file. The “See also” section lists the appropriate references for more information on generating these resources.

The `mdlDialog_publish...` functions are usually called near the beginning of an MDL application program. They must be called before any dialog boxes that use C expression strings in their resource definitions are opened.

Some examples:

C variable:	<code>long globalLong;</code>
expression	<code>'globalLong'</code>
function	<code>mdlDialog_publishBasicVariable(setP, &longType, "globalLong", &globalLong);</code>

C variable:	<code>MSStateData statedata;</code>
expression	<code>"statedata.precision"</code>
function	<code>mdlDialog_publishComplexVariable(setP, "msStateData", "statedata", &statedata);</code>

C variable:	<code>long *globalLongP;</code>
expression	<code>"*globalLongP"</code>
function	<code>mdlDialog_publishBasicPtr(setP, &longType, "globalLongP", &globalLongP);</code>

C variable:	<code>tcb *tcb;</code>
expression	<code>"tcb->control.grid_lock"</code>
function	<code>mdlDialog_publishComplexPtr(setP, "tcb", "tcb", &tcb);</code>

C variable	<not applicable>
expression	"((struct toggletest *) itemUserDataP)->mallocedValue"
function	mdlDialog_publishStructure(setP, "toggletest");

Returns The mdlDialog_publishBasicVariable, mdlDialog_publishComplexVariable, mdlDialog_publishBasicPtr, mdlDialog_publishComplexPtr and mdlDialog_publishStructure functions return TRUE if an error occurs.

See Also "Generating Resource Files from C Type Definitions," "Referencing application variables from resource files," "C Expression Handling".

mdlDialog_synonymsSynch

```
#include <dlogitem.h>

boolean mdlDialog_synonymsSynch /* <= TRUE if error */
(
  RscFileHandle rFileH,      /* => NULL means use opened */
  int            synonymsId,  /* => synonyms resource ID */
  void          *ownerMD     /* => owner MDL task */
);
```

Description The mdlDialog_synonymsSynch function forces the appearance of all items in the synonym resource specified by *synonymsId* to match their external state. It does this by calling the mdlDialog_itemSynch function for each occurrence of all the items in the list no matter what dialog box they appear in.

The *rFileH* parameter specifies the resource file to search for the synonym resource. If NULL, all the calling application's open resource files and then MicroStation's open resource files will be searched.

The *ownerMD* parameter should be set to NULL.

Returns The mdlDialog_synonymsSynch function returns TRUE if an error occurs. This means that the synonym resource could not be found.

See Also mdlDialog_itemsSynch, mdlDialog_itemSynch, "Synonym resources", "Dialog item state: internal value versus external state", "Item synchronization".

mdlDialog_setFilterString

```
int mdlDialog_setFilterString
(
  char    *filterString
);
```

Description mdlDialog_setFilterString is used to set the file filter string of a user-defined File Open or File Create dialog box. This, in turn, will control the files that get displayed

in the file name list box. This function must be called while the target dialog box is displayed on the screen (implying that the mdlDialog_setFilterString function call takes place from within a hook function of one of the dialog's items).

filterString is a file filter, for example, "*.dgn".

Returns mdlDialog_setFilterString returns SUCCESS if it can find the target dialog and change its filter string, otherwise it returns ERROR.

See Also mdlDialog_fileOpen, mdlDialog_defFileOpen, mdlDialog_fileCreate, mdlDialog_defFileCreate.

Dialog Box Item Functions

The dialog box item functions manipulate dialog box items. Items can be moved, drawn, hidden, shown, added or deleted using functions described in this section.

The following table lists the dialog box item functions:

Function	Used to
mdlDialog_itemGetState	get the external state of an item.
mdlDialog_itemGetValue	get the internal value of an item.
mdlDialog_itemGetByIndex	get a pointer to an item by specifying its position in the dialog item list.
mdlDialog_itemGetByTypeAndId	get a pointer to an item by specifying its resource type and ID.
mdlDialog_itemDraw	draw an item.
mdlDialog_itemHide	hide an item.
mdlDialog_itemShow	show a previously hidden item.
mdlDialog_itemLoad	load an item into a dialog box.
mdlDialog_itemMove	move an item.
mdlDialog_itemsSwapOrder	swap the positions of two items.
mdlDialog_itemSetEnabledState	set the enabled state (enabled or disabled) of an item.
mdlDialog_itemSetExtent	set the location and size of an item.
mdlDialog_itemSetLabel	set the label of an item.
mdlDialog_itemSetState	set the external state of an item.
mdlDialog_itemSetValue	set the internal value of an item.
mdlDialog_itemSynch	make an item's appearance match its external state.

Function	Used to
<code>mdlDialog_itemsSynch</code>	make the appearance of all the items in a dialog box match their external state.
<code>mdlDialog_itemSynchByTypeAndId</code>	send a synch message to the item matching the given type and ID in a dialog box.
<code>mdlDialog_itemsApply</code>	make the external state of all items in a dialog box match the item's appearance.
<code>mdlDialog_itemsGetNumberOf</code>	get the number of items in a dialog box.
<code>mdlDialog_itemsFree</code>	free all the items in a dialog box.
<code>mdlDialog_itemsLoad</code>	load a list of items into a dialog box.
<code>mdlDialog_focusItemIndexGet</code>	get the index of the item that has the keyboard focus in a dialog box.
<code>mdlDialog_focusItemIndexSet</code>	set the keyboard focus item for a dialog box.
<code>mdlItem_colorChanged</code>	query whether an item's color has been changed from the default.

mdlDialog_itemGetState, mdlDialog_itemGetValue

```
#include dlogitem.h>

boolean mdlDialog_itemGetState /* <= TRUE if error */
(
    int          *formatTypeP,          /* <= NULL, format of item's value */
    ValueUnion   *valueUnionP,          /* <= NULL, underlying app state */
    char         *stringValueP,         /* <= NULL, place for returned str */
    DialogBox    *db,                  /* => dialogBox that contains item */
    int          itemIndex,             /* => item index */
    int          maxStringSize          /* => size of string buffer */
);

boolean mdlDialog_itemGetValue /* <= TRUE if error */
(
    int          *formatTypeP,          /* <= NULL, format of *valueUnionP */
    ValueUnion   *valueUnionP,          /* <= NULL, item's internal value */
    char         *stringValueP,         /* <= NULL, place for returned str */
    DialogBox    *db,                  /* => dialogBox that contains item */
    int          itemIndex,             /* => item index */
    int          maxStringSize          /* => size of string buffer */
);
```

Description The `mdlDialog_itemGetState` function retrieves the specified dialog box item's external state. This is the value of the application data that the item controls, possibly filtered through a mask associated with the item.

The `mdlDialog_itemGetValue` function retrieves the specified dialog box item's internal value. This is the value that is used to determine the item's appearance when the item is drawn.

The *formatTypeP* parameter points to an integer variable which will be set to the type of value returned. Most dialog items types will set this to `FMT_LONG`. In this case, the *valueUnionP* parameter points to a `ValueUnion` variable whose *sLongFormat* member will be set to the item's state or value. The *stringValueP* and *maxStringSize* parameters are not used and can be set to `NULL` and 0, respectively.

Note that with multi-line text item, *valueUnionP* is used to extract the string and *formatTypeP* should be set to `FMT_STRING`.

Text items, however, store their internal state as strings and will set the variable pointed to by the *formatTypeP* parameter to `FMT_STRING`. In this case, *valueUnionP* is not used and should be set to `NULL`. *stringValueP* should then point to the buffer which is to receive the item's string value, and *maxStringSize* should contain the size of that buffer.

The *db* parameter specifies a dialog box.

The *itemIndex* parameter specifies the index of the item whose external state or internal value the function is retrieving. *itemIndex* must be greater than or equal to 0 and less than the number of items in the dialog box. Use `mdlDialog_itemsGetNumberOf` to determine the number of items in a dialog box.

Returns `mdlDialog_itemGetState` and `mdlDialog_itemGetValue` return `TRUE` if an error occurs. Either *db* doesn't point at a dialog box, or *itemIndex* is out of range.

See Also `mdlDialog_itemsGetNumberOf`, "Dialog item state: internal value versus external state".

mdlDialog_itemGetByIndex, mdlDialog_itemGetByTypeAndId

```
#include <dlogitem.h>

DialogItem *mdlDialog_itemGetByIndex /* <= item requested */
(
    DialogBox      *db,           /* => dialogBox that contains item */
    int            itemIndex     /* => index of item to get ptr to */
);
```

```
DialogItem *mdlDialog_itemGetByTypeAndId /* <= item requested */
(
DialogBox      *db,          /* => dialogBox that contains item */
long           type,         /* => type of the item to get */
long           id,           /* => resource ID of the item to get */
int            startingIndex /* => starting item index (usually 0) */
);
```

Description The `mdlDialog_itemGetByIndex` function retrieves a pointer to the requested dialog item contained in the dialog box specified by *db*.

The *itemIndex* parameter specifies the index of the item to retrieve a pointer to. *itemIndex* must be greater than or equal to 0, and less than the number of items in the dialog box. Use the `mdlDialog_itemsGetNumberOf` function to determine the number of items in a dialog box.

The `mdlDialog_itemGetByTypeAndId` retrieves a pointer to the dialog item of *type* and resource ID *id* that is contained in the dialog box specified by *db*. The list of possible values for *type* is contained in `dlogbox.h`. For example, the type of a text item is `RTYPE_Text`.

The *startingIndex* parameter specifies where to start looking for the dialog item. It is usually set to 0, which means start the search from the beginning of the dialog item list.

Generally speaking, to make an MDL application more robust, pointers to items should be obtained by calling the `mdlDialog_itemGetByTypeAndId` function, not `mdlDialog_itemGetByIndex`. In this way, the application does not depend on items being in fixed positions within the dialog item list. New items can be added at the beginning or items removed from the middle of a dialog without affecting program code. Since the dialog item list is searched for the specified item, its position can change if `mdlDialog_itemGetByTypeAndId` is used.

A number of other dialog box functions require item indexes. To make an application more item position independent when an item index is required, first call `mdlDialog_itemGetByTypeAndId`, and then use the

pointer that is returned to determine the item index of the item that is found. For example:

```
{
    ...
    DialogItem *diP;
    int itemIndex;
    RawItemHdr *textP;
    diP=mdlDialog_itemGetByTypeAndId(db, RTYPE_Text,
                                     TEXTID_ElementColor,0);

    itemIndex=diP->itemIndex;
    textP=diP->rawItemP;
    ...
}
```

A number of other dialog box functions require a pointer to a `RawItemHdr` structure not a pointer to a `DialogItem` structure. The above example also shows how to get a pointer to a `RawItemHdr` given a pointer to a `DialogItem`.

Pointers to `DialogItem` structures or `RawItemHdr` structures will remain valid as long as the referenced item exists. The pointers will be valid even if other items are inserted or deleted from the dialog box. A pointer to a `DialogItem` that is retrieved when the item is first created, (obtained upon response to the `DITEM_MESSAGE_CREATE` message) will be valid until the item is destroyed.

Returns The `mdlDialog_itemGetByIndex` and `mdlDialog_itemGetByTypeAndId` functions return a pointer to a `DialogItem` structure, or `NULL` if the specified item is not found.

See Also `mdlDialog_itemsGetNumberOf`, "DialogItem structure".

mdlDialog_itemDraw

```
#include <dlogitem.h>

boolean mdlDialog_itemDraw /* <= TRUE if error */
(
    DialogBox    *db,          /* => dialogBox containing item to draw */
    int          itemIndex    /* => item index of item to draw */
);
```

Description The `mdlDialog_itemDraw` function redraws an item within the dialog box specified by *db*. The item is drawn based on its internal value, not its external state.

itemIndex specifies the index of the item to draw. *itemIndex* must be greater than or equal to 0 and less than the number of items in the dialog box. Use the `mdlDialog_itemsGetNumberOf` function to determine the number of items in a dialog box.

Hidden items can not be drawn. Call `mdlDialog_itemShow` to draw hidden items. Whether an item is hidden or not can be determined by looking at the item's `DialogItem` *attributes.hidden* bit.

Returns The `mdlDialog_itemDraw` function returns `TRUE` if an error occurs. This means that *db* is not a pointer to a dialog box or *itemIndex* is out of range.

See Also `mdlDialog_itemsGetNumberOf`, `mdlDialog_itemGetByIndex`, `mdlDialog_itemGetByTypeAndId`, "Dialog item state: internal value versus external state," "DialogItem structure".

mdlDialog_itemHide

```
#include <dlogitem.h>

boolean mdlDialog_itemHide /* <= TRUE if error */
(
    DialogBox    *db,                /* => dialogBox */
    int          itemIndex,          /* => index of item to hide */
    boolean      ignoreFocusOutErrors /* => usually FALSE */
);
```

Description The `mdlDialog_itemHide` function hides an item within the dialog box *db*. *itemIndex* specifies the index of the item to hide. *itemIndex* must be greater than or equal to 0, and less than the number of items in the dialog box. Use the `mdlDialog_itemsGetNumberOf` function to determine the number of items in a dialog box.

The *ignoreFocusOutErrors* parameter can be used to ignore any focusing out errors (such as the current value being out of range) that may occur if the item to be hidden has the keyboard focus. It should usually be set `FALSE`, which indicates that focus out errors should not be ignored.

Whether an item is hidden or not can be determined by looking at the item's `DialogItem` *attributes.hidden* bit.

Returns The `mdlDialog_itemHide` function returns `TRUE` if an error occurs. This means that *db* is not a pointer to a dialog box, *itemIndex* is out of range, or the item has the focus and the focus can not be removed (because the current value is out of range).

See Also `mdlDialog_itemShow`, `mdlDialog_itemsGetNumberOf`, `mdlDialog_itemGetByIndex`, `mdlDialog_itemGetByTypeAndId`, "DialogItem structure," "Keyboard focus".

mdlDialog_itemShow

```
#include <dlogitem.h>
boolean mdlDialog_itemShow /* <= TRUE if error */
(
    DialogBox    *db,           /* => dialogBox containing item to show */
    int          itemIndex     /* => index of item to show */
);
```

Description The mdlDialog_itemShow function shows a previously hidden item within the dialog box specified by *db*.

itemIndex specifies the index of the item to show. *itemIndex* must be greater than or equal to 0, and less than the number of items in the dialog box. Use mdlDialog_itemsGetNumberOf to determine the number of items in a dialog box.

Whether an item is hidden or not can be determined by looking at the item's DialogItem *attributes.hidden* bit.

Returns The mdlDialog_itemShow function returns TRUE if an error occurs. This means that *db* is not a pointer to a dialog box or *itemIndex* is out of range.

See Also mdlDialog_itemHide, mdlDialog_itemsGetNumberOf, "DialogItem structure".

mdlDialog_itemLoad

```
#include <dlogitem.h>
DialogItem *mdlDialog_itemLoad /* <= NULL if error */
(
    DialogBox    *db,           /* => dialogBox to contain item */
    DialogItemRsc *diRP,       /* => item to load */
    RscFileHandle rFileH,      /* => NULL=use opened rsc files */
    void         *ownerMD,     /* => should be NULL */
    int          beforeItemIndex, /* => item to load before */
    Point2d      *originP      /* => loads relative to this pt */
);
```

Description The mdlDialog_itemLoad function loads the item specified by data pointed at by *diRP* into the dialog box specified by *db*.

The *diRP* parameter points at a DialogItemRsc structure that specifies the item to load. The fields of the DialogItemRsc structure should be set just as if a definition in a resource file were being created.

The *rFileH* parameter specifies the resource file to search for the dialog box item resource. If NULL, all the calling application's open resource files, then MicroStation's open resource files will be searched.

The *ownerMD* parameter should be set to NULL.

The *beforeItemIndex* parameter specifies the index of the item before which the new item will be loaded. Specify -1 to indicate that the item should be appended at the end of the dialog item list.

The *originP* parameter is used to specify the origin of a temporary coordinate system that will be used when loading the dialog item. The units of the point pointed at by *originP* should be in dialog coordinate units. The `extent` field of the `DialogItemRsc` structure pointed at by *diRP* will be interpreted with an origin at *originP*. Specify `NULL` for *originP* to indicate the origin should be at (0, 0), the upper left corner of the dialog box.

Returns The `mdlDialog_itemLoad` function returns a pointer to the dialog item that is loaded or `NULL` if an error occurs. This means that *db* is not a pointer to a dialog box or the item resource could not be found.

See Also `mdlDialog_itemsLoad`, “DialogItemRsc Structure”.

mdlDialog_itemMove

```
#include <dlogitem.h>

boolean mdlDialog_itemMove /* <= TRUE if error */
(
    DialogBox    *db,           /* => dialogBox containing item to move */
    int          itemIndex,     /* => index of item to move */
    Point2d      *ptP,          /* => where to move item to */
    boolean      redraw         /* => TRUE means redraw item after move */
);
```

Description `mdlDialog_itemMove` moves an item within the dialog box specified by *db*.

itemIndex specifies the index of the item to move. *itemIndex* must be greater than or equal to 0, and less than the number of items in the dialog box. Use `mdlDialog_itemsGetNumberOf` to determine the number of items in a dialog box.

The *ptP* parameter points at a variable that specifies the new location of the item. Positive values for the *x* or *y* member of `Point2d` indicate that dialog coordinate units are being specified. Negative values should be used to specify the new location in pixels. In general, pixel coordinates should be avoided since problems may occur if the dialog box is moved to a screen that uses a different text font size.

If the *redraw* parameter is `TRUE`, the item will be drawn in its new location.

Returns The `mdlDialog_itemMove` function returns `TRUE` if an error occurs. This means that *db* is not a pointer to a dialog box or *itemIndex* is out of range.

See Also `mdlDialog_itemsGetNumberOf`.

mdlDialog_itemsSwapOrder

```
#include <dlogitem.h>

boolean mdlDialog_itemsSwapOrder /* <= TRUE if error */
(
    DialogBox    *dbP,          /* => dialog to swap items in */
    int          itemIndex1,    /* => item to swap */
    int          itemIndex2     /* => item to swap it with */
);
```

Description mdlDialog_itemsSwapOrder is used to swap the positions of two dialog items in a dialog box. This does not affect their display coordinates within the dialog box.

dbP points to the dialog box that contains the items.

itemIndex1 and *itemIndex2* identify the items to be swapped.

Returns mdlDialog_itemsSwapOrder returns SUCCESS if the items could be swapped.

mdlDialog_itemSetEnabledState

```
#include <dlogitem.h>

boolean mdlDialog_itemSetEnabledState /* <= TRUE if error */
(
    DialogBox    *db,          /* => dialogBox that contains item */
    int          itemIndex,    /* => index of item to set enabled */
    boolean      enabled,      /* => FALSE means disable item */
    boolean      ignoreFocusOutErrors /* => usually FALSE */
);
```

Description mdlDialog_itemSetEnabledState sets the enabled state (enabled or disabled) of an item within the dialog box specified by *db*.

The *itemIndex* parameter specifies the index of the item to hide. *itemIndex* must be greater than or equal to 0, and less than the number of items in the dialog box. Use mdlDialog_itemsGetNumberOf to determine the number of items in a dialog box.

The *enabled* parameter should be set to TRUE to enable the item, or FALSE to disable the item. The user can interact with an enabled item; a disabled item is drawn with dim text (if any) and ignores mouse presses and keyboard events.

The *ignoreFocusOutErrors* parameter can be used to ignore any focusing out errors (such as the current value being out of range) that may occur if the item to be disabled has the keyboard focus. It should usually be set FALSE which indicates that focus out errors should not be ignored.

Returns mdlDialog_itemSetEnabledState returns TRUE if an error occurs. This means that *db* is not a pointer to a dialog box, *itemIndex* is out of range, or the item has the focus and the focus can not be removed (because the current value is out of range).

See Also `mdlDialog_itemsGetNumberOf`, “Keyboard focus”.

mdlDialog_itemSetExtent

```
#include <dlogitem.h>

boolean mdlDialog_itemSetExtent /* <= TRUE if error */
(
    DialogBox    *db,           /* => dialogBox that contains item */
    int           itemIndex,    /* => index of item to set extent of */
    Sextent       *sextentP,    /* => new extent of item */
    boolean       redraw        /* => TRUE=item redrawn after resizing */
);
```

Description The `mdlDialog_itemSetExtent` function sets the extent of an item within the dialog box specified by *db*.

The *itemIndex* parameter specifies the index of the item to set the extent of. *itemIndex* must be greater than or equal to 0, and less than the number of items in the dialog box. Use the `mdlDialog_itemsGetNumberOf` function to determine the number of items in a dialog box.

The *sextentP* parameter points at a variable that specifies the new location and size of the item. Positive values for the *origin.x*, *origin.y*, *width* or *height* fields of *Sextent* indicate that dialog coordinate units are being specified. Negative values should be used to specify the new location or dimension in pixels. In general, pixel coordinates should be avoided since problems may occur if the dialog box is moved to a screen that uses a different text font size.

If *redraw* is `TRUE` (as it usually is), the item will be drawn in its new location.

Returns The `mdlDialog_itemSetExtent` function returns `TRUE` if an error occurs. This means that *db* is not a pointer to a dialog box or *itemIndex* is out of range.

See Also `mdlDialog_itemsGetNumberOf`.

mdlDialog_itemSetLabel

```
#include <dlogitem.h>

boolean mdlDialog_itemSetLabel /* <= TRUE if error */
(
    DialogBox    *db,           /* => dialogBox that contains item */
    int           itemIndex,    /* => index of item to set label */
    char         *stringP       /* => new label */
);
```

Description The `mdlDialog_itemSetLabel` function sets the label of an item within the dialog box specified by *db*.

The *itemIndex* parameter specifies the index of the item to set the label of. *itemIndex* must be greater than or equal to 0 and less than the number of items in the dialog box. Use the `mdlDialog_itemsGetNumberOf` function to determine the number of items in a dialog box.

stringP points at the NULL terminated string that will be the item's new label.

Returns The `mdlDialog_itemSetLabel` function returns TRUE if an error occurs. Either *db* is not a pointer to a dialog box, or *itemIndex* is out of range.

See Also `mdlDialog_itemsGetNumberOf`.

mdlDialog_itemSetState, mdlDialog_itemSetValue

```
#include <dlogitem.h>

boolean mdlDialog_itemSetState /* <= TRUE if error */
(
    boolean      *stateChangedP,      /* <= state changed, NULL = ignored */
    DialogBox    *db,                  /* => dialogBox that contains item */
    int          itemIndex              /* => index of item to set state */
);

boolean mdlDialog_itemSetValue /* <= TRUE if error */
(
    boolean      *valueChangedP,      /* <= TRUE=new value was different */
    int          formatType,          /* => format of *valueUnionP */
    ValueUnion   *valueUnionP,        /* => item's new internal value */
    char         *stringValueP,       /* => item's new string value */
    DialogBox    *db,                  /* => dialogBox that contains item */
    int          itemIndex            /* => item index */
);
```

Description `mdlDialog_itemSetState` forces the specified dialog box item's external state to match its internal value. This is the opposite behavior of `mdlDialog_itemSynch`, which forces the appearance and internal value of an item to match its external state.

The *stateChangedP* parameter points at an integer variable which is set to TRUE if the item's internal value was different than its external state.

If the item has a synonym item list resource attached to it, synchronize messages will be sent to all items in the list.

The `mdlDialog_itemSetValue` function sets the internal value of the specified dialog box item. This is the value that is used to determine the item's appearance when the item is drawn. It does not affect the item's external state.

The *valueChangedP* parameter points at an integer variable which is set to TRUE if the new value is different than the item's old internal value.

The *formatType* parameter is ignored and can be set to 0.

Most dialog items store their internal value as a signed long integer and the *valueUnionP* parameter should point to a `ValueUnion` variable whose *sLongFormat* member contains the item's new internal value. In this case, the *stringValueP* parameter is also not used and can be set to NULL.

Text items, however, store their internal value as strings; *valueUnionP* is not used and should be set to NULL. If the internal value of a text item is being set, *stringValueP* should point to a buffer which contains the item's new internal string value.

The *db* parameter specifies a dialog box.

The *itemIndex* parameter specifies the index of the item whose external state or internal value the function is setting. *itemIndex* must be greater than or equal to 0 and less than the number of items in the dialog box. Use the `mdlDialog_itemsGetNumberOf` function to determine the number of items in a dialog box.

Returns The `mdlDialog_itemSetState` and `mdlDialog_itemSetValue` functions return TRUE if an error occurs. Either *db* doesn't point at a dialog box, or *itemIndex* is out of range.

See Also `mdlDialog_itemsGetNumberOf`, `mdlDialog_itemSynch`, "Dialog item state: internal value versus external state", "Synonym resources".

mdlDialog_itemSynch

```
#include <dlogitem.h>

boolean mdlDialog_itemSynch /* <= TRUE if error */
(
    DialogBox    *db,           /* => dialogBox with item to synch */
    int          itemIndex     /* => index of item to synch */
);
```

Description The `mdlDialog_itemSynch` function forces the appearance of an item to match its external state. This is the opposite behavior of `mdlDialog_itemSetState`, which forces the external state of an item to match its appearance.

The *db* parameter specifies a dialog box.

The *itemIndex* parameter specifies the index of the item to synchronize. *itemIndex* must be greater than or equal to 0, and less than the number of items in the dialog box. Use the `mdlDialog_itemsGetNumberOf` function to determine the number of items in a dialog box.



Be careful when using `mdlDialog_itemSynch` from the `DIALOG_MESSAGE_SYNCH` case of a dialog box hook. If you do not take any

special precautions this will result in a recursive infinite loop and corrupt your stack.

Returns The mdlDialog_itemsSynch function returns `TRUE` if an error occurs. This means that *db* is not a pointer to a dialog box, or *itemIndex* is out of range.

See Also mdlDialog_itemSetState, mdlDialog_itemsGetNumberOf, "Item synchronization".

mdlDialog_itemsSynch

```
#include <dlogitem.h>

boolean mdlDialog_itemsSynch /* <= TRUE if error */
(
    DialogBox    *db          /* => dialog to synch */
);
```

Description mdlDialog_itemsSynch forces the appearance of all items in the dialog box specified by *db* to match their external state. It does this by calling mdlDialog_itemSynch for each visible item.

Returns The mdlDialog_itemsSynch function returns `TRUE` if an error occurs. This means that *db* is not a pointer to a dialog box.

See Also mdlDialog_itemSynch, mdlDialog_itemsApply, "Dialog item state: internal value versus external state", "Item synchronization".

mdlDialog_itemSynchByTypeAndId

```
boolean mdlDialog_itemSynchByTypeAndId /* <= TRUE if error */
(
    DialogBox    *dbP,          /* => dBox that contains item to synch */
    ULong        itemType,      /* => type of item to synch */
    int          itemId         /* => id of item to synch */
);
```

Description mdlDialog_itemSynchByTypeAndId causes a `DITEM_MESSAGE_SYNCHRONIZE` message to be sent to the dialog item indicated by *itemType* and *itemId* in the dialog box indicated by *dbP*.

dbP points to the dialog box containing the item to be synched.

itemType indicates the type or resource class of the item.

itemId is the resource ID of the item.

Returns mdlDialog_itemSynchByTypeAndId returns `SUCCESS` if the item could be found and a synch message could be sent to it.

mdlDialog_itemsApply

```
#include <dlogitem.h>
boolean mdlDialog_itemsApply /* <= TRUE = error (outOfRange) */
(
  DialogBox    *db          /* => dialogBox to send messages to */
);
```

Description The mdlDialog_itemsApply function forces the external state of all enabled items in the dialog box specified by *db* to match their appearance. It does this by calling the mdlDialog_itemSetState function for each enabled item.

To insure that the current input focus item (if any) within the dialog contains a valid value, the focus is removed and then restored to that item. This causes the contents of the item to be validated as part of the standard focus out processing.

This function is called automatically when the standard OK push button is activated by the user. Most MDL dialog programmers will never need to call this function.

Returns The mdlDialog_itemsApply function returns TRUE if an error occurs. This means that *db* is not a pointer to a dialog box, or the current input focus item (if any) within the dialog contains a value that is out of range.

See Also mdlDialog_itemSetState, mdlDialog_itemsSynch, "Dialog item state: internal value versus external state", "Keyboard focus".

mdlDialog_itemsGetNumberOf

```
#include <dlogitem.h>
int mdlDialog_itemsGetNumberOf /* <= # of items in dialogBox */
(
  DialogBox    *db          /* => dialogBox to get # of items in */
);
```

Description The mdlDialog_itemsGetNumberOf function retrieves the number of dialog items contained by the dialog box specified by *db*.

Returns The mdlDialog_itemsGetNumberOf function returns the number of dialog items contained in the specified dialog box. It returns -1 if *db* does not point at a dialog box.

mdlDialog_itemsFree

```
#include <dlogitem.h>
boolean mdlDialog_itemsFree /* <= TRUE if error */
(
    DialogBox      *db,          /* => dialogBox w/ items to free */
    int             startItemIndex, /* => starting index of items */
    int             maxItemIndex   /* => last index of items */
);
```

Description The mdlDialog_itemsFree function frees all the specified dialog items from the dialog box specified by *db*.

The *startItemIndex* and *maxItemIndex* parameters specify the starting and ending indexes of the items to free. *startItemIndex* and *maxItemIndex* must be greater than or equal to 0, and less than the number of items in the dialog box. Use mdlDialog_itemsGetNumberOf to determine the number of items in a dialog box.

Returns The mdlDialog_itemsFree function returns TRUE if an error occurs. This means that *db* is not a pointer to a dialog box.

See Also mdlDialog_itemsGetNumberOf.

mdlDialog_itemsLoad

```
#include <dlogitem.h>
boolean mdlDialog_itemsLoad /* <= TRUE if error */
(
    DialogBox      *db,          /* => dialogBox in which to load items */
    DialogItemListRsc *dilRP,    /* => list of items to load */
    RscFileHandle   rFileH,      /* => NULL = use opened rsc files */
    void           *ownerMD,     /* => should be NULL */
    int             beforeItemIndex, /* => item to load before */
    Point2d        *originP,     /* => load relative to this pt */
    boolean         drawItems     /* => TRUE = draw items after loading */
);
```

Description The mdlDialog_itemsLoad function loads the items specified by data pointed at by *dilRP* into the dialog box specified by *db*.

The *dilRP* parameter points at a DialogItemListRsc structure that specifies the items to load. A DialogItemListRsc structure (defined in dlogbox.h) contains the number of items to load and an array of DialogItemRsc structures. The fields of the DialogItemRsc structures should be set just as if a definition in a resource file was being created.

The *rFileH* parameter specifies the resource file to search for the dialog box item resources. If NULL, all the calling application's open resource files, then MicroStation's open resource files will be searched.

The *ownerMD* parameter should be set to NULL.

The *beforeItemIndex* parameter specifies the index of the item before which the new items will be loaded. Specify -1 to indicate that the items should be appended at the end of the dialog item list.

The *originP* parameter is used to specify the origin of a temporary coordinate system that will be used when loading the dialog items. The units of the point pointed at by *originP* should be in dialog coordinate units. The *extent* field of the `DialogItemRsc` structures contained by the structure pointed at by *dilRP* will be interpreted with an origin at *originP*. Specify `NULL` for *originP* to indicate the origin should be at (0, 0), the upper left corner of the dialog box.

If the *drawItems* parameter is `TRUE`, the items will be drawn after they are loaded.

Returns The `mdlDialog_itemsLoad` function returns `TRUE` if an error occurs. This means that *db* is not a pointer to a dialog box, or one of item resources could not be found.

See Also `mdlDialog_itemLoad`, “DialogItemRsc Structure”.

mdlDialog_focusItemIndexGet, mdlDialog_focusItemIndexSet

```
#include <dlogitem.h>

int mdlDialog_focusItemIndexGet /* <= current focus item */
(
    DialogBox    *db                /* => dialogBox whose focus item to get */
);

boolean mdlDialog_focusItemIndexSet /* <= TRUE if error */
(
    DialogBox    *db,                /* => dialogBox we're setting */
    int          itemIndex,          /* => index of new focus item */
    boolean      ignoreFocusOutErrors /* => usually FALSE */
);
```

Description The `mdlDialog_focusItemIndexGet` function retrieves the index of the dialog item which currently has the keyboard focus in the dialog box specified by *db*. Note that the item will only actually have the keyboard focus if the dialog box has the focus. Otherwise, it is the item which will gain the focus the next time the dialog box becomes the keyboard focus dialog box.

The `mdlDialog_focusItemIndexSet` function sets the keyboard focus item in the dialog box specified by *db*. When the dialog box is the keyboard focus dialog box, that item will receive user keystrokes.

The *itemIndex* parameter specifies the new keyboard focus item index. *itemIndex* must be greater than or equal to 0 and less than the number of items in the dialog box. Use the `mdlDialog_itemsGetNumberOf` function to determine the number of items in a dialog box. The type of item specified must be able to accept user keystrokes.

The *ignoreFocusOutErrors* parameter can be used to ignore any errors (such as the current value being out of range) that may occur when taking the focus away from the current focus item. It should usually be set to `FALSE` which indicates that focus out errors should not be ignored.



`mdlDialog_focusItemIndexSet` will not let you set the focus to a specific item on a `DITEM_MESSAGE_FOCUSOUT`; attempting to do so will put you into a focus out loop. Instead, you should set `nextFocusItemIndex` in the dialog item message structure to the index of the item you want the focus to be in. For example:

```
case DITEM_MESSAGE_FOCUSOUT:
    dimP->u.focusOut.nextFocusItemIndex = 4
```

Returns `mdlDialog_focusItemIndexGet` returns the index of the item that currently has the keyboard focus. It returns -1 if no item within the dialog box has the focus.

`mdlDialog_focusItemIndexSet` returns `TRUE` if an error occurs. This means that *db* is not a pointer to a dialog box, *itemIndex* is out of range, or the focus can not be removed from the current focus item.

See Also `mdlDialog_itemsGetNumberOf`, “Keyboard focus”.

mdlItem_colorChanged [ditemlib.ml]

```
boolean mdlItem_colorChanged /* <= TRUE if no longer default */
(
    RawItemHdr *riP,
    int         itemColorType /* => DITEM_COLORTYPE_BACKGROUND, etc */
);
```

Description The `mdlItem_colorChanged` function queries whether an item’s color has been changed from the default, and if so, gives the color type to which it has been changed.

riP points to the raw item header of the dialog item in question, and *itemColorType* indicates the color of the item.

Returns `mdlItem_colorChanged` returns `FALSE` if the item’s color is unchanged, or a non-zero value if has been changed.

Dialog Box Drawing Functions

The dialog box drawing functions draw into dialog boxes. Diamonds, ellipses, arrows and text can all be drawn using the functions in this section.

A number of these functions make references to color indexes. See the “Color” section in the “Dialog Box Manager Basic Concepts” chapter for more information on specifying color in dialog boxes.

The following table lists the dialog box drawing functions:

Function	Used to
<code>mdlDialog_toPixels</code>	convert dialog coordinate units to pixels.
<code>mdlDialog_toDCoord</code>	convert pixels to dialog coordinate units.
<code>mdlDialog_diamondDrawBeveled</code>	draw a beveled diamond.
<code>mdlDialog_diamondFill</code>	draw a filled diamond.
<code>mdlDialog_ellipseFill</code>	draw a filled ellipse.
<code>mdlDialog_lineStyleSet</code>	set the color, style and line weight for subsequent drawing.
<code>mdlDialog_scrollArrowDraw</code>	draw a scroll-bar arrow.
<code>mdlDialog_stringWidth</code>	get the width of a string.
<code>mdlDialog_stringnWidth</code>	get the width of the first n chars of a string.
<code>mdlDialog_textDraw</code>	draw a text string.
<code>mdlDialog_textDrawN</code>	draw the first n characters of a text string.
<code>mdlDialog_textDrawCD</code>	draw text to a dialog, color provided by a color descriptor.
<code>mdlDialog_textDrawNCD</code>	same as <code>mdlDialog_textDrawCD</code> but with a maximum character length.
<code>mdlDialog_underlineDraw</code>	underline text in a dialog.
<code>mdlDialog_underlineDrawCD</code>	underline text in a dialog in a color provided by a color descriptor.

mdlDialog_toPixels

```
#include <dlogitem.h>

int mdlDialog_toPixels /* <= pixels */
(
    DialogBox    *db,      /* => dialogBox where conversion takes place */
    int          dialogCoord /* => dialog coord to convert */
);
```

Description The mdlDialog_toPixels function converts a value that is in dialog coordinate units to the corresponding pixel value. The font currently associated with the dialog box is used in the calculation.

Returns The mdlDialog_toPixels function returns the result of converting to pixels a value originally in dialog coordinate units.

See Also mdlDialog_toDCoord, the “Specifying coordinates” section of the Dialog Overview chapter of the MDL Programmer’s Guide.

mdlDialog_toDCoord

```
#include <dlogitem.h>

int mdlDialog_toDCoord /* <= dialog coordinates */
(
    DialogBox    *db,          /* => where conversion takes place */
    int          pixels        /* => pixels to convert */
);
```

Description mdlDialog_toDCoord converts a value that is in pixels to the corresponding dialog coordinate units value. The font currently associated with the dialog box is used in the calculation.

Returns The mdlDialog_toDCoord function returns the result of converting to dialog coordinate units a value originally in pixels.

See Also mdlDialog_toPixels, the “Specifying coordinates” section of the Dialog Overview chapter of the MDL Programmer’s Guide.

mdlDialog_diamondDrawBeveled, mdlDialog_diamondFill

```
#include <dlogitem.h>

void mdlDialog_diamondDrawBeveled
(
    DialogBox    *db,          /* => dialogBox to draw diamond in */
    Point2d      *ptP,         /* => upper left corner of diamond */
    int          size,         /* => WARNING, must be odd */
    boolean      raised        /* => FALSE diamond looks recessed */
);

void mdlDialog_diamondFill
(
    DialogBox    *db,          /* => dialogBox to draw diamond in */
    Point2d      *ptP,         /* => upper left corner of diamond */
    int          size,         /* => WARNING, must be odd */
    int          colorIndex    /* => fixed color index fill color */
);
```

Description The `mdlDialog_diamondDrawBeveled` function draws a diamond with beveled edges in the dialog box specified by *db*.

If the *raised* parameter is `TRUE`, the diamond appears raised from the surface of the dialog box. Otherwise it appears recessed into the dialog box surface.

The `mdlDialog_diamondFill` function draws a filled diamond of color *colorIndex* in the dialog box specified by *db*.

The *ptP* parameter specifies the location (in pixels) of the upper left corner of a square of size *size* which surrounds where the diamond will be drawn.

Returns These functions are of type `void`. They return no value.

mdlDialog_ellipseFill

```
#include <dlogitem.h>

void mdlDialog_ellipseFill
(
    DialogBox    *db,           /* => dialog to draw in */
    Dpoint2d     *originP,      /* => center point of ellipse */
    double       primary,       /* => primary axis */
    double       secondary,     /* => secondary axis */
    double       rotation,      /* => rotation of entire ellipse */
    int          colorIndex     /* => fixed color index fill color */
);
```

Description The `mdlDialog_ellipseFill` function draws a filled ellipse of color *colorIndex* in the dialog box specified by *db*.

The origin of the ellipse is given in a double-precision 2D point, *originP*. The primary axis (the axis at 0° in the arc/ellipse coordinate system) is passed in *primary*, and the secondary axis (the axis at 90°) is passed in *secondary*. You can produce a circle by setting *primary* equal to *secondary*. The entire ellipse can be rotated by *rotation* degrees.

Returns The `mdlDialog_ellipseFill` function is of type `void`. It returns no value.

See Also `mdlWindow_ellipseFill`.

mdlDialog_lineStyleSet

```
#include <dlogitem.h>
void mdlDialog_lineStyleSet
(
  DialogBox    *db,           /* => dialogBox to set lineStyle in */
  int          pattern,       /* => index into pattern array */
  int          colorIndex,    /* => line fixed color index */
  int          mode,          /* => drawing mode: 0=OR, 1=XOR */
  int          lweight        /* => line weight */
);
```

Description The `mdlDialog_lineStyleSet` function sets the characteristics of the graphics that are subsequently drawn. The *db* argument designates the dialog box that you intend to draw to. On some platforms (such as the PC), the graphic attributes are shared for all windows on a given screen and are not maintained for each window individually. Therefore, as a general rule, set the graphic attributes when you draw on the screen.

pattern specifies an on/off pattern for the line to be drawn. Possible values are 0 through 7, resulting in solid, dotted, medium-dashed, long-dashed, dot-dashed short-dashed, dash double-dot and long dash-short dashlines, respectively.

colorIndex specifies a color index for the graphics. Its usual value is `WHITE_INDEX`, `BLACK_INDEX`, `LGREY_INDEX` or `DGREY_INDEX`. See `msdefs.h` for the complete list of possible color indexes.

mode determines whether the lines will be drawn on the screen (mode 0), or combined with the current screen contents using the XOR (exclusive OR) operation (mode 1).

lweight specifies the line thickness. Possible values are 0 to 31. Sometimes two adjacent line thicknesses will display the same on the screen because of the limited resolution available on some displays. (For example 0 and 1 may set only one pixel).

Returns The `mdlDialog_lineStyleSet` function is of type `void`. It returns no value.

See Also `mdlDialog_textDraw`.

mdlDialog_scrollArrowDraw

```
#include <dlogitem.h>
void mdlDialog_scrollArrowDraw
(
DialogBox    *db,           /* => dialogBox to draw in */
Point2d      *ptP,         /* => upper left corner of arrow */
int          size,         /* => size of arrow base (must be odd) */
boolean      raised,       /* => FALSE arrow looks recessed */
boolean      fillBlack,    /* => usually FALSE */
boolean      fat,          /* => TRUE = two pixel edge */
int          type          /* => SCROLL_ARROW_UP, SCROLL_ARROW_DOWN, etc */
);
```

Description The `mdlDialog_scrollArrowDraw` function draws a scroll-bar-item-like arrow in the dialog box specified by *db*.

The *ptP* parameter specifies the location (in pixels) of the upper left corner of a square of size *size* which surrounds where the arrow will be drawn.

If the *raised* parameter is `TRUE`, the arrow appears raised from the surface of the dialog box. Otherwise it appears recessed into the dialog box surface.

If the *fillBlack* parameter is `TRUE`, the inside of the arrow is drawn in black instead of light gray. *fillBlack* is usually `FALSE`.

If the *fat* parameter is `TRUE`, the arrow will have two pixel wide edges. Otherwise, the edges will only be one pixel wide.

The *type* parameter indicates the type of arrow to draw. Its value can be `SCROLL_ARROW_UP`, `SCROLL_ARROW_DOWN`, `SCROLL_ARROW_LEFT` or `SCROLL_ARROW_RIGHT`.

Returns The `mdlDialog_scrollArrowDraw` function is of type `void`. It returns no value.

See Also `mdlDialog_scrollBarSetRange`.

mdlDialog_stringWidth, mdlDialog_stringnWidth

```
#include <dlogitem.h>

int mdlDialog_stringWidth /* <= string width in pixels */
(
DialogBox    *db,           /* => dialogBox string will be drawn in */
int          fontIndex,    /* => -1 = use font associated with db */
char         *stringP      /* => string to measure */
);

int mdlDialog_ *stringP, /* => string to measure */
int          nChars      /* => # of chars to measure */
);
```

Description The mdlDialog_stringWidth function returns the width in pixels of the string pointed at by *stringP*.

The mdlDialog_stringnWidth function returns the width in pixels of the first *nChars* characters of the string pointed at by *stringP*.

db specifies which dialog box the string would be drawn in.

The *fontIndex* parameter specifies the font the string would be drawn in. -1 indicates the font currently associated with the dialog box should be used.

Returns mdlDialog_stringWidth returns the width in pixels of the string pointed at by *stringP*.

mdlDialog_stringnWidth returns the width in pixels of the first *nChars* characters of the string pointed at by *stringP*.

See Also "Text font".

mdlDialog_textDraw, mdlDialog_textDrawN

```
#include <dlogitem.h>

void mdlDialog_textDraw
(
    DialogBox    *db,           /* => dialogBox to draw in */
    Point2d      *ptP,          /* => upper left of drawing area */
    BSIRect      *clipRectP,     /* => clip rectangle (local coords) */
    char         *stringP,       /* => string to draw */
    boolean      dimText        /* => TRUE = text is "dimmed" */
);

void mdlDialog_textDrawN
(
    DialogBox    *db,           /* => dialogBox to draw in */
    Point2d      *ptP,          /* => upper left of drawing area */
    BSIRect      *clipRectP,     /* => clip rectangle (local coords) */
    char         *stringP,       /* => string to draw */
    boolean      dimText,        /* => TRUE = text is "dimmed" */
    int          nChars          /* => number of chars to draw */
);
```

Description The mdlDialog_textDraw function draws the string pointed at by *stringP* into the dialog box pointed at by *db*.

The mdlDialog_textDrawN function draws the first *nChars* characters of the string pointed at by *stringP* into the dialog box pointed at by *db*.

The *ptP* parameter specifies the upper left corner of where the text will be drawn.

A clipping rectangle can be specified with *clipRectP*, or the value `NULL` causes the routine to use the dialog box's content rectangle.

If *dimText* is `TRUE`, the text is drawn with a “dimmed” appearance. On color screens this is a medium gray color, on monochrome screens the text looks half-toned. If *dimText* is `FALSE`, the text is drawn in black.

The text is always drawn using the font currently associated with the dialog box.

Returns `mdlDialog_textDraw` and `mdlDialog_textDrawN` are of type `void`. They return no value.

See Also `mdlDialog_textDrawCD`, `mdlDialog_textDrawNCD`, `mdlDialog_fontIndexGet`, `mdlDialog_fontIndexSet`, `mdlDialog_fontGetInfo`, “Text font”.

mdlDialog_textDrawCD, mdlDialog_textDrawNCD

```
#include <mcolor.h>
#include <dlogitem.h>

void mdlDialog_textDrawCD
(
    DialogBox    *dbP,          /* => dialogBox to draw in */
    Point2d      *ptP,          /* => upper left of drawing area */
    BSIRect      *clipRectP,    /* => clip rectangle */
    char         *stringP,      /* => string to draw */
    boolean      dimText,       /* => TRUE=text is "dimmed" */
    BSIColorDescr *foregroundP, /* => fg color descr, NULL=default */
    BSIColorDescr *backgroundP, /* => bg color descr, NULL=default */
    BSIColorDescr *dimColorP,   /* => dimText color, NULL=default */
    RawItemHdr   *riP           /* => default color item */
);

void mdlDialog_textDrawNCD
(
    DialogBox    *dbP,          /* => dialogBox to draw in */
    Point2d      *ptP,          /* => upper left of drawing area */
    BSIRect      *clipRectP,    /* => clip rectangle */
    char         *stringP,      /* => string to draw */
    boolean      dimText,       /* => TRUE=text is "dimmed" */
    int          nChars,        /* => number of chars to draw */
    BSIColorDescr *foregroundP, /* => fg color descr, NULL=default */
    BSIColorDescr *backgroundP, /* => bg color descr, NULL=default */
    BSIColorDescr *dimColorP,   /* => dimText color, NULL=default */
    RawItemHdr   *riP           /* => default color item */
);
```

Description `mdlDialog_textDrawCD` is used to draw text in a dialog box with specific foreground and background colors. `mdlDialog_textDrawNCD` is identical in function to `mdlDialog_textDrawCD` except that the string being drawn may be truncated to a maximum length.

dbP is the dialog to draw in.

ptP indicates the position for the text.

clipRectP is a clipping rectangle (any drawing outside this rectangle is suppressed).

stringP is the text to be drawn.

dimText indicates that the text should be dimmed.

nChars indicates the maximum number of characters in *stringP* to draw.

foregroundP specifies the foreground color for the text. *backgroundP* specifies the background color for the text.

dimColorP specifies the color to use for dim text. Only valid if *dimText* is TRUE.

riP is only used if *foregroundP* or *backgroundP* is NULL. In that case, it indicates a dialog item from which to use get the default foreground or background color.

Returns mdlDialog_textDrawCD and mdlDialog_textDrawNCD have no return value.

See Also mdlDialog_textDraw, mdlDialog_textDrawN.

mdlDialog_underlineDraw

```
#include <mscolor.h>
#include <dlogitem.h>

void mdlDialog_underlineDraw
(
    DialogBox    *dbP,           /* => dialog w/ text to underline */
    Point2d      *ptP,           /* => location of string */
    char         *stringP,        /* => string to underline */
    int          charIndex,       /* => index of char to underline */
    int          fontIndex,       /* => string's font */
    boolean      dimLine          /* => TRUE=dimColor, FALSE=fg color */
);
```

Description mdlDialog_underlineDraw is used to underline text in a dialog box.

dbP points to the dialog where the text is to be underlined.

ptP is the position of the text, *stringP* (not necessarily the underlining).

stringP is the string of which all or part is to be underlined.

charIndex is the index of the first character in *stringP* to be underlined.

fontIndex specifies the font being used for the string. Usually this will be FONT_INDEX_DIALOG.

dimLine indicates the type of color to be used for drawing the underline. If TRUE, the default dim color is used, otherwise the default foreground color is used.

Returns `mdlDialog_underlineDraw` has no return value.

See Also `mdlDialog_underlineDrawCD`.

mdlDialog_underlineDrawCD

```
#include <mcolor.h>
#include <dlogitem.h>

void mdlDialog_underlineDrawCD
(
DialogBox      *dbP,           /* => dialog w/ text to underline */
Point2d        *ptP,           /* => location of string */
char           *stringP,       /* => string to underline */
int            charIndex,      /* => index of char to underline */
int            fontIndex,      /* => string's font */
boolean        dimLine,        /* => TRUE=dimColor, FALSE=foreground */
BSIRect        *clipRectP,     /* => clip rect */
BSIColorDescr  *foregroundP,    /* => fg color descr, NULL= default */
BSIColorDescr  *dimColorP,     /* => color if dim, NULL= default */
RawItemHdr     *riP           /* => item to use for default colors */
);
```

Description `mdlDialog_underlineDrawCD` is used to underline text in a dialog box with a particular color.

dbP points to the dialog where the text is to be underlined.

ptP is the position of the text, *stringP* (not necessarily the underlining).

stringP is the string of which all or part is to be underlined.

charIndex is the index of the first character in *stringP* to be underlined.

fontIndex specifies the font being used for the string. Usually this will be `FONT_INDEX_DIALOG`.

dimLine indicates the type of color to be used for drawing the underline. If `TRUE`, the *dimColorP* color descriptor is used, otherwise the *foregroundP* color descriptor is used.

clipRectP is a clipping rectangle (any drawing outside this rectangle is suppressed).

foregroundP specifies the foreground color for the underlining if the underline is not dim.

dimColorP specifies the color to use for dim underlining.

riP is only used if *foregroundP* or *dimColorP* is `NULL`. In that case, it indicates a dialog item from which to use get the default foreground or dimmed color.

Returns `mdlDialog_underlineDrawCD` has no return value.

See Also mdlDialog_underlineDraw.

Dialog Box Rectangle Drawing Functions

The dialog box rectangle drawing functions draw a variety of rectangle shapes in a dialog box.

The following table lists the dialog box rectangle drawing functions:

Function	Used to
mdlDialog_rectDraw	draw the outline of a rectangle.
mdlDialog_rectDrawCD	draw a rectangle, color provided by a color descriptor.
mdlDialog_rectFill	draw a filled rectangle.
mdlDialog_rectFillCD	draw a filled rectangle, color provided by a color descriptor.
mdlDialog_rectInvert	invert a rectangle's color.
mdlDialog_rectClearBevel	clear the bevel or edge of a rectangle.
mdlDialog_rectDrawBeveled	draw a beveled rectangle.
mdlDialog_rectDrawEdge	draw an edge around a rectangle.
mdlDialog_rectInset	inset a rectangle.
mdlDialog_rectOffset	offset a rectangle.
mdlDialog_rectSet	set up a rectangle.
mdlDialog_rectWidth	get the width of a rectangle.
mdlDialog_rectHeight	get the height of a rectangle.
mdlDialog_rectPointInside	determine if a point is inside a rectangle.
mdlDialog_rectEqual	determine if two rectangles are equal.
mdlDialog_rectOverlap	construct a rectangle representing the overlapping region of two input rectangles.

mdlDialog_rectDraw, mdlDialog_rectDrawCD

```
#include <mcolor.h>
#include <dlogitem.h>

void mdlDialog_rectDraw
(
    DialogBox    *db,           /* => dialogBox to draw rect outline in */
    BSIRect      *rP,           /* => rect to draw */
    int          colorIndex     /* => fixed color index of rect */
);

void mdlDialog_rectDrawCD
(
    DialogBox    *dbP,           /* => dialogBox to draw rect outline in */
    BSIRect      *rP,           /* => rect to draw */
    BSIRect      *clipRectP,     /* => clip rect */
    BSIColorDescr *colorP,       /* => NULL = use foreground color */
    RawItemHdr   *riP           /* => item to use for default clr */
);
```

Description `mdlDialog_rectDraw` draws the outline in a dialog box in a color specified by a color index. `mdlDialog_rectDrawCD` draws a rectangle outline in a dialog box in a color specified by a color descriptor.

dbP indicates the dialog box to draw in.

rP defines the rectangle to be drawn.

clipRectP is a clipping rectangle (any drawing outside this rectangle is suppressed).

The *colorIndex* parameter specifies a color index for drawing. Its usual value is `WHITE_INDEX`, `BLACK_INDEX`, `LGREY_INDEX` or `DGREY_INDEX`. See `msdefs.h` for the complete list of possible color indexes.

colorP is a `BSIColorDescr` pointer, usually obtained from a `BSIColorPalette`.

riP is only used if *colorP* is `NULL`. In that case, it indicates a dialog item from which to use get the value of *colorP*. The foreground color associated with the item is used.

Returns `mdlDialog_rectDraw` and `mdlDialog_rectDrawCD` are of type `void`. They return no value.

See Also `mdlDialog_rectFill`, `mdlDialog_rectInvert`, `mdlDialog_rectClearBevel`, `mdlDialog_rectDrawBeveled`, `mdlDialog_rectDrawEdge`, "Color".

mdlDialog_rectFill, mdlDialog_rectFillCD

```
void mdlDialog_rectFill
(
    DialogBox      *db,           /* => dialogBox to draw in */
    BSIRect        *rP,           /* => rect to fill */
    int            colorIndex     /* => fixed color index fill color */
);
void mdlDialog_rectFillCD
(
    DialogBox      *dbP,           /* => dialogBox to draw in */
    BSIRect        *rP,           /* => rect to fill */
    BSIRect        *clipRectP,    /* => clip rectangle */
    BSIColorDescr  *fillP,        /* => fill color descr */
    RawItemHdr     *riP           /* => item to use for default clrs */
);
```

Description *mdlDialog_rectFill* draws a filled rectangle in a dialog box in a color specified by a color index. *mdlDialog_rectFillCD* draws a filled rectangle in a dialog box in a color specified by a color descriptor.

dbP indicates the dialog box to draw in.

rP defines the rectangle to be drawn.

The *colorIndex* parameter specifies a color index for drawing. Its usual value is `WHITE_INDEX`, `BLACK_INDEX`, `LGREY_INDEX` or `DGREY_INDEX`. See `msdefs.h` for the complete list of possible color indexes.

clipRectP is a clipping rectangle (any drawing outside this rectangle is suppressed).

fillP is a `BSIColorDescr` pointer, usually obtained from a `BSIColorPalette`.

riP is only used if *fillP* is `NULL`. In that case, it indicates a dialog item from which to use get the value of *fillP*. The foreground color associated with the item is used.

Returns *mdlDialog_rectFill* and *mdlDialog_rectFillCD* are of type `void`. They return no value.

See Also *mdlDialog_rectDraw*, *mdlDialog_rectInvert*, *mdlDialog_rectClearBevel*, *mdlDialog_rectDrawBeveled*, *mdlDialog_rectDrawEdge*, "Color".

mdlDialog_rectInvert, mdlDialog_rectClearBevel, mdlDialog_rectDrawBeveled, mdlDialog_rectDrawEdge

```

void mdlDialog_rectInvert
(
    DialogBox    *db,          /* => dialogBox to affect */
    BSIRect      *rP           /* => area to invert color of */
);

void mdlDialog_rectClearBevel
(
    DialogBox    *db,          /* => dialogBox to clear */
    BSIRect      *rP           /* => area to clear */
);

void mdlDialog_rectDrawBeveled
(
    DialogBox    *db,          /* => dialogBox to draw beveled rect in */
    BSIRect      *rP,          /* => beveled rect */
    boolean      raised,        /* => FALSE=enclose rect looks recessed */
    boolean      wide           /* => TRUE=bevel is two pixels wide */
);

void mdlDialog_rectDrawEdge
(
    DialogBox    *db,          /* => dialogBox to draw 3d rect edge in */
    BSIRect      *rP,          /* => rect edge */
    boolean      raised         /* => FALSE=edge looks like groove instead of wall */
);

```

Description The `mdlDialog_rectInvert` function inverts the color of the rectangle pointed at by *rP*, in the dialog box specified by *db*.

The `mdlDialog_rectClearBevel` function erases the bevel or edge of the rectangle pointed at by *rP* from the dialog box specified by *db*.

The `mdlDialog_rectDrawBeveled` function draws a bevel around the rectangle pointed at by *rP* in the dialog box specified by *db*.

The `mdlDialog_rectDrawEdge` function draws a raised or recessed edge around the rectangle pointed at by *rP* in the dialog box specified by *db*.

If the *raised* parameter is `TRUE`, the rectangle edge appears raised from the surface of the dialog box. Otherwise it appears recessed into the dialog box surface. For example, the group box item is drawn with the raised parameter set to `FALSE`.

If the *wide* parameter is `TRUE`, the rectangle will have a two pixel wide bevel. Otherwise, the bevel will only be one pixel wide.

Returns The `mdlDialog_rectInvert`, `mdlDialog_rectClearBevel`, `mdlDialog_rectDrawBeveled` and `mdlDialog_rectDrawEdge` functions are of type `void`. They return no value.

See Also `mdlDialog_rectDraw`, `mdlDialog_rectFill`, “Color”.

mdlDialog_rectInset, mdlDialog_rectOffset, mdlDialog_rectSet

```
#include <dlogitem.h>

void mdlDialog_rectInset
(
    BSIRect      *rP,
    int           deltaX,    /* positive moves in */
    int           deltaY     /* positive moves in */
);

void mdlDialog_rectOffset
(
    BSIRect      *rP,        /* => rectangle to offset */
    int           dx,        /* => x offset amount */
    int           dy         /* => y offset amount */
);

void mdlDialog_rectSet
(
    BSIRect      *rP,        /* => rectangle to set */
    int           left,      /* => new left edge of rect */
    int           top,       /* => new top edge of rect */
    int           right,     /* => new right edge of rect */
    int           bottom     /* => new bottom edge of rect */
);
```

Description `mdlDialog_rectInset` insets the rectangle pointed at by *rP*. The *deltaX* parameter specifies the distance the left and right edges are moved in by. The *deltaY* parameter specifies the distance the top and bottom edges are moved in by.

`mdlDialog_rectOffset` offsets the rectangle pointed at by *rP* by *dx* in the x direction and *dy* in the y direction.

`mdlDialog_rectSet` sets the rectangle pointed at by *rP* to have its left, right, top and bottom edges set to *left*, *right*, *top* and *bottom*, respectively.

Returns `mdlDialog_rectInset`, `mdlDialog_rectOffset` and `mdlDialog_rectSet` are of type void. They return no value.

mdlDialog_rectWidth, mdlDialog_rectHeight

```
#include <dlogitem.h>
int mdlDialog_rectWidth /* <= width of rect */
(
    BSIRect      *rP      /* => rectangle */
);
int mdlDialog_rectHeight /* <= height of rect */
(
    BSIRect      *rP      /* => rectangle */
);
```

Description mdlDialog_rectWidth returns the width of the rectangle pointed at by *rP*.
mdlDialog_rectHeight returns the height of the rectangle pointed at by *rP*.

Returns mdlDialog_rectWidth returns the width of the rectangle pointed at by *rP*.
mdlDialog_rectHeight returns the height of the rectangle pointed at by *rP*.

mdlDialog_rectPointInside

```
#include <dlogitem.h>
boolean mdlDialog_rectPointInside /* <= TRUE if pt inside */
(
    BSIRect      *rP,      /* => rectangle to check */
    Point2d      *ptP      /* => point to check */
);
```

Description mdlDialog_rectPointInside is used to check if a given point is within a given rectangle.

rp points to the rectangle to check, and *ptP* points to the point to check.

Returns mdlDialog_rectPointInside returns TRUE if the point pointed to by *ptP* is within the rectangle pointed to by *rP*.

mdlDialog_rectEqual

```
#include <dlogitem.h>
boolean mdlDialog_rectEqual /* <= TRUE if rects are equal */
(
    BSIRect      *rect1P,  /* => rectangle to compare */
    BSIRect      *rect2P  /* => other rectangle to compare with */
);
```

Description mdlDialog_rectEqual is used to check two rectangle definitions for equality. Two rectangles are equal if they have identical coordinates at all four corners.

rect1P first rectangle to compare.

rect2P second rectangle to compare.

Returns mdlDialog_rectEqual returns TRUE if the rectangles are identical.

mdlDialog_rectOverlap

```
#include <dlogitem.h>

boolean mdlDialog_rectOverlap /* <= TRUE if rectangles overlap */
(
    BSIRect      *overlapRectP,      /* <= rectangle of overlap */
    BSIRect      *rect1P,            /* => first rectangle to check */
    BSIRect      *rect2P            /* => second rectangle to check */
);
```

Description mdlDialog_rectOverlap will determine if two rectangles intersect, and if they do, will return a third rectangle representing their intersection.

overlapRectP points to a rectangle to receive the definition of the intersection of the first two rectangles.

rect1P points to one of the rectangles to check for intersection.

rect2P points to the second rectangle to check for intersection.

Returns mdlDialog_rectOverlap returns TRUE if an intersection of the two rectangles *rect1P* and *rect2P* exists.

Dialog Box Font Functions

The dialog box font functions return and set information on fonts being used in dialog boxes.

The following table lists the dialog box font functions:

Function	Used to
mdlDialog_fontGetCurHeight	get the height of the current font associated with a dialog box.
mdlDialog_fontGetHeight	get the height of a specified font.
mdlDialog_fontGetInfo	get various information about a specified font.
mdlDialog_fontNameGet	get the name of the font that is currently associated with a dialog box.
mdlDialog_fontIndexGet	get the index of the current font associated with a dialog box.
mdlDialog_fontIndexSet	set the font to be associated with a dialog box.

mdlDialog_fontGetCurHeight, mdlDialog_fontGetHeight

```
#include <dlogitem.h>

int mdlDialog_fontGetCurHeight /* <= current font height */
(
    DialogBox    *db          /* => dialogBox's font height to get */
);

int mdlDialog_fontGetHeight /* <= fontHeight */
(
    DialogBox    *db,          /* => dialogBox in which font would be drawn */
    int          fontIndex /* => index of font to get height of */
);
```

Description `mdlDialog_fontGetCurHeight` returns the height of the font currently associated with a dialog box.

`mdlDialog_fontGetHeight` returns the height of a font assuming that it will be used in a specified dialog box.

db is the dialog box to query.

fontIndex is the font whose height to get.

Returns `mdlDialog_fontGetCurHeight` returns the font height currently associated with the dialog box specified by *db*.

`mdlDialog_fontGetHeight` returns the height of the font, assuming that it will be used in the dialog box specified by *db*.

The values returned by these functions are specified in pixels.

See Also `mdlDialog_fontGetInfo`, `mdlDialog_fontIndexGet`, `mdlDialog_fontIndexSet`, `mdlDialog_fontNameGet`, `mdlDialog_textDraw`, `mdlDialog_textDrawN`, `mdlDialog_stringWidth`, "Text font".

mdlDialog_fontGetInfo, mdlDialog_fontNameGet

```
#include <dlogitem.h>

boolean mdlDialog_fontGetInfo /* <= TRUE if error */
(
    char    *fontNameP,      /* <= font name */
    int     *avgWidthP,      /* <= avg width of alphanumeric chars */
    int     *maxWidthP,      /* <= maximum width */
    int     *heightP,        /* <= font height */
    DialogBox *db,           /* => dialogBox font would be drawn in */
);
```

```
int      fontIndex      /* => index of font to get info about */
);

boolean mdlDialog_fontNameGet /* <= TRUE if error */
(
char      *fontNameP,      /* <= name of current font */
DialogBox *db              /* => dialogBox whose font name to get */
);
```

Description mdlDialog_fontGetInfo retrieves information about the font specified by *fontIndex* assuming that it will be used in the dialog box specified by *db*.

fontNameP points at a buffer which will be set to the name of the font. The *avgWidthP*, *maxWidthP* and *heightP* parameters point at integers which will be set to the average width, maximum width and height of the font, respectively. Units are specified in pixels. Any of these parameters can be NULL, which indicates that the caller doesn't need the corresponding value.

mdlDialog_fontNameGet retrieves the name of the font currently associated with the dialog box specified by *db* into the buffer pointed at by *fontNameP*.

Returns mdlDialog_fontGetInfo and mdlDialog_fontNameGet return TRUE if an error occurs. This means that *db* is not a pointer to a dialog box.

See Also mdlDialog_fontIndexGet, mdlDialog_fontIndexSet, mdlDialog_fontGetCurHeight, mdlDialog_fontGetHeight, mdlDialog_textDraw, mdlDialog_textDrawN, mdlDialog_stringWidth, "Text font".

mdlDialog_fontIndexGet, mdlDialog_fontIndexSet

```
#include <dlogitem.h>

int mdlDialog_fontIndexGet /* <= current font index */
(
DialogBox *db              /* => dialogBox whose font index to get */
);

boolean mdlDialog_fontIndexSet /* <= TRUE if error */
(
DialogBox *db,              /* => dialogBox to get new current font */
int      fontIndex          /* => index of new font to use */
);
```

Description mdlDialog_fontIndexGet returns the index of the font currently associated with the dialog box specified by *db*.

mdlDialog_fontIndexSet sets the font associated with the dialog box specified by *db* to be *fontIndex*. All subsequent text will be drawn using this font.

Returns `mdlDialog_fontIndexGet` returns the index of the font currently associated with the dialog box specified by *db*.

`mdlDialog_fontIndexSet` returns `TRUE` if an error occurs. This means that *db* is not a pointer to a dialog box.

See Also `mdlDialog_fontGetInfo`, `mdlDialog_fontNameGet`, `mdlDialog_fontGetCurHeight`, `mdlDialog_fontGetHeight`, `mdlDialog_textDraw`, `mdlDialog_textDrawN`, `mdlDialog_stringWidth`, "Text font".

Command Queuing Functions

The following table lists command queuing functions:

Function	Used to
<code>mdlDialog_closeCommandQueue</code>	queue a close dialog box command.
<code>mdlDialog_cmdNumberQueue</code>	queue a command number.
<code>mdlDialog_cmdNumberQueueByDb</code>	queue a command number to the owner of a dialog.
<code>mdlDialog_cmdNumberQueueByTaskId</code>	queue a command number to an explicit task.
<code>mdlDialog_cmdNumberQueue</code>	queue a command.
<code>mdlDialog_cmdNumQueueExt</code>	queue a command and optionally journal it.
<code>mdlDialog_cmdNumberQueueByDb</code>	queue a command to the owner of a dialog.
<code>mdlDialog_cmdNumQByDbExt</code>	queue a command to the owner of a dialog and optionally journal it.
<code>mdlDialog_cmdNumberQueueByTaskId</code>	queue a command to an explicit task.
<code>mdlDialog_cmdNumQByTaskIdExt</code>	queue a command to an explicit task and optionally journal it.

mdlDialog_closeCommandQueue

```
#include <dlogitem.h>

boolean mdlDialog_closeCommandQueue /* <= TRUE if error */
(
    DialogBox    *db          /* => dialog to queue close command for */
);
```

Description The `mdlDialog_closeCommandQueue` function queues a `DMSG_CANCEL` command to close the dialog box specified by *db*.

Returns The `mdlDialog_closeCommandQueue` function returns `TRUE` if an error occurs. This means that *db* is not a pointer to a dialog box.

mdlDialog_cmdNumberQueue

```
#include <dlogitem.h>

void mdlDialog_cmdNumberQueue
(
    boolean localCmd,          /* => FALSE = use uStation's cmdtable */
    long   cmdNum,            /* => command number to queue */
    char   *unparsed,         /* => unparsed (string) part of command */
    boolean atEndOfQueue      /* => FALSE = put at start of queue */
);
```

Description The `mdlDialog_cmdNumberQueue` function queues the command specified by *cmdNum* onto the input queue.

The *localCmd* parameter should be set to `TRUE` if the command is from the calling application's command table. Otherwise the command will be assumed to be from MicroStation's command table.

The *unparsed* parameter points at a `NULL` terminated string which will be placed on the input queue along with *cmdNum*. Specifying *unparsed* simulates the user keying-in the command indicated by *cmdNum*, a space character, and then the *unparsed* string.

The *atEndOfQueue* parameter should be set to `TRUE` if the command is to be placed at the end of the input queue. Otherwise, it will be placed at the beginning of the input queue.

Returns The `mdlDialog_cmdNumberQueue` function is of type `void`. It returns no value.

mdlDialog_cmdNumberQueueByDb

```
#include <cmdlist.h>
#include <dlogitem.h>

void mdlDialog_cmdNumberQueueByDb
(
    DialogBox *dbP,           /* => if localCmd=TRUE, dbP = MDL task */
    boolean   localCmd,       /* => TRUE = use uStn cmd tables */
    long      cmdnum,         /* => command number to queue */
    char      *unparsed,      /* => unparsed (string) part of command */
    boolean   atEndOfQueue    /* => FALSE=put at beginning of queue */
);
```

Description `mdlDialog_cmdNumberQueueByDb` is used to send a command number input queue element to the owner of a particular dialog box.

dbP is the dialog box from which the target MDL task is determined.

localCmd, if `TRUE`, indicates that the command number being queued is local to the application and not a MicroStation command.

cmdnum is the command number to queue to the target task.

unparsed is any unparsed (string) portion of the command that the calling application wants parsed by the target task.

atEndOfQueue indicates where in the input queue to put the command queue element. `FALSE` indicates at the beginning of the queue.

Returns `mdlDialog_cmdNumberQueueByDb` has no return value.

See Also `mdlDialog_cmdNumberQueueByTaskId`.

mdlDialog_cmdNumberQueueByTaskId

```
#include <cmdlist.h>
#include <dlogitem.h>

void mdlDialog_cmdNumberQueueByTaskId
(
    char    *taskIdP,      /* => task id string */
    long    cmdnum,        /* => command number to queue */
    char    *unparsed,     /* => unparsed (string) part of command */
    boolean atEndOfQueue   /* => FALSE = put at beginning of queue */
);
```

Description `mdlDialog_cmdNumberQueueByTaskId` is used to send a command number input queue element to a task.

taskIdP is the name of the target task, also known as the task ID, to receive the input command queue element.

cmdnum is the command number to queue to the target task.

unparsed is any unparsed (string) portion of the command that the calling application wants parsed by the target task.

atEndOfQueue indicates where in the input queue to put the command queue element. `FALSE` indicates at the beginning of the queue.

Returns `mdlDialog_cmdNumberQueueByTaskId` has no return value.

See Also `mdlDialog_cmdNumberQueueByDb`.

mdlDialog_cmdNumberQueue, mdlDialog_cmdNumQueueExt

```
#include <msdialog.fdf>

void mdlDialog_cmdNumberQueue
(
  BoolInt localCmd,      /* => FALSE = use MicroStation's cmdtable */
  long   cmdnum,         /* => command number to queue */
  char   *unparsedP,     /* => unparsed (string) part of command */
  BoolInt atEndOfQueue   /* => FALSE = put at start of queue */
);

void mdlDialog_cmdNumQueueExt
(
  BoolInt localCmd,      /* => FALSE = use MicroStation's cmdtable */
  long   cmdnum,         /* => command number to queue */
  char   *unparsedP,     /* => unparsed (string) part of command */
  BoolInt atEndOfQueue,  /* => FALSE = put at start of queue */
  BoolInt journal        /* => TRUE = CAD input journal the command */
);
```

Description The *mdlDialog_cmdNumberQueue* function queues the command specified by *cmdnum* onto the input queue and journals the command if CAD input journaling is active. The *mdlDialog_cmdNumQueueExt* function also queues the command specified by *cmdnum* onto the input queue and but it optionally journals the command if CAD input journaling is active.

localCmd should be set to **TRUE** if the command is from the calling application's command table. Otherwise, the command will be assumed to be from MicroStation's command table.

unparsedP points to a null terminated string which will be placed on the input queue along with *cmdnum*. Specifying *unparsedP* simulates the user keying-in the command indicated by *cmdnum*, a space character, and then the *unparsedP* string.

atEndOfQueue should be set to **TRUE** if the command is to be placed at the end of the input queue. Otherwise, it will be placed at the beginning of the input queue.

journal should be set to **FALSE** if the command should not be journaled when CAD input journaling is active.



These functions were implemented in MicroStation 95.

Returns The *mdlDialog_cmdNumberQueue* and *mdlDialog_cmdNumQueueExt* functions are of type **void**. They return no values.

mdlDialog_cmdNumberQueueByDb, mdlDialog_cmdNumQByDbExt

```
#include <msdialog.fdf>

void mdlDialog_cmdNumberQueueByDb
(
  DialogBox    *dbP,          /* => if localCmd=TRUE, dbP = MDL task */
  BoolInt      localCmd,      /* => FALSE = use uStn cmd tables */
  long         cmdnum,        /* => command number to queue */
  char         *unparsed,     /* => unparsed (string) part of command */
  BoolInt      atEndOfQueue   /* => FALSE = put at beginning of queue */
);

void mdlDialog_cmdNumQByDbExt
(
  DialogBox    *dbP,          /* => if localCmd=TRUE, dbP = MDL task */
  BoolInt      localCmd,      /* => FALSE = use uStn cmd tables */
  long         cmdnum,        /* => command number to queue */
  char         *unparsed,     /* => unparsed (string) part of command */
  BoolInt      atEndOfQueue,  /* => FALSE = put at beginning of queue */
  BoolInt      journal        /* => TRUE = CAD input journal the command */
);
```

Description The `mdlDialog_cmdNumberQueueByDb` function sends a command number input queue element to the owner of a particular dialog box. The command is also journaled if CAD input journaling is active.

The `mdlDialog_cmdNumQByDbExt` function also sends a command number input queue element to the owner of a particular dialog box. However, if CAD input journaling is active, the command is journaled only if `journal` is `TRUE`.

dbP is the dialog box from which the target MDL task is determined.

localCmd, if `TRUE`, indicates that the command number being queued is local to the application and not a MicroStation command.

cmdnum is the command number to queue to the target task.

unparsed is any unparsed (string) portion of the command that the calling application wants parsed by the target task.

atEndOfQueue indicates where in the input queue to put the command queue element. `FALSE` indicates at the beginning of the queue.



These functions were implemented in MicroStation 95.

Returns `mdlDialog_cmdNumberQueueByDb` and `mdlDialog_cmdNumQByDbExt` are of type `void` and return no values.

See Also `mdlDialog_cmdNumberQueueByTaskId`, `mdlDialog_cmdNumQByTaskIdExt`.

mdlDialog_cmdNumberQueueByTaskId, mdlDialog_cmdNumQByTaskIdExt

```
#include <msdialog.fdf>

void mdlDialog_cmdNumberQueueByTaskId
(
  char   *taskIdP,      /* => task id string */
  long   cmdnum,        /* => command number to queue */
  char   *unparsed,     /* => unparsed (string) part of command */
  BoolInt atEndOfQueue /* => FALSE = put at beginning of queue */
);

void mdlDialog_cmdNumQByTaskIdExt
(
  char   *taskIdP,      /* => task id string */
  long   cmdnum,        /* => command number to queue */
  char   *unparsed,     /* => unparsed (string) part of command */
  BoolInt atEndOfQueue, /* => FALSE = put at beginning of queue */
  BoolInt journal       /* => TRUE= CAD input journal the command */
);
```

Description The `mdlDialog_cmdNumberQueueByTaskId` function sends a command number input queue element to a task. The command is also journaled if CAD input journaling is active.

The `mdlDialog_cmdNumQByTaskIdExt` function also sends a command number input queue element to a task. However, if CAD input journaling is active, the command is journaled only if `journal` is `TRUE`.

taskIdP is the name of the target task, also known as the task ID, to receive the input command queue element.

cmdnum is the command number to queue to the target task.

unparsed is any unparsed (string) portion of the command that the calling application wants parsed by the target task.

atEndOfQueue indicates where in the input queue to put the command queue element. `FALSE` indicates at the beginning of the queue.



These functions were implemented in MicroStation 95.

Returns `mdlDialog_cmdNumberQueueByTaskId` and `mdlDialog_cmdNumQByTaskIdExt` are void and return no values.

See Also `mdlDialog_cmdNumberQueueByDb`, `mdlDialog_cmdNumQByDbExt`.

Track Bar Window Functions

The following table lists track bar window functions:

Function	Used to
<code>mdlDialog_trackBarStartProcessing</code> [ditemlib.ml, ditemlib.msl]	open track bar window and start work operations.
<code>mdlDialog_trackBarStopProcessing</code> [ditemlib.ml, ditemlib.msl]	cancel track bar operations and close the window.
<code>mdlDialog_trackBarUpdateControlParms</code> [ditemlib.ml, ditemlib.msl]	update control information relating to the track bar window operation currently active.
<code>mdlDialog_trackBarUpdateDisplayInfo</code> [ditemlib.ml, ditemlib.msl]	update the message text and percentage completion displayed in the track bar window.

mdlDialog_trackBarStartProcessing [ditemlib.ml, ditemlib.msl]

```
#include <dlogitem.h>
#include <dlogman.fdf>

int mdlDialog_trackBarStartProcessing
(
    int      (*workFuncP)(),           /* => work function pointer */
    void     *funcParmsP,             /* => function parameters */
    void     (*completionFuncP)(),    /* => completion func. ptr. or NULL */
    void     (*cancelFuncP)(),        /* => cancel func. pointer or NULL */
    char     *cancelMessage,          /* => cancel confirm msg or NULL */
    int      stackedCompletionBars,   /* => double track bars if TRUE */
    TrackBarInfo *trackBarInfoP,      /* => track bar data */
    char     *windowTitle             /* => busy bar window title or NULL */
);
```

Description The `mdlDialog_trackBarStartProcessing` function initializes the processing environment, opens the track bar window, and starts the processing for the calling application by calling the application's MDL work function as specified by the required *workFuncP* parameter after a small time delay. The MDL function pointed to by *workFuncP* should be written to perform an incremental amount of work with each call to the function. The rest of the parameters are optional.

The *funcParmsP* parameter is a pointer to the parameter which is passed to all functions which can be called by the track bar window.

completionFuncP is an MDL function pointer which points to the application completion function which will get control when all track bar processing has been completed. A value of `NULL` indicates no completion function.

cancelFuncP is an MDL function pointer which points to the application cancel function which will get control when the user activates the CANCEL button in the track bar window. A value of `NULL` indicates no cancel function.

cancelMessage is a pointer to a message to be displayed to the user to confirm the user's desire to cancel the current operation. A value of `NULL` indicates that the standard confirmation message of "Cancel current operation?" should be used.

stackedCompletionBars is a flag indicating whether the track bar should display a single completion bar as is done in the completion bar window or two stacked completion bars so that two separate granularities of processing or events can be tracked. A value of 0 indicates a single completion bar should be used.

trackBarInfoP is a pointer to the track bar control information indicating percent completion and messages to be displayed above each completion bar. The information in this structure is used to set the initial display values for the track bar window. The update field of this structure contains an indication as to which fields in this structure contain valid information. The structure of this information and associated definitions are:

```
#define      UPDATE_Percent1 1
#define      UPDATE_Percent2 2
#define      UPDATE_Msg1     4
#define      UPDATE_Msg2     8
#define      UPDATE_All      15

typedef struct trackbarinfo
{
    int        update;                /* Which fields to update */
    long       percentComplete1;      /* Percent complete - bar 1 (top) */
    long       percentComplete2;      /* Percent complete - bar 2 (bottom) */
    char       msgText1[256];         /* Message text - bar 1 (top) */
    char       msgText2[256];         /* Message text - bar 2 (bottom) */
} TrackBarInfo;
```

windowTitle is a pointer to a string which is to be used as the title of the track bar window. A value of `NULL` indicates that the default title of "Working" is to be used.



This function was implemented in MicroStation 95.

Returns `mdlDialog_trackBarStartProcessing` returns `ERROR` if an error occurred or a busy/track bar is currently active and `SUCCESS` upon successful completion.

See Also `mdlDialog_trackBarStopProcessing`, `mdlDialog_trackBarUpdateMessage`, `mdlDialog_trackBarUpdateControlParms`, `mdlDialog_completionBarOpen`, `mdlDialog_busyBarStartProcessing`, `userBar_workFunction`, `userBar_cancelFunction`, `userBar_completionFunction`.

mdlDialog_trackBarStopProcessing [ditemlib.ml, ditemlib.msl]

```
#include <dlogman.fdf>

void mdlDialog_trackBarStopProcessing(void);
```

Description The `mdlDialog_trackBarStopProcessing` function closes down the track bar processing environment and closes the track bar window. The application completion function is not called as a result of calling this function.



This function was implemented in MicroStation 95.

Returns The `mdlDialog_trackBarStopProcessing` function is void and returns no value.

See Also `mdlDialog_trackBarStartProcessing`, `mdlDialog_trackBarUpdateMessage`, `mdlDialog_trackBarUpdateControlParms`, `mdlDialog_completionBarClose`, `mdlDialog_busyBarStopProcessing`.

mdlDialog_trackBarUpdateControlParms [ditemlib.ml, ditemlib.msl]

```
#include <dlogitem.h>
#include <dlogman.fdf>

int mdlDialog_trackBarUpdateControlParms
(
    int      (*workFuncP)(),      /* => work function pointer or NULL */
    void      *funcParmsP,        /* => function parameters or NULL */
    void      (*completionFuncP)(), /* => completion func. pointer or NULL */
    void      (*cancelFuncP)(),    /* => cancel function pointer or NULL */
    char      *cancelMessage,      /* => cancel confirmation msg or NULL */
    TrackBarInfo *trackBarInfoP /* => track bar data or NULL */
);
```

Description The `mdlDialog_trackBarUpdateControlParms` function updates control information relating to the track bar window operation currently active. Any portion of the control information may be updated and `NULL` may be specified for any parameter which is to remain unchanged. This function is useful in applications which will perform a number of different operations while still maintaining the track bar window.

workFuncP is an MDL function pointer which points to the application work function which will get control to allow the application to perform a piece of work at short timer intervals. The MDL function pointed to by *workFuncP* should be written to perform an incremental amount of work with each call to the function. A value of `NULL` indicates no change to this control information.

The *funcParmsP* parameter is a pointer to the parameter which is passed to all functions which can be called by the track bar window. A value of `NULL` indicates no change to this control information.

completionFuncP is an MDL function pointer which points to the application completion function which will get control when all track bar processing has been completed. A value of `NULL` indicates no change to this control information.

cancelFuncP is an MDL function pointer which points to the application cancel function which will get control when the user activates the CANCEL button in the track bar window. A value of `NULL` indicates no change to this control information.

cancelMessage is a pointer to a message to be displayed to the user to confirm the user's desire to cancel the current operation. A value of `NULL` indicates no change to this control information.

trackBarInfoP is a pointer to the track bar control information indicating percent completion and messages to be displayed above each completion bar. The information in this structure is used to update the current display information in the track bar window. A value of `NULL` indicates no change to this control information.



This function was implemented in MicroStation 95.

Returns `mdlDialog_trackBarUpdateControlParms` returns `ERROR` if an error occurred or a track bar is not currently active and `SUCCESS` upon successful completion.

See Also `mdlDialog_trackBarStartProcessing`, `mdlDialog_trackBarStopProcessing`, `mdlDialog_trackBarUpdateMessage`, `mdlDialog_busyBarUpdateControlParms`, `userBar_workFunction`, `userBar_cancelFunction`, `userBar_completionFunction`.

mdlDialog_trackBarUpdateDisplayInfo [ditemlib.ml, ditemlib.msl]

```
#include <dlogitem.h>
#include <dlogman.fdf>
void mdlDialog_trackBarUpdateDisplayInfo
(
  TrackBarInfo      *trackBarInfoP /* => Display information */
);
```

Description The `mdlDialog_trackBarUpdateDisplayInfo` function updates the track bar window with the information supplied in the information pointed to by *trackBarInfoP* as an indication of the current processing state of the application. The fields to be updated from information within this structure are defined using the update field of this structure.



This function was implemented in MicroStation 95.

Returns The `mdlDialog_trackBarUpdateDisplayInfo` function is of type `void` and returns no value.

See Also mdlDialog_trackBarStartProcessing, mdlDialog_trackBarStopProcessing, mdlDialog_trackBarUpdateMessage, mdlDialog_trackBarUpdateControlParms, mdlDialog_completionBarOpen, mdlDialog_busyBarUpdateMessage, userBar_workFunction, userBar_cancelFunction, userBar_completionFunction.

Busy Bar Window Functions

The following table lists the Dialog Box Busy Bar functions:

Function	Used to
mdlDialog_busyBarStartProcessing [ditemlib.ml, ditemlib.msl]	open a busy bar window and start work operations.
mdlDialog_busyBarStopProcessing [ditemlib.ml, ditemlib.msl]	cancel busy bar operations and close the window.
mdlDialog_busyBarUpdateControlParms [ditemlib.ml, ditemlib.msl]	update the control parameters associated with busy bar operations.
mdlDialog_busyBarUpdateMessage [ditemlib.ml, ditemlib.msl]	update the message text displayed in the busy bar window.

The following table lists the User functions.

Function	Used to
userBar_cancelFunction	user function called by track/busy bar window to allow the application to cancel the work in progress.
userBar_completionFunction	called by track/busy bar window to allow the application to clean up.
userBar_workFunction	called by track/busy bar window to allow the application to perform work.

mdlDialog_busyBarStartProcessing [ditemlib.ml, ditemlib.msl]

```
#include <dlogman.fdf>

int mdlDialog_busyBarStartProcessing
(
  int      (*workFuncP)(),      /* => work function pointer */
  void     *funcParmsP,        /* => function parameters */
  void     (*completionFuncP)(), /* => completion func. pointer or NULL */
  void     (*cancelFuncP)(),    /* => cancel function pointer or NULL */
  char     *cancelMessage,     /* => cancel confirmation msg or NULL */
  int      rectangle,          /* => rectangle or shape bar */
  char     *msgStr,            /* => display message */
  char     *windowTitle        /* => busy bar window title or NULL */
);
```

Description The `mdlDialog_busyBarStartProcessing` function initializes the processing environment, opens the busy bar window, and starts the processing for the calling application by calling the application's MDL work function as specified by the required *workFuncP* parameter after a small time delay. The MDL function pointed to by *workFuncP* should be written to perform an incremental amount of work with each call to the function. The rest of the parameters are optional.

The *funcParmsP* parameter is a pointer to the parameter which is passed to all functions which can be called by the busy bar window.

completionFuncP is an MDL function pointer which points to the application completion function which will get control when all busy bar processing has been completed. A value of `NULL` indicates no completion function.

cancelFuncP is an MDL function pointer which points to the application cancel function which will get control when the user activates the CANCEL button in the busy bar window. A value of `NULL` indicates no cancel function.

cancelMessage is a pointer to a message to be displayed to the user to confirm the user's desire to cancel the current operation. A value of `NULL` indicates that the standard confirmation message of "Cancel current operation?" should be used.

rectangle is a flag indicating whether the busy bar style used displays sliding rectangles or sliding parallelograms. A value of 0 indicates parallelograms should be used.

msgStr is a pointer to the initial message string to be displayed in the busy bar window describing the current operational status of the work in progress. A value of `NULL` indicates no message is to be displayed.

windowTitle is a pointer to a string which is to be used as the title of the busy bar window. A value of `NULL` indicates that the default title of "Busy" is to be used.



This function was implemented in MicroStation 95.

Returns `mdlDialog_busyBarStartProcessing` returns `ERROR` if an error occurred or a busy/track bar is currently active and `SUCCESS` upon successful completion.

See Also `mdlDialog_busyBarStopProcessing`, `mdlDialog_busyBarUpdateMessage`, `mdlDialog_busyBarUpdateControlParms`, `mdlDialog_completionBarOpen`, `mdlDialog_trackBarStartProcessing`, `userBar_workFunction`, `userBar_cancelFunction`, `userBar_completionFunction`.

mdlDialog_busyBarStopProcessing [ditemlib.ml, ditemlib.msl]

```
#include <dlogman.fdf>

void mdlDialog_busyBarStopProcessing
(
void
);
```

Description The `mdlDialog_busyBarStopProcessing` function closes down the busy bar processing environment and closes the busy bar window. The application completion function is not called as a result of calling this function.



This function was implemented in MicroStation 95.

Returns The `mdlDialog_busyBarStopProcessing` function is of type `void` and returns no value.

See Also `mdlDialog_busyBarStartProcessing`, `mdlDialog_busyBarUpdateMessage`, `mdlDialog_busyBarUpdateControlParms`, `mdlDialog_completionBarClose`, `mdlDialog_trackBarStopProcessing`.

mdlDialog_busyBarUpdateControlParms [ditemlib.ml, ditemlib.msl]

```
#include <dlogman.fdf>

int mdlDialog_busyBarUpdateControlParms
(
int      (*workFuncP)(),      /* => work function pointer or NULL */
void     *funcParmsP,        /* => function parameters or NULL */
void     (*completionFuncP)(), /* => completion func. pointer or NULL */
void     (*cancelFuncP)(),    /* => cancel function pointer or NULL */
char     *cancelMessage,      /* => cancel confirmation msg or NULL */
char     *msgStr              /* => display message or NULL */
);
```

Description The `mdlDialog_busyBarUpdateControlParms` function updates control information relating to the busy bar window operation currently active. Any portion of the control information may be updated and `NULL` may be specified for any parameter which is to remain unchanged. This function is useful in applications which will

perform a number of different operations while still maintaining the busy bar window.

workFuncP is an MDL function pointer which points to the application work function which will get control to allow the application to perform a piece of work at short timer intervals. The MDL function pointed to by *workFuncP* should be written to perform an incremental amount of work with each call to the function. A value of `NULL` indicates no change to this control information.

The *funcParmsP* parameter is a pointer to the parameter which is passed to all functions which can be called by the busy bar window. A value of `NULL` indicates no change to this control information.

completionFuncP is an MDL function pointer which points to the application completion function which will get control when all busy bar processing has been completed. A value of `NULL` indicates no change to this control information.

cancelFuncP is an MDL function pointer which points to the application cancel function which will get control when the user activates the CANCEL button in the busy bar window. A value of `NULL` indicates no change to this control information.

cancelMessage is a pointer to a message to be displayed to the user to confirm the user's desire to cancel the current operation. A value of `NULL` indicates no change to this control information.

msgStr is a pointer to the message string to be displayed in the busy bar window describing the current operational status of the work in progress. A value of `NULL` indicates no change to this control information.



This function was implemented in MicroStation 95.

Returns mdlDialog_busyBarUpdateControlParms returns `ERROR` if an error occurred or a busy bar is not currently active and `SUCCESS` upon successful completion.

See Also mdlDialog_busyBarStartProcessing, mdlDialog_busyBarStopProcessing, mdlDialog_busyBarUpdateMessage, mdlDialog_trackBarUpdateControlParms, userBar_workFunction, userBar_cancelFunction, userBar_completionFunction.

mdlDialog_busyBarUpdateMessage [ditemlib.ml, ditemlib.msl]

```
#include <dlogman.fdf>

void mdlDialog_busyBarUpdateMessage
(
char    *msgStr    /* => display message */
);
```

Description The `mdlDialog_busyBarUpdateMessage` function displays the message pointed to by the *msgStr* parameter in the busy bar window as an indication of the current processing state of the application.



This function was implemented in MicroStation 95.

Returns The `mdlDialog_busyBarUpdateMessage` function is void and returns no value.

See Also `mdlDialog_busyBarStopProcessing`, `mdlDialog_busyBarUpdateMessage`, `mdlDialog_busyBarUpdateControlParms`, `mdlDialog_completionBarOpen`, `mdlDialog_trackBarStartProcessing`, `userBar_workFunction`, `userBar_cancelFunction`, `userBar_completionFunction`.

userBar_cancelFunction

```
#include <dlogitem.h>
#include <dlogman.fdf>

void userBar_cancelFunction
(
    void      *funcParmsP      /* <= function parameters */
);
```

Description The *userBar_cancelFunction* function is called by the track/busy bar window to allow the application to cancel the work being visually indicated by the displayed window and is set using one of `mdlDialog_trackBarStartProcessing`, `mdlDialog_busyBarStartProcessing`, `mdlDialog_trackBarUpdateControlParms` or `mdlDialog_busyBarUpdateControlParms` functions. This function is called as a result of the user pressing the CANCEL button on the busy/track bar window to and the OK button in the subsequent acknowledgement dialog to abort the work currently being performed by the application. This function should clean up from the work in progress in this function since the application completion function is NOT called as a result of a CANCEL operation by the user.

The *funcParmsP* parameter is a pointer to application specific data as defined by one of the `mdlDialog_trackBarStartProcessing`, `mdlDialog_busyBarStartProcessing`, `mdlDialog_trackBarUpdateControlParms` or `mdlDialog_busyBarUpdateControlParms` functions. The busy/track bar window functions do nothing with this parameter.

Returns The *userBar_cancelFunction* function is of type `void` and returns no value.

See Also `mdlDialog_busyBarStartProcessing`, `mdlDialog_busyBarUpdateControlParms`, `mdlDialog_trackBarStartProcessing`, `mdlDialog_trackBarUpdateControlParms`, `userBar_workFunction`, `userBar_completionFunction`.

userBar_completionFunction

```
#include <dlogitem.h>
#include <dlogman.fdf>

void userBar_completionFunction
(
int      status,          /* <= completion status from workFunc */
void     *funcParmsP     /* <= function parameters */
);
```

Description The *userBar_completionFunction* function is called by the track/busy bar window to allow the application to clean up after completion of the work being visually indicated by the displayed window and is set using one of `mdlDialog_trackBarStartProcessing`, `mdlDialog_busyBarStartProcessing`, `mdlDialog_trackBarUpdateControlParms` or `mdlDialog_busyBarUpdateControlParms` functions. This function is called when the application work function returns a non-zero value to the busy/track bar control logic.

The *status* parameter is the return code from the work function passed to the busy/track bar control logic indicating why application processing terminated. The value of this parameter is application dependent.

The *funcParmsP* parameter is a pointer to application specific data as defined by one of the `mdlDialog_trackBarStartProcessing`, `mdlDialog_busyBarStartProcessing`, `mdlDialog_trackBarUpdateControlParms` or `mdlDialog_busyBarUpdateControlParms` functions. The busy/track bar window functions do nothing with this parameter.

Returns The *userBar_completionFunction* function is of type `void` and returns no value.

See Also `mdlDialog_busyBarStartProcessing`, `mdlDialog_busyBarUpdateControlParms`, `mdlDialog_trackBarStartProcessing`, `mdlDialog_trackBarUpdateControlParms`, `userBar_cancelFunction`, `userBar_workFunction`.

userBar_workFunction

```
#include <dlogitem.h>
#include <dlogman.fdf>

int userBar_workFunction
(
void     *funcParmsP     /* <= function parameters */
);
```

Description The *userBar_workFunction* function is called by the track/busy bar window to allow the application to perform the work being visually indicated by the displayed window and is set using one of `mdlDialog_trackBarStartProcessing`, `mdlDialog_busyBarStartProcessing`, `mdlDialog_trackBarUpdateControlParms` or `mdlDialog_busyBarUpdateControlParms` functions. This function should

perform an integral portion of work in a time period long enough to perform a reasonable and productive amount of work while remaining small enough to allow the track/busy bar display to show activity and allow the user the opportunity to abort the operation.

The *funcParmsP* parameter is a pointer to application specific data as defined by one of the `mdlDialog_trackBarStartProcessing`, `mdlDialog_busyBarStartProcessing`, `mdlDialog_trackBarUpdateControlParms` or `mdlDialog_busyBarUpdateControlParms` functions. The busy/track bar window functions do nothing with this parameter.

Returns *userBar_workFunction* returns a non-ZERO value if an error occurred or if the work the application was doing has been completed and `SUCCESS` to indicate that more work needs to be performed by the work function. A non-ZERO return value will cause the busy/track bar control logic to terminate processing and call the application completion function, if defined.

See Also `mdlDialog_busyBarStartProcessing`, `mdlDialog_busyBarUpdateControlParms`, `mdlDialog_trackBarStartProcessing`, `mdlDialog_trackBarUpdateControlParms`, `userBar_cancelFunction`, `userBar_completionFunction`.

Miscellaneous Dialog Box Functions

The following table lists miscellaneous dialog box functions:

Function	Used to
<code>mdlDialog_labelSetAttributes</code>	set attributes of a label item.
<code>mdlDialog_dmsgsPrint</code>	send a string to the dialog manager debug window.
<code>mdlDialog_dump</code>	dump the contents of a dialog box.
<code>mdlDialog_callFunction</code>	call an MDL function.
<code>mdlDialog_userPrefFileOpen</code>	open the current user preferences resource file.
<code>mdlDialog_fixResourceAddress</code>	fix the alignment of a dialog resource.
<code>mdlDialog_openAdvisoryBox</code>	provide an unclosable message dialog.
<code>mdlDialog_openMessageBox</code>	open a modal dialog box containing up to three push buttons, a multi-line label item and a specified icon.
<code>mdlDialog_overallTitleBarGet</code>	get a pointer to the dialog box that contains the overall application title bar and main application menu.

Function	Used to
mdlDialog_statusAreaGet	get a pointer to the dialog box that displays the current snap icon, current level, element selection status, etc.
mdlDialog_commandWindowGet	get a pointer to the dialog that display prompts, current command and error messages.
mdlDialog_fileOpenExt	select a file through a dialog box to open or create.
mdlDialog_keyinWindowGet	get a pointer to the dialog box that contains the main key-in area.

mdlDialog_labelSetAttributes

```
boolean mdlDialog_labelSetAttributes /* <= TRUE if error */
(
  RawItemHdr *labelP,      /* => label to set */
  int         attributes    /* => new label attributes */
);
```

Description mdlDialog_labelSetAttributes sets the attributes of the label item indicated by *labelP*.

labelP is a pointer to the raw item header of the label whose attributes are to be changed.

attributes is a bitmapped integer that may contain combinations of the constants ALIGN_LEFT, ALIGN_RIGHT, ALIGN_CENTER, LABEL_FONT_BOLD, LABEL_WORDWRAP, and so on.

Returns mdlDialog_labelSetAttributes returns FALSE on success or TRUE if there is an error.

mdlDialog_dmsgsPrint

```
void mdlDialog_dmsgsPrint
(
  char *stringP      /* => string to print in "messages" dbox */
);
```

Description mdlDialog_dmsgsPrint allows application developers to display status or diagnostic messages to the Dialog Manager's messages dialog box. If the messages dialog box is not already visible on the screen, calling this function will make it so.

stringP is a free-form string to display in the Dialog Manager's messages dialog box.

Returns mdlDialog_dmsgsPrint has no return value.

See Also mdlDialog_dump.

mdlDialog_dump

```
void mdlDialog_dump
(
DialogBox    *dbP      /* => dialog box to print list of items about */
);
```

Description mdlDialog_dump is used to display diagnostic information about a dialog box and its constituent items. The information is displayed in the Dialog Manager's messages dialog box. If the messages dialog box is not already visible, calling this function will make it so. This is a debugging tool.

dbP is the dialog box about which information will be displayed.

Returns mdlDialog_dump is of type void and has no return value.

See Also mdlDialog_dmsgsPrint.

mdlDialog_callFunction

```
#include <msdialog.fdf>

int mdlDialog_callFunction /* <= result of function called */
(
void          *funcMD,      /* => MDL descr of function to call */
MdlFunctionP *funcOffset,  /* => function to call */
char          *argP        /* => argument(s) to function */
);
```

Description The mdlDialog_callFunction function calls a function in an MDL application.

funcMD is the MDL descriptor of the task owning the function to call, and *funcOffset* is the address of the function within the application. *argP* represents the list of arguments to be passed to the MDL function.

Returns mdlDialog_callFunction returns the value returned by the called function.

mdlDialog_userPrefFileOpen

```
#include <rsdefs.h>

boolean mdlDialog_userPrefFileOpen /* <= TRUE if error */
(
RscFileHandle *userPrefHP,      /* <= handle to user pref file */
int           readWriteAccess  /* => Desired R/W access mode */
);
```

Description mdlDialog_userPrefFileOpen is used to obtain a resource file handle corresponding to the active user preferences file for the current MicroStation

session. Other resource manager calls such as `mdlResource_addByAlias` and `mdlResource_loadByAlias` can be called using this resource file handle.



If an application does store resources in the MicroStation user preferences file, it should follow two rules. First, it should always use positive resource IDs. Negative resource IDs are reserved for MicroStation. Second, the resources added by an application should always be qualified by an alias name equal to the application's task ID to keep its resources from being overwritten by user preference resources belonging to other applications.

userPrefHP points to a location to receive the resource file handle for the user preferences file.

readWriteAccess is one of the valid file access modes defined in `mdlResource_openFile`.

Returns `mdlDialog_userPrefFileOpen` returns `SUCCESS` if the open succeeds.

See Also `mdlResource_openFile`, `mdlResource_loadByAlias`, `mdlResource_addByAlias`.

mdlDialog_fixResourceAddress

```
void *mdlDialog_fixResourceAddress /* <= fixed address */
(
    byte    *resP,           /* => base of structure next ptr is in */
    byte    *memberP,       /* => member that may need alignment fixing */
    int     sizeMember      /* => member alignment */
);
```

Description The `mdlDialog_fixResourceAddress` function is used to obtain a properly aligned pointer to a structure member within a resource. This function is useful for traversing resources which contain variable sized arrays (variable sized arrays make static dereferences impossible). The function uses three parameters to accomplish this as follows:

First *resP*, the base address of the resource, is subtracted from *memberP*, the caller's best guess at the location of the desired structure member. This calculates the effective offset of the "best guess" relative to the beginning of the resource. ("Best guess" means the caller calculated the address without taking padding or alignment considerations into account).

The third argument, the proposed alignment of the target member, is used to adjust the caller's proposed member offset required by the current hardware platform. That adjusted pointer is returned to the caller.

Returns `mdlDialog_fixResourceAddress` returns the address of the desired resource member.

See Also "Finding your way around a resource" section in the *MDL Programmer's Guide*.

mdlDialog_openAdvisoryBox

```
#include <msdialog.fdf>
#include <dlogids.h>

int mdlDialog_openAdvisoryBox /* <= SUCCESS or ERROR */
(
    long    dialogId, /* => DialogId of AdvisoryBox */
    char    *titleP,  /* => title string, NULL for default */
    char    *stringP, /* => message string, NULL for default */
    long    whichIcon /* => number of image to be displayed, 0 for ! */
);
```

Description The mdlDialog_openAdvisoryBox function is used to present a message that cannot be dismissed. Once opened, it will stay visible until the current drawing is closed. Typically, an application operating as a MS_DGNAPPS will hook the SYSTEM_NEWFILE_COMPLETE event in the SYSTEM_NEW_DESIGN_FILE asynchronous function. An example advisory resource is shown below. The published hook "HOOKDIALOGID_AdvisoryBox" must be used to obtain the necessary hook processing. The attribute DIALOGATTR_DOCKABLE in combination with this hook processing causes the dialog to always stay within the window area. If DIALOGATTR_DOCKABLE is not used, the dialog may be moved past the edges of the main window.

Example

```
/*-----+
|                                     |
|      Sample Advisory Dialog      |
|                                     |
+-----*/

#undef XC
#define XC      (DCOORD_RESOLUTION/2) * ASPECT_ADVISORYBOX
#define XSIZE   50*XC
#define YSIZE   10*YC
DialogBoxRsc DIALOGID_Advisory=
{
    DIALOGATTR_DOCKABLE|DIALOGATTR_UNCLOSEABLE|DIALOGATTR_NORIGHTICONS,
    XSIZE, YSIZE, NOHELP, MHELPTOPIC, HOOKDIALOGID_AdvisoryBox,
    NOPARENTID,
    "A Default Title",
    {
        {{10*XC,YC,35*XC,0}, MLText, MLTEXTID_Advisory, ON, 0, "", ""},
        {{GENX,GENY(2),5*XC,5*YC},Generic,GENERICID_WarningIcon,ON,0,"",""},
    }
};
DItem_MultilineTextRsc MLTEXTID_Advisory=
{
    NOSYNONYM, NOHELP, LHELPTOPIC, NOHOOK, NOARG,
    MLTEXTATTR_READONLY | MLTEXTATTR_NODISPLAYNLCHAR |
```



```
MLTEXTATTR_AUTOHIDEDECOR, 10, ""
};
Ditem_GenericRsc GENERICID_WarningIcon =
{
    NOHELP, MHELPTOPIC, HOOKITEMID_Generic_WarningIcon, NOARG
};
```



This function was implemented in MicroStation 95.

Returns mdlDialog_openAdvisoryBox returns SUCCESS or ERROR.

See Also mdlSystem_setFunction.

mdlDialog_openMessageBox

```
#include <dlogids.h>
#include <msdialog.fdf>

int mdlDialog_openMessageBox /* <= usually ACTIONBUTTON_ */
(
    long    dialogId,      /* => Dialog Id of message box to create */
    char    *stringP,      /* => string to display in message area */
    long    whichIcon      /* => Id of symbol to be drawn in box */
);
```

Description The mdlDialog_openMessageBox function opens a modal dialog box which contains one to three push buttons, a multi-line label item that will be set to the string pointed to by *stringP*, and one of four icons specified by whichIcon. It does not return to it's caller the user closes the modal dialog box.

dialogId is the ID of the message box which will be displayed, and will be used to specify which push buttons are included in the dialog. The following *dialogs* are available for use with mdlDialog_openMessageBox:

dialogId	Description
DIALOGID_MsgBoxOK	The message box contains one button: OK.
DIALOGID_MsgBoxOKCancel	The message box contains one button: OK and Cancel.
DIALOGID_MsgBoxYesNo	The message box contains one button: Yes and No.
DIALOGID_MsgBoxYesNoCancel	The message box contains one button: Yes, No and Cancel.

mdlDialog_openMessageBox can also be used to display Icons in the following, dialogs. If, for example, you want to display a large information box - perhaps because you have a very long message - you can use:

```
mdlDialog_openMessageBox(DIALOGID_LargeInfoBox, stringP,
                        MSGBOX_ICON_INFORMATION)

DIALOGID_StandardAlert
DIALOGID_MediumAlert
DIALOGID_LargeAlert
DIALOGID_StandardInfoBox
DIALOGID_MediumInfoBox
DIALOGID_LargeInfoBox
DIALOGID_YesNoCancelAlert
```

stringP is the multi-line text string to be displayed as the alert message of the dialog.

whichIcon specifies the raster icon to be drawn in the dialog box to the left of the alert message.

mdlDialog_openMessageBox automatically detects the current GUIMODE and displays icons appropriate to Windows or Motif interface styles.



This function was implemented in MicroStation 95.

Returns mdlDialog_openMessageBox returns ACTIONBUTTON_OK, ACTIONBUTTON_CANCEL, etc. returns -1 if failure.

mdlDialog_overallTitleBarGet

```
#include <msdialog.fdf>

DialogBox *mdlDialog_overallTitleBarGet(void);
/* <= pointer to title bar */
```

Description The mdlDialog_overallTitleBarGet function is provided so that the dialog box that contains MicroStation's overall application title and main menu can be referenced without having to consider the currently selected interface style. This function is not typically needed. The mdlDialog_menuBarGetCmdWinP function should be used if an application needs to dynamically alter MicroStation's main menu.



This function was implemented in MicroStation 95.

Returns mdlDialog_overallTitleBarGet returns a pointer to the DialogBox that houses the overall application title and main menu.

See Also mdlDialog_commandWindowGet, mdlDialog_keyinWindowGet, mdlDialog_statusAreaGet, mdlDialog_menuBarGetCmdWinP.

mdlDialog_statusAreaGet

```
#include <msdialog.fdf>

DialogBox *mdlDialog_statusAreaGet(void);
```

Description The mdlDialog_statusAreaGet function is used to reference the dialog box that contains the current snap icon, active level, etc. Since the command window style interface does not have a status area, this function can be used to determine the currently selected interface style.



This function was implemented in MicroStation 95.

Returns mdlDialog_statusAreaGet returns a pointer to the DialogBox that contains the status area. In the command window style interface, this function returns NULL. In the status bar style interface, this function returns a pointer to the status bar.

See Also mdlDialog_commandWindowGet, mdlDialog_keyinWindowGet, mdlDialog_overallTitleBarGet, mdlDialog_menuBarGetCmdWinP.

mdlDialog_commandWindowGet

```
#include <msdialog.fdf>

DialogBox *mdlDialog_commandWindowGet(void);
/* <= pointer to cmd window */
```

Description The mdlDialog_commandWindowGet function is provided so that MicroStation's prompt and message area can be referenced without having to consider the currently selected interface style.



This function was implemented in MicroStation 95.

Returns mdlDialog_commandWindowGet returns a pointer to the DialogBox that contains the prompt and message area. In the command window style interface, this function returns a pointer to the command window. In the status bar style interface, this function returns a pointer to the status bar.

See Also mdlDialog_keyinWindowGet, mdlDialog_statusAreaGet, mdlDialog_overallTitleBarGet, mdlDialog_menuBarGetCmdWinP.

mdlDialog_fileOpenExt

```
#include <filelist.h>
#include <rscdefs.h>
#include <msdialog.fdf>

int mdlDialog_fileOpenExt
(
    char                *fileNameP,      /* <= fully specified filename */
    FileOpenExtraInfo *extraInfoP,      /* <= extra selection info (or NULL) */
    FileOpenParams    *paramsP,        /* => file open dialog parameters */
    int                attributes       /* => file open dialog attributes */
);
```

Description The `mdlDialog_fileOpenExt` function lets the user conveniently select a file (through a dialog box) to open or create. The `mdlDialog_fileOpenExt` function can be used in place of `mdlDialog_fileOpen`, `mdlDialog_fileCreate`, `mdlDialog_defFileOpen`, `mdlDialog_defFileCreate` and `mdlDialog_fileCreateFromSeed`.

fileNameP returns the name of the file to be opened or created. On Open, *fileNameP* will be an existing file. On Create, if the file exists, an alert will display asking the user if the existing file should be overwritten. If the user chooses CANCEL, the user can choose another file name. If the user chooses OK, the file name will be returned.

extraInfoP returns additional information about the selected file. This structure will contain the name of the configuration variable used to select a directory (as in attaching a reference file) or the name of the seed file that should be used to create a file (as in creating a new design file). If this information is not needed, pass NULL.

paramsP is a structure containing file open parameters. The entire structure should be memset to 0 and then individual fields set as necessary.

```
typedef struct fileopenparams
{
    RscFileHandle  dialogRsch;
    int            dialogId;
    int            openCreate;
    char           *suggestedFileNameP;
    char           *defaultFilterP;
    char           *filterInfoStrP;
    char           *defaultDirP;
    char           *dirCfgVarP;
    char           *titleP;
    int            defFileId;
    RscFileHandle  defFileRsch;
    void           *dialogOwnerMD;
    char           *defSeedFileNameP;
    char           *defSeedDirP;
    char           *defSeedFilterP;
    int            defSeedFileId;
    int            futureUse[7];
    char           *futureUseCharP[4];
} FileOpenParams;
```

dialogRsch and *dialogId* allow custom file open dialogs to have the functionality of standard file open dialogs. *dialogRsch* is a handle to the resource file to use for loading a user-specified custom dialog box. If *dialogRsch* is NULL, the default resource file will be used.

dialogId is the ID of the dialog to use within the resource file. If *dialogId* is 0, the default dialog box will be used.

openCreate determines whether a file will be opened or created. It may be set to FILELISTATTR_OPEN, FILELISTATTR_CREATE or FILELISTATTR_CREATEFROMSEED.

suggestedFileNameP suggests a filename for creating a file. If a directory is attached to the filename, this argument serves as the default directory and the *defaultDirP* field is ignored. This field should be NULL when opening files.

defaultFilterP contains the filter to use for determining which files to include in the file list. It is useful for limiting files displayed to a particular type. Simple wildcarding is supported. An asterisk (*) matches any string, a question mark (?) matches any single character. If *defaultFilterP* is NULL, the filter string will match all files (*.*) .

filterInfoStrP contains a comma separated list of descriptive filter strings that will be used to populate the filter option button. Each filter entry requires a filter and a description. An example is: "*.dgn, MicroStation Design Files [*.dgn], *.h*, Hidden Line Files [*.h*]". If *filterInfoStrP* is NULL, *defaultFilterP* will be used as both the filter and the description.

defaultDirP contains the default starting directory for the file open dialog. It can also be a configuration variable. However, this directory can be overridden in two ways. First, if a *defFileId* is specified and a filename is retrieved from the saved history, its directory is used. Second, if no filename was retrieved from the history and *suggestedFileNameP* is set and contains a directory, it is used. If the default directory is not overridden then *defaultDirP* is used. If *defaultDirP* is NULL, the current working directory is used.

dirCfgVarP is the name of a configuration variable. This variable will be used to add directories to the "Directory" menu on standard default file open dialogs. A standard default file open dialog is selected by setting *dialogId* to 0 and specifying a *defFileId*.

titleP contains the title of the dialog box.

defFileId identifies a resource that stores a history of selected files. This resource is used to set the starting location for the file open dialog. If the user selects another file, this new filename is saved back to the *defFileRsch* resource file.

defFileRsch is the handle of a resource file opened by the calling application or NULL if the MicroStation user preferences file is to be used. This is where the default file information is loaded from and saved after the user makes a new file selection.

dialogOwnerMD only needs to be set when a custom file open dialog is supplied. It is the MDL descriptor of the application handling any hooks

on the custom file open dialog. `mdlSystem_getCurrMd1Desc` should be used to set *dialogOwnerMD*.

defSeedFileNameP only applies when *openCreate* is set to `FILELISTATTR_CREATEFROMSEED`. This field specifies the default file to copy when creating a new file (as in creating a new design file). This is only the default file to copy. The actual seed file that was used is returned in *extraInfoP->seedFileName*.

defSeedDirP, *defSeedFilterP*, *defSeedFileId* only apply when *openCreate* is set to `FILELISTATTR_CREATEFROMSEED`. They are identical to *defaultDirP*, *defaultFilterP* and *defFileId* except that these are used for the seed file dialog box. The seed file dialog box can be invoked from the create dialog box and lets the user specify which seed file to use in the creation of the new file.

futureUse and *futureUseCharP* are not currently used. These fields are reserved for future use.

attributes is `FILEOPENEXTATTR_CENTERONSCREEN` if the file open dialog is to be centered on the screen. Pass 0 if the default or saved dialog position is desired.



This function was implemented in MicroStation 95.

Returns `mdlDialog_fileOpenExt` returns `TRUE` if the CANCEL button is pressed, `FALSE` if the OK button is pressed, and `ERROR` if an error occurred while the dialog box was being created.

See Also `mdlDialog_fileOpen`, `mdlDialog_fileCreate`, `mdlDialog_defFileOpen`, `mdlDialog_defFileCreate`, `mdlDialog_fileCreateFromSeed`, `mdlFileList_edit`, `mdlSystem_getCurrMd1Desc`.

mdlDialog_keyinWindowGet

```
#include <msdialog.fdf>
```

```
DialogBox *mdlDialog_keyinWindowGet(void);
/* <= ptr to key-in window */
```

Description The `mdlDialog_keyWindowGet` function is provided so that MicroStation's main key-in area can be referenced without having to consider the currently selected interface style.



This function was implemented in MicroStation 95.

Returns `mdlDialog_keyinWindowGet` returns a pointer to the `DialogBox` that contains the main key-in area. In the command window style interface, this function returns a pointer to the command window. In the status bar style interface, this function returns a pointer to the key-in dialog if it is open and `NULL` otherwise.

See Also mdlDialog_commandWindowGet, mdlDialog_statusAreaGet,
mdlDialog_overallTitleBarGet, mdlDialog_menuBarGetCmdWinP.

Tab Page List and Tab Page Functions

The tab page list and tab page functions are used to manipulate the state of tab page lists and tab pages. See the tab page list and tab page section of the standard dialog box items chapter in the MDL Programmer's Reference for the item list and resource specifications of these items. Unless otherwise noted, these functions were initially implemented in MicroStation SE.

The following table lists tab page list and tab page functions:

Function	Used to
mdlDialog_tabPageListGetInfo	retrieve detailed information about tab page list.
mdlDialog_tabPageListSetInfo	set various options and attributes in a tab page list.
mdlDialog_tabPageListGetPageById	get the <code>DialogItem</code> pointer of the tab page with the given id.
mdlDialog_tabPageListGetPageByIndex	get the <code>DialogItem</code> pointer of the page with the given index within the tab page list.
mdlDialog_tabPageListFreePages	free the specified pages from the tab page list.
mdlDialog_tabPageListLoadPages	insert pages into the tab page list.
mdlDialog_tabPageGetInfo	retrieve detailed information about the tab page.
mdlDialog_tabPageSetInfo	set various options in the tab page.
mdlDialog_tabPageGetItemByTypeAndId	get the <code>DialogItem</code> pointer of the item with the given type and id.
mdlDialog_tabPageGetItemByIndex	get the <code>DialogItem</code> pointer of the item with the given index within the tab page.
mdlDialog_tabPageFreeItems	free the specified items from the tab page.
mdlDialog_tabPageLoadItems	insert items into the tab page.

mdlDialog_tabPageListGetInfo

```

BoolInt mdlDialog_tabPageListGetInfo
(
    int      *numPagesP,          /* <= NULL - don't want Num of pages */
    int      *firstPageIndexP,   /* <= NULL - don't want Index of 1st page */
    int      *lastPageIndexP,    /* <= NULL - don't want Index of last pg */
    ULONG    *commandNumberP,    /* <= NULL - don't want cmd number */
    ULONG    *commandSourceP,    /* <= NULL - don't want cmd source */
    ULONG    *attributesP,       /* <= NULL - don't want attributes */
    RawItemHdr *tPListRiP       /* => tabpagelist item to get info on */
);

```

Description mdlDialog_tabPageListGetInfo is used to retrieve information about a specific tab page list.

numPagesP points to a variable to receive number of pages in the tab page list.

firstPageIndexP points to a variable to receive index of the first page in the list.

lastPageIndexP points to a variable to receive index of the last page in the list.

attributesP points to a variable to receive the attributes of this list, such as TABATTR_DEFAULT, TABATTR_TABSBOTTOM and TABATTR_MULTITROW.

tPListRiP points to the raw item header of the tab page list to process.

Returns mdlDialog_tabPageListGetInfo returns TRUE if an error occurs, otherwise SUCCESS.

See Also mdlDialog_tabPageListSetInfo.

mdlDialog_tabPageListSetInfo

```

BoolInt mdlDialog_tabPageListSetInfo
(
    RawItemHdr *tPListRiP,       /* => tabpagelist item to set info */
    ULONG      *commandNumberP,  /* => NULL if not setting cmd number */
    ULONG      *commandSourceP,  /* => NULL if not setting cmd source */
    ULONG      *attributesP,     /* => NULL if not setting attributes */
    BoolInt    redraw           /* => TRUE means redraw */
);

```

Description mdlDialog_tabPageListSetInfo is used to set various options and attributes in a tab page list.

tPListRiP points to the raw item header of the tab page list to process.

attributesP points to a variable containing the new attributes for this list. The variable is constructed by combining the constants from the tab page list attributes table.

If *redraw* is TRUE, the tab page display area, including tabs and tab page items, will be redrawn after the attributes are set.

Returns mdlDialog_tabPageListGetSetInfo returns TRUE if an error occurs, otherwise SUCCESS.

See Also mdlDialog_tabPageListGetPageById.

mdlDialog_tabPageListGetPageById

```
DialogItem *mdlDialog_tabPageListGetPageById
(
  RawItemHdr  *tPListRiP,    /* => tabpagelist item to get page from */
  long        tabPageId,     /* => tab page id */
  int         startingIndex /* => starting db item index (usually 0) */
);
```

Description mdlDialog_tabPageListGetPageById is used get a pointer to the requested tab page item with the given ID.

tPListRiP points to the raw item header of the tab page list to process.

tabPageId specifies the ID of the tab page to get.

startingIndex indicates where to start looking for page within the tab page list.

Returns mdlDialog_tabPageListGetPageById returns a pointer to a DialogItem structure, or NULL if the specified page is not found.

See Also mdlDialog_tabPageGetItemByIndex.

mdlDialog_tabPageListGetPageByIndex

```
DialogItem *mdlDialog_tabPageListGetPageByIndex
(
  RawItemHdr  *tPListRiP,    /* => TabPageList to process */
  int         index          /* => index of the page to get */
);
```

Description mdlDialog_tabPageListGetPageByIndex is used to get a pointer to the requested tab page item with the given index within the tab page list.

tPListRiP points to the raw item header of the tab page list to process.

index specifies the index, within the tab page list, of the tab page to retrieve.

Returns mdlDialog_tabPageListGetPageByIndex returns a pointer to a DialogItem structure, or NULL if the specified page is not found.

See Also mdlDialog_tabPageListGetPageById.

mdlDialog_tabPageListFreePages

```
BoolInt    mdlDialog_tabPageListFreePages
(
RawItemHdr *tPListRiP,    /* => tabPageList contain pages to free */
int         startPageIndex, /* => start index of pages to free */
int         maxPageIndex   /* => last index of pages to free */
);
```

Description mdlDialog_tabPageListFreePages is used to free the specified pages from the tab page list.

tPListRiP points to the raw item header of the tab page list to process.

startPageIndex indicates the index of the first page within the list to free.

maxPageIndex indicates the index of the last page to free.

Returns mdlDialog_tabPageListFreePages returns TRUE if an error occurs, otherwise SUCCESS.

See Also mdlDialog_tabPageListLoadPages.

mdlDialog_tabPageListLoadPages

```
BoolInt mdlDialog_tabPageListLoadPages
(
RawItemHdr *tPListRiP,    /* => tabpagelist to insert pages into */
DialogItemListRsc *dilRP, /* => list of pages to load */
RscFileHandle rFileH,    /* => NULL - use opened resource files */
void         *ownerMD,    /* => NULL - owner items is owner of db */
int          beforePageIndex /* => item to load before, -1=append */
);
```

Description mdlDialog_tabPageListLoadPages is used to insert pages into tab page list.

tPListRiP points to the raw item header of the tab page list to process.

dilRP points to a DialogItemListRsc structure that specifies the pages to load. A DialogItemListRsc structure (defined in dlogbox.h) contains the number of pages to load and an array of DialogItemRsc structures.

rFileH specifies the resource file to search for the tab page item resources. If NULL is specified, all the calling application's open resource files will be searched, followed by MicroStation's open resource files.

ownerMD should be set to NULL.

beforePageIndex specifies the index of the page before which the new pages will be loaded. Specify -1 to indicate that the pages should be appended at the end of the tab page list.

Returns mdlDialog_tabPageListLoadPages returns TRUE if an error occurs, otherwise SUCCESS.

See Also mdlDialog_tabPageListFreePages, mdlResource_load.

mdlDialog_tabPageGetInfo

```
BoolInt mdlDialog_tabPageGetInfo
(
    int      *numItemsP,           /* <= Number of items on the page */
    int      *firstItemIndexP,    /* <= Index of 1st item on page */
    int      *lastItemIndexP,     /* <= Index of last item on page */
    ULong    *commandNumberP,     /* <= NULL if don't want cmd number */
    ULong    *commandSourceP,     /* <= NULL if don't want cmd source */
    ULong    *iconTypeP,          /* <= NULL if don't want type */
    long     *iconIdP,            /* <= NULL if don't want id */
    ULong    *attributesP,        /* <= NULL if don't want attributes */
    RawItemHdr *tPageRiP         /* => tabpage item to get info on */
);
```

Description mdlDialog_tabPageGetInfo is used to retrieve information about a specific tab page.

numItemsP points to a variable to receive the number of items on the tab page.

firstItemIndexP points to a variable to receive the index of first item on page.

lastItemIndexP points to a variable to receive the index of last item on page.

iconTypeP points to a variable to receive the type of icon on the tab.

iconIdP points to a variable to receive the Id of the icon on the tab.

attributesP points to a variable to receive the attributes of this page (not currently used).

tPageRiP points to the raw item header of the tab page to process.

Returns mdlDialog_tabPageGetInfo returns TRUE if an error occurs, otherwise SUCCESS.

See Also mdlDialog_tabPageSetInfo.

mdlDialog_tabPageSetInfo

```

BoolInt mdlDialog_tabPageSetInfo
(
    RawItemHdr *tPageRiP,      /* => tabpage item to set info in */
    ULong *commandNumberP,     /* => NULL if not setting cmd number */
    ULong *commandSourceP,     /* => NULL if not setting cmd source */
    ULong *iconTypeP,          /* => NULL if not setting type */
    long *iconIdP,             /* => NULL if not setting id */
    ULong *attributesP,        /* => NULL if not setting attributes */
    BoolInt redraw              /* => TRUE means redraw */
);

```

Description mdlDialog_tabPageSetInfo is used to set various options in the tab page.

tPageRiP points to the raw item header of the tab page.

iconTypeP points to a variable containing the updated type of the icon on tab.

iconIdP points to a variable containing the updated ID of the icon on the tab.

attributesP is not currently used. Set this to NULL.

If *redraw* is TRUE, the tab page display area, including tabs and tab page items, will be redrawn after the options are set.

Returns mdlDialog_tabPageSetInfo returns TRUE if an error occurs, otherwise SUCCESS.

See Also mdlDialog_tabPageListGetInfo.

mdlDialog_tabPageGetItemByTypeAndId

```

DialogItem *mdlDialog_tabPageGetItemByTypeAndId
(
    RawItemHdr *tPageRiP,      /* => tabPage to process */
    long type,                  /* => type of the item to get */
    long id                     /* => resource id of the item to get */
);

```

Description mdlDialog_tabPageGetItemByTypeAndId is used get a pointer to the requested dialog item with the given type and ID.

tPageRiP points to the raw item header of the tab page to search.

type is the type of the dialog item to search for.

id is the ID of the dialog item to search for.

Returns mdlDialog_tabPageGetItemByTypeAndId returns a pointer to a DialogItem structure, or NULL if the specified item is not found.

See Also mdlDialog_tabPageGetItemByIndex.

mdlDialog_tabPageGetItemByIndex

```
DialogItem *mdlDialog_tabPageGetItemByIndex  
(  
    RawItemHdr *tPageRiP,      /* => TabPage to process */  
    int         index          /* => index of the item to get */  
);
```

Description mdlDialog_tabPageGetItemByIndex is used get a pointer to the requested dialog item with the given index within the tab page.

tPageRiP points to the raw item header of the tab page to search.

index contains the index, within the tab page, of the dialog item to search for.

Returns mdlDialog_tabPageGetItemByIndex returns a pointer to a DialogItem structure, or NULL if the specified item is not found.

See Also mdlDialog_tabPageGetItemByTypeAndId.

mdlDialog_tabPageFreeItems

```
BoolInt mdlDialog_tabPageFreeItems  
(  
    RawItemHdr *tPageRiP,      /* => TabPage containing items to free */  
    int         startItemIndex, /* => start index of items to free */  
    int         maxItemIndex    /* => last index of items to free */  
);
```

Description mdlDialog_tabPageFreeItems is used to free the specified items from the tab page.

tPageRiP points to the raw item header of the tab page to process.

startItemIndex indicates the index of the first item to free from the page.

maxItemIndex indicates the index of the last item to free.

Returns mdlDialog_tabPageFreeItems returns TRUE if error occurs, otherwise SUCCESS.

See Also mdlDialog_tabPageLoadItems.

mdlDialog_tabPageLoadItems

```

BoolInt mdlDialog_tabPageLoadItems
(
    RawItemHdr      *tPageRiP,      /* => tabPage on which to load items */
    DialogItemListRsc *dilRP,        /* => list of items to load */
    RscFileHandle    rFileH,         /* => NULL, use open resource files */
    void             *ownerMD,       /* => NULL=owner items is owner of db */
    int              beforeItemIndex, /* => item to load before, -1=append */
    Point2d          *originP,       /* => items loaded relative to this pt.
                                     x,y > 0 in dcoord units else in
                                     pixels */
    BoolInt          drawItems        /* => TRUE means draw items after loading */
);

```

Description mdlDialog_tabPageListLoadPages is used to insert pages into the tab page list.

tPageRiP points to the raw item header of the tab page to process.

dilRP points to a `DialogItemListRsc` structure that specifies the items to load. A `DialogItemListRsc` structure (defined in `dlogbox.h`) contains the number of items to load and an array of `DialogItemRsc` structures.

rFileH specifies the resource file to search for the dialog item resources. If `NULL`, all the calling application's open resource files will be searched, followed by MicroStation's open resource files.

ownerMD should be set to `NULL`.

beforeItemIndex specifies the index of the item before which the new items will be loaded. Specify -1 to indicate that the items should be appended to the end of the tab page's item list.

originP specifies the origin used when loading the dialog items. If units of the point are greater than 0, they will be interpreted as dialog coordinate units; otherwise, they will be interpreted as pixel units. `NULL` sets the origin at (0, 0), the upper left corner of the tab page display area.

Returns mdlDialog_tabPageLoadItems returns `TRUE` if an error occurs, otherwise `SUCCESS`.

See Also mdlDialog_tabPageFreeItems, mdlResource_load.

ComboBox Item Functions

The combo box is a dialog box item provided with MicroStation SE that allows for the compact, option button-like selection among a various amount of selections. It differs from the option button in a few respects. Firstly, the size of the combo box is not based on the number of selections. Instead, the combo box allows for a predefined number of selections to display with the others accessible through scrolling. Also, the number of selections in a combo box is not static as in an option button. Additional

selections may be written to the combo box. However, selections may not be deleted or modified.

See the combo box item section of the standard dialog box items chapter in the MDL Programmer's Reference for the item list and resource specifications of this item. Unless otherwise noted, these functions were initially implemented in MicroStation SE.

The following table lists combo box item functions:

Function	Used to
<code>mdlDialog_comboBoxGetInfo</code>	get the attributes of a combo box item.
<code>mdlDialog_comboBoxSetInfo</code>	set the attributes of a combo box item.
<code>mdlDialog_comboBoxGetTextP</code>	return the <code>RawItemHdr</code> pointer of the attached text item.
<code>mdlDialog_comboBoxGetListBoxP</code>	return the <code>RawItemHdr</code> pointer of the attached popup list box.
<code>mdlDialog_comboBoxGetStrListP</code>	return a pointer to the string list that is currently connected to the combo box item.
<code>mdlDialog_comboBoxSetStrListP</code>	set the string list that the combo box item will manipulate.
<code>mdlDialog_comboBoxSetPopupState</code>	to programmatically open or close the popup list box.

`mdlDialog_comboBoxGetInfo, mdlDialog_comboBoxSetInfo`

```

BoolInt mdlDialog_comboBoxGetInfo
(
    ULong    *commandNumberP,          /* <= NULL, "don't want" cmd number */
    ULong    *commandSourceP,         /* <= NULL, " " cmd source */
    int      *maxSizeP,               /* <= NULL, " " max size */
    char     *formatToDisplayP,       /* <= NULL, " " format */
    char     *formatToInternalP,      /* <= NULL, " " format */
    char     *minimumP,              /* <= NULL, " " minimum */
    char     *maximumP,              /* <= NULL, " " maximum */
    ULong    *maskP,                 /* <= NULL, " " mask */
    UShort   *nRowsP,                /* <= NULL, " " Num rows in list */
    UShort   *gapWidthP,              /* <= NULL, " " Gap textedit/button */
    UShort   *listWidthP,            /* <= NULL, " " Width of listbox */
    ULong    *attributesP,           /* <= NULL, " " attributes */
    RawItemHdr *comboBoxP            /* => ComboBox item to get info on */
);

BoolInt mdlDialog_comboBoxSetInfo
(
    ULong    *commandNumberP,          /* <= NULL, "not setting" cmd number */

```

```

ULong *commandSourceP,          /* <= NULL, " " cmd source */
int *maxSizeP,                  /* <= NULL, " " max size (<256) */
char *formatToDisplayP,         /* <= NULL, " " format */
char *formatToInternalP,        /* <= NULL, " " format */
char *minimumP,                 /* <= NULL, " " minimum */
char *maximumP,                 /* <= NULL, " " maximum */
ULong *maskP,                   /* <= NULL, " " mask */
UShort *nRowsP,                 /* <= NULL, " " Num rows in list */
UShort *gapWidthP,              /* <= NULL, " " Gap textedit/button */
UShort *listWidthP,             /* <= NULL, " " Width of listbox */
ULong *attributesP,             /* <= NULL, " " attributes */
BoolInt redraw,                 /* => TRUE means redraw */
RawItemHdr *comboBP            /* => ComboBox item to get info on */
);

```

Description `mdlDialog_comboBoxGetInfo` and `mdlDialog_comboBoxSetInfo` are used to retrieve and set attributes of a combo box item. For all of the pointer parameters, `NULL` indicates to `mdlDialog_comboBoxGetInfo` that the field should not be modified, and indicates to `mdlDialog_comboBoxSetInfo` that the field is not desired.

maxSizeP points to a variable that indicates the maximum allowable size of the text string that can be contained by the combo box item, the maximum is 256 characters.

formatToDisplayP points to a `sprintf` format string to convert the value of the variable associated with the combo box item into a displayable string.

formatToInternalP points to a `scanf` format string to convert from the combo box item's string value to the format expected by the variable associated with the combo box item.

minimumP points to a string that contains the minimum value of the combo box item.

maximumP points to a string that contains the maximum value of the combo box item.

nRowsP points to an `int` that contains the number of rows displayed when the proper list box appears.

gapWidthP points to an `int` that contains the width, in dialog coordinate units (dcoords), between the selection field and the popup list activation button.

listWidthP points to an `int` that contains the width, in dialog coordinate units (dcoords), of the popup list box.

attributesP points to an `int` that contains the combo box item's attributes. The bits that can be included are defined in `dlogbox.h`.

redraw is a boolean flag that indicates whether `mdlDialog_comboBoxSetInfo` should cause the combo box to be redrawn after the information is modified.

Returns mdlDialog_comboBoxGetInfo and mdlDialog_comboBoxSetInfo return SUCCESS upon encountering no error and TRUE upon encountering an error.

mdlDialog_comboBoxGetTextP

```
RawItemHdr *mdlDialog_comboBoxGetTextP
(
    RawItemHdr *comboBP      /* => comboBox to get item rawItemP of */
);
```

Description mdlDialog_comboBoxGetTextP returns a RawItemHdr pointer of the text item attached to the combo box item.

mdlDialog_comboBoxGetListBoxP

```
RawItemHdr *mdlDialog_comboBoxGetListBoxP
(
    RawItemHdr *comboBP      /* => comboBox to get listBox rawItemP of */
);
```

Description mdlDialog_comboBoxGetListBoxP returns a RawItemHdr pointer of the list box attached to the combo box item.

mdlDialog_comboBoxGetStrListP, mdlDialog_comboBoxSetStrListP

```
StringList *mdlDialog_comboBoxGetTextP
(
    RawItemHdr *comboBP      /* => comboBox to get strListP of */
);

ErrorCode mdlDialog_comboBoxSetStrListP
(
    RawItemHdr *comboBP,      /* => comboBox to set strList of */
    StringList *strListP,     /* => list's string list */
    int nColumns              /* => number of columns in list */
);
```

Description mdlDialog_comboBoxGetStrListP retrieves a pointer to a string list that the combo box item specified by *comboBP* is manipulating. mdlDialog_comboBoxSetStrListP sets the string list pointed to by *strListP* to be the string list used by the combo box item specified by *comboBP*.

nColumns is the number of columns per row in the combo box popup list.

Returns mdlDialog_comboBoxSetStrListP returns TRUE if the combo box item's string list could not be properly set. Usually, this implies that either *comboBP* or *strListP* is invalid.

Description mdlDialog_comboBoxSetPopupState

```
BoolInt mdlDialog_comboBoxSetPopupState
(
    RawItemHdr *comboBoxP,      /* => comboBox to process */
    BoolInt     bOpen,          /* => TRUE=open, FALSE=close */
);
```

Description mdlDialog_comboBoxSetPopupState provides developers the means of programmatically opening or closing the popup list box.

Returns mdlDialog_comboBoxSetPopupState returns SUCCESS if the popup list box was properly opened or closed, and returns TRUE if there is an error.

SpinBox Item Functions

The spin box is dialog box item that allows for the compact selection of a value between a maximum and minimum value. While the functionality provided by the spin box is very similar to that of the scale item, it is more consistent with functionality provided by the Windows platforms. See the spin box item section of the standard dialog box items chapter in the MDL Programmer's Reference for the item list and resource specifications of this item. Unless otherwise noted, these functions were initially implemented in MicroStation SE.

The following table lists spin box item functions:

Function	Used to
mdlDialog_spinBoxGetInfo	retrieve attributes of SpinBox items.
mdlDialog_spinBoxSetInfo	set attributes of SpinBox items.

mdlDialog_spinBoxGetInfo and mdlDialog_spinBoxSetInfo

```
#include <dlogman.fdf>

BoolInt mdlDialog_spinBoxGetInfo
(
    ULONG *commandNumberP,    /* <= set NULL if don't want cmd number */
    ULONG *commandSourceP,    /* <= set NULL "" cmd source */
    int *maxSizeP,            /* <= set NULL "" max size */
    char *formatToDisplayP,    /* <= set NULL "" format */
    char *formatToInternalP,   /* <= set NULL "" format */
    double *minimumP,          /* <= set NULL "" minimum */
    double *maximumP,          /* <= set NULL "" maximum */
    double *incAmountP,        /* <= set NULL "" incAmount */
    ULONG *maskP,              /* <= set NULL "" mask */
    ULONG *attributesP,        /* <= set NULL "" attributes */
);
```

```
RawItemHdr *spinBP          /* => SpinBox item to get info on */
);

BoolInt mdlDialog_spinBoxSetInfo
(
    ULong *commandNumberP,    /* => set NULL "if not setting" cmd num */
    ULong *commandSourceP,    /* => set NULL "" cmd source */
    int *maxSizeP,            /* => set NULL "" maxSize<256 */
    char *formatToDisplayP,    /* => set NULL "" format */
    char *formatToInternalP,   /* => set NULL "" format */
    double *minimumP,          /* => set NULL "" minimum */
    double *maximumP,          /* => set NULL "" maximum */
    double *incAmountP,        /* <= set NULL "" incAmount */
    ULong *maskP,              /* => set NULL "" mask */
    ULong *attributesP,        /* => set NULL "" attributes */
    BoolInt redraw,            /* => TRUE means redraw */
    RawItemHdr *spinBP        /* => SpinBox item to get info on */
);
```

Description mdlDialog_spinBoxGetInfo and mdlDialog_spinBoxSetInfo are used to retrieve and set attributes of SpinBox items. For all of the pointer parameters, NULL indicates to mdlDialog_spinBoxGetInfo that the field should not be modified, and indicates to mdlDialog_spinBoxSetInfo that the field is not desired.

maxSizeP points to a variable indicating the maximum allowable size of the text string that can be contained by the SpinBox item, up to 256 characters.

formatToDisplayP points to a sprintf format string to convert the value of the variable associated with the SpinBox item into a displayable string.

formatToInternalP points to a sscanf format string to convert from the SpinBox item's string value to the format expected by the variable associated with the SpinBox item.

minimumP points to a double that contains the minimum value of the SpinBox item.

maximumP points to a double that contains the maximum value of the SpinBox item.

incAmountP points to a double that contains the increment (and decrement) value.

attributesP points to an int that contains the SpinBox item's attributes. The bits that can be included are defined in dlogbox.h.

redraw is a boolean flag indicating whether mdlDialog_spinBoxSetInfo should cause the SpinBox to be redrawn after the information is modified.

Returns mdlDialog_spinBoxGetInfo and mdlDialog_spinBoxGetInfo return SUCCESS on success or TRUE if there is an error.

4

System Functions

The system functions provide operating system functionality. They provide an interface to MDL's **runtime environment**, not to the underlying operating system.

MDL distinguishes between the terms **program**, **application** and **task**. These terms are all defined in “MDL Applications.” The relationship between the terms can be summarized as follows:

- MDL loads and unloads programs.
- A program is loaded from an application file. An application file can contain more than one program.
- When a program is loaded, MDL creates a task.
- When a program is unloaded, MDL destroys the task.

This chapter contains the following sections:

- System Functions
- Configuration Variable Functions

System Functions

The following table lists system functions:

Function	Used to
<code>mdlSystem_exit</code>	terminate the MDL application.
<code>mdlSystem_loadMdlProgram</code>	load the specified MDL program.
<code>mdlSystem_unloadMdlProgram</code>	unload the specified MDL program.
<code>mdlSystem_createStartupElement</code>	create an MDL program startup element.
<code>mdlSystem_deleteStartupElement</code>	delete an MDL program startup element.
<code>mdlSystem_enterGraphics</code>	put MicroStation in graphic mode.
<code>mdlSystem_enterGraphicsExtended</code>	put MicroStation in graphic mode without opening a design file.
<code>mdlSystem_newDesignFile</code>	close the open (active) design file and open a new one.
<code>mdlSystem_closeDesignFile</code>	close the open design file.
<code>mdlSystem_fileDesign</code>	save MicroStation's settings.
<code>mdlSystem_getChar</code>	poll the keyboard.
<code>mdlSystem_getCurrTaskID</code>	retrieve the pointer to the current MDL task id.
<code>mdlSystem_getMdlTaskList</code>	list names of MDL tasks.
<code>mdlSystem_getTaskStatistics</code>	get statistics on the specified MDL task.
<code>mdlSystem_pauseTicks</code>	pause MicroStation.
<code>mdlSystem_getTicks</code>	determine elapsed time.
<code>mdlSystem_abortRequested</code>	determine if the user has tried to abort an MDL task.
<code>mdlSystem_userAbortEnable</code>	control whether the user can abort an MDL task.
<code>mdlSystem_flushDesignFile</code>	flush the design file.
<code>mdlSystem_extendedAbortEnable</code>	control whether the user can abort an MDL task.
<code>mdlSystem_extendedAbortRequested</code>	determine if the user has tried to abort an MDL task.
<code>mdlSystem_enterDebug</code>	invoke the MDL debugger from the program.
<code>mdlSystem_computeDesignRange</code>	compute range of a design file's geometry.
<code>mdlSystem_findMdlDesc</code>	get the MDL descriptor corresponding to a task ID.
<code>mdlSystem_getCurrMdlDesc</code>	retrieve a pointer to the current MDL descriptor.

Function	Used to
<code>mdlSystem_getMdlTaskID</code>	get the task ID corresponding to an MDL descriptor.
<code>mdlSystem_nextMdlApp</code>	step through MicroStation's list of MDL tasks.
<code>mdlSystem_getUstn0SPid</code>	get MicroStation's process ID.
<code>mdlSystem_getProcessNumberFromMdlDesc</code>	get an MDL task's process number.
<code>mdlSystem_getMdlDescFromProcessNumber</code>	derive an MDL descriptor from a process number.
<code>mdlSystem_setMdlAppClass</code>	modify an application's application class.
<code>mdlSystem_setTimerFunction</code>	set up a timer function, <code>userSystem_timerExpired</code> , to be called after a specified time.
<code>mdlSystem_cancelTimer</code>	cancel an outstanding timer request.
<code>mdlSystem_flushWriteCache</code>	causes MicroStation to flush any information in the write cache to disk.
<code>mdlSystem_compressDgnFile</code>	compress deleted elements from master design file and write to disk.
<code>mdlSystem_compressLibrary</code>	compress deleted cells in current cell library and write to disk.
<code>mdlSystem_startBusyCursor</code> (Macintosh only)	causes the graphical cursor to indicate that work is in progress.
<code>mdlSystem_stopBusyCursor</code> (Macintosh only)	changes the graphical cursor back to normal after a call to <code>mdlSystem_startBusyCursor</code> (Macintosh only).
<code>mdlSystem_setFunction</code>	designate a <i>userSystem</i> user function.
<code>mdlSystem_CADInputJournalActive</code>	check whether Cad input journaling is active.
<code>mdlSystem_elapsedTime</code>	return elapsed time in seconds since the MicroStation session was started.
<code>mdlSystem_exchangeDesignFile</code>	open a design file with the view configuration of the active design file.
<code>mdlSystem_getWorkspaceList</code>	get information about available workspaces, project configuration files and custom user interfaces.
<code>mdlSystem_journalAccessStrByTaskID</code>	describe access string.
<code>mdlSystem_journalAccessString</code>	journal the described access string, if Cad input journaling is active.
<code>mdlSystem_journalAppMessage</code>	journal the application-specific string, if Cad input journaling is active.

Function	Used to
mdlSystem_journalCommand	journal described command, if Cad input journaling is active.
mdlSystem_journalCommandString	journal the given string as a command, if Cad input journaling is active.
mdlSystem_journalDataPoint	journal described data point, if Cad input journaling is active.
mdlSystem_journalKeyin	journal the key-in, if Cad input journaling is active.
mdlSystem_journalReset	journal a reset, if Cad input journaling is active.
mdlSystem_journalTentativePoint	journal described tentative point, if Cad input journaling is active.
mdlSystem_saveDesignFile	simulate File>Save when “save design changes” is off.

The following table lists system user functions. The programmers determines the user function names. These functions are all designated to MDL through function pointers. MDL does not use the names. The following names are used merely as illustrations.

Function	MicroStation calls when
userSystem_mdIChildTerminated	a child task terminates.
userSystem_reloadProgram	MicroStation receives a load request for a program that is already loaded.
userSystem_unloadProgram	MicroStation is trying to unload the program.
userSystem_newDesignFile	MicroStation loads a new design file.
userSystem_timerExpired	a user-specified timer expires.
userSystem_writeToFile	MicroStation writes to a file.
userSystem_saveAs	called after MicroStation saves the design file under a different name.
userSystem_elmDscrToFile	an element descriptor is about to be written to the design file.
userSystem_colorMapChange	a new color mapping is generated between the design file colortable and the video hardware colors.
userSystem_menuBarChange	a menu bar is activated or deactivated.
userSystem_referenceAttach	a reference file is about to be attached.
userSystem_allMDLUnloads	learn about MDL applications being unloaded.

Function	MicroStation calls when
<code>userSystem_exitDesignFileState</code>	learn when MicroStation is exiting design file state.
<code>userInput_fileOpenDialogPreprocess</code>	called before a file open dialog is opened.
<code>userShare_sharedLibNoMoreClients</code>	a shared library has no more client MDL applications.
<code>userSystem_fenceChanged</code>	a fence is either created or cleared.
<code>userSystem_save</code>	called before and after saving design file changes.

Example

See the `system.mc` example.

`mdlSystem_exit`

```
void mdlSystem_exit
(
    int      exitStatus,    /* => status for parent task */
    int      unload        /* => 1 to unload program */
);
```

Description `mdlSystem_exit` terminates the MDL task. MDL does not return to the MDL task. An MDL task can use this function to abort regardless of how deeply it is nested in function calls.

exitStatus is the function's exit status. If the MDL task was started by another MDL task, the exit status is returned to the parent task.

A non-zero value for *unload* tells MDL to unload the program. If *unload* is zero, MDL aborts the MDL task and reinitializes the stack pointer.

If the program is unloaded, MDL frees all memory, flushes, and closes all files opened by the MDL task. There are two known exceptions to this: element descriptors and string lists. An MDL program must specifically free all of the string lists and element descriptors created for it. Use the `mdlElmdscr_freeAll` function to free element descriptors. Use `mdlStringList_destroy` to free memory allocated through string list functions.

If the program remains loaded, MDL performs no clean-up other than resetting the stack pointer to the initial value.

Please keep the following points in mind when using `mdlSystem_exit` and `exit`:

- It is very dangerous to call `exit` or `mdlSystem_exit` from any hook function. It is better to queue a request to unload the application. See

the application described in the “A Complete Example” chapter of the *MDL Programmer's Guide* for an example of how to do this. It queues an unload request when it receives `DIALOG_MESSAGE_DESTROY` message. If your application needs to set an exit code, then your application can queue a command to itself and then in the command it can call `exit` or `mdlSystem_exit`.

- To detect problems related to calling `exit` or `mdlSystem_exit` at an improper time, run MicroStation with `MS_DEBUGHEAP` set to 1 and `MS_DEBUGMDLHEAP` set to ALL or the name of your application. See the “Debugging” chapter of the *MDL Programmer's Guide* for more information on these environment variables.

Returns The `mdlSystem_exit` function does not return to the MDL task.

See Also `exit`, `userSystem_md1ChildTerminated`.

mdlSystem_loadMdlProgram, mdlSystem_unloadMdlProgram

```
int mdlSystem_loadMdlProgram
(
    char    *appFileP,      /* => application file name */
    char    *taskIdP,       /* => task name, can be NULL */
    char    *argumentP      /* => argument for task's main */
);

int mdlSystem_unloadMdlProgram
(
    char    *taskIdP        /* => task name */
);
```

Description `mdlSystem_loadMdlProgram` loads the program specified by *taskIdP* from the resource file specified by *appFileP*. If *appFileP* is NULL, the new program is loaded from the same resource file as the parent task. If the task has a function *main*, *main* is executed before the calling function is resumed. *argumentP* is passed to *main*.

If the program is already loaded and does not have a reload user function, an error occurs. If the program is already loaded and has a reload user function, MicroStation calls that reload user function.

The `mdlSystem_unloadMdlProgram` function unloads the MDL program specified by *taskIdP*. The MDL task to be unloaded can reject the unload request by returning a non-zero value from the `userSystem_unloadProgram` user function.

Returns The `mdlSystem_loadMdlProgram` function returns `SUCCESS` if it is successful. Otherwise, it returns -1.

The `mdlSystem_unloadMdlProgram` function returns zero if it is successful. Otherwise, it returns a non-zero value and more detailed error information is available in `mdlErrno`.

The values for `mdlErrno` are defined in `mdlerrs.h`.

See Also `userSystem_md1ChildTerminated`, `userSystem_unloadProgram`,
`userSystem_reloadProgram`.

mdlSystem_createStartupElement, mdlSystem_deleteStartupElement

```
#include <mselems.h>

int mdlSystem_createStartupElement
(
MSElementUnion    *outElementP,  /* <= the created element */
char               *startStringP  /* => string to start program */
);

int mdlSystem_deleteStartupElement
(
char               *programNameP  /* => program name */
);
```

Description The `mdlSystem_createStartupElement` program creates a MicroStation start-up element to be placed in the design file. After creating a startup element, use the `mdlElement_add` function to add it to the design file.

If MicroStation finds a startup element when loading a design file, it queues a command to start the MDL program. Once MicroStation has completely loaded the design file, it begins processing all queued commands, including commands resulting from startup elements.

startStringP should contain the exact string the user needs to start the program. It can contain command line arguments that are passed to the MDL program when it is started.

The `mdlSystem_deleteStartupElement` function finds all start-up elements in the current design file that start *programNameP*. The function then deletes these elements.

Returns The `mdlSystem_createStartupElement` returns zero if it is successful. If the specified string is over 1000 characters, the function returns `MDLERR_BADSTRING`.

The `mdlSystem_deleteStartupElement` function returns zero if it is successful. If the specified string is empty or too long, the function returns `MDLERR_BADSTRING`.

See Also `mdlElement_add`.

mdlSystem_enterGraphics

```
void mdlSystem_enterGraphics();
```

Description The `mdlSystem_enterGraphics` function initializes graphics. If graphics have been initialized, `mdlSystem_enterGraphics` returns. The `mdlSystem_enterGraphics` function is intended for programs started before MicroStation initializes graphics.

Returns The `mdlSystem_enterGraphics` function is of type `void`. It returns no value.

See Also `mdlSystem_enterGraphicsExtended`, `mdlSystem_newDesignFile`, `mdlSystem_closeDesignFile`, the “Using MS_INITAPPS Applications” section in the “Developing Applications” chapter of the MDL Programmer’s Guide.

mdlSystem_enterGraphicsExtended

```
void mdlSystem_enterGraphicsExtended
(
    int      flags      /* 0=display views, 1=don't display views */
);
```

Description The `mdlSystem_enterGraphicsExtended` function serves the same purpose as the function `mdlSystem_enterGraphics`, except that it allows applications to initialize MicroStation’s graphics environment without displaying design files. This is useful, for example, in `INITAPPS` applications that process design files and want to display dialog boxes, but don’t want the drawing displayed on the screen.

The *flags* parameter determines whether MicroStation displays design file views or not. If *flags* is 0, MicroStation does display them, if *flags* is 1 it does not. Other values for *flags* are reserved.

Returns `mdlSystem_enterGraphicsExtended` is of type `void`. It does not return a status.

See Also `mdlSystem_enterGraphics`.

mdlSystem_newDesignFile

```
#include <mdl.h>

int mdlSystem_newDesignFile
(
    char      *nameP      /* => name of design file */
);
```

Description The `mdlSystem_newDesignFile` function opens a design file after closing the open (active) design file.

nameP specifies the name of the file. The filename path or extension does not need to be supplied. MicroStation assumes the `.dgn` extension. It searches all directories specified by the MicroStation environment variable `MS_DEF`.

The `mdlSystem_newDesignFile` function reinitializes most of MicroStation. For example, it clears the state functions, reads the type 9 and 66 elements and initializes the Terminal Control Block (TCB) accordingly, and resets all undo pointers.

If the mdlSystem_newDesignFile function is unable to initialize the TCB, it displays a message and exits MicroStation.

If the end-of-file is missing, the mdlSystem_newDesignFile function asks the user if the file should be fixed. It attempts to fix the file upon user request.

The mdlSystem_newDesignFile function places queue elements in the input queue and invokes MicroStation's queue processing loop. This action could affect a task that has command or queue filters (such as those set using mdlInput_setMonitorFunction) or that is queuing elements before calling mdlSystem_newDesignFile.

Returns The mdlSystem_newDesignFile returns SUCCESS if no errors occur.

mdlSystem_closeDesignFile

```
void mdlSystem_closeDesignFile();
```

Description The mdlSystem_closeDesignFile function closes the current design file. If no MS_INITAPPS applications are installed, mdlSystem_closeDesignFile terminates MicroStation. Otherwise, the MS_INITAPPS task is resumed.

Returns The mdlSystem_closeDesignFile function is of type void. It returns no value.

mdlSystem_fileDesign

```
int mdlSystem_fileDesign();
```

Description The mdlSystem_fileDesign function produces the same result as selecting Save Settings from the File menu. All savable settings (those stored in the type 9 and 66 elements) are saved in the design file.

Returns The mdlSystem_fileDesign function returns SUCCESS if no errors occur. Otherwise, it returns a non-zero value.

mdlSystem_getChar

```
int mdlSystem_getChar();
```

Description The mdlSystem_getChar function polls the keyboard looking for a character. It returns immediately, regardless of whether a character is available.

Returns The mdlSystem_getChar function returns 0 if a character is not available. Otherwise, it returns the character.

See Also userState_keyin, mdlDialog_openAlert.

mdlSystem_getCurrTaskID

```
char *mdlSystem_getCurrTaskID();
```

Description The `mdlSystem_getCurrTaskID` function returns a pointer to the task ID of the application that calls it. Applications that require a task ID should use this function rather than relying on a hard-coded value.

Returns The `mdlSystem_getCurrTaskID` function returns a pointer to the application's task ID.

mdlSystem_getMdlTaskList

```
char *mdlSystem_getMdlTaskList();
```

Description The `mdlSystem_getMdlTaskList` function generates a list of installed MDL task names. The task calling `mdlSystem_getMdlTaskList` must call `free` once to free the entire list.

The first four bytes of the list contain a count of names in the list. The first task name begins after the count.

Returns `mdlSystem_getMdlTaskList` returns a pointer to the list of names.

See Also `mdlSystem_getMdlTaskID`.

mdlSystem_getTaskStatistics

```
#include <system.h>

int mdlSystem_getTaskStatistics
(
    MdlAppStatistics *statisticsP, /* <=> statistics buffer */
    char             *taskIdP     /* <=> name of task */
);
```

Description The `mdlSystem_getTaskStatistics` function collects statistics on the task specified by *taskIdP*. It returns them in the structure that *statisticsP* points to. *taskIdP* points to the task ID of the desired MDL task.

statisticsP can be `NULL` if `mdlSystem_getTaskStatistics` is used only to determine whether a program is installed.

Returns The `mdlSystem_getTaskStatistics` function returns `SUCCESS` if the task is still installed. Otherwise, it returns a non-zero value.

mdlSystem_pauseTicks, mdlSystem_getTicks

```
void mdlSystem_pauseTicks
(
    int    sixtyeths /* => pause interval */
);

int mdlSystem_getTicks();
```

Description The `mdlSystem_pauseTicks` function pauses the MDL program and MicroStation for at least the specified interval. MicroStation pauses in an idle loop. It is inactive during the interval.

The `mdlSystem_getTicks` function returns the number of ticks since an arbitrary time. The arbitrary time is constant, so two calls to `mdlSystem_getTicks` determines the interval between calls.

The ticks are approximately, but not exactly, one-sixtieth of a second. On some platforms a time resolution this fine is not available. MicroStation attempts to be as close as possible to sixtieths.

Returns `mdlSystem_pauseTicks` is of type `void`. It returns no value.

`mdlSystem_getTicks` returns the number of ticks since an arbitrary time.

See Also `mdlSystem_setTimerFunction`.

mdlSystem_abortRequested, mdlSystem_userAbortEnable

```
int mdlSystem_abortRequested();  
void mdlSystem_userAbortEnable  
(  
    int    enableOn /* => 0 prevents asynchronous aborts */  
);
```

Description The `mdlSystem_userAbortEnable` function controls whether the user can abort an MDL task. If an MDL task never calls `mdlSystem_userAbortEnable`, the user can abort the MDL task by pressing <Ctrl-C>. The MDL task can disable this capability by calling `mdlSystem_userAbortEnable` with an argument of 0. The MDL task can enable this capability by calling `mdlSystem_userAbortEnable` with a non-zero argument. By default, this capability is enabled.

An MDL task can call `mdlSystem_abortRequested` to determine whether the user has tried to abort the task. `mdlSystem_abortRequested` works even if the task disables the abort capability. `mdlSystem_abortRequested` does not actually abort the task. It simply allows the task to determine whether an abort has been requested.

Returns The `mdlSystem_userAbortEnable` function is of type `void`. It returns no value.

The `mdlSystem_abortRequested` function returns a non-zero value if the user requested an abort.

See Also `mdlSystem_extendedAbortEnable, mdlSystem_extendedAbortRequested`.

mdlSystem_flushDesignFile

```
#include <mssystem.fdf>  
void mdlSystem_flushDesignFile(void);
```

Description The `mdlSystem_flushDesignFile` function causes MicroStation to flush the design file. When elements are added MicroStation writes them immediately. However, the operating system may buffer them. The `mdlSystem_flushDesignFile` function causes MicroStation to force the operating system to flush the design file.

This function is rarely needed. It is only needed for external programs that access the design file separately. For example, if an MDL program spawned an external program to copy the design file, it would be appropriate to call `mdlSystem_flushDesignFile` prior to spawning the program.

Returns `mdlSystem_flushDesignFile` is of type `void`. It does not return anything.

mdlSystem_extendedAbortEnable, mdlSystem_extendedAbortRequested

```
void mdlSystem_extendedAbortEnable
(
    boolean enable    /* <= TRUE to enable, FALSE to disable */
);

int mdlSystem_extendedAbortRequested(void);
```

Description `mdlSystem_extendedAbortEnable` and `mdlSystem_extendedAbortRequested` allow an MDL application to check for a user abort using the same mechanism as MicroStation's update and fence processing. This lets the user abort a process using Escape, Reset or <Ctrl-C>. The other methods of signaling abort limit the user to pressing <Ctrl-C> to abort the application.

If *enable* is `TRUE`, MicroStation turns on the extended abort processing. In doing so, it resets the MicroStation flag that indicates whether an abort has been requested. It also turns off the standard abort processing for that application by calling `mdlSystem_userAbortEnable` with an argument of 0.

If *enable* is `FALSE`, MicroStation turns off extended abort processing. In doing so, it does not reset the MicroStation flag that indicates whether an abort has been requested. That value can still be tested using `mdlSystem_extendedAbortRequested`. MicroStation also enables standard abort processing for the application by calling `mdlSystem_userAbortEnable` with an argument of 0.

The `mdlSystem_extendedAbortRequested` function allows an MDL application to determine if the user requested anytime since the last call to `mdlSystem_extendedAbortEnable`.

An MDL application should not keep the abort testing constantly enabled. It should enable it at the beginning of time-consuming task, and disable it at the end.

Returns `mdlSystem_extendedAbortEnable` is of type `void`. It returns no value. `mdlSystem_extendedAbortRequested` returns a non-zero value if the user has requested an abort. It returns 0 if the user has not requested an abort.

See Also mdlSystem_userAbortEnable, mdlSystem_abortRequested.

mdlSystem_enterDebug

```
void mdlSystem_enterDebug();
```

Description The mdlSystem_enterDebug function invokes the MDL debugger, loading the program's debugging information if it is not already loaded.

Returns The mdlSystem_enterDebug function is of type void. It returns no value.

mdlSystem_computeDesignRange

```
int mdlSystem_computeDesignRange
(
    Dpoint3d      *minP,           /* <= range minimum */
    Dpoint3d      *maxP,           /* <= range maximum */
    boolean       *elementFoundP, /* <= TRUE if element found (or NULL) */
    int           view,            /* => view number (or -1) */
    RotMatrix     *rotMatrixP,     /* => rot matrix (NULL for identity) */
    ULong         *fileMaskP      /* => file mask (NULL for all files) */
);
```

Description mdlSystem_computeDesignRange computes the range of design file geometry. This function scans through the file(s) and computes the union of the element range blocks, in a similar manner to the MicroStation “fit” command.

minP and *maxP* point to the range minimum and maximum.

elementFoundP points to an integer that is set to TRUE if an element is found. If no elements are found, the range values are not valid.

If *view* is not negative, only elements displayed in *view* are processed. The range calculations are performed in a coordinate system rotated by the inverse of the view rotation matrix (unless overridden with *rotMatrixP*).

rotMatrixP is a pointer to a rotation matrix for the range calculations. If NULL is passed (and *view* is negative), the range calculations are performed in design file coordinates (no rotation).

If *fileMaskP* is NULL, the master file and all reference files are processed, otherwise it points to an array of eight long integers that control the files that are processed. See mdlView_updateMulti for a description of the file bitmap control.

Returns mdlSystem_computeDesignRange returns SUCCESS if the range is calculated successfully and an appropriate error code otherwise.

mdlSystem_findMdlDesc, mdlSystem_getCurrMdlDesc, mdlSystem_getMdlTaskID

```
#include <mssystem.fdf>

void *mdlSystem_findMdlDesc
(
    char    *taskIdP          /* => task whose descriptor to get */
);

void *mdlSystem_getCurrMdlDesc(void);

char *mdlSystem_getMdlTaskID
(
    void    *mdlDescP        /* => task whose task ID to get */
);
```

Description The `mdlSystem_findMdlDesc` function gets a pointer to the MDL descriptor associated with the task ID.

The `mdlSystem_getCurrMdlDesc` gets a pointer to the MDL descriptor of the currently active MDL application. That is, it returns a pointer to the MDL descriptor of the application that makes the call to `mdlSystem_getCurrMdlDesc`.

The `mdlSystem_getMdlTaskID` function gets a pointer to the task ID of the application identified by *mdlDescP*.

The `mdlSystem_findMdlDesc` function gets a pointer to the MDL descriptor associated with the task ID.

An MDL descriptor is a structure that MicroStation uses to track the status of an MDL application. The format of the structure itself is not published. Wherever a pointer to an MDL descriptor is used, it is just declared as `void *`. Typically, MDL programs do not need to be concerned with the MDL descriptor.

Usually the task ID is just the base name of the application's file name. In some cases, a specific task ID is specified via the MDL linker. The task ID is not case sensitive. Prior to using the task ID, MicroStation always makes a copy of it and converts it to upper case.

Both the task ID of an MDL application and the pointer to the MDL descriptor uniquely identify the MDL program. To derive the MDL descriptor from the task ID, use the built-in function `mdlSystem_findMdlDesc`. To derive the task ID from the MDL descriptor, use the built-in function `mdlSystem_getMdlTaskID`. A program can retrieve its own MDL descriptor using the built-in function `mdlSystem_getCurrMdlDesc`.

Returns `mdlSystem_findMdlDesc` returns a pointer to an MDL descriptor. If the named task is not loaded, it returns `NULL`. `mdlSystem_getCurrMdlDesc` returns a pointer to an MDL descriptor. `mdlSystem_getMdlTaskID` returns a pointer to a task ID.

See Also `mdlSystem_getMdlTaskList`.

mdlSystem_nextMdlApp

```
#include <mssystem.fdf>

void *mdlSystem_nextMdlApp
(
    void *descP    /* => MDL Descriptor from prev. call or NULL */
);
```

Description The mdlSystem_nextMdlApp function allows an MDL program to step through MicroStation's list of MDL tasks.

descP points to an MDL descriptor. It typically is NULL, or the value returned from the previous call to mdlSystem_nextMdlApp.

To step through MicroStation's list of MDL tasks, first call mdlSystem_nextMdlApp passing NULL for *descP*. Next call mdlSystem_nextMdlApp specifying the MDL descriptor returned from the previous call to mdlSystem_nextMdlApp. Continue this until mdlSystem_nextMdlApp returns NULL.

Returns mdlSystem_nextMdlApp returns a pointer to an MDL descriptor if *descP* is NULL, or points to the MDL descriptor that is next in the list. If *descP* points to the last entry in the list, mdlSystem_nextMdlApp returns NULL.

See Also mdlSystem_findMdlDesc.

mdlSystem_getUstnOSPid

```
#include <mssystem.fdf>
#include <msextern.h>

void mdlSystem_getUstnOSPid
(
    ProcessID *pidP    /* <= location of process ID */
);
```

Description The mdlSystem_getUstnOSPid function gets MicroStation's OS process number. This is called a process ID on some systems. The process ID is assigned by the operating system to the MicroStation process.

pidP points to a location to receive the process ID.

Returns mdlSystem_getUstnOSPid is of type void. It does not return anything.

**mdlSystem_getProcessNumberFromMdlDesc,
mdlSystem_getMdlDescFromProcessNumber**

```

long mdlSystem_getProcessNumberFromMdlDesc
(
    void      *mdlDescP          /* => MDL descriptor */
);

long mdlSystem_getMdlDescFromProcessNumber
(
    void      *processNumber     /* => MDL task ID */
);

```

Description The `mdlSystem_getProcessNumberFromMdlDesc` function returns the process number of the specified MDL task. The `mdlSystem_getMdlDescFromProcessNumber` function returns a pointer to the MDL descriptor corresponding to the specified process number.

processNumber is a number that uniquely identifies an MDL task.

mdlDescP is a pointer to an MDL descriptor.

MicroStation assigns a number to the MDL task when it loads the application. MicroStation does not reuse any process numbers during a MicroStation session. Even after an application is unloaded, that application's process number will not refer to another MDL application.

MicroStation uses the process number as a task identifier in places where it is absolutely essential that the identifier cannot refer to another task. In such cases, MicroStation cannot use the task ID or MDL descriptor because either of these may be reused after the application is unloaded.

MicroStation stores the process number with UNDO information to identify the MDL task that caused the transaction. This process number is part of the UNDO information that is passed to `userUndo_addToBuffer` and `userUndo_command` user functions.

Returns `mdlSystem_getProcessNumberFromMdlDesc` returns the process number that uniquely identifies the MDL task. If the corresponding MDL application is still loaded, `mdlSystem_getMdlDescFromProcessNumber` returns a pointer to the task's MDL descriptor. Otherwise, it returns `NULL`.

See Also `mdlSystem_getCurrMdlDesc`, `mdlSystem_getMdlTaskID`, `mdlSystem_findMdlDesc`, `userUndo_addToBuffer` and `userUndo_command`.

mdlSystem_setMdlAppClass

```
#include <mdl.h>
#include <msdefs.h>
#include <mssystem.fdf>

int mdlSystem_setMdlAppClass
(
    char    *taskIdP,      /* => application whose class to modify */
    int     newClass       /* => new class to be assigned */
);
```

Description The mdlSystem_setMdlAppClass function modifies the application class of a loaded MDL application.

taskIdP specifies the application to be effected. If an application is modifying its own application class, it can specify NULL for *taskIdP*.

newClass specifies the new class. It must be one of the APPLICATION_ values defined in msdefs.h. Application programs normally have class APPLICATION_INITAPP if the application was started from the MS_INITAPPS environment variable, APPLICATION_DGNAPP if the application was started from the MS_DGNAPPS environment variable, or APPLICATION_USER if it was started any other way. Some MS_INITAPPS applications that start other MDL applications need the child application to remain loaded. They want to prevent the application from being unloaded if MicroStation exits from the design file state. To prevent this, set the child application's class to APPLICATION_INITAPP after starting it.

Returns mdlSystem_setMdlAppClass returns -1 if the application is not loaded. Otherwise, it returns the application's old class.

mdlSystem_setTimerFunction, mdlSystem_cancelTimer

```
int mdlSystem_setTimerFunction
(
    int          *timerHandleP,      /* <= handle of timer */
    long         duration,           /* => time before calling functionP */
    MdlFunctionP function,           /* => address of user function */
    long         userArg,            /* => argument to user function */
    boolean      continuous          /* => TRUE for repeating calls */
);

void mdlSystem_cancelTimer
(
    int          timerHandle          /* => timer to cancel */
);
```

Description The mdlSystem_setTimerFunction function designates a timer function to be called after *duration* timer ticks. A tick is approximately (but not exactly) one-sixtieth of a second. The output *timerHandleP* is set to a unique integer handle that cancels the

timer using `mdlSystem_cancelTimer`. When the user-designated function *function* is called, the long-sized argument *userArg* is passed to it. If *continuous* is `TRUE`, *function* is called every *duration* ticks until cancelled.

An MDL application can have multiple timers simultaneously. When an MDL application exits, all of its timers are automatically canceled.

Returns The `mdlSystem_setTimerFunction` function returns `SUCCESS` if it is successful. It returns `ERROR` if *duration* is less than zero or system resources are exhausted. `mdlSystem_cancelTimer` is of type `void`. It returns no value.

See Also `userSystem_timerExpired`, `mdlSystem_getTicks`.

mdlSystem_flushWriteCache

```
void mdlSystem_flushWriteCache(void);
```

Description MicroStation buffers all output to the design file in an internal buffer, or “write cache,” to improve efficiency. The cache is automatically flushed by MicroStation every time through its input loop. However, there can be occasions (albeit rare) where MDL applications wish to make sure that the information they have written to the design file is actually on the disk.

The `mdlSystem_flushWriteCache` function causes MicroStation to flush any information in the write cache to disk.

Returns `mdlSystem_flushWriteCache` is of type `void`. It does not return a value.

mdlSystem_compressDgnFile, mdlSystem_compressLibrary

```
void mdlSystem_compressDgnFile(void);
void mdlSystem_compressLibrary(void);
```

Description The `mdlSystem_compressDgnFile` function compresses the deleted elements from the master design file and writes the file out to disk. This is equivalent to executing the MicroStation COMPRESS DESIGN command. The `mdlSystem_compressLibrary` function does the same thing for the currently attached cell library.

Returns Both `mdlSystem_compressDgnFile` and `mdlSystem_compressLibrary` are of type `void`. They do not return a value.

mdlSystem_startBusyCursor (Macintosh only)

```
#include <mssystem.fdf>

void mdlSystem_startBusyCursor(void);
```

Description The `mdlSystem_startBusyCursor` function will change the graphical cursor to a form which indicates to the user that work is in progress and that the user should

wait for the operation to complete. On platforms other than the Macintosh, the function has no effect.

Returns `mdlSystem_startBusyCursor` is of type `void`. It returns no value.

See Also `mdlSystem_stopBusyCursor` (Macintosh only).

mdlSystem_stopBusyCursor (Macintosh only)

```
#include <mssystem.fdf>

void mdlSystem_stopBusyCursor(void);
```

Description The `mdlSystem_stopBusyCursor` function changes the cursor back to normal following a call to `mdlSystem_startBusyCursor`. On platforms other than the Macintosh, the function has no effect.

Returns `mdlSystem_stopBusyCursor` is of type `void`. It returns no value.

See Also `mdlSystem_startBusyCursor` (Macintosh only).

mdlSystem_setFunction

```
MdlFunctionP mdlSystem_setFunction
(
    int          type,          /* => class of user function */
    MdlFunctionP function      /* => function in MDL program */
);
```

Description The `mdlSystem_setFunction` function designates one system user function.

Valid values for *type* are `SYSTEM_NEW_DESIGN_FILE`,
`SYSTEM_RELOAD_PROGRAM`, `SYSTEM_UNLOAD_PROGRAM`,
`SYSTEM_MDLCHILD_TERMINATED`, `SYSTEM_EXIT_DESIGN_FILE_STATE`,
`SYSTEM_ALL_MDL_UNLOADS`, `SYSTEM_MENUBAR_CHANGE`,
`SYSTEM_REFERENCE_ATTACH`, `SYSTEM_WINDOW_CLOSE`,
`SYSTEM_COLORMAP_CHANGE`, `SYSTEM_SAVEAS`, `SYSTEM_ELMDSCR_TO_FILE`,
`SYSTEM_WRITE_TO_FILE`, `SYSTEM_ROOT_CHANGE`, `SYSTEM_PROCESS_CHANGE`,
`SYSTEM_MCSL_HOOK_CHANGE`, `SYSTEM_CMD_WINDOW_OPEN`,
`SYSTEM_GUIMODE_CHANGE`, `SYSTEM_COLOR_INTERPOLATION`,
`SYSTEM_APPLICATION_NOT_FOUND`, `SYSTEM_JOURNAL_EVENTS`,
`SYSTEM_PLAYBACK_FUNC`, `SYSTEM_ASSOC_DEPENDENT`,
`SYSTEM_NATIVE_TO_RASTFONT_XLT`, `SYSTEM_RASTFONT_TO_NATIVE_XLT`,
`SYSTEM_RASTFONT_TO_DGN_XLT`, `SYSTEM_DGN_TO_RASTFONT_XLT`,
`SYSTEM_FENCE_CHANGED`, `SYSTEM_SAVE`, `SYSTEM_JOURNAL_CADINPUT`,
`SYSTEM_DESIGN_FIND`, `SYSTEM_REFERENCE_DETACH`,
`SYSTEM_REFERENCE_ATTACHED`, `SYSTEM_CELL_LIB_FIND`,
`SYSTEM_DISPLAY_MODE_CHANGED`, `SYSTEM_RASTREF_CHANGED` **and**
`SYSTEM_DIALOG_FIND`.

functionP must be `NULL` or a valid MDL function pointer.

Returns mdlSystem_setFunction returns a pointer to the user function (of the same type) that was previously set using mdlSystem_setFunction. If *type* is invalid, mdlSystem_setFunction returns -1.

See Also userSystem_newDesignFile, userSystem_unloadProgram, userSystem_reloadProgram, userSystem_md1ChildTerminated, userSystem_writeToFile, userSystem_exitDesignFileState, userSystem_allMDLUnloads, userSystem_referenceAttach, userSystem_elmDscrToFile, userSystem_colorMapChange, userSystem_saveAs, userSystem_menuBarChange, userSystem_timerExpired.

mdlSystem_CADInputJournalActive

```
#include <mssystem.fdf>
```

```
BoolInt mdlSystem_CADInputJournalActive(void);
```

Description The mdlSystem_CADInputJournalActive function checks whether CAD input journaling is currently active. Currently, CAD input journaling is used to generate macros.



This function was implemented in MicroStation 95.

Returns mdlSystem_CADInputJournalActive returns TRUE if CAD input journaling is active and FALSE if it is not.

mdlSystem_elapsedTime

```
#include <mssystem.fdf>
```

```
double mdlSystem_elapsedTime(void);
```

Description The mdlSystem_elapsedTime function returns the elapsed time, in seconds, since the MicroStation session was started. It returns the time as the maximum resolution that it can get from the system, which varies from platform to platform. The primary use of this function is for timing operations in MicroStation.



This function was implemented in MicroStation 95.

Returns mdlSystem_elapsedTime returns the elapsed time since the MicroStation session started.

See Also mdlSystem_setTimer.

mdlSystem_exchangeDesignFile

```
#include <mssystem.fdf>
int mdlSystem_exchangeDesignFile
(
    char    *fileName,      /* => name of file to make master file */
    ULong   option          /* => function options */
);
```

Description The `mdlSystem_exchangeDesignFile` function opens a file as the master file while retaining the view configuration of the current master file. This provides a convenient way to switch the role of master and reference files without changing the view configuration. `mdlSystem_exchangeDesignFile` provides the same functionality as the `EXCHANGEFILE` command (XD=).

The *fileName* parameter specifies the name of the file which is to be opened as the master file. If the file specified by *fileName* is already attached as a reference file, it will be made the master file and the current master file will be made a reference file.

The *option* parameter specifies options for the function. At present there is only one option supported. Set *option* to 1 to indicate that an update is not to be performed.



This function was implemented in MicroStation 95.

Returns `mdlSystem_exchangeDesignFile` returns `SUCCESS` if no errors occur.

See Also `mdlSystem_newDesignFile`.

mdlSystem_getWorkspaceList

```
#include <msdefs.h>
#include <mssystem.fdf>

int mdlSystem_getWorkspaceList
(
    StringList **workspaceStrListPP, /* <= workspace information */
    int         attributes           /* => type of workspace checked returned */
);
```

Description The `mdlSystem_getWorkspaceList` function is used to get information about the available workspaces (user configuration files), project configuration files, and custom user interfaces.

workspaceStrListPP is the address of a `StringList` pointer. `NULL` may be passed if `mdlSystem_getWorkspaceList` is just used to check the number of entries. Otherwise, a `StringList` is created by `mdlSystem_getWorkspaceList` based on the *attributes* parameter. The calling application must use `mdlStringList_destroy` to free the `StringList` if the number of returned entries is greater than 0.

attributes defines the type of workspace information that will be checked/returned. Possible values are:

```
WORKSPACE_LIST_WORKSPACES
WORKSPACE_LIST_PROJECTS
WORKSPACE_LIST_INTERFACES
```

Only one of the above may be specified. `WORKSPACE_LIST_PARSENAME` may be combined with the type selection if base file names, instead of full path file names, are desired.



This function was implemented in MicroStation 95.

Returns `mdlSystem_getWorkspaceList` returns -1 if there is an error. Otherwise the number of entries as determined by the *attributes* parameter is returned.

See Also `mdlStringList_getMember`, `mdlStringList_destroy`.

mdlSystem_journalAccessString, mdlSystem_journalAccessStrByTaskId

```
#include <mssystem.fdf>
#include <cexpr.h>

int mdlSystem_journalAccessString
(
    void    *ownerMD,      /* => MDL descriptor for the owner task */
    char    *accessStrP,   /* => Access string */
    int     dataType,      /* => Data type */
    ULong   dataMask,      /* => Bit mask if only using bits in *dataP */
    void    *dataP         /* => Pointer to the data, matches "dataType" */
);

int mdlSystem_journalAccessStrByTaskId
(
    char    *taskIdP,      /* => Task's id string */
    char    *accessStrP,   /* => Access string */
    int     dataType,      /* => Data type */
    ULong   dataMask,      /* => Bit mask if only using bits in *dataP */
    void    *dataP         /* => Pointer to the data, matches "dataType" */
);
```

Description If CAD input journaling is active, the `mdlSystem_journalAccessString` and `mdlSystem_journalAccessStrByTaskId` functions journal the described access string setting. When the macro generator is using CAD input journaling, these functions add `MbeSetAppVariable` statements to the macro.

ownerMD is the MDL descriptor of the task owning the access string.

taskIdP is the name of the task owning the access string.

accessStrP is the C variable whose value is to be journaled.

dataType is the type of C variable. Possible types are CEXPR_TYPE_POINTER, CEXPR_TYPE_LONG, CEXPR_TYPE_ULONG and CEXPR_TYPE_DOUBLE.

dataMask is the bit mask that indicates which bits in *dataP* are being journaled. The mask only applies to the long and unsigned long data types. Use NOMASK when all bits are to be journaled.

dataP is a pointer to the data being journaled. The type must match that of *dataType*.



These functions were implemented in MicroStation 95.

Returns mdlSystem_journalAccessString and mdlSystem_journalAccessStrByTaskId return SUCCESS.

mdlSystem_journalAppMessage

```
#include <mssystem.fdf>
#include <msinputq.h>

int mdlSystem_journalAppMessage
(
    char    *taskIdP, /* => Task's id string */
    char    *msgP      /* => Message text, may contain newlines */
);
```

Description If CAD input journaling is active, the mdlSystem_journalAppMessage function journals the given application specific string. When the macro generator is using CAD input journaling, this function adds an MbeSendAppMessage statement to the macro.

taskIdP is the name of the task that understands the message text. When the message is added to a macro, this is the task that receives the APPLICATION_EVENT input queue element containing the message.

msgP is the application-specific text to be journaled.



This function was implemented in MicroStation 95.

Returns mdlSystem_journalAppMessage returns SUCCESS.

mdlSystem_journalCommand

```
#include <mssystem.fdf>

int mdlSystem_journalCommand
(
  char    *taskIdP,      /* => task id string (may be NULL) */
  long    cmdnum,        /* => command number */
  char    *unparsedP     /* => unparsed part of command (may be NULL) */
);
```

Description If CAD input journaling is active, the `mdlSystem_journalCommand` function journals the described command. When the macro generator is using CAD input journaling, this function adds an `MbeSendCommand` statement to the macro.

taskIdP is the name of the task owning the command. If it is NULL, MicroStation is assumed.

cmdnum is the command number.

unparsedP is any unparsed part of the command string. This argument may be NULL.



This function was implemented in MicroStation 95.

Returns `mdlSystem_journalCommand` returns SUCCESS.

mdlSystem_journalCommandString

```
#include <mssystem.fdf>

int mdlSystem_journalCommandString
(
  char    *stringP /* => String to journal as a command */
);
```

Description If CAD input journaling is active, the `mdlSystem_journalCommandString` function journals the given string as a command. When the macro generator is using CAD input journaling, this function adds an `MbeSendCommand` statement to the macro.

stringP is the string to be journaled as a command.



This function was implemented in MicroStation 95.

Returns `mdlSystem_journalCommandString` returns SUCCESS.

mdlSystem_journalDataPoint

```
#include <mssystem.fdf>
#include <msdefs.h>
#include <keys.h>

int mdlSystem_journalDataPoint
(
    Point3d      *uorsP,          /* => Data point in dgn coordinates */
    Point3d      *rawUorsP,       /* => Unadjust data pt coord, can be NULL */
    int          view,            /* => View that data point is in (0 relative) */
    int          buttonTrans,     /* => Button transition */
    int          qualifierMask    /* => qualifying keys, if any */
);
```

Description If CAD input journaling is active, the mdlSystem_journalDataPoint function journals the described data point. When the macro generator is using CAD input journaling, this function adds an MbeSendDataPoint statement to the macro.

uorsP is the coordinates of the data point in design file coordinates.

rawUorsP is the coordinates of the data point in design file coordinates before it was adjusted for locks. This argument can be NULL.

view is the number of the view in which the data point was entered. The view numbers are 0 relative.

buttonTrans is the transition of the button, for example, BUTTONTRANS_DOWN or BUTTONTRANS_UP. If a 0 is passed to the function, BUTTONTRANS_CLICK is assumed.

qualifierMask is a mask of the keys held down when the data point was entered, for example, TOGGLESELECT_MODKEY or PANOVERLAP_MODKEY.



This function was implemented in MicroStation 95.

Returns mdlSystem_journalDataPoint returns SUCCESS.

mdlSystem_journalKeyin

```
#include <mssystem.fdf>

int mdlSystem_journalKeyin
(
    char      *stringP  /* => String keyed in */
);
```

Description If CAD input journaling is active, the mdlSystem_journalKeyin function journals the given key-in. When the macro generator is using CAD input journaling, this function adds an MbeSendKeyin statement to the macro.

stringP is the string to be added.



This function was implemented in MicroStation 95.

Returns `mdlSystem_journalKeyin` returns `SUCCESS`.

mdlSystem_journalReset

```
#include <mssystem.fdf>

int mdlSystem_journalReset(void);
```

Description If CAD input journaling is active, the `mdlSystem_journalReset` function journals a reset. When the macro generator is using CAD input journaling, this function adds an `MbeSendReset` statement to the macro.



This function was implemented in MicroStation 95.

Returns `mdlSystem_journalReset` returns `SUCCESS`.

mdlSystem_journalTentativePoint

```
#include <mssystem.fdf>

int mdlSystem_journalTentativePoint
(
    Point3d *uorsP,    /* => Tentative point in design file coordinates */
    int    view        /* => View that tentative point is in (0 relative) */
);
```

Description If CAD input journaling is active, the `mdlSystem_journalTentativePoint` function journals the described tentative point. When the macro generator is using CAD input journaling, this function adds an `MbeSendTentPoint` statement to the macro.

uorsP is the coordinates of the tentative point in design file coordinates.

view is the number of the view in which the tentative point was entered. This view number is 0 relative.



This function was implemented in MicroStation 95.

Returns `mdlSystem_journalTentativePoint` returns `SUCCESS`.

mdlSystem_saveDesignFile

```
#include <mssystem.fdf>

int mdlSystem_saveDesignFile
(
    void
);
```

Description The mdlSystem_saveDesignFile function simulates “Save” being selected from the “File” menu. It only applies when the user preference to immediately save design changes is off. However, the preference does not need to be checked before calling this function. mdlSystem_saveDesignFile saves any changes to the current design file and causes a new backup file to be created.



This function was implemented in MicroStation 95.

Returns mdlSystem_saveDesignFile returns ERROR if the backup file cannot be created. Otherwise, SUCCESS is returned.

See Also mdlSystem_fileDesign, mdlSystem_closeDesignFile.

userSystem_md1ChildTerminated

```
#include <userfnc.h>

void userSystem_md1ChildTerminated
(
    Md1ChildTerminated    *childInfoP    /* <=> receives exit status */
);
```

Description An MDL application designates an MDL child-terminated function by calling mdlSystem_setFunction. The application programmer determines the function name; *userSystem_md1ChildTerminated* is used merely as an example.

MicroStation calls an MDL task child-terminated function if one of the task’s child programs terminates. For example, if program A starts program B and program A has the *userSystem_md1ChildTerminated* function, MicroStation calls *userSystem_md1ChildTerminated* when program B terminates.

When MicroStation calls the user function, *childInfoP->pName* gives the name of the terminated task. *childInfoP->exitReason* is one of the SYSTEM_TERMINATED values defined in userfnc.h. If the terminated task called mdlSystem_exit, *childInfoP->exitReason* is set to SYSTEM_TERMINATED_EXIT, and *childInfoP->exitStatus* is the value that the terminated task specified in the mdlSystem_exit call.

Returns *userSystem_md1ChildTerminated* is of type void. MDL ignores the return value.

See Also mdlSystem_setFunction.

userSystem_reloadProgram

```
void userSystem_reloadProgram
(
    int    argc,        /* => count of entries in argv */
    char   *argv[]      /* => array of pointers to arguments */
);
```

Description An MDL program designates an MDL reload function by calling `mdlSystem_setFunction`. The application programmer determines the function name; *userSystem_reloadProgram* is used merely as an example.

MicroStation calls a program's reload function if it receives a request to load that program after it has already been loaded. When MicroStation calls the user function, it passes the arguments that it would supply to the task's `main` if the program was not already loaded.

argc is a count of pointers in *argv*. *argv* is an array of pointers to the arguments. *argv[0]* points to the name of the file that contains the MDL application. *argv[1]* points to a string that describes the load request's source. The remaining entries in *argv* point to parameters for the program.

Returns The *userSystem_reloadProgram* is of type `void`. Return values are ignored.

See Also `mdlSystem_setFunction`.

userSystem_unloadProgram

```
#include <userfnc.h>

int userSystem_unloadProgram
(
    int      unloadType          /* => SYSTEM_TERMINATED_SHUTDOWN */
);
```

Description An MDL application designates an MDL unload function by calling `mdlSystem_setFunction`. The application programmer determines the function name; *userSystem_unloadProgram* is used merely as an example.

MicroStation calls the *userSystem_unloadProgram* function if it is directed to unload the program.

The possible values for *unloadType* are `SYSTEM_TERMINATED_SHUTDOWN` if MicroStation is shutting down, `SYSTEM_TERMINATED_COMMAND` if the user entered an MDL UNLOAD command, `SYSTEM_TERMINATED_EXIT` if the current task called `mdlSystem_exit`, and `SYSTEM_TERMINATED_BY_APP` if another MDL task called `mdlSystem_unloadMdlProgram`.

Returns If *unloadType* is negative, MDL ignores the return value. If *unloadType* is positive, the function can return a non-zero value to abort the unload.

See Also `userSystem_exitDesignFileState`, `userSystem_allMDLUnloads`, `mdlSystem_unloadMdlProgram`, `mdlSystem_setFunction`.

userSystem_newDesignFile

```
#include <userfnc.h>
void userSystem_newDesignFile
(
char    *filenameP,    /* => name of file */
int     state          /* => SYSTEM_NEWFILE_CLOSE, SYSTEM_NEWFILE_COMPLETE */
);
```

Description An MDL application designates an MDL design file function by calling `mdlSystem_setFunction`. The application programmer determines the function name; *userSystem_newDesignFile* is used merely as an example.

The *userSystem_newDesignFile* function is called before MicroStation closes a design file and after MicroStation finishes opening a newly entered design file.

When calling *userSystem_newDesignFile* before closing the old design file, MicroStation provides a value of `SYSTEM_NEWFILE_CLOSE` for *state*.

filenameP points to the name of the old design file.

When calling *userSystem_newDesignFile* after opening the new design file, MicroStation provides a value of `SYSTEM_NEWFILE_COMPLETE` for *state*.

filenameP points to the name of the new design file.

Returns MDL ignores the return value of the new design file user function.

See Also `mdlSystem_setFunction`, `userSystem_saveAs`.

userSystem_timerExpired

```
void userSystem_timerExpired
(
long    userArg,        /* => user argument */
int     timerHandle     /* => timer handle */
);
```

Description An MDL application designates an MDL timer function by calling `mdlSystem_setTimerFunction`. The application programmer determines the function name; *userSystem_timerExpired* is used merely as an example.

The *userSystem_timerExpired* function is called when the timer duration expires.

When MicroStation calls the user function, *userArg* is the value set in the call to `mdlSystem_setTimerFunction`. *timerHandle* is the timer handle set by the call to `mdlSystem_setTimerFunction`.

If the timer was specified as continuous in the call to `mdlSystem_setTimerFunction`, the *userSystem_timerExpired* function is called repeatedly until the timer is canceled with `mdlSystem_cancelTimer`.

Returns MDL ignores the return value of the timer user function.

See Also mdlSystem_setTimerFunction, mdlSystem_cancelTimer.

userSystem_writeToFile

```
void userSystem_writeToFile
(
    mdlElementUnion*fileElementP,    /* => elm in file format */
    ULong          filePos,          /* => file position */
    int            nBytes,           /* => # of bytes to write */
    boolean        eof,              /* => TRUE means EOF */
    MSElementUnion *internalElementP /* => elm in working format */
);
```

Description An MDL application designates an MDL write-to-file function by calling mdlSystem_setFunction. The application programmer determines the function name; *userSystem_writeToFile* is used merely as an example.

MicroStation calls the *userSystem_writeToFile* user function each time it writes to the design file. If a *userSystem_writeToFile* function is defined, it will degrade performance.

fileElementP points to the element that results from the conversion to file format.

internalElementP points to the element in the format that matches the structure definitions provided in "mselems.h."

filePos specifies the location in the file where the element is being written. This argument is a byte offset relative to the start of the file.

eof is TRUE if the element is being written to the end of the file.

nBytes specifies the number of bytes that will be written.

Returns MDL ignores the return value.

See Also userSystem_elmDscrToFile, userUndo_addToBuffer.

userSystem_saveAs

```
void userSystem_saveAs
(
    int      afterSave    /* => TRUE if after the file is saved */
);
```

Description MicroStation calls the *userSystem_saveAs* user functions both at the beginning and end of processing a request to save the design file. When MicroStation calls the *userSystem_saveAs* function with *exitReason* TRUE, MicroStation has already updated the tcb with the new file name.

A *userSystem_saveAs* user function is identified to MicroStation using the function call

```
mdlSystem_setFunction (SYSTEM_SAVEAS, <hook function name>).
```

Returns MicroStation ignores the return value from *userSystem_saveAs* user hook functions.

See Also *userSystem_newDesignFile*.

userSystem_elmDscrToFile

```
#include <userfnc.h>

int userSystem_elmDscrToFile
(
    int          action,          /* => operation being performed */
    int          fileNum,        /* => file number for this element */
    ULong        filePos,       /* => file position to be modified */
    MSElementDescr *newEdP,     /* => new element descriptor */
    MSElementDescr *oldEdP,     /* => old element descriptor */
    MSElementDescr **replacmentEdPP /* <= replacement descriptor */
);
```

Description An MDL application designates an MDL element-descriptor-to-file function by calling *mdlSystem_setFunction*. The application programmer determines the function name; *userSystem_elmDscrToFile* is used merely as an example.

The *userSystem_elmDscrToFile* function is called by MicroStation before an element descriptor is written to the design file. The function may modify the element(s) before they are written, or it may stop the operation from occurring at all.

The parameter *action* indicates the reason that MicroStation called *userSystem_elmDscrToFile*. Possible values are:

Value for <i>action</i>	Called when
ELMDTF_ACTION_APPEND	New elements are added to the end of the file.
ELMDTF_ACTION_DELETE	Existing elements are deleted from the file.
ELMDTF_ACTION_REPLACE	Elements are overwritten in place.

The parameters *fileNum* and *filePos* indicate the file number and file position of the element(s) being written to the file.

The parameters *newEdP* and *oldEdP* are pointers to element descriptors that have the new and old element(s) respectively. When the value of *action* is *ELMDTF_ACTION_APPEND*, *oldEdP* is NULL. When the value of *action* is *ELMDTF_ACTION_DELETE*, *newEdP* is NULL.

The parameter *replacmentEdPP* is a pointer to a pointer to an element descriptor that is allocated and then returned by *userSystem_elmDscrToFile*. If *userSystem_elmDscrToFile* returns *ELMDTF_STATUS_REPLACE*, MicroStation writes this element descriptor to the file rather than the one pointed to by *newEdP*. Ownership of

replacementEdPP is passed to MicroStation. It frees this element descriptor when it is done with it.

Returns Your *userSystem_elmDscrToFile* function must return one of the following values:

Return Status	Meaning
ELMDTF_STATUS_SUCCESS	Proceed with operation unchanged.
ELMDTF_STATUS_ABORT	Do not allow operation. Command that initiated the write receives a “write failed” status.
ELMDTF_STATUS_REPLACE	Continue with the operation using the element descriptor pointed to by <i>replacementEdPP</i> . Not valid if <i>action</i> is ELMDTF_ACTION_DELETE.

See Also *userSystem_writeToFile*, *userUndo_addToBuffer*, *mdlSystem_setFunction*.

userSystem_colorMapChange

```
#include <userfnc.h>

void userSystem_colorMapChange
(
    int      screen, /* => mapping is for this screen */
    char     *colortbl, /* => master or ref file color table */
    ULONG    *colorMap, /* => colorMap corresponding to colortbl */
    int      fileNum /* => 0=master file. 1-255 = ref file */
);
```

Description An MDL application designates an MDL colorMap-change function by calling *mdlSystem_setFunction*. The application programmer determines the function name; *userSystem_colorMapChange* is used merely as an example.

MicroStation calls the *userSystem_colorMapChange* function when a new color mapping is generated between the design file colortable and the video hardware colors loaded for a particular screen. A new color mapping can be generated for a variety of reasons, such as:

- Entry to a new design file
- Attachment of a new colortable
- Color configuration change such as the number of Exact Colors

The main reason why an application might want to monitor color mapping changes would be to know when it is time to regenerate its color maps for its own RGB arrays. See *mdlColor_matchLongColorMap* for details on generating application-specific color maps.

screen indicates for which screen the color mapping took place.

colortbl is the array of RGB values that got mapped. The first RGB represents the design file background color. The second RGB corresponds to element color number zero.

colorMap is the resulting array of draw values corresponding to each RGB entry in *colortable*. The first color map entry corresponds to element color number zero. The last entry (255) corresponds to the design file background color.

fileNum indicates the file from which the color table was obtained.



With the advent of Color Palettes in MicroStation Version 5.0 the need for monitoring color map changes and generating application-specific color maps is greatly reduced if not eliminated. For more information on Color Palettes, see the Color Palette Functions section of the Color Manager Chapter.

Returns *userSystem_colorMapChange* has no return value.

See Also *mdlSystem_setFunction*, *mdlColor_matchLongColorMap*.

userSystem_menuBarChange

```
int userSystem_menuBarChange /* <= stop menu bar from changing */
(
    char    *oldTaskIdP,    /* => task id of deactivating menu bar's owner */
    char    *newTaskIdP,    /* => task id of activating menu bar's owner */
    boolean activating,     /* => TRUE=newTaskIdP's bar is being activated */
    boolean afterMenuBarActivated /* => TRUE=newTaskIdP menubar activated */
);
```

Description The *userSystem_menuBarChange* function is called twice each time a menu bar owned by the task is activated or deactivated; once before a new menu bar is activated and once after, indicated by the *activating* and *afterMenuBarActivated* parameters.

An MDL program designates a call-back function for menu bar changes by calling *mdlSystem_setFunction*. The application programmer determines the function name; *userSystem_menuBarChange* is used merely as an example.

Returns If *userSystem_menuBarChange* returns TRUE when *afterMenuBarActivated* is FALSE, it stops the menu bar from changing.

See Also *mdlSystem_setFunction*.

userSystem_referenceAttach

```
#include <userfnc.h>

int userSystem_referencAttach
(
    char    *fileName      /* => reference file being attached */
);
```

Description An MDL application designates an MDL reference-attach function by calling `mdlSystem_setFunction`. The application programmer determines the function name; *userSystem_referenceAttach* is used merely as an example.

MicroStation calls the *userSystem_referenceAttach* function when a reference file is about to be attached. This occurs either as MicroStation loads a new design file and encounters reference attachment elements, or when the user attempts to manually attach a new reference file.

userSystem_referenceAttach is passed the name of the reference file about to be attached in *fileName*. It may modify the file name (to a maximum of `MAXFILELENGTH` characters), and cause MicroStation to attempt to attach the reference file using the new file name.

Returns If *userSystem_referenceAttach* returns `SUCCESS`, MicroStation continues with the reference file attachment, otherwise it aborts the attachment request.

See Also `mdlSystem_setFunction`.

userSystem_allMDLUnloads

```
#include <userfnc.h>

void userSystem_allMDLUnloads
(
void    *mdlDescP,      /* => application being unloaded */
int     exitReason,     /* => reason for unloading */
char    *taskIdP       /* => application being unloaded */
);
```

Description MicroStation calls the *userSystem_allMDLUnloads* user functions every time an MDL application is unloaded. It also calls these hooks if it has to abort an unload for some reason.

Both *mdlDescP* and the *taskIdP* identify the application being unloaded. They may be `NULL` if the user hooks are being called after a failed attempt to load an MDL application.

exitReason identifies why the application is being unloaded. It can be any of the `SYSTEM_TERMINATED_` values defined in `userfnc.h`.

The *userSystem_allMDLUnloads* user function for the task being unloaded is not called.

A *userSystem_allMDLUnloads* user function is identified to MicroStation using the function call
`mdlSystem_setFunction(SYSTEM_ALL_MDL_UNLOADS, <hook function name>).`

Returns MicroStation ignores the value returned by *userSystem_allMDLUnloads* user functions.

See Also `userSystem_exitDesignFileState`, `userSystem_unloadProgram`, `mdlSystem_setFunction`.

userSystem_exitDesignFileState

```
void userSystem_exitDesignFileState(void);
```

Description MicroStation calls the *userSystem_exitDesignFileState* user functions as the first step in exiting design file state. This occurs when MicroStation executes a Close or Exit command while a design file is open.

This hook function is particularly useful for MDL applications that have MicroCSL applications. MicroStation terminates all MicroCSL programs as a part of the processing that occurs when MicroStation is exiting design file state. This user function allows applications to better control shutting down their MicroCSL applications.

MicroStation calls the *userSystem_exitDesignFileState* user functions before terminating the MicroCSL applications, before closing the design file and reference files, before closing the windows, and before unloading the non-essential MDL applications. The system is still in a good state for any work the application must do.

A *userSystem_exitDesignFileState* user function is identified to MicroStation using the function call
mdlSystem_setFunction(SYSTEM_EXIT_DESIGN_FILE_STATE, <hook
function name>).

Returns MicroStation ignores the return value from *userSystem_exitDesignFileState* user hook functions.

See Also mdlSystem_setFunction, userSystem_unloadProgram,
userSystem_allMDLUnloads.

userInput_fileOpenDialogPreprocess

```
#include <filelist.h>
#include <deffiles.h>
#include <userfnc.h>

int userInput_fileOpenDialogPreprocess
(
    char          *fileNameP,      /* <= choosen file name */
    char          *extraOutArgP,   /* <= not currently used */
    FileOpenParams *fopenParamsP, /* => file open parameters */
    void          *extraInArgP     /* => not currently used */
);
```

Description An MDL application designates an MDL file-open-dialog-preprocess function by calling mdlInput_setFunction with type INPUT_FILEOPENDIALOG_PREPROCESS. The application programmer determines the function name; *userInput_fileOpenDialogPreprocess* is used merely as an example.

MicroStation calls the *userInput_fileOpenDialogPreprocess* function right before each file open dialog is displayed. This offers the application programmer an opportunity to supply a file name via a custom dialog box.

However, calls to `mdlDialog_fileOpenExt`, `mdlDialog_fileOpen`, `mdlDialog_defFileOpen`, `mdlDialog_fileCreate` or `mdlDialog_fileCreateFromSeed` may not be made inside the `userInput_fileOpenDialogPreprocess` function as this would result in calling the user function again.

fileNameP is an output parameter supplied by the user function. It should contain a fully specified file name when the user function returns `INPUT_FOPENPREPROCESS_HANDLED`.

extraOutArgP is not currently used.

fopenParamsP points to a structure containing initialization information about the file open dialog being preprocessed by the `userInput_fileOpenDialogPreprocess` user function.

fopenParamsP->openCreate should be checked to determine if the file is to be opened or created. *fopenParamsP->defFileId* should be checked to determine the type of file that is to be opened or created. Refer to the documentation of `mdlDialog_fileOpenExt` for complete information about the `FileOpenParams` structure.

extraInArgP is not currently used.



This function was implemented in MicroStation 95.

Returns Your `userInput_fileOpenDialogPreprocess` function must return one of the following values:

Return Status	Meaning
<code>INPUT_FOPENPREPROCESS_HANDLED</code>	The event has been handled by the user function. MicroStation will use the filename returned in <i>fileNameP</i> .
<code>INPUT_FOPENPREPROCESS_CANCELED</code>	The event has been handled by the user function and has been canceled. Similar to pushing the “Cancel” button on a file open dialog.
<code>INPUT_FOPENPREPROCESS_IGNORED</code>	The event has been ignored by the user function. Let MicroStation or another application handle the event.

See Also `mdlInput_setFunction`, `mdlDialog_fileOpenExt`.

userShare_sharedLibNoMoreClients

```
#include <userfnc.h>

void userShare_sharedLibNoMoreClients(void);
```

Description An MDL application designates an MDL shared-library-no-more-clients function by calling `mdlShare_setFunction` with type `SHARED_LIB_NO_MORE_CLIENTS`. The

application programmer determines the function name;
userShare_sharedLibNoMoreClients is used merely as an example.

When MicroStation unloads a client application, it checks each of the client application's shared libraries to determine if there are any that do not have more clients loaded. MicroStation calls the *userShare_sharedLibNoMoreClients* asynchronous function for each library that no does not have any other clients.



This function was implemented in MicroStation 95.

Returns *userShare_sharedLibNoMoreClients* is of type `void`. It returns no value.

See Also *userSystem_unloadProgram*, *userSystem_allMDLUnloads*, *mdlShare_setFunction*

userSystem_fenceChanged

```
#include <userfnc.h>

void userSystem_fenceChanged
(
    int      changeReason
);
```

Description An MDL application designates an MDL fence-changed function by calling *mdlSystem_setFunction* with type `SYSTEM_FENCE_CHANGED`. The application programmer determines the function name; *userSystem_fenceChanged* is used merely as an example.

MicroStation calls the *userSystem_fenceChanged* function when a fence is either created or cleared.

changeReason will either be `SYSTEM_FENCE_CLEARED` or `SYSTEM_FENCE_CREATED`.



This function was implemented in MicroStation 95.

Returns *userSystem_fenceChanged* is of type `void`. It returns no value.

See Also *mdlSystem_setFunction*.

userSystem_save

```
#include <userfnc.h>

void userSystem_save
(
    BoolInt      postSave,
    BoolInt      intermediateSave
);
```

Description An MDL application designates an MDL save function by calling `mdlSystem_setFunction` with type `SYSTEM_SAVE`. The application programmer determines the function name; `userSystem_save` is used merely as an example.

MicroStation calls the `userSystem_save` function before and after saving design file changes. `userSystem_save` only applies when the “Immediately Save Design Changes” user preference is off.

`postSave` indicates if `userSystem_save` is being called before or after saving design file changes. It will be `TRUE` if after save has been completed.

`intermediateSave` will be `TRUE` if the reason for calling `userSystem_save` is because “Save” was selected from the “File” menu or the “Save Design” command was otherwise issued. `intermediateSave` will be `FALSE` when saving changes at design file close.



This function was implemented in MicroStation 95.

Returns `userSystem_save` is of type `void`. It has no return value.

See Also `mdlSystem_setFunction`, `userSystem_saveAs`.

Configuration Variable Functions

Configuration variable files provide a mechanism for communicating information to MicroStation about the system and the user running it. Some of this information is defined when MicroStation is installed (such as directory names), some of it is defined by the hardware (such as graphics card type and plotter model), and some of it is chosen by users according to their needs (such as additional symbology files).

Configuration variable files are generally created once, usually automatically by the MicroStation installation procedure or the Configuration Variables Dialog, and are rarely changed thereafter. All levels of the configuration variables files are then processed by MicroStation every time it starts up. This processing creates a configuration variable table that MDL applications can access. To get the value of a configuration variable MicroStation checks the configuration variable table for an entry of the same name. If a match is found, MicroStation searches down through the levels (in the following order: user, project, site, application, system) to obtain a value. These levels correspond to the configuration file processing levels which by default correspond to subdirectories of the MicroStation config directory. If a match is not found in the configuration variable table then MicroStation searches for an operating system environment variable of the same name. If an operating system environment variable is defined, MicroStation uses that value as if it were defined in the configuration variable table.

In addition, MDL applications can define, delete and lock configuration variables.

The following table lists configuration variable functions:

Function	Used to
<code>mdlSystem_getenv</code> (obsolete)	retrieve the expanded value of a MicroStation configuration variable.
<code>mdlSystem_putenv</code> (obsolete)	define a system-level environment variable.
<code>mdlSystem_getCfgVar</code>	get the unexpanded value of a configuration variable.
<code>mdlSystem_getCfgVarByIndex</code>	get unexpanded value of a configuration variable by its index.
<code>mdlSystem_expandCfgVar</code>	expand configuration variable references.
<code>mdlSystem_getCfgVarLevel</code>	get the level on which a configuration variable is defined.
<code>mdlSystem_defineCfgVar</code>	define a configuration variable.
<code>mdlSystem_deleteCfgVar</code>	remove a variable from the configuration variable table.
<code>mdlSystem_deleteCfgVarAtLevel</code>	remove a value from a particular level of a config. variable.
<code>mdlSystem_lockCfgVar</code>	lock a configuration variable.
<code>mdlSystem_isCfgVarLocked</code>	query the status of the lock on a configuration variable.
<code>mdlSystem_processCfgVarFile</code>	process a configuration variable file.
<code>mdlSystem_createListFromCfgVarValue</code>	parse a configuration variable's value into a <code>StringList</code> .
<code>mdlSystem_getCfgVarAtLevel</code>	retrieve the unexpanded value of the configuration variable, <i>cfgVarNameP</i> , at a particular level.
<code>mdlSystem_getExpandedCfgVar</code>	retrieve the expanded value of <i>cfgVarNameP</i> .
<code>mdlSystem_rewriteCfgVarFile</code>	rewrite the <i>cfgVarFileNameP</i> file so that <i>cfgVarNameP</i> will get defined to <i>cfgVarValueP</i> .

Examples

See `excfgar.mc`.

mdlSystem_getenv (obsolete)

```
int mdlSystem_getenv
(
char    *valueP,          /* <=> receives value */
char    *envvarP,         /* => name of environment variable */
int     maxlen            /* => maximum size of value */
);
```

Description The `mdlSystem_getenv` function retrieves the expanded value of a MicroStation configuration variable *envvarP*. If no configuration variable exists, MicroStation attempts to find a system level environment variable with that name.

valueP points to the buffer that receives the value of the environment variable. *valueP* can be NULL if `mdlSystem_getenv` is used to check for the existence of the environment variable.

envvarP points to the name of the configuration variable.

maxlen specifies the size of the buffer pointed to by *valueP*.

The standard C function `getenv` is also available in MDL for retrieving operating system environment variables only (not configuration variables).



The function `mdlSystem_getenv` is obsolete with version 5.0 of MicroStation. New MDL programs should begin using `mdlSystem_getCfgVar` or `mdlSystem_getExpandedCfgVar` instead.

Returns The `mdlSystem_getenv` function returns `SUCCESS` if the environment variable is defined. Otherwise, it returns a non-zero value.

See Also `mdlSystem_getCfgVar`, `getenv`, `mdlSystem_defineCfgVar`.

mdlSystem_putenv (obsolete)

```
void mdlSystem_putenv
(
char    *variableNameP,   /* => name of environment variable */
char    *valueP           /* => new value */
);
```

Description The `mdlSystem_putenv` function defines a system level environment variable. On most systems, environment variables set with `mdlSystem_putenv` are lost after the MicroStation session ends. Under DOS, `mdlSystem_putenv` adds an entry in the BSI environment table, which does retain its value between MicroStation sessions.

variableNameP defines the name of the variable. *valueP* defines the value of the environment variable.

If *variableNameP* is defined before the call, the new value replaces the old value.



mdlSystem_putenv is obsolete and is kept only for compatibility with version 4.0 of MicroStation. New MDL applications should use mdlSystem_defineCfgVar.

Returns The mdlSystem_putenv function is of type void. It returns no value.

See Also mdlSystem_defineCfgVar, mdlSystem_getCfgVar.

mdlSystem_getCfgVar

```
int mdlSystem_getCfgVar
(
char    *valueP,          /* <= receives value */
char    *cfgVarP,         /* => name of configuration variable */
int     maxlen            /* => maximum size of value */
);
```

Description The mdlSystem_getCfgVar function retrieves the unexpanded value of the configuration variable *cfgVarP*. If no configuration variable exists, MicroStation attempts to find a system level environment variable with that name.

mdlSystem_getCfgVar differs from mdlSystem_getenv (obsolete) in that it does not expand the value of the variable before returning it. This is helpful in cases where the application has no way of knowing what the maximum size of the expanded variable might be.

valueP points to the buffer that receives the value of the configuration variable. *valueP* can be NULL if mdlSystem_getCfgVar is used to check for the existence of the configuration variable.

cfgVarP points to the name of the configuration variable.

maxlen specifies the size of the buffer pointed to by *valueP*.



valueP may contain references to nested configuration variables. Use mdlSystem_expandCfgVar to obtain the expanded value(s).

Returns The mdlSystem_getCfgVar function returns SUCCESS if the configuration variable is defined. Otherwise, it returns a non-zero value.

See Also mdlSystem_getenv (obsolete), mdlSystem_expandCfgVar, mdlSystem_getCfgVarByIndex, mdlSystem_getCfgVarLevel, getenv.

mdlSystem_getCfgVarByIndex

```
#include <msdefs.h>
int mdlSystem_getCfgVarByIndex
(
    char    **name,          /* <= configuration variable name */
    char    **translation,  /* <= current value */
    int     *level,         /* <= level at which it was defined */
    boolean *locked,        /* <= TRUE means cfg var is locked */
    int     index           /* => index number */
);
```

Description The `mdlSystem_getCfgVarByIndex` function returns the name and value of a configuration variable by its index into the table of configuration variables. Any pointers can be `NULL`.

name is the address of a character pointer. On successful return, *name* points to the string that holds the name of the configuration variable. MDL applications should never change the string pointed to by *name*.

translation is the address of a character pointer. On successful return, *translation* points to the string that holds the current value of configuration variable *name*. MDL applications should never change the string pointed to by *translation*.

level is a pointer to an integer. On successful return, *level* holds the level at which the variables was defined. Constants for the configuration variable processing levels are defined in `msdefs.h` (`CFGVAR_LEVEL_xxx`).

locked is a pointer to a boolean. On successful return, *locked* holds `TRUE` if the configuration variable *name* is locked and `FALSE` if it is not.

index is an index into the table of configuration variables. The most common use for `mdlSystem_getCfgVarByIndex` is to get a list of all currently defined variables. To do so, start at index 0 and loop, incrementing *index* by 1, until the function returns `ERROR`.

Returns `mdlSystem_getCfgVarByIndex` returns `SUCCESS` if a configuration variable at *index* exists and *name*, *translation*, *level* and *locked* are valid. Otherwise it returns `ERROR`.

See Also `mdlSystem_getCfgVar`, `mdlSystem_isCfgVarLocked`, `mdlSystem_getCfgVarLevel`.

mdlSystem_expandCfgVar

```
char *mdlSystem_expandCfgVar
(
    char    *cfgVarStringP    /* => string to expand */
);
```

Description The `mdlSystem_expandCfgVar` function expands any variable references in the string *cfgVarStringP*. This string can either be a string returned by

mdlSystem_getCfgVar or an application generated string. All nested variable definitions are also expanded.

Returns mdlSystem_expandCfgVar allocates memory to hold the resultant string and returns a pointer to that memory. The application must free this memory when finished with it.

See Also mdlSystem_getCfgVar.

mdlSystem_getCfgVarLevel

```
#include <msdefs.h>

int mdlSystem_getCfgVarLevel
(
    int      *levelP,          /* <= destination level */
    char     *varNameP        /* => configuration variable to find */
);
```

Description The mdlSystem_getCfgVarLevel function gets the level at which the passed in configuration variable is defined.

levelP points to the location where the definition level will be copied. Valid values are the CFGVAR_LEVEL_XXX constants defined in msdefs.h and correspond to the configuration file processing levels.

varNameP points to the name of the configuration variable to find.

Returns mdlSystem_getCfgVarLevel returns SUCCESS if the configuration variable is defined and ERROR if the configuration variable is not defined. If ERROR is returned, the value pointed to by *levelP* is not valid.

See Also mdlSystem_getCfgVar.

mdlSystem_defineCfgVar

```
int mdlSystem_defineCfgVar
(
    char     *cfgVarP,         /* => name of variable */
    char     *valueP,          /* => new value */
    int      level             /* => definition level */
);
```

Description The mdlSystem_defineCfgVar function defines a configuration variable. If the variable already exists, its value is redefined.

cfgVarP is the name of the configuration variable. Configuration variable names are not case sensitive.

valueP is the new value. There is no limit on the length of a variable value.

level is used to identify the class (definition level) of configuration variable. Valid values are the CFGVAR_LEVEL_XXX constants defined in msdefs.h.

Typically, applications should use `CFGVAR_LEVEL_USER` since this definition level is the first searched when getting the value of a configuration variable. Also this is equivalent to making a change from the Configuration Variable dialog box.

Returns `mdlSystem_defineCfgVar` return `SUCCESS` if the variable is successfully defined. Otherwise it returns `ERROR`.

See Also `mdlSystem_getCfgVar`, `mdlSystem_deleteCfgVar`.

mdlSystem_deleteCfgVar, mdlSystem_deleteCfgVarAtLevel

```
int mdlSystem_deleteCfgVar
(
char    *cfgVarP          /* => name of variable */
);

int mdlSystem_deleteCfgVarAtLevel
(
char    *cfgVarP,          /* => name of variable */
int     level              /* => level to remove */
);
```

Description The `mdlSystem_deleteCfgVar` function removes the variable *cfgVarP* from the configuration variable table.

The `mdlSystem_deleteCfgVarAtLevel` differs from the `mdlSystem_deleteCfgVar` function in that it only removes the a definition at a single level. For example, if `MS_DEF` had a value at the system level and had been overridden at the user level MicroStation would use the value defined at the user level. The `mdlSystem_deleteCfgVarAtLevel` function could be used to delete the value at the user level. After successful completion of this function MicroStation would then use the value of `MS_DEF` that was defined at the system level. If the `mdlSystem_deleteCfgVarAtLevel` function is used to remove the level that contains the only definition of a configuration variable then it functions exactly as `mdlSystem_deleteCfgVar`.



`mdlSystem_deleteCfgVar` and `mdlSystem_deleteCfgVarAtLevel` cannot be used to remove values for/instances of operating system environment variables.

Returns `mdlSystem_defineCfgVar` returns `SUCCESS` if the variable is successfully deleted. If it cannot find *cfgVarP* in the variable table, it returns `ERROR`.

See Also `mdlSystem_getCfgVar`, `mdlSystem_defineCfgVar`.

mdlSystem_lockCfgVar, mdlSystem_isCfgVarLocked

```
int mdlSystem_lockCfgVar
(
    char    *cfgVarP          /* => name of variable to lock */
);
boolean mdlSystem_isCfgVarLocked
(
    char    *cfgVarP          /* => name of variable */
);
```

Description The `mdlSystem_lockCfgVar` function locks the variable *cfgVarP* in the configuration variable table. Once a configuration variable is locked the configuration variable cannot be deleted and its value cannot be changed.

`mdlSystem_isCfgVarLocked` queries the state of the lock for a given configuration variable. If it is of concern to the application, a call to `mdlSystem_getCfgVar` can be used to determine the configuration variable's existence.

Returns `mdlSystem_lockCfgVar` returns `SUCCESS` if the variable is successfully locked. If it cannot find *cfgVarP* in the variable table or in the system environment, it returns `ERROR`.

`mdlSystem_isCfgVarLocked` returns `TRUE` if the variable is locked and `FALSE` if it is not locked. If it cannot find *cfgVarP* in the variable table or in the system environment, it also returns `FALSE`.

See Also `mdlSystem_getCfgVar`, `mdlSystem_defineCfgVar`.

mdlSystem_processCfgVarFile

```
int mdlSystem_processCfgVarFile
(
    char    *fileName,          /* => name of file to process */
    int     startingProcessLevel /* => starting processing level */
);
```

Description Normally, configuration variable files are only processed during MicroStation's startup procedure. The `mdlSystem_processCfgVarFile` function can be used to initialize additional configuration variables inside of MicroStation.

fileName is the name of the configuration variable file to process.

startingProcessLevel is the level at which to start processing the file. Valid values are the `CFGVAR_LEVEL_XXX` constants defined in `msdefs.h` and correspond to the configuration file processing levels. Most applications will want to use `CFGVAR_LEVEL_USER`.



`mdlSystem_processCfgVarFile` should only be used to set directory search paths or application specific configuration variables that require no additional initialization. Certain MicroStation configuration variables,

(MS_SYMBRSRC and MS_USERPREF for example) require additional initialization upon MicroStation startup. It would therefore be invalid to set these variables through the mdlSystem_processCfgVarFile call.

Returns mdlSystem_processCfgVarFile returns ERROR if any errors occur during the processing of the configuration variable file. Otherwise it returns SUCCESS.

See Also mdlSystem_getCfgVar, mdlSystem_deleteCfgVar.

mdlSystem_createListFromCfgVarValue

```
#include <msdefs.h>
#include <mssystem.fdf>

int mdlSystem_createListFromCfgVarValue
/* <= # of values, -1 error */
(
StringList  **valueStrListPP, /* <= string list to hold parsed value */
char        *cfgVarValueP    /* => config. variable value to parse */
);
```

Description The mdlSystem_createListFromCfgVarValue function parses a configuration variable's value into a StringList. For example, the value of MS_DEF, which is a list of directories to search for design files, would have individual directories separated by the PATH_SEPARATOR_CHAR so it can be stored as a single buffer. mdlSystem_createListFromCfgVarValue breaks up the value at the PATH_SEPARATOR_CHAR and makes each item an entry in the StringList. This function would be called after mdlSystem_getCfgVar or mdlSystem_getExpandedCfgVar.

valueStrListPP is the address of a StringList pointer. A StringList will be created to hold the individual entries of a configuration variable's value. The calling application must free this StringList by calling mdlStringList_destroy. *valueStrListPP* can be NULL if the number of entries is the only item of interest.

cfgVarValueP points to a buffer containing a configuration variable's value.



This function was implemented in MicroStation 95.

Returns mdlSystem_createListFromCfgVarValue returns the number of entries in the created StringList or -1 if there is an error.

See Also mdlSystem_getCfgVar, mdlSystem_getExpandedCfgVar, mdlStringList_destroy.

mdlSystem_getCfgVarAtLevel

```
#include <msdefs.h>
#include <mssystem.fdf>
int mdlSystem_getCfgVarAtLevel
(
    char    *cfgVarValueP,      /* <= value of cfgVarNameP */
    char    *cfgVarNameP,      /* => name of variable to look up */
    int     maxlen,            /* => size of cfgVarValueP buffer */
    int     level               /* => level at which to look */
);
```

Description The `mdlSystem_getCfgVarAtLevel` function retrieves the unexpanded value of the configuration variable *cfgVarNameP* at a particular level. This is different from `mdlSystem_getCfgVar` which checks all levels.

The *cfgVarValueP* parameter points to the buffer that receives the value of the configuration variable. *cfgVarValueP* can be `NULL` if `mdlSystem_getCfgVarAtLevel` is just used to check for the existence of *cfgVarNameP* at a particular level.

The *cfgVarNameP* parameter points to the name of the configuration variable.

The *maxLen* parameter specifies the size of the buffer pointed to by *cfgVarValueP*.

The *level* parameter specifies the level to look for a value for *cfgVarNameP*. Valid values are the `CFGVAR_LEVEL_XXX` constants defined in `msdefs.h` and correspond to the configuration file processing levels.



This function was implemented in MicroStation 95.

Returns `mdlSystem_getCfgVarAtLevel` returns `SUCCESS` if the configuration variable is defined at level. Otherwise, it returns a non-zero value.

See Also `mdlSystem_getCfgVar`, `mdlSystem_getCfgVarByIndex`, `mdlSystem_expandCfgVar`.

mdlSystem_getExpandedCfgVar

```
#include <mssystem.fdf>
char *mdlSystem_getExpandedCfgVar
(
    char    *cfgVarNameP
);
```

Description The `mdlSystem_getExpandedCfgVar` function retrieves the expanded value of *cfgVarNameP*. Calling `mdlSystem_getExpandedCfgVar` would be similar to calling `mdlSystem_getCfgVar` and then immediately calling `mdlSystem_expandCfgVar`. *cfgVarNameP* points to the name of the configuration variable.



This function was implemented in MicroStation 95.

Returns `mdlSystem_getExpandedCfgVar` returns a pointer to a dynamically allocated buffer containing the value of *cfgVarNameP* or `NULL` if no value was found. The calling application must free this memory when the return value is not `NULL`.

See Also `mdlSystem_getCfgVar`, `mdlSystem_expandCfgVar`.

mdlSystem_rewriteCfgVarFile

```
#include <mssystem.fdf>

int mdlSystem_rewriteCfgVarFile
(
char    *cfgVarFileNameP,    /* => file name to rewrite */
char    *cfgVarNameP,        /* => name of variable */
char    *cfgVarValueP        /* => value of variable */
);
```

Description The `mdlSystem_rewriteCfgVarFile` function rewrites the *cfgVarFileNameP* file so that *cfgVarNameP* will get defined to *cfgVarValueP*. It does so without disturbing comments, etc. elsewhere in the file. `mdlSystem_rewriteCfgVarFile` does not affect the current session's environment. Use `mdlSystem_defineCfgVar` to change the current session.

cfgVarFileNameP is the name of the configuration variable file to rewrite. If `NULL` is passed, the current user configuration file is assumed.

cfgVarNameP is the name of the configuration variable.

cfgVarValueP is the value of *cfgVarNameP* that should be written to *cfgVarFileNameP*.



This function was implemented in MicroStation 95.

Returns `mdlSystem_rewriteCfgVarFile` returns `SUCCESS` if *cfgVarFileNameP* was updated successfully. Otherwise, a non-zero value is returned.

See Also `mdlSystem_processCfgVarFile`, `mdlSystem_defineCfgVar`, `mdlSystem_getCfgVar`

5

CAD Engine

Most MDL applications need to create, modify, and extract information from elements. MDL provides a set of functions to accomplish these tasks without requiring you to know the format of MicroStation elements. Even though the format for MicroStation elements is defined in `mselems.h`, the programmer should use the MDL create and extract functions since they isolate applications from changes in element structure.

This chapter discusses:

- Element creation functions
- Element information extraction functions
- Miscellaneous element functions
- Element descriptor functions
- Boolean functions
- Measurement functions
- Element intersection functions
- View functions
- Sectioning and hidden line viewing functions
- Auxiliary coordinate system functions
- Current transformation functions
- Transient element functions

Element Creation Functions

The following table lists the element creation functions:

Function	Used to
mdlLine_create	create a line element.
mdlLineString_create	create a line string.
mdlShape_create	create a shape element.
mdlCurve_create	create a curve element.
mdlLine_createI	create a line element with <i>points</i> as 32-bit integers.
mdlLineString_createI	create a line string with <i>points</i> as 32-bit integers.
mdlShape_createI	create a shape element with <i>points</i> as 32-bit integers.
mdlCurve_createI	create a curve element with <i>points</i> as 32-bit integers.
mdlArc_create	create an arc element.
mdlArc_createByPoints	create an arc element defined by three points.
mdlArc_createByCenter	create an arc element defined by its center and two end points.
mdlEllipse_create	create an ellipse from the parameters passed.
mdlCircle_createBy3Pts	create a circular ellipse element defined by three points.
mdlText_create	create a text element.
mdlTextNode_create	create an empty text node element.
mdlTextNode_createWithStrings	create a text node element with text strings.
mdlCone_create (3D only)	create a cone element.
mdlCone_createWithRotMatrix	create a cone element whose orientation is defined by a rotation matrix.
mdlCone_createRightCylinder (3D only)	create a right cylinder (cone element).
mdlCell_create	create a cell header.
mdlPointString_create	create a point string element.

The element creation functions have a varying number of arguments, depending on the type of element being created. In many cases, the arguments are pointers that define various aspects of the element. Often, the programmer can omit these arguments by passing `NULL`. MDL uses reasonable default values for the argument.

The first two parameters to all MDL creation functions point to the element being created and to an existing element, as the following example illustrates:

```
mdlLine_create(out, in ...)
```

The first argument, *out*, must always be declared as type `MSElementUnion` so it is large enough to hold the newly created element. *out* must always be non-NULL.

The second argument, *in*, points to an existing MicroStation element with the same type as the element being created. This element defines the display parameters (such as color, level, weight and line style) for the new element. If *in* is NULL, the current MicroStation settings for these parameters are used. This construct simplifies modifying existing elements.

Example

See `create.mc`.

mdlLine_create

```
#include <mselems.h>

int mdlLine_create
(
    MSElementUnion  *out,      /* <= line element created */
    MSElementUnion  *in,      /* => template element */
    Dpoint3d         *points   /* => end points */
);
```

Description The `mdlLine_create` function creates a MicroStation line element in *out* from the end points in the *points* parameter. The points are transformed by the current transform if one exists.

If *in* is NULL, the display parameters for the created element are taken from the active MicroStation settings when the function is called. Otherwise, the display parameters from *in* are used. All attribute information from *in* is retained in *out*.

Returns `mdlLine_create` returns `SUCCESS` if a valid MicroStation element is created even if the coordinates are beyond the design plane.

See Also `mdlLine_createI`, `mdlLinear_extract`.

mdlLineString_create

```
#include <mselems.h>

int mdlLineString_create
(
  MSElementUnion  *out,          /* <= line string created */
  MSElementUnion  *in,          /* => template element */
  Dpoint3d         *points,      /* => vertices */
  int              numVerts,     /* => number of vertices */
);
```

Description The `mdlLineString_create` function creates a line string element.

numVerts indicates the number of vertices in *points* and must be $2 \leq \text{numVerts} \leq 101$. The points are transformed by the current transform if one exists.

If *in* is `NULL`, the display parameters for the created element are taken from the active MicroStation settings when the function is called. Otherwise, the display parameters from *in* are used. All attribute information from *in* is retained in *out*.

Returns `mdlLineString_create` returns `SUCCESS` if a valid MicroStation element is created. It returns `MDLERR_BADELEMENT` if the coordinates are beyond the design plane.

See Also `mdlLineString_createI`, `mdlShape_create`, `mdlCurve_create`, `mdlLinear_extract`.

mdlShape_create, mdlCurve_create

```
#include <mselems.h>

int mdlShape_create
(
  MSElementUnion  *out,          /* <= shape created */
  MSElementUnion  *in,          /* => template element */
  Dpoint3d         *points,      /* => vertices */
  int              numVerts,     /* => number of vertices */
  int              fillMode      /* => filled or not, -1, 0 or 1 */
);

int mdlCurve_create
(
  MSElementUnion  *out,          /* <= curve created */
  MSElementUnion  *in,          /* => template element */
  Dpoint3d         *points,      /* => vertices */
  int              numVerts     /* => number of vertices */
);
```


Description The mdlShape_create function creates a shape element and the mdlCurve_create function creates a curve element.

numVerts indicates the number of vertices in *points* and must be $2 \leq \text{numVerts} \leq 101$. The points are transformed by the current transform if one exists.

fillMode for the mdlShape_create function determines whether the shape is filled or open. Possible values are as follows:

<i>fillMode</i>	Meaning
-1	Use the active <i>fillMode</i> from MicroStation.
0	The shape is not filled.
1	The shape is filled.

If *in* is NULL, the display parameters for the created element are taken from the active MicroStation settings when the function is called. Otherwise, the display parameters from *in* are used. All attribute information from *in* is retained in *out*.

Returns The mdlShape_create and mdlCurve_create functions return SUCCESS if a valid MicroStation element is created. They return MDLERR_BADELEMENT if the coordinates are beyond the design plane.

See Also mdlLineString_create, mdlShape_createI, mdlCurve_createI, mdlLinear_extract.

mdlLine_createI, mdlLineString_createI, mdlShape_createI, mdlCurve_createI

```
#include <mselems.h>

int mdlLine_createI
(
MSElementUnion  *out,      /* <= line element created */
MSElementUnion  *in,       /* => template element */
Point3d          *points    /* => points */
);

int mdlLineString_createI
(
MSElementUnion  *out,      /* <= line string created */
MSElementUnion  *in,       /* => template element */
Point3d          *points,   /* => points */
int              numVerts   /* => number of points in array */
);

int mdlShape_createI
(
MSElementUnion  *out,      /* <= shape created */
MSElementUnion  *in,       /* => template element */
```

```

Point3d      *points, /* => points */
int          numVerts, /* => number of vertices in shape */
int          fillMode /* => filled or not, -1, 0 or 1 */
);

int mdlCurve_createI
(
MSElementUnion *out, /* <= curve created */
MSElementUnion *in, /* => template element */
Point3d *points, /* => points */
int numVerts, /* => number of vertices */
int computeStart, /* => TRUE=compute slope */
int computeEnd /* => TRUE=compute slope */
);

```

Description `mdlLine_createI`, `mdlLineString_createI` and `mdlShape_createI` are identical to `mdlLine_create`, `mdlLineString_create` and `mdlShape_create` except *points* are 32-bit integers rather than double-precision floating point coordinates.

`mdlCurve_createI` is identical to `mdlCurve_create`, except that *points* is an array of 32-bit integers coordinates, and it has two additional parameters, *computeStart* and *computeEnd*.

The first two points and the last two points of a curve are used to indicate the starting and ending slope of the curve, and are non-displayable. If *computeStart* is `TRUE`, MicroStation will ignore any values in the first two coordinates of *points*, and calculate them based on the rest of the points. If *computeStart* is `FALSE`, MicroStation assumes that you have already calculated the slope and that the first two coordinates of *points* are valid. *computeEnd* has the same meaning for the last two points.

These functions are provided for compatibility with applications developed on other CAD systems with integer databases or developed for MicroStation before MDL. In general, using these functions is discouraged since they do not use the current transform, offer little or no performance benefits, and are not likely to be compatible with future MicroStation enhancements.

Returns See the `mdlLine_create`, `mdlLineString_create`, `mdlShape_create` and `mdlCurve_create` elements.

See Also `mdlLine_create`, `mdlLineString_create`, `mdlShape_create`, `mdlCurve_create`, `mdlLinear_extract`.

mdlArc_create

```
#include <mselems.h>

int mdlArc_create
(
  MSElementUnion  *out,          /* <= arc created */
  MSElementUnion  *in,          /* => template element */
  Dpoint3d         *center,      /* => center of arc (or NULL) */
  double           primary,      /* => primary axis */
  double           secondary,    /* => secondary axis */
  RotMatrix        *rMatrix,     /* => rotation matrix (or NULL) */
  double           start,        /* => starting angle */
  double           sweep         /* => sweep angle */
);
```

Description The `mdlArc_create` function creates a MicroStation arc element in *out* from the parameters passed. All parameters other than *start* and *sweep* are transformed by the current transform if one exists.

center defines the center point for the arc. If *center* is NULL, the center is placed at (0, 0, 0) in the current coordinate system.

primary and *secondary* are the sizes of the primary and secondary arc axes.

rMatrix is the arc's rotation matrix. If *rMatrix* is NULL, the identity matrix (no rotation) is used.

start and *sweep* are the arc's starting and sweep angles in radians.

If *in* is NULL, the display parameters for the created element are taken from the active MicroStation settings when the function is called. Otherwise, the display parameters from *in* are used. All attribute information from *in* is retained in *out*.

Returns The `mdlArc_create` function returns SUCCESS if a valid MicroStation element is created. It returns MDLERR_BADELEMENT if the resulting arc is beyond the design plane.

See Also `mdlArc_createByPoints`, `mdlArc_extract`.

mdlArc_createByPoints, mdlArc_createByCenter

```
#include <mselems.h>

int mdlArc_createByPoints
(
  MSElementUnion  *out,          /* <= arc created */
  MSElementUnion  *in,          /* => template element */
  Dpoint3d         arcPts[3]     /* => points on arc */
);

int mdlArc_createByCenter
```

```
(
MSElementUnion  *out,      /* <= arc created */
MSElementUnion  *in,      /* => template element */
Dpoint3d         arcPts[3], /* => center and end points */
int              useRadius, /* => 1=use radius, 0=use points */
double           radius,    /* => radius (if useRadius is TRUE) */
int              view       /* => view number */
);
```

Description The `mdlArc_createByPoints` function creates a MicroStation arc element in *out* defined by three points. The points are transformed by the current transform if one exists. *arcPts[0]* is the starting point, *arcPts[2]* is the ending point, and *arcPts[1]* is another point on the arc.

The `mdlArc_createByCenter` function creates a MicroStation arc element in *out* defined by its center and two end points. The points are transformed by the current transform if one exists. *arcPts[0]* is the starting point, *arcPts[1]* is the center point, and *arcPts[2]* is the ending point.

If *useRadius* is TRUE, the radius is specified by *radius*, and *arcPts[0]* and *arcPts[2]* determine only the arc's start and sweep. Otherwise, the radius is determined by the distance between the center and the first point. If all three *arcPts* are colinear in a 3D file, the rotation matrix of *view* determines the arc's plane.

If *in* is NULL, the display parameters for the created element are taken from the active MicroStation settings when the function is called. Otherwise, the display parameters from *in* are used. All attribute information from *in* is retained in *out*.

Returns Both `mdlArc_createByPoints` and `mdlArc_createByCenter` return SUCCESS if a valid MicroStation element is created. They return MDLERR_BADELEMENT if the resulting arc is beyond the design plane or if the three points are colinear.

See Also `mdlArc_create`, `mdlArc_extract`.

mdlEllipse_create

```
#include <mselems.h>

int mdlEllipse_create
(
MSElementUnion  *out,      /* <= ellipse created */
MSElementUnion  *in,      /* => template element */
Dpoint3d         *center,  /* => center of ellipse (or NULL) */
double           primary,  /* => primary axis */
double           secondary, /* => secondary axis */
RotMatrix        *rMatrix, /* => rotation matrix (or NULL) */
int              fillMode  /* => filled or not, -1, 0 or 1 */
);
```

Description mdlEllipse_create creates a MicroStation ellipse element in *out* from the parameters passed. All parameters are transformed by the current transform if one exists.

center defines the ellipse's center point. If *center* is NULL, the center is placed at (0, 0, 0) in the current coordinate system.

The primary axis (the axis at 0° in the arc/ellipse coordinate system) is passed in *primary*, and the secondary axis (the axis at 90°) is passed in *secondary*. You can produce a circle by setting *primary* equal to *secondary*.

rMatrix is the arc's rotation matrix. If *rMatrix* is NULL, the identity matrix (no rotation) is used.

fillMode determines whether the created ellipse is filled. See mdlShape_create for possible values of *fillMode*.

If *in* is NULL, the display parameters for the created element are taken from the active MicroStation settings when the function is called. Otherwise, the display parameters from *in* are used. All attribute information from *in* is retained in *out*.

Returns The mdlEllipse_create function returns SUCCESS if a valid MicroStation element is created. It returns MDLERR_BADELEMENT if the resulting ellipse is beyond the design plane.

See Also mdlCircle_createBy3Pts, mdlArc_extract.

mdlCircle_createBy3Pts

```
#include <mselems.h>

int mdlCircle_createBy3Pts
(
  MSElementUnion  *out,           /* <= circle created */
  MSElementUnion  *in,           /* => template element */
  Dpoint3d         points[3],     /* => perimeter points */
  int              fillMode       /* => filled or not, -1, 0, 1 */
);
```

Description The mdlCircle_createBy3Pts function creates a MicroStation circular ellipse element in *out* defined by three points in the perimeter. The points in the *point* parameter are transformed by the current transform if one exists.

fillMode determines whether the created circle is filled. See mdlShape_create for possible values of *fillMode*.

If *in* is NULL, the display parameters for the created element are taken from the active MicroStation settings when the function is called. Otherwise, the display parameters from *in* are used. All attribute information from *in* is retained in *out*.

Returns The `mdlCircle_createBy3Pts` function returns `SUCCESS` if a valid MicroStation element is created. It returns `MDLERR_BADELEMENT` if the resulting ellipse is beyond the design plane.

See Also `mdlEllipse_create`, `mdlArc_extract`.

mdlText_create

```
#include <mdl.h>
#include <mselems.h>

int mdlText_create
(
  MSElementUnion  *out,          /* <= text element created */
  MSElementUnion  *in,          /* => template element */
  char             *stringParam, /* => characters */
  Dpoint3d         *origin,      /* => origin (or NULL) */
  TextSizeParam    *txSize,      /* => size (or NULL) */
  RotMatrix        *rMatrix,     /* => rotation matrix (or NULL) */
  TextParam        *txtParams,   /* => parameters (or NULL) */
  TextEDParam      *edParam      /* => enter data info (or NULL) */
);
```

Description The `mdlText_create` function creates a MicroStation text element in *out* based on the parameters and an existing text element, *in*. If the *stringParam*, *origin*, *txSize*, *rMatrix*, *txtParams* or *edParam* is `NULL`, its value is derived from the text element, *in*.

stringParam points to an ASCII character string containing the text element's characters.

origin specifies the coordinates for the text element's snap point. The text element is placed relative to the *origin* depending on the text justification. If *origin* and *in* are `NULL`, the text element is placed at (0, 0, 0) in the current coordinate system.

txSize points to a `TextSizeParam` structure (defined in `mdl.h`) that determines the resulting text element's size. The text element's size depends on one of the following values of the *txSize->mode* field:

<i>txSize->mode</i>	Meaning of <i>txSize->size</i> field
<code>TXT_BY_TILE_SIZE</code>	The size of a one-character cell.
<code>TXT_BY_MULT</code>	The multiplication factor for one character cell. (Text elements are defined in the MicroStation reference guide).
<code>TXT_BY_TEXT_SIZE</code>	The total size of the text element.
<code>TXT_BY_WIDTH_ASPECT</code>	<i>txSize->size.width</i> defines length of text element, and <i>txSize->aspectRatio</i> defines the aspect ratio of the character cells.

The text size is scaled by the current transform. To specify sizes in absolute units (without using the current transformation), combine the *txSize->mode* field with the `TXT_NO_TRANSFORM` constant using the OR operator. If *txSize* and *in* are `NULL`, the text element is created with the active text size when the function is called.

rMatrix is a rotation matrix to be applied to the text element. If *rMatrix* and *in* are `NULL`, the identity matrix is used.

txtParams points to a `TextParam` structure defined in `mdl.h`. This structure defines the created text element's font number, style, justification, and view-independence. If any member of *txtParams* is negative, the default value from *in* (or from the current value if *in* is `NULL`) is used. If *txtParams* and *in* are `NULL`, the active font, style and justification are used and the text element is not view-independent. Possible values for *txtParams->just* are defined in `mdl.h`.

edParam points to a `TextEdParam` structure, defined in `mdl.h` that specifies the number and locations of enter-data fields within the text string. The enter-data fields are specified with an array of `TextEdField` structures of size *edParam->numEdFields* pointed to by *edParam->edField*. Each entry in the `TextEdField` array defines the starting character position, the length and the justification for a single enter-data field. If both *edParam* and *in* are `NULL`, the text element has no enter-data fields.

If *in* is `NULL`, the display parameters for the created element are taken from the active MicroStation settings when the function is called. Otherwise, the display parameters from *in* are used. All attribute information from *in* is retained in *out*.

Returns The `mdlText_create` function returns `SUCCESS` if a valid MicroStation text element is created. If *in* and *stringParam* are `NULL`, `mdlText_create` returns `MDLERR_INSFINFO`. It returns `MDLERR_BADELEMENT` if the text is beyond the design plane.

See Also `mdlTextNode_create`, `mdlText_extract`, `mdlText_extractShape`.

mdlTextNode_create

```

#include <mdl.h>
#include <mselems.h>
int mdlTextNode_create
(
  MSElementUnion    *out,          /* <= text node created */
  MSElementUnion    *in,          /* => template element */
  Dpoint3d           *origin,      /* => origin (or NULL) */
  RotMatrix           *rMatrix,    /* => rotation matrix (or NULL) */
  double             *lineSpacing, /* => line spacing (or NULL) */
  TextSize            *size,       /* => deflt text size (or NULL) */
  TextParam           *txtParams   /* => text parameters (or NULL) */
);

```

Description `mdlTextNode_create` creates an empty MicroStation text node element in *out*.

origin specifies the coordinates for the text node's snap point. The text attached to a text node is placed relative to the *origin* depending on the text node justification. If *origin* is NULL, the text node is placed at (0, 0, 0) in the current coordinate system.

rMatrix is a rotation matrix to be applied to the text element. If *rMatrix* is NULL, the identity matrix is used.

lineSpacing specifies the distance between lines in the text node element when text is attached. If *lineSpacing* is NULL, the active line spacing is used.

size is the height and width of the character cell for text attached to this text node. If *size* is NULL, the active text height and width are used.

txtParams points to a `TextParam` structure defined in `mdl.h`. This structure defines the created text node element's font number, style, justification, and view-independence. If any member of *txtParams* is negative, the default value from *in* (or from the current value if *in* is NULL) is used. If *txtParams* and *in* are NULL, the active font, style and justification are used and the text element is not view-independent. Possible values for *txtParams->just* are defined in `mdl.h`.

If *in* is NULL, the display parameters for the created element are taken from the active MicroStation settings when the function is called. Otherwise, the display parameters from *in* are used. All attribute information from *in* is retained in *out*.

Returns `mdlTextNode_create` returns `SUCCESS` if a valid MicroStation element is created. It returns `MDLERR_BADELEMENT` if the resulting text node is beyond the design plane.

See Also `mdlText_create`, `mdlTextNode_extract`, `mdlTextNode_extractShape`.

mdlTextNode_createWithStrings

```
#include <mdl.h>
#include <mselems.h>

int mdlTextNode_createWithStrings
(
    MSElementDescr    **out,          /* <= text node created */
    MSElementUnion    *in,           /* => template element */
    Dpoint3d           *origin,       /* => origin (or NULL) */
    RotMatrix           *rMatrix,     /* => rotation matrix (or NULL) */
    double              *lineSpacing, /* => line spacing (or NULL) */
    TextSize            *size,        /* => default text size (or NULL) */
    TextParam           *txtParams,   /* => text parameters (or NULL) */
    UChar               *strings[],   /* => array of string pointers */
    int                 totalLines,   /* => number of lines in strings */
    TextEDParam         *textEDParam /* => array of ED field structures */
);
```

Description The `mdlTextNode_createWithStrings` function creates a MicroStation text node element with text strings in *out*.

origin specifies the coordinates for the text node's snap point. The text attached to a text node is placed relative to the *origin* depending on the text node justification. If *origin* is `NULL`, the text node is placed at (0, 0, 0) in the current coordinate system.

rMatrix is a rotation matrix to be applied to the text element. If *rMatrix* is `NULL`, the identity matrix is used.

lineSpacing specifies the distance between lines in the text node element when text is attached. If *lineSpacing* is `NULL`, the active line spacing is used.

size is the height and width of the character cell for text attached to this text node. If *size* is `NULL`, the size from *in* is used. If *in* is also `NULL`, the active text height and width are used.

txtParams points to a `TextParam` structure defined in `mdl.h`. This structure defines the created text node element's font number, style, justification, and view-independence. If any member of *txtParams* is negative, the default value from *in* (or from the current value if *in* is `NULL`) is used. If *txtParams* and *in* are `NULL`, the active font, style and justification are used and the text element is not view-independent. Possible values for *txtParams*->*just* are defined in `mdl.h`.

The argument *Strings* is an array of string pointers, similar to the standard argument *argv* which is passed to every C program main function. It is *not* a two-dimensional array of characters. MDL programs working with `RTYPE_MultilineText` dialog items may use `mdlText_getStringArrayFromBuffer` to create a compatible argument. The total number of string pointers is given by *totalLines*. The enter data

field information for each line is given in an array of `TextEDParam` structures, *textEDParam*.

If *in* is `NULL`, the display parameters for the created element are taken from the active MicroStation settings when the function is called. Otherwise, the display parameters from *in* are used. All attribute information from *in* is retained in *out*.

Returns The `mdlTextNode_createWithStrings` function returns `SUCCESS` if a valid MicroStation element is created. It returns `MDLERR_BADELEMENT` if the resulting text node is beyond the design plane.

See Also `mdlText_create`, `mdlTextNode_extract`, `mdlText_getStringArrayFromBuffer`.

mdlCone_create (3D only)

```
#include <mselems.h>

int mdlCone_create
(
    MSElementUnion  *out,          /* <= cone element */
    MSElementUnion  *in,          /* => template element */
    double           topRadius,    /* => radius of top of cone */
    double           bottomRadius, /* => radius at bottom */
    Dpoint3d         *base,        /* => center of base */
    Dpoint3d         *top,         /* => center of top */
    RotMatrix        *coneRMatrix /* => rotate (skew) matrix, or NULL */
);
```

Description The `mdlCone_create` function creates a MicroStation cone element in *out*. The cone is defined by two circles specified by their radii, *topRadius*, and *bottomRadius* and their center points, *base* and *top*. If *base* or *top* is `NULL`, the coordinate (0, 0, 0) in the current coordinate system is used.

coneRMatrix specifies a skew rotation to be applied to the top and bottom circles. If *coneRMatrix* is `NULL`, a right cone is created.

Returns The `mdlCone_create` function returns `SUCCESS` if a valid MicroStation element is created. If the design file is 2D, `mdlCone_create` returns `MDLERR_FILE2SUB3`. It returns `MDLERR_BADELEMENT` if the resulting cone is beyond the design plane.

See Also `mdlCone_createWithRotMatrix`, `mdlCone_createRightCylinder`, `mdlCone_extract`.

mdlCone_createWithRotMatrix

```
int mdlCone_createWithRotMatrix
(
  MSElement *out,          /* <= element to be created */
  MSElement *in,          /* => optional starting element */
  double topRadius,        /* => top radius */
  double bottomRadius,     /* => bottom radius */
  Dpoint3d *base,          /* => center of base */
  Dpoint3d *top,           /* => center of top */
  RotMatrix *coneRMMatrix /* => general cone Transformation */
);
```

Description The `mdlCone_createWithRotMatrix` function creates a MicroStation cone element in *out*. The cone is defined by two circles specified by their radii *topRadius* and *bottomRadius* and their center points, *base* and *top*. The orientation of both circles is specified by *coneRMMatrix*. This rotation matrix specifies the circle orientation directly and is not a skew matrix as used in `mdlCone_create`; it is the same rotation matrix that is returned by `mdlCone_extract`.

Returns `mdlCone_createWithRotMatrix` returns `SUCCESS` if a valid MicroStation element is created. If the design file is 2D, `mdlCone_create` returns `MDLERR_FILE2SUB3`. It returns `MDLERR_BADELEMENT` if the resulting cone is beyond the design plane.

See Also `mdlCone_create`, `mdlCone_extract`.

mdlCone_createRightCylinder (3D only)

```
#include <mselems.h>

int mdlCone_createRightCylinder
(
  MSElementUnion *out,      /* <= cylinder element */
  MSElementUnion *in,      /* => template element */
  double radius,           /* => radius */
  Dpoint3d *base,          /* => center of base */
  Dpoint3d *top,           /* => center of top */
);
```

Description The `mdlCone_createRightCylinder` function is provided as a convenience. It produces the same result as calling `mdlCone_create` with *topRadius* and *bottomRadius* equal and *coneRMMatrix* set to `NULL`:

```
mdlCone_create(out, in, radius, radius, base, top, NULL);
```

Returns `mdlCone_createRightCylinder` returns `SUCCESS` if a valid MicroStation element is created. If the design file is 2D, it returns `MDLERR_FILE2SUB3`. It returns `MDLERR_BADELEMENT` if the resulting cone is beyond the design plane.

See Also `mdlCone_create`, `mdlCone_extract`.

mdlCell_create

```
#include <mselems.h>

void mdlCell_create
(
  MSElementUnion  *cell,          /* <= cell header element */
  char             *cellName,      /* => cell name */
  Dpoint3d         *origin,        /* => cell origin */
  boolean          pointCell       /* => TRUE = point cell */
);
```

Description *mdlCell_create* creates a cell header element in *cell*. *cellName* is the cell name and should be an ASCII string that has six characters or less. It should consist only of valid Radix 50 characters. (See *mdlCnv_fromAsciiToR50* for a discussion of valid Radix 50 characters). To create an orphan cell header, set *cellName* to *NULL*.

origin is the cell origin. If *NULL*, it is set to (0, 0, 0) in the current coordinate system.

If *pointCell* is *TRUE*, the cell header will be a point cell header. Otherwise, it is a normal cell.



mdlCell_create is normally used to create header elements for use with *mdlElmdscr_new*.

Returns The *mdlCell_create* function is of type *void*. It returns no value.

See Also *mdlCell_extract*, *mdlElmdscr_new*, *mdlCnv_fromAsciiToR50*.

mdlPointString_create

```
#include <mselems.h>

void mdlPointString_create
(
  MSElementUnion  *out,          /* <= point string element */
  MSElementUnion  *in,          /* => template element */
  Point3d          *points,       /* => vertices */
  double           *rMatrices,    /* => array of rotation matrices */
  int              numVerts,      /* => number of vertices */
  int              disjoint       /* => disjoint or continuous */
);
```

Description The *mdlPointString_create* function creates a MicroStation point string element. A point string element consists of vertices with orientations at each vertex.

points specifies the point string's vertices. Unlike all other MDL element creation routines, *mdlPointString_create* accepts *Point3d* points rather than *DPoint3d* points and is not transformed by the current transform.

rMatrices is an array of rotation matrices, one matrix for each vertex. In 2D, the rotation matrices are 2×2 matrices, and in 3D they are 3×3 matrices.

numVerts is the number of vertices in the point string. *points* and *rMatrices* should be dimensioned to *numVerts*. The maximum number of vertices for a point string is 48.

disjoint determines whether *out* is created as a disjoint or continuous point string. Continuous point strings display with lines connecting the vertices.

Returns The `mdlPointString_create` function is of type `void`. It returns no value.

Element Information Extraction Functions

Element information extraction functions let the programmer extract information from MicroStation elements without needing to know element formats. The element information extraction functions return information about a specific MicroStation element type and expect to be called only with an element of that type. Functions in the “Miscellaneous Element Functions” section extract the header information (the information that is common to all element types).

The following table lists element information extraction functions:

Function	Used to
<code>mdlLinear_extract</code>	extract an array of coordinates from a linear element.
<code>mdlLinear_getClosestSegment</code>	get the line string segment that is closest to a point.
<code>mdlArc_extract</code>	extract information from an arc or ellipse element.
<code>mdlText_extract</code>	extract information from a text element.
<code>mdlText_extractShape</code>	return a rectangle that is the smallest bounding box around the text element.
<code>mdlText_extractString</code>	return a character string contained in a text element.
<code>mdlTextNode_extract</code>	extract information from a text node element.
<code>mdlTextNode_extractShape</code>	return a rectangle that is the smallest bounding box around the text node.
<code>mdlCone_extract</code>	extract information from a cone element.
<code>mdlCell_extract</code>	extract information from a cell header element.

Function	Used to
mdlSharedCell_extract	extract information from a shared cell instance element or shared cell definition element.
mdlElement_extractRange	provide the range of an element.

mdlLinear_extract

```
#include <mselems.h>

int mdlLinear_extract
(
  Dpoint3d      *points,      /* <= vertices */
  int            *numVerts,    /* <= number of vertices */
  MSElementUnion *in,        /* => linear element */
  int            filenum      /* => 0 for master, 1-72 for reference */
);
```

Description The `mdlLinear_extract` function extracts an array of coordinates from a linear element pointed to by *in*. The coordinates are transformed into the current coordinate system if one exists.

The *numVerts* argument is set to the number of vertices in *points*. Sufficient memory should be allocated in *points* to receive MAX_VERTICES vertices.



`mdlLinear_extract` *does* copy hidden points in a curve element into the points array. Be aware that the hidden points consist of two points at the beginning and two points at the end of the curve that are used only as curve definition data and are not displayed.

Returns `mdlLinear_extract` returns SUCCESS if a valid MicroStation element of type LINE_ELM, LINE_STRING_ELM, SHAPE_ELM, CONIC_ELM, CURVE_ELM, MULTILINE_ELM or BSPLINE_POLE_ELM is passed in *in*. Otherwise, it returns MDLERR_BADELEMENT.

See Also `mdlLine_create`, `mdlLineString_create`, `mdlShape_create`, `mdlCurve_create`.

mdlLinear_getClosestSegment

```
#include <mselems.h>

int mdlLinear_getClosestSegment
(
  Dvector3d      *segment, /* <= segment closest to point */
  Dpoint3d       *point,   /* <=> point to test (and return) */
  MSElementUnion *linear  /* => linear element */
);
```

Description The mdlLinear_getClosestSegment function finds a line segment of the linear element *linear*. This segment is closest to the point *point*. It also adjusts *point* to the closest point on *linear*.

Returns The mdlLinear_getClosestSegment function returns the segment number (relative to 0) from which *segment* was derived. Only MicroStation elements of type LINE_ELM, LINE_STRING_ELM, SHAPE_ELM or CURVE_ELM are valid.

mdlArc_extract

```
#include <mselems.h>

int mdlArc_extract
(
    Dpoint3d      *startEndPts, /* <= 2 points (or NULL) */
    double        *start,      /* <= starting angle (or NULL) */
    double        *sweep,      /* <= sweep angle (or NULL) */
    double        *primary,     /* <= primary axis (or NULL) */
    double        *secondary,   /* <= secondary axis (or NULL) */
    RotMatrix     *rMatrix,     /* <= rotation matrix (or NULL) */
    Dpoint3d      *center,      /* <= center of arc (or NULL) */
    MSElementUnion *in         /* => arc element */
);
```

Description The mdlArc_extract function extracts information from a MicroStation arc or ellipse element, *in*. The coordinates and rotation matrix are transformed into the current coordinate system.

startEndPts points to an array of two Dpoint3ds to receive the coordinates of the arc's starting and ending points. If *in* is an ellipse, the two coordinates are the same. If these coordinates are not needed, pass NULL for *startEndPts*.

start and *sweep* point to doubles to receive the starting and sweep angle in radians. The sweep will be 2π for ellipses.

primary and *secondary* point to doubles to receive the arc or ellipse's primary and secondary axes.

rMatrix points to a rotation matrix describing the arc or ellipse's rotation.

center points to a Dpoint3d structure to receive the coordinates of the arc or ellipse's center.



MicroStation always creates arcs with a start angle of 0 and uses the rotation matrix to rotate the arc if it does not start at 0 degrees in the world coordinate system.

Returns The mdlArc_extract function returns SUCCESS if a valid MicroStation element of type ARC_ELM or ELLIPSE_ELM is passed in *in*. Otherwise, it returns MDLERR_BADELEMENT.

See Also mdlArc_create, mdlArc_createByPoints, mdlEllipse_create, mdlCircle_createBy3Pts.

mdlText_extract

```
#include <mselems.h>

int mdlText_extract
(
    Dpoint3d          *origin,          /* <= origin */
    Dpoint3d          *userOrigin,      /* <= snap point */
    int               *numEdfields,     /* <= num of enter data fields */
    TextEdField       *edFields,        /* <= ed fields structures */
    char              *string,          /* <= character string */
    RotMatrix         *rMatrix,         /* <= rotation matrix */
    TextStyleInfo     *textStyleInfo,   /* <= font and style */
    int               *just,            /* <= justification mode */
    TextSize          *tileSize,        /* <= tile size */
    TextSize          *textSize,        /* <= total text size */
    MSElementUnion   *in               /* => text element */
);
```

Description The mdlText_extract function extracts information from the MicroStation text element, *in*. The coordinates, sizes, and rotation matrices are transformed into the current coordinate system.

All information about the text element is returned in memory pointed to by the arguments. The program must allocate the memory for these structures. If the information in any parameter is not needed, pass NULL and MicroStation does not fill it in.

The *origin* of the text element is always the coordinate of the lower-left corner. The *userOrigin* is the text element's snap point. The position of *userOrigin* relative to *origin* depends on the text element's justification.

The number of enter-data fields in the text element is returned in *numEdfields* and the array of the TextEDField structure is returned in *edFields*.

The ASCII character string contained in the text element is returned in *string*.

The text element's rotation matrix is returned in *rMatrix*.

The font and style are returned in the TextStyleInfo structure pointed to by *textStyleInfo*.

The justification for the text element is returned in *just*. Possible values for *just* are defined in mdl.h.

The text element's size is returned in two forms. First, the size of a single character cell is returned in the TextSize structure pointed to by *tileSize*.

Second, the overall text element size is returned in the `TextSize` structure pointed to by *textSize*.



In versions 4.2 or newer of MicroStation, `mdlText_extract` can return double- or wide-byte strings if used in that context. To indicate that such a string is being returned, `$fffe` will be the first two bytes in the string.

Returns The `mdlText_extract` function returns `SUCCESS` if *in* is a valid MicroStation element of type `TEXT_ELM`. Otherwise, it returns `MDLERR_BADELEMENT`.

See Also `mdlText_create`, `mdlText_extractString`, `mdlText_extractShape`.

mdlText_extractShape

```
#include <mselems.h>

int mdlText_extractShape
(
    Dpoint3d      *points,          /* <= shape vertices */
    Dpoint3d      *userOrigin,     /* <= snap point */
    MSElementUnion *in,           /* => text element */
    int           addSnapTol,      /* => add snap tolerance */
    int           view             /* => view number if addSnapTol TRUE */
);
```

Description The `mdlText_extractShape` function returns a rectangle (five `Dpoint3ds`) in *points*. This rectangle is the smallest bounding box around the text element. If *addSnapTol* is `TRUE`, MicroStation makes the bounding box larger by the current snap tolerance. This feature is useful for deciding whether a point is close to the text element. The view for the snap tolerance is given by *view*.

The snap point for the text element is returned in *userOrigin*. If this coordinate is not needed, pass `NULL` for *userOrigin*.

Returns The `mdlText_extract` function returns `SUCCESS` if *in* is a valid MicroStation element of type `TEXT_NODE_ELM`. Otherwise, it returns `MDLERR_BADELEMENT`.

See Also `mdlText_create`, `mdlText_extractString`, `mdlText_extract`.

mdlText_extractString

```
#include <mselems.h>

int mdlText_extractString
(
    char          *string,         /* <= character string */
    MSElementUnion *in           /* => text element */
);
```

Description The `mdlText_extractString` function returns the ASCII character string contained in a MicroStation text element in *string*.

Returns The `mdlText_extractString` function returns `SUCCESS` if *in* is a valid MicroStation element of type `TEXT_ELM`. Otherwise, it returns `MDLERR_BADELEMENT`.

See Also `mdlText_create`, `mdlText_extractShape`, `mdlText_extract`.

mdlTextNode_extract

```
#include <mselems.h>

int mdlTextNode_extract
(
    Dpoint3d      *origin,          /* <= origin of text node */
    RotMatrix     *rMatrix,         /* <= rotation matrix */
    TextSize      *tileSize,        /* <= tile size */
    double        *lineSpacing,     /* <= line spacing */
    TextParam     *textParams,      /* <= font, just, etc. */
    int           *nodeNumber,      /* <= number of this node */
    MSElementUnion *node           /* => text node element */
);
```

Description The `mdlTextNode_extract` function extracts information from the MicroStation text node element *node*. The origin, sizes and rotation matrix are transformed into the current coordinate system.

All information about the text node is returned in memory pointed to by the arguments. The program must allocate the memory for these structures. If the information in any of the output parameters is not needed, pass `NULL` and MicroStation does not fill in these parameters.

The node origin is returned in *origin*.

The rotation matrix for the text node is returned in *rMatrix*.

The default size for text elements attached to this text node is returned in the `TextSize` structure pointed to by *tileSize*. These values are the height and width of a single-character cell.

The line spacing is returned in *lineSpacing*.

The font, justification, style, and view-independence are returned in the `TextParam` structure pointed to by *textParams*.

The node number is returned in *nodeNumber*.

Returns The `mdlTextNode_extract` function returns `SUCCESS` if *node* is a valid MicroStation element of type `TEXT_NODE_ELM`. Otherwise, it returns `MDLERR_BADELEMENT`.

See Also `mdlTextNode_create`, `mdlTextNode_extractShape`.

mdlTextNode_extractShape

```
#include <mselems.h>

int mdlTextNode_extractShape
(
    Dpoint3d      *points,      /* <= shape vertices */
    Dpoint3d      *origin,      /* <=> snap point or NULL */
    MSElementUnion *in,        /* => text element */
    int           addsnaptol,    /* => add snap tolerance */
    int           view          /* => view number if addSnapTol TRUE */
);
```

Description The mdlTextNode_extractShape function returns a rectangle (five Dpoint3ds) in *points*. This rectangle is the smallest bounding box around the text node. If *addSnapTol* is TRUE, MicroStation makes the bounding box larger by the current snap tolerance. This feature is useful for deciding whether the text node is close to another point. The view for the snap tolerance is given by *view*.

The snap point for the text node is returned in *origin*. Pass NULL for *origin* if this coordinate is not needed.

Returns The mdlTextNode_extractShape function returns SUCCESS if *in* is a valid MicroStation element of type TEXT_ELM. Otherwise, it returns MDLERR_BADELEMENT.

See Also mdlTextNode_create, mdlTextNode_extract.

mdlCone_extract

```
#include <mselems.h>

int mdlCone_extract
(
    double      *topRadius,      /* <= radius at top */
    double      *bottomRadius,   /* <= radius at bottom */
    Dpoint3d    *topCenter,      /* <= coordinate of top center */
    Dpoint3d    *bottomCenter,   /* <= coord of bottom center */
    RotMatrix   *rMatrix,        /* <= skew matrix */
    MSElementUnion *cone        /* => cone element */
);
```

Description The mdlCone_extract function extracts the information from a cone element, *cone*. If any parameters are NULL, this function does not attempt to fill them in.

topRadius and *bottomRadius* are the radii for the cone's top and bottom.

topCenter and *bottomCenter* are the cone's top and bottom center coordinates.

rMatrix is the orientation of the cone's top and bottom.

Returns The mdlCone_extract function returns SUCCESS if *cone* is a valid MicroStation element of type CONE_ELM. Otherwise, it returns MDLERR_BADELEMENT.

See Also mdlCone_create (3D only).

mdlCell_extract

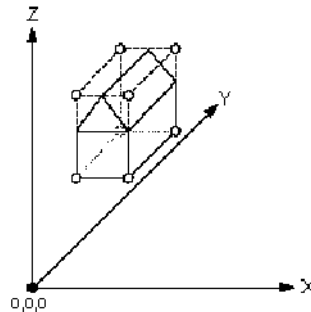
```
#include <mselems.h>

int mdlCell_extract
(
  Dpoint3d    *origin,      /* <= origin of cell */
  Dpoint3d    *shape,       /* <= bounding shape (8 Dpoint3ds) */
  RotMatrix   *rMatrix,     /* <= orientation of cell */
  Dpoint3d    *scale,       /* <= scaling vector */
  char        *cellName,    /* <= cell name (ASCII) */
  MSElementUnion *cell      /* => cell element */
);
```

Description The mdlCell_extract function extracts the information from a cell header element, *cell*. If any parameters are NULL, this function does not attempt to fill them in. All parameters are returned in the current (design file) coordinate system.

origin is the cell origin.

shape returns an array of eight Dpoint3ds (shown as *s* in the diagram below) which represent the minimum bounding box for the cell in the coordinate system of the cell. This idea can be illustrated by using MicroStation's element selection tool to select a cell. MicroStation places handles on the boundary which defines the cell, and these handles correlate to the eight Dpoint3ds returned by mdlCell_extract.



The points returned in shape for mdlCell_extract



The points in *shape* are transformed into the current coordinate system before they are returned by `mdlCell_extract`. They do not necessarily represent a minimum bounding box in the current coordinate system.

rMatrix is the rotation matrix for the cell. This is a matrix that holds the cell's rotation.

scale is a `Dpoint3d` that holds the cell's X, Y and Z scale factors.

cellName is the cell's name in ASCII.

Returns The `mdlCell_extract` function returns `SUCCESS` if *cell* is a valid MicroStation element of type `CELL_HEADER_ELM`. Otherwise, it returns `MDLERR_BADELEMENT`.

See Also `mdlCell_create`.

mdlSharedCell_extract

```
#include <mselems.h>

int mdlSharedCell_extract
(
    Dpoint3d    *origin,        /* <= shared cell origin */
    Dpoint3d    *shape,        /* <= bounding shape (8 Dpoint3ds) */
    RotMatrix   *rMatrix,      /* <= orientation of cell */
    Dpoint3d    *scale,        /* <= scaling vector */
    char        *cellName,     /* <= cell name (ASCII) */
    SCOVERRIDE  *override,     /* <= values overridden */
    MSElementUnion *sharedCell, /* => shared cell element */
    int         fileName       /* => design file (master or reference) */
);
```

Description The `mdlSharedCell_extract` function extracts information from a shared cell instance element or shared cell definition element. If output parameters are `NULL`, the function does not attempt to fill them in. All parameters are returned in the current coordinate system.

origin is the shared cell origin.

shape returns an array of eight `Dpoint3ds`, which constitute a box that defines the extents of the cell. This idea can be illustrated if you were to use MicroStation's element selection tool to select a cell. MicroStation places handles on the boundary which defines the cell, and these handles correlate to the eight `Dpoint3ds` returned by `mdlCell_extract`.



The points returned in *shape* are in cell coordinates and not design file coordinates. The diagram on the previous page illustrates the difference between the two systems, with the cube surrounding the cell indicating the cell coordinate system.

rMatrix is the shared cell's rotation matrix. This is a (possibly non-orthogonal) matrix that holds the cell's rotation.

scale is a `Dpoint3d` that holds the cell's X, Y and Z scale factors.

cellName is the shared cell's name in ASCII.

override is a structure that defines the fields in the shared cell instance element. These fields override the values in the shared cell definition elements.

sharedCell points to the shared cell element.

fileNumber is the file that contains the shared cell (MASTERFILE for the master file and 1-256 for a reference file).

Returns The `mdlCell_extract` function returns `SUCCESS` if *sharedCell* is a valid MicroStation element of type `SHARED_CELL_ELM`. Otherwise, it returns `MDLERR_BADELEMENT`.

See Also `mdlSharedCell_create`.

mdlElement_extractRange

```
int mdlElement_extractRange
(
    Dvector3d    *rangeP,
    MSElement   *elementP
);
```

Description `mdlElement_extractRange` generates the range of the element in the current transform. It should be used for non-complex elements. For B-splines, it only provides the bounds of the control polygon. For view-independent elements, the range encompasses any orientation of the element.

Returns `mdlElement_extractRange` returns `SUCCESS` or `ERROR`.

Miscellaneous Element Functions

Each element function operates on a single MicroStation element. For operating on complex elements (such as cells, text nodes and complex shapes), use the element descriptor functions in the “Element Descriptor” section.

The following table lists the element functions:

Function	Used to
mdlElement_add	add a new MicroStation element to the design file.
mdlElement_append	add a modified MicroStation element to the design file.
mdlElement_rewrite	overwrite an existing MicroStation element with a different element.
mdlElement_display	display an element in all active views.
mdlElement_displayInSelectedViews	display an element in all active views and specify the views into which the element will be drawn.
mdlElement_undoableDelete	delete an element.
mdlElement_extractAttributes	extract attribute information from the element.
mdlElement_appendAttributes	append attribute information to the end of any of the element's existing attributes.
mdlElement_stripAttributes	remove all attribute information from an element.
mdlElement_size	return an element's size in internal format.
mdlElement_igdsSize	return an element's size in design file format.
mdlElement_read	read an element from a design file to the buffer.
mdlElement_getFilePos	return an element's file position.
mdlElement_setFilePos	set an element's file position.
mdlElement_getSymbology	extract symbology information from an element.
mdlElement_setSymbology	set symbology information in an element.
mdlElement_getType	extract the element type from the element.
mdlElement_createColorTable	create a type 5 color table element.
mdlElement_setFillColor	set an element's fill color.
mdlElement_getFillColor	query an element's fill color.
mdlElement_getProperties	extract property information from an element.

Function	Used to
<code>mdlElement_setProperties</code>	set property information in an element.
<code>mdlElement_isFilled</code>	query an element's fill status.
<code>mdlElement_cnvToFileFormat</code>	convert an element from internal to external file format.
<code>mdlElement_cnvFromFileFormat</code>	convert an element from external to internal file format.
<code>mdlElement_transform</code>	transform an element by a transformation matrix.
<code>mdlElement_offset</code>	offset an element by distance.
<code>mdlElement_stroke</code>	stroke an element into vectors.
<code>mdlElement_getLineStyle</code>	retrieve the line style attribute linkage information from an element.
<code>mdlElement_setLineStyle</code>	set the line style of an element to a new specified MicroStation line style.

The following table lists Element functions that are used to create type 66 elements. These elements are used by MicroStation to store various settings in a design file. All of these functions create the type 66 from the applicable, current active settings.

Function	Used to
<code>mdlElement_createDigitizerSettings</code>	create a digitizer settings type 66 element.
<code>mdlElement_createDimensionSettings</code>	create a dimension settings type 66 element.
<code>mdlElement_createViewSettings</code>	create a view settings type 66 element.
<code>mdlElement_createExtendedTCB</code>	create an extended TCB type 66 element.

Example

See `element.mc`.

mdlElement_add, mdlElement_append

```
#include <mselems.h>

ULong mdlElement_add
(
MSElement    *element    /* => element to add */
);
ULong mdlElement_append
(
MSElement    *element    /* => element to append */
);
```

Description The mdlElement_add function adds a **new** MicroStation element to the design file. *element* points to this new element. The mdlElement_add function creates new elements. Before writing the element to the file, mdlElement_add sets the properties bits in the element header to not locked, new element and not modified.

The mdlElement_append function adds the **modified** MicroStation element pointed to by *element* to the design file. The mdlElement_append function is needed when an application modifies an element and changes its size. mdlElement_rewrite can also be used for this purpose, but this function requires that you know the old file position.



MicroStation remembers the mdlElement_add and mdlElement_append functions, so the user can undo them.

Returns The mdlElement_add and mdlElement_append functions return the file position of the element added to the design file. If an error occurs, the file position is set to zero and the global variable mdlErrno is set to the specific error cause. Possible values for mdlErrno are MDLERR_READONLY, MDLERR_DISKFULL, MDLERR_WRITEINHIBIT, MDLERR_BADELEMENT and MDLERR_WRITEFAILED.

See Also mdlElement_rewrite.

mdlElement_rewrite

```
#include <mselems.h>

ULong mdlElement_rewrite
(
MSElement    *newElem,      /* => new element */
MSElement    *oldElem,      /* => old element (or NULL) */
ULong         filePos        /* => file position of element */
);
```

Description mdlElement_rewrite overwrites an existing MicroStation element pointed to by *oldElem* with a new element pointed to by *newElem* at file position *filePos*.

If the two elements have different sizes, MicroStation deletes the old one and appends the new element to the end of the file. Otherwise, it overwrites the old element at the same file position.

MicroStation needs the old element to save in the undo buffer. If you do not have a copy of the old element when you want to overwrite it, pass `NULL` for *oldElem* and MicroStation will re-read the old element from the cache. This process adds overhead to `mdlElement_rewrite`, so always pass the old element if you have it.



MicroStation remembers the `mdlElement_rewrite` function, so the user can undo it.

Returns `mdlElement_rewrite` returns the file position of the element added to the design file. If the two elements are the same size, the file position will be the same as *filePos*.

If an error occurs, the file position is set to zero and the global variable `mdlErrno` is set to the specific error cause. Possible values for `mdlErrno` are `MDLERR_READONLY`, `MDLERR_DISKFULL`, `MDLERR_WRITEINHIBIT`, `MDLERR_BADELEMENT` and `MDLERR_WRITEFAILED`.

See Also `mdlElement_add`, `mdlElement_append`.

mdlElement_display, mdlElement_displayInSelectedViews

```
#include <mselems.h>

void mdlElement_display
(
MSElement    *element,      /* => element to be displayed */
int           drawMode      /* => drawing mode */
);

void mdlElement_displayInSelectedViews
(
MSElement    *element,      /* => element to be displayed */
int           drawMode,      /* => drawing mode */
int           viewMask       /* => one bit per view */
);
```

Description `mdlElement_display` displays the MicroStation element pointed to by *element* in all active views. It displays non-complex elements that may or may not actually be in the design file. If *element* is `NULL`, MicroStation draws the element in `dgnBuf`.

drawMode determines how MicroStation displays the element. Possible values for *drawMode* are as follows:

<i>drawMode</i>	Meaning of <i>drawMode</i> field
NORMALDRAW	Draw the element in its normal color.
ERASE	Erase the element.
HILITE	Draw the element in the current highlight color.

The `mdlElement_displayInSelectedViews` function is identical to `mdlElement_display` except that it has an additional parameter, *viewMask*, that specifies the views into which the element will be drawn. *viewMask* is a bit mask, with one bit per view; the lowest bit specifies view 1.

Returns The `mdlElement_display` and `mdlElement_displayInSelectedViews` functions are of type `void`. They return no values.

See Also `mdlElmdscr_display`, `mdlElmdscr_displayFromFile`.

mdlElement_undoableDelete

```
#include <mselems.h>

int mdlElement_undoableDelete
(
MSElement    *element,      /* => element to delete (or NULL) */
ULong         filePos,       /* => filepos of element */
int           display        /* => TRUE = erase from screen */
);
```

Description The `mdlElement_undoableDelete` function deletes the element pointed to by *element* at file position *filePos*. MicroStation needs the element to save in the undo buffer. If you do not have a copy of the element when you want to delete it, pass `NULL` for *element* and MicroStation will re-read the element from the file. This process adds overhead to `mdlElement_undoableDelete`, so always pass the element if it is known.

If *display* is `TRUE`, MicroStation erases the element from the screen as it deletes it. Otherwise, it does not erase the element.



MicroStation remembers the element deleted by `mdlElement_undoableDelete`, so that the user can undo the delete if necessary.

Returns If the element is deleted, `mdlElement_undoableDelete` returns `SUCCESS`. If an error occurs, the global variable `mdlErrno` is set to the specific error cause. Possible values for `mdlErrno` are `MDLERR_READONLY`, `MDLERR_WRITEINHIBIT`, `MDLERR_BADELEMENT` and `MDLERR_MODIFYCOMPLEX`.

See Also `mdlElmdscr_undoableDelete`.

mdlElement_extractAttributes

```
#include <mselems.h>

void mdlElement_extractAttributes
(
    int          *length,          /* <= length of returned attrs */
    short        *attributes,      /* <= attribute data */
    MSElement   *element         /* => element to extract from */
);
```

Description The mdlElement_extractAttributes function extracts attribute information from the element pointed to by *element*. The attributes are copied to the *attributes* buffer and *length* is set to the size of the attribute data in words.

The *attributes* array should be large enough to hold MAX_ATTRIBSIZE words.



element is not changed by mdlElement_extractAttributes.

Returns The mdlElement_extractAttributes function is of type void. It returns no value.

See Also mdlElement_appendAttributes.

mdlElement_appendAttributes

```
#include <mselems.h>

void mdlElement_appendAttributes
(
    MSElement*element,          /* <=> element to append to */
    int      length,             /* => length of attr data */
    short    *attributes         /* => attribute data */
);
```

Description The mdlElement_appendAttributes function appends attribute information in the *attributes* buffer to the end of any existing attributes on the element pointed to by *element*. The size of the attribute data in words is specified by *length*.



No validity checking is performed on the attribute information in attributes. All attribute rules should be adhered to.

Returns The mdlElement_appendAttributes function returns no value.

See Also mdlElement_extractAttributes, mdlElement_stripAttributes.

mdlElement_stripAttributes

```
#include <mselems.h>

void mdlElement_stripAttributes
(
  MSElementUnion   *out,      /* <= element with no attrs */
  MSElementUnion   *in       /* => original element */
);
```

Description The mdlElement_stripAttributes function strips attribute information from the input element *in* and puts the result in *out*. *in* and *out* can point to the same element.



Certain element types (CMPLX_STRING_ELM and CMPLX_SHAPE_ELM) must always have attributes. mdlElement_stripAttributes does not remove such required attributes.

Returns The mdlElement_stripAttributes function is of type void. It returns no status.

See Also mdlElement_appendAttributes, mdlElement_extractAttributes, mdlDB_detachAttributesElement, mdlDB_detachAttributesDscr.

mdlElement_size, mdlElement_igdsSize

```
#include <mselems.h>

int mdlElement_igdsSize
(
  MSElement   *element,      /* => element */
  int          fileNumber     /* => used to tell 2D/3D */
);

int mdlElement_size
(
  MSElement   *element       /* => element */
);
```

Description MicroStation stores elements in memory using a format that differs from the one used in the design file. The mdlElement_size and mdlElement_igdsSize functions determine an element's size in internal or design file format.

The mdlElement_size function returns the size of *element* in internal (memory) format.

The mdlElement_igdsSize function returns the size of *element* in external (design file) format. *fileNumber* is 0 for the master file and 1 through MAX_REFS for reference files. MAX_REFS is defined in mdl.h.



fileNumber is needed since 2D reference files can be attached to 3D master files. The mdlElement_igdsSize function is needed only to determine the file position of the next element in a design file.

Returns The `mdlElement_size` function returns the size of *element* in bytes. The `mdlElement_igdsSize` function returns the size of *element* in disk format in words.

See Also `mdlElmdscr_igdsSize`.

mdlElement_read

```
#include <mselems.h>

int mdlElement_read
(
MSElement    *element,      /* <= must be of type MSElement */
int           fileName,      /* => file number to read */
ULONG         filePos        /* => file position */
);
```

Description The `mdlElement_read` function reads an element from a design file to the buffer pointed to by *element*. *element* should point to an `MSElementUnion` structure so it is large enough to hold the largest MicroStation element.

fileName indicates the file from which the function will read. File 0 means read from the master file, file 1 through `MAX_REFS` are for reference files, and file `CELL_LIB` is for the cell library. `MAX_REFS` and `CELL_LIB` are defined in `msdefs.h`.

filePos is the element's file position.



The `mdlElement_read` function uses the design file cache when possible.

Returns The `mdlElement_read` function returns `SUCCESS` if an element is read into *element*. If *fileName* is invalid because it is out of the 0 through `MAX_REFS` range or it indicates an unattached reference file, `mdlElement_read` returns `MDLERR_BADFILENAME`. If *filePos* points to the end of file marker, it returns `MDLERR_ENDOFFILE`.

See Also `mdlScan_file`.

mdlElement_getFilePos

```
#include <mdl.h>

ULONG mdlElement_getFilePos
(
int    type,          /* => type of file position */
int    *fileName      /* <= file number */
);
```

Description The `mdlElement_getFilePos` function returns the file position for various elements.

fileName points to an integer indicating the file number for the file position returned. If you are not interested in the file number, pass `NULL` for *fileName*.

type indicates the type of file position. Possible values are as follows:

<i>type</i>	mdlElement_getFilePos returns
FILEPOS_EOF	the current end of file position for the master file (<i>tcb->dfsect</i> , <i>tcb->dfbyte</i>).
FILEPOS_CURRENT	the file position of the current element (<i>tcb->curebl</i> , <i>tcb->cureby</i>). <i>fileNumber</i> is set to the current file number (<i>tcb->cureff</i>).
FILEPOS_FIRST_ELE	the file position of the first element in the master file.
FILEPOS_NEXT_ELE	the file position of the next element (<i>tcb->eleblk</i> , <i>tcb->elecmt</i>). <i>fileNumber</i> is set to the current file number (<i>tcb->cureff</i>).
FILEPOS_WORKING_SET	the file position of the working set, if one is active (<i>tcb->wssect</i> , 0).
FILEPOS_WORKING_WINDOW	the file position of the working window (<i>tcb->wwsect</i> , <i>tcb->wwbyte</i>). <i>fileNumber</i> is set to <i>tcb->wwfile</i> .
FILEPOS_COMPONENT	the component offset of the element that caused a complex element to be located. Will be zero if a non-complex element was located.

Returns The mdlElement_getFilePos function returns the file position of the element indicated by *type*. If an invalid value is passed for *type*, the function returns 0xffffffff.

See Also mdlElement_setFilePos.

mdlElement_setFilePos

```
#include <mdl.h>

ULong mdlElement_setFilePos
(
    int    type,           /* => parameter name */
    int    fileNumber,     /* => file number */
    ULong  filePos         /* => file position */
);
```

Description The mdlElement_setFilePos function sets the file position for various elements.

filePos is the file position to be set and *fileNumber* is an integer indicating the file number for that file position. If you do not want to change the file number, pass -1 for *fileNumber*.

type indicates the type of file position to be set. Possible values are as follows:

<i>type</i>	mdlElement_setFilePos <i>sets</i>
FILEPOS_CURRENT	the file position of the current element (<i>tcb->curebl</i> , <i>tcb->cureby</i>). <i>tcb->curefl</i> is set to <i>fileNumber</i> .
FILEPOS_NEXT_ELE	the file position of the next element (<i>tcb->eleblk</i> , <i>tcb->elecn</i>). <i>tcb->curefl</i> is set to <i>fileNumber</i> .
FILEPOS_WORKING_WINDOW	the file position of the working window (<i>tcb->wwsect</i> , <i>tcb->wwbyte</i>). <i>tcb->wwfile</i> is set to <i>fileNumber</i> .



Some file positions returned by `mdlElement_getFilePos` are read-only and cannot be set with `mdlElement_setFilePos`.

Returns The `mdlElement_setFilePos` function returns `SUCCESS` if the indicated file position is set. It returns `-1` if an invalid *type* is passed.

See Also `mdlElement_getFilePos`.

mdlElement_getSymbology, mdlElement_setSymbology, mdlElement_getType

```
#include <mselems.h>

void mdlElement_getSymbology
(
    int          *color,          /* <= color (or NULL) */
    int          *weight,         /* <= weight (or NULL) */
    int          *style,          /* <= style (or NULL) */
    MSElement   *element         /* => element */
);

void mdlElement_setSymbology
(
    MSElement   *element,        /* <=> element */
    int          *color,          /* => color (or NULL) */
    int          *weight,         /* => weight (or NULL) */
    int          *style           /* => style (or NULL) */
);

int mdlElement_getType
(
    MSElement   *element         /* => element */
);
```

Description The `mdlElement_getSymbology` function extracts the color, weight and style from the element pointed to by *element*. This function puts the extracted values in the

variables pointed to by *color*, *weight* and *style*. If parameters other than *element* are NULL, MicroStation does not fill in the value.

The mdlElement_setSymbology function changes the color, weight, and style in the element pointed to by *element* to the values pointed to by *color*, *weight* and *style*. If parameters other than *element* are NULL, MicroStation does not change the value in *element*.

The mdlElement_getType function returns the type of the element pointed to by *element*.

Returns mdlElement_getSymbology and mdlElement_setSymbology return no values.

The mdlElement_getType function returns the element type. Element type values are defined in mselems.h. For example, for an ellipse element, the mdlElement_getType function returns ELLIPSE_ELM.

See Also mdlElement_getProperties, mdlElement_setProperties, mdlElmdscr_setProperties.

mdlElement_createColorTable

```
void mdlElement_createColorTable
(
MSElement    *elementP,      /* <= color table element */
byte          *colorP,        /* => color table (768 bytes) */
char          *cnameP         /* => color table name */
);
```

Description The mdlElement_createColorTable function creates a type 5 color table element. Only one color table element should exist in a design file. A color table contains the RGB colors for each of the element colors and the background color.

elementP points to the element to be created.

colorP is a pointer to an array of 256 RGB color triplets. Each color contains a single byte for the red, green and blue color intensities. The total size of the color table is therefore $3 \times 256 = 768$ bytes.

cnameP is the color table name.

Returns mdlElement_createColorTable is of type void; it returns no value.

mdlElement_setFillColor, mdlElement_getFillColor

```

void mdlElement_setFillColor
(
MSElement  *elementP,      /* element to set */
int         fillColor       /* fill color to set */
);
int mdlElement_getFillColor
(
MSElement  *elementP,      /* element to query */
);

```

Description `mdlElement_setFillColor` sets an element's interior fill color, while `mdlElement_getFillColor` queries the existing fill color of an element. In MicroStation version 5 and later, the interior fill color can differ from the outline color. The `mdlElmdscr_addFill` function sets the fill color to match the outline color.

elementP is the element to be queried or whose fill color is to be set. If *elementP* is not filled, *elementP* is not modified.

fillColor is the color index to be assigned to the element.

Returns `mdlElement_setFillColor` is of type `void`. `mdlElement_getFillColor` returns the color index that was used to fill the queried element.

See Also `mdlElement_isFilled`, `mdlElmdscr_addFill`, `mdlElmdscr_stripFill`.

mdlElement_getProperties, mdlElement_setProperties

```

#include <mselems.h>

void mdlElement_getProperties
(
int      *level,           /* => level (or NULL) */
int      *ggNumber,        /* => graphic group # (or NULL) */
int      *class,           /* => class (or NULL) */
int      *locked,          /* => locked (or NULL) */
int      *new,             /* => new element (or NULL) */
int      *modified,        /* => modified (or NULL) */
int      *viewInd,         /* => view independent (or NULL) */
int      *solidHole,        /* => solid/hole (or NULL) */
MSElement *element /* => element */
);

void mdlElement_setProperties
(

```

```

MSElement  *element,
int          *level,      /* <= level (or NULL) */
int          *ggNumber,   /* <= graphic group # (or NULL) */
int          *class,      /* <= class (or NULL) */
int          *locked,     /* <= locked (or NULL) */
int          *new,        /* <= new element (or NULL) */
int          *modified,   /* <= modified (or NULL) */
int          *viewInd,    /* <= view independent (or NULL) */
int          *solidHole   /* <= solid/hole (or NULL) */
);

```

Description The `mdlElement_getProperties` function extracts the level, graphic group number, class, locked status, new status, modified status, view independence status, and solid-hole status from the element pointed to by *element*. This function puts the extracted values in the variables pointed to by *level*, *ggNumber*, *class*, *locked*, *new*, *modified*, *viewInd* and *solidHole*. If parameters other than *element* are `NULL`, MicroStation does not fill in the value.

The `mdlElement_setProperties` function changes the level, graphic group number, class, locked status, new status, modified status, view independence status, and solid-hole status in the element pointed to by *element*. This function changes these values to the values pointed to by *level*, *ggNumber*, *class*, *locked*, *new*, *modified*, *viewInd* and *solidHole*. If parameters other than *element* are `NULL`, MicroStation does not change the value in *element*.

Returns `mdlElement_getProperties` and `mdlElement_setProperties` return no values.

See Also `mdlElement_getSymbology`, `mdlElement_setSymbology`, `mdlElement_getType`, `mdlElmdscr_setProperties`.

mdlElement_isFilled

```

int mdlElement_isFilled
(
MSElement  *element /* => element */
);

```

Description The `mdlElement_isFilled` function indicates *element* fill status. The fill status is stored in attribute information at the end of the element.

Returns The `mdlElement_isFilled` function returns `TRUE` if an element is filled and `FALSE` if it is not filled.

mdlElement_cnvToFileFormat, mdlElement_cnvFromFileFormat

```
# include <mselems.h>

int mdlElement_cnvToFileFormat
(
    byte          *out,
    MSElement    *in,
    int           threeD
);
int mdlElement_cnvFromFileFormat
(
    MSElement    *out,          /* <= elem in internal format */
    byte          *in,          /* => file format */
    int           conversionType /* => conversion flag */
);
```

Description MicroStation stores elements in the design file in a format that is different from the one used internally. Normally this format is transparent to applications, since MicroStation automatically converts from internal to external format and vice versa. Most MDL applications use elements in internal format. Applications may occasionally need to operate directly on elements in file format. The `mdlElement_cnvToFileFormat` and `mdlElement_cnvFromFileFormat` functions are provided to convert between formats.

The `mdlElement_cnvToFileFormat` function converts the element *in* to external format in the buffer *out*. *out* should be a buffer of size `MAX_ELEMENT_SIZE`. Once the element is in file format, `mdlElement_cnvFromFileFormat` is the only function that can use it. *threeD* is TRUE if the element is in 3D and FALSE if it is in 2D.

The `mdlElement_cnvFromFileFormat` function converts the element *in* from external to internal format in *out*. *conversionType* indicates the type of conversion to be performed: 0 is 2D to 2D conversion, 1 is 3D to 3D conversion, and 2 is 2D to 3D conversion.

Returns The `mdlElement_cnvToFileFormat` and `mdlElement_cnvFromFileFormat` functions return `SUCCESS` if the conversion is successful. Otherwise, they return `ERROR`.

mdlElement_transform, mdlElement_offset

```
int mdlElement_transform
(
MSElement  *out,          /* <= transformed element */
MSElement  *in,           /* => original element */
Transform   *tMatrix       /* => transformation matrix */
);

int mdlElement_offset
(
MSElement  *out,          /* <= offset element */
MSElement  *in,           /* => original element */
Dpoint3d    *offset        /* => distance to offset */
);
```

Description The mdlElement_transform function transforms the element *in* by the transformation matrix *tMatrix* and places the result in *out*. *in* and *out* can point to the same element. The transformation matrix relates to the current coordinate system if one exists.

The mdlElement_offset function moves the element *in* by the distance given by *offset*. The function then places the result in *out*. *in* and *out* can point to the same element. The distance is given in the current coordinate system if one exists.



Since the transformation matrix for mdlElement_transform includes translation, the mdlElement_offset function is technically unnecessary. However, since elements are frequently offset, mdlElement_offset is a highly optimized version of mdlElement_transform.

Returns The mdlElement_transform and mdlElement_offset functions return SUCCESS if the element is transformed properly and ERROR if the resulting element is beyond the design plane.

mdlElement_stroke

```
int mdlElement_stroke
(
Dpoint3d    **points,      /* <= point array allocation */
int         *numPoints,    /* <= number of points in points */
MSElement  *element,      /* => element to be stroked */
double      tolerance      /* => arc tolerance */
);
```

Description The mdlElement_stroke function strokes the element pointed to by *element* into vectors in an array pointed to by *points*. This array should be the address of a pointer to a Dpoint3d structure. MicroStation allocates the memory for the vectors and returns the starting address in *points*. The application programmer frees this

memory when it is no longer needed. The number of points in the array is returned in *numPoints*.

tolerance is the maximum distance between the actual curve and the approximating vectors for curved elements.

Returns The `mdlElement_stroke` function returns `SUCCESS` if *element* is stroked and *points* and *numPoints* are valid. It returns `ERROR` if *element* is not valid.

See Also `mdlElmdscr_stroke`.

mdlElement_getLineStyle

```
#include <mselems.h>
#include <mslstyle.h>
#include <mselemen.fdf>

int mdlElement_getLineStyle
(
    char          *pStyleName,    /* <= Element style name */
    StyleParam    *pParams,      /* <= Element style params */
    MSElement    *pElement,     /* => Element to extract info from */
    int           fileNo,        /* => File element is from */
    int           lsIndex        /* => Line style index */
);
```

Description The `mdlElement_getLineStyle` function retrieves the line style attribute linkage information from the element pointed to by *pLineStyle*.

The name of the line style will be copied to the buffer pointed to by *pStyleName*. Pass `NULL` for this parameter if you do not want the name to be returned.

If *pParams* is not `NULL` the parameters extracted from the line style linkage will be copied there.

The *pElement* parameter is the address of the element from which the line style linkage will be extracted.

The *fileNo* parameter must contain the file number of the design file from which the element came. Pass `MASTERFILE` (zero) for the master file of 1-255 for reference files.

If *pElement* is a multi-line element it can contain up to nineteen different line styles; one for each of the sixteen possible profile lines, one for each end cap, and one for all joint lines. A value of 0 for *lsIndex* indicates the default line style used for all components that do not specify overrides. Values from 1 - 16 indicate the corresponding profile lines 1 - 16 (or 0 - 15 zero based). A value of 17 indicates the origin cap, 18 indicates the end cap and 19 indicates the line style used for joint lines.

Returns mdlElement_getLineStyle returns SUCCESS (zero) if the element contained a line style linkage and the information was extracted successfully. Otherwise an error status is returned.

See Also mdlElement_setLineStyle.

mdlElement_setLineStyle

```
#include <mselems.h>
#include <mslstyle.h>
#include <mselemen.fdf>

int mdlElement_setLineStyle
(
MSElement    *pElement,      /* <= Element to modify */
int           fileNo,         /* => Design file number */
int           lsIndex,        /* => Line style index */
char          *pStyleName,     /* => Line style name */
StyleParam    *pParams        /* => Line style parameters */
);
```

Description The mdlElement_setLineStyle function sets the line style of the element pointed to by *pElement* to the new MicroStation line style defined by *pStyleName* and *pParams*. Setting the element line style involves creating a line style attribute linkage and attaching it to the element. The element must be small enough to accommodate the new line style linkage and the buffer pointed to by *pElement* must be large enough to accommodate the new linkage.

pElement contains the address of the element to be modified.

fileNo indicates which file the element is to be placed in. Currently, the only valid option is MASTERFILE (zero).

If *pElement* is a multi-line element it can contain up to nineteen different line styles; one for each of the sixteen possible profile lines, one for each end cap, and one for all joint lines. A value of 0 for *lsIndex* indicates the default line style used for all components that do not specify overrides. Values from 1 - 16 indicate the corresponding profile lines 1 - 16 (or 0 - 15 zero based). A value of 17 indicates the origin cap, 18 indicates the end cap and 19 indicates the line style used for joint lines.

pStyleName specifies the name of the line style that will be attached to the element *pElement*.

If *pParams* is not NULL the parameters extracted from the line style linkage will be copied there. If *pParams* is NULL, the element's attribute linkage will be removed.

Returns mdlElement_setLineStyle returns SUCCESS (zero) if the operation was completed successfully. Otherwise an error status is returned.

See Also mdlElement_getLineStyle.

**mdlElement_createDigitizerSettings, mdlElement_createDimensionSettings,
mdlElement_createViewSettings, mdlElement_createExtendedTCB**

```
#include <mselems.h>

int mdlElement_createDigitizerSettings
(
MSElement    *elementP      /* <= Type 66 with digitizer settings */
);

int mdlElement_createDimensionSettings
(
MSElement    *elementP      /* <= Type 66 with dimension settings */
);

int mdlElement_createViewSettings
(
MSElement    *elementP      /* <= Type 66 with view settings */
);

int mdlElement_createExtendedTCB
(
MSElement    *elementP      /* <= Type 66 w/ extended TCB settings */
);
```

Description mdlElement_createDigitizerSettings creates a type 66 element with the current digitizer settings. The digitizer settings include the transformation from digitizer coordinates to design file coordinates created DIGITIZER SETUP command and the digitizer partition created by the DIGITIZER PARTITION command.

mdlElement_createDimensionSettings creates a type 66 element with the current dimension settings.

mdlElement_createViewSettings creates a type 66 element with the location of the view windows.

mdlElement_createExtendedTCB creates a type 66 element with the extended TCB parameters. The extended TCB parameters consist of TCB data that is stored in the design file but did not exist in the original IGDS type 9 design file header elements. These parameters include the extended view information, extended locks, active cell settings, the full cell library specification and the design file specific rendering settings.

Returns All functions return SUCCESS or an appropriate error code (see mdlerrs.h).

Element Descriptor Functions

Elements are stored sequentially in design files. Each primitive element has a fixed format header that contains the element's size so that the next element can be found. Any primitive element has a maximum of 768 words, so MDL programs can create or manipulate elements by declaring a buffer of type `MSElement`. This technique works well for primitive elements, since each one can be manipulated on its own.

However, it does not work well with complex elements. Complex elements (such as cells, text nodes, complex shapes, complex chains and B-splines) are stored in the design file with a complex header element followed by a series of component elements. The component elements cannot be modified without information in the header element being updated. Since the number and size of component elements can vary, MDL programs are unlikely to declare a buffer large enough to hold complex elements.



For this reason, the element descriptor functions are provided. The element descriptor functions enable components of complex elements to be manipulated without the need to maintain the headers. MicroStation automatically updates header information. Automatic header update greatly simplifies the task of modifying complex elements and virtually eliminates a common cause of file corruption.

The main difference between using the element descriptor functions and using the simple element functions is that the element descriptor functions operate on memory that MicroStation allocates, and the programmer must provide memory for simple element functions. The memory allocated for element descriptors can be freed with `mdlElement_freeAll`. MDL applications should never allocate element descriptors directly. Instead, the element descriptor routines allocate them. MDL applications should always use pointers to element descriptors.

Element descriptors are composed of a linked list of elements. These lists generally hold complex elements, but they can also hold a single simple element or a list of simple elements.

A common source of errors when using element descriptors is when you need to copy a element from an element descriptor. The obvious thing to do is:

```
ElementUnion newElement;
memcpy(&newElement, &edP->el, sizeof(newElement));
```

However, this does not work because the element descriptor only has enough memory allocated to it to hold the element (which will always be less than or equal to the size of an `ElementUnion`). As a result, programs end up reading past the end of allocated memory. This does not cause problems in MS-DOS, but can cause intermittent core dumps (SIGSEGV) on UNIX systems.

The proper way to construct the above code is:

```
ElementUnion newElement;
memcpy(&newElement, &edP->el, mdlElement_size(&edP->el));
```



Userdata fields, areas where custom data may be stored, are kept in the element descriptor header. This information is not written back into the design file; only the actual element is written. See the `elemdescr.mc` example to see how to handle userdata fields.

The following table lists element descriptor functions:

Function	Used to
<code>mdlElmdscr_read</code>	read element(s) from the design file and allocate a new element descriptor.
<code>mdlElmdscr_readToMaster</code>	read element(s) from the design file and allocate a new element descriptor. Transform elements to master file coordinates.
<code>mdlElmdscr_add</code>	add the <i>new</i> elements contained in the element descriptor to the design file.
<code>mdlElmdscr_append</code>	add the modified elements contained in the element descriptor to the design file.
<code>mdlElmdscr_undoableDelete</code>	delete the elements contained in the element descriptor from the design file.
<code>mdlElmdscr_transform</code>	transform the element descriptor.
<code>mdlElmdscr_rewrite</code>	overwrite the existing elements with the new elements.
<code>mdlElmdscr_display</code>	display the element descriptor in all active views.
<code>mdlElmdscr_displayFromFile</code>	read and display an element in the design file.
<code>mdlElmdscr_displayInSelectedViews</code>	display the element descriptor in selected views.
<code>mdlElmdscr_displayFromFileViews</code>	read and display an element in selected views in the design file.
<code>mdlElmdscr_displaySingle</code>	display a single element descriptor.
<code>mdlElmdscr_stroke</code>	stroke the element descriptor to vectors.
<code>mdlElmdscr_operation</code>	call a user function for each element descriptor member.
<code>mdlElmdscr_igdsSize</code>	determine the element descriptor file size.
<code>mdlElmdscr_singleIgdsSize</code>	return size of a single element descriptor.
<code>mdlElmdscr_markElement</code>	mark the element in the element descriptor for later identification.

Function	Used to
<code>mdlElmdscr_duplicate</code>	copy the element descriptor.
<code>mdlElmdscr_duplicateSingle</code>	duplicate a single element descriptor.
<code>mdlElmdscr_validate</code>	update a complex header for the element descriptor.
<code>mdlElmdscr_reverse</code>	reverse direction of element.
<code>mdlElmdscr_reverseNormal</code>	reverse the normal direction for a surface, solid or closed element.
<code>mdlElmdscr_partialDelete</code>	partially delete an element.
<code>mdlElmdscr_extractEndPoints</code>	extract endpoints of element.
<code>mdlElmdscr_new</code>	create a new element descriptor from an element.
<code>mdlElmdscr_appendElement</code>	append the element to the element descriptor.
<code>mdlElmdscr_insertElement</code>	insert the element in an existing element descriptor.
<code>mdlElmdscr_removeElement</code>	remove the element from the element descriptor.
<code>mdlElmdscr_replaceElement</code>	replace an element in the element descriptor.
<code>mdlElmdscr_addToChain</code>	add the element descriptor to another element descriptor.
<code>mdlElmdscr_appendDscr</code>	append an element descriptor to another element descriptor.
<code>mdlElmdscr_replaceDscr</code>	replace an element descriptor.
<code>mdlElmdscr_createFromVertices</code>	create an element with an arbitrary number of vertices.
<code>mdlElmdscr_generatePartial</code>	generate and return a portion of an element descriptor.
<code>mdlElmdscr_createShapeWithHoles</code>	create an orphan cell entity that associates a solid element with one or more hole entities.
<code>mdlElmdscr_hiddenLineRemoval</code>	perform same function as <code>mdlView_hiddenLineRemoval</code> but on an element descriptor.
<code>mdlElmdscr_addFill</code>	add fill to a closed element.
<code>mdlElmdscr_stripFill</code>	remove fill from a closed element.
<code>mdlElmdscr_freeAll</code>	free all memory allocated for the element descriptor chain.
<code>mdlElmdscr_convertTo2D</code>	convert a 3D element descriptor to a 2D element descriptor.

Function	Used to
<code>mdlElmdscr_convertTo3D</code>	convert a 2D element descriptor to a 3D element descriptor.
<code>mdlElmdscr_fromCompoundElement</code>	converts a compound element to a set of simple (pre-MicroStation 4.0 compatible) elements.
<code>mdlElmdscr_isClosed</code>	query whether element is closed.
<code>mdlElmdscr_isOpen</code>	query whether element is open.
<code>mdlElmdscr_isGroupedHole</code>	query whether element is grouped hole.
<code>mdlElmdscr_distanceAtPoint</code>	get distance along element at point.
<code>mdlElmdscr_pointAtDistance</code>	get point at distance along element.
<code>mdlElmdscr_open</code>	convert a closed element to an open element.
<code>mdlElmdscr_close</code>	convert an open element to a closed element.
<code>mdlElmdscr_stripAttributes</code>	strip all attribute data from elements in an element descriptor.
<code>mdlElmdscr_appendAttributes</code>	append attribute data to elements in an element descriptor.
<code>mdlElmdscr_extractAttributes</code>	extract attribute data from first element in an element descriptor.
<code>mdlElmdscr_setProperties</code>	perform same function as <code>mdlElement_setProperties</code> but on an element descriptor.
<code>mdlElmdscr_extractNormal</code>	get the normal vector to a planar element.
<code>mdlElmdscr_displayToWindow</code>	display an element in a window.
<code>mdlElmdscr_extendedDisplayToWindow</code>	display rendered images within a window.
<code>mdlElmdscr_copyParallel</code>	create a copy of an element descriptor that is offset from the original element.
<code>mdlProject_tangent</code>	project point to element along tangent.
<code>mdlProject_perpendicular</code>	project point to element along perpendicular.

Example

See `elemdscr.mc` and `doc.mc`.

mdlElmdscr_read, mdlElmdscr_readToMaster

```

#include <mselems.h>

ULong mdlElmdscr_read
(
  MSElementDescr  **elemDscrPP, /* <= element descriptor */
  ULong            filePos,       /* => file pos to read from */
  int              fileNum,       /* => file number */
  int              expandSharedCells, /* => for shared cells */
  ULong            *readFilePos   /* <= actual file pos */
);

ULong mdlElmdscr_readToMaster
(
  MSElementDescr  **elemDscrPP, /* <= element descriptor */
  ULong            filePos,       /* => file pos to read from */
  int              fileNum,       /* => file number */
  int              expandSharedCells, /* => for shared cells */
  ULong            *readFilePos   /* <= actual file pos */
);

```

Description mdlElmdscr_read reads the element at address *filePos* in file *fileNum* and creates a new element descriptor. A pointer to the element descriptor is returned in *elemDscrPP*. mdlElmdscr_readToMaster is similar, but also transforms the elements in the element descriptor to the master file coordinate system if *fileNum* indicates a reference file (*fileNum* > 0). If the element is from the master file (*fileNum* = 0), mdlElmdscr_readToMaster is equivalent to mdlElmdscr_read.

If the element at address *filePos* is a complex header, these functions read and allocate memory for the header and all of its components.

If the element is a shared cell instance:

Value of <i>expandSharedCells</i> <i>elemDscrPP</i> contains <i>Is</i>	
0	Only the shared cell instance element.
1	The expanded shared cell definition as it was stored.
2	The expanded shared cell definition translated to design file coordinates.



mdlElmdscr_read and mdlElmdscr_readToMaster never return deleted elements; they are skipped. Therefore, the file position for the element descriptor *elemDscrPP* can differ from *filePos* if *filePos* points to a deleted element. The actual file position from which the elements that comprise the element descriptor were read is returned in *readFilePos*. If the actual file position is not needed, pass NULL for *readFilePos*.



Applications must free the memory pointed to by *elemDscrPP* using `mdlElmdscr_freeAll` when they no longer need this memory.

Applications should almost always set *expandSharedCells* to `FALSE`.

Returns `mdlElmdscr_read` and `mdlElmdscr_readToMaster` return the file position of the next element in the file. If the read fails (such as at the end of file), they return 0 for the file position and *elemDscrPP* is not valid. `mdlErrno` is set to the specific error cause.

See Also `mdlElmdscr_freeAll`.

mdlElmdscr_add ,mdlElmdscr_append

```
#include <mselems.h>

ULong mdlElmdscr_add
(
  MSElementDescr    *elemDscrP      /* => added to file */
);
ULong mdlElmdscr_append
(
  MSElementDescr    *elemDscrP      /* => appended to file */
);
```

Description The `mdlElmdscr_add` function adds the *new* element(s) contained in the element descriptor pointed to by *elemDscrP* to the design file. You should use `mdlElmdscr_add` when creating new elements. Before writing the element to the file, `mdlElmdscr_add` sets the properties bits in the element header to not locked, new element, and not modified.

The `mdlElmdscr_append` function adds the *modified* element(s) contained in the element descriptor pointed to by *elemDscrP* to the design file. You should use `mdlElmdscr_append` if you are modifying an element and changing its size. The `mdlElmdscr_rewrite` function is also useful for this purpose, but requires that the old file position be known.



The `mdlElmdscr_add` and `mdlElmdscr_append` functions validate the element descriptor before adding it to the file.

MicroStation remembers the `mdlElmdscr_add` and `mdlElmdscr_append` functions, so the user can undo them.

Returns The `mdlElmdscr_add` and `mdlElmdscr_append` functions return the file position of the first element added to the design file. If an error occurs, the file position is set to zero and the global variable `mdlErrno` is set to the specific error cause. Possible values for `mdlErrno` are `MDLERR_READONLY`, `MDLERR_DISKFULL`, `MDLERR_WRITEINHIBIT`, `MDLERR_BADELEMENT` and `MDLERR_WRITEFAILED`.

See Also `mdlElmdscr_rewrite`, `mdlElement_add`, `mdlElement_append`.

mdlElmdscr_undoableDelete

```
#include <mselems.h>

int mdlElmdscr_undoableDelete
(
  MSElementDescr   *elemDscrP,    /* => element descr to delete */
  ULong             filePos,        /* => file position */
  int                display        /* => TRUE = erase from windows */
);
```

Description The `mdlElmdscr_undoableDelete` function deletes the element(s) pointed to by *elemDscrP* at file position *filePos*. MicroStation needs the element(s) to save in the undo buffer. If the element descriptor does not exist, pass NULL for *elemDscrP* and MicroStation will re-read the elements from the cache. Doing so adds some overhead to `mdlElmdscr_undoableDelete`, so always pass the element descriptor if it exists.

If *display* is TRUE, MicroStation erases the elements from the screen as it deletes them. Otherwise, it does not erase them.



MicroStation remembers the `mdlElmdscr_undoableDelete` function, so the user can undo it.

Returns If the element is deleted, `mdlElmdscr_undoableDelete` returns SUCCESS. If it fails, it sets `mdlErrno` and returns one of the following: MDLERR_READONLY, MDLERR_WRITEINHIBIT, MDLERR_BADELEMENT or MDLERR_MODIFYCOMPLEX.

See Also `mdlElement_undoableDelete`.

mdlElmdscr_transform

```
# include <mselems.h>

int mdlElmdscr_transform
(
  MSElementDescr   *elemDscrP,    /* <=> element descriptor */
  Transform          *tMatrix       /* => transformation matrix */
);
```

Description The `mdlElmdscr_transform` function transforms the element descriptor pointed to by *elemDscrP* by the transformation matrix *tMatrix*. The transformation matrix relates to the current coordinate system if one exists.

Returns If the element is deleted, `mdlElmdscr_transform` returns SUCCESS. If it fails, it returns a non-zero value.

See Also `mdlElement_transform`, `mdlElmdscr_operation`.

mdlElmdscr_rewrite

```
#include <mselems.h>

ULong mdlElmdscr_rewrite
(
  MSElementDescr    *newElemDscrP,    /* => new elements */
  MSElementDescr    *oldElemDscrP,    /* => original elements */
  ULong              filePos           /* => file position */
);
```

Description The `mdlElmdscr_rewrite` function overwrites the existing MicroStation element(s) pointed to by *oldElemDscrP* with the new element(s) pointed to by *newElemDscrP* at file position *filePos*.

If the sizes of the two element descriptors differ, MicroStation deletes the old one and appends the new one to the end of the file. Otherwise, it overwrites the old element(s) in the same position.

MicroStation needs the old element(s) to save in the undo buffer. If a copy of the old element descriptor does not exist, pass `NULL` for *oldElemDscrP* and MicroStation will re-read the old element(s) from the cache. Passing `NULL` for *oldElemDscrP* adds some overhead to `mdlElmdscr_rewrite`, so always pass the old element descriptor if it exists.



MicroStation remembers the `mdlElmdscr_rewrite` function, so the user can undo it.

Returns The `mdlElmdscr_rewrite` function returns the file position of the element added to the design file. If the two elements are the same size, this file position is the same as *filePos*.

If an error occurs, the file position is set to zero and the global variable `mdlErrno` is set to the specific error cause. Possible values for `mdlErrno` are `MDLERR_READONLY`, `MDLERR_DISKFULL`, `MDLERR_WRITEINHIBIT`, `MDLERR_BADELEMENT` and `MDLERR_WRITEFAILED`.

See Also `mdlElmdscr_add`, `mdlElmdscr_append`.

mdlElmdscr_display, mdlElmdscr_displayFromFile

```

#include <mselems.h>

void mdlElmdscr_display
(
  MSElementDescr   *elemDscrP,    /* => elements to display */
  int                file,          /* => file for elemDscrP */
  int                drawMode       /* => NORMALDRAW, ERASE, etc. */
);

int mdlElmdscr_displayFromFile
(
  ULong             filePos,        /* => file position */
  int                file,          /* => file number */
  MSElement         *element,      /* => element (or NULL) */
  int                drawMode       /* => NORMALDRAW, ERASE, etc. */
);

```

Description The `mdlElmdscr_display` function displays the element descriptor pointed to by *elemDscrP* in all active views.

file determines the display transformation and clipping to be applied to the element(s) as they are drawn.

The `mdlElmdscr_displayFromFile` function is similar, but operates on elements in the design file rather than from an element descriptor supplied by the application. *element* is an optional parameter that points to the element at address *filePos* from file *file*. If the element does not exist, pass NULL. `mdlElmdscr_displayFromFile` uses the following logic internally:

```

if (element is non-NULL and points to a simple element)
{
  mdlElement_display(element, ...)
}
else
{
  mdlElmdscr_read(..., filePos, file, ...)
  mdlElmdscr_display(...)
  mdlElmdscr_freeAll(...)
}

```

drawMode determines how MicroStation displays the element(s). Possible values for *drawMode* are as follows:

<i>drawMode</i>	<i>drawMode</i> field meaning
NORMALDRAW	Draw the element(s) in its normal color.
ERASE	Erase the element(s).
HILITE	Draw the element(s) in the current highlight color.

Returns The `mdlElmdscr_display` function returns no value. `mdlElmdscr_displayFromFile` function returns `SUCCESS` if the element is read and displayed and `ERROR` if *filePos* is invalid.

See Also `mdlElmdscr_displaySingle`, `mdlElement_display`,
`mdlElmdscr_displayInSelectedViews`, `mdlElmdscr_displayFromFileViews`.

mdlElmdscr_displayInSelectedViews, mdlElmdscr_displayFromFileViews

```
void mdlElmdscr_displayInSelectedViews
(
  MSElementDescr *elemDP, /* => ptr to elemDscr displayed */
  int             fileNum,  /* => file for elemDP */
  int             drawMode, /* => drawing mode */
  int             viewMask  /* => one bit per view */
);

int mdlElmdscr_displayFromFileViews
(
  ULong          filePos, /* => file position of element */
  int            fileNum, /* => file number */
  MSElement     *el,     /* => optional */
  int            drawMode, /* => drawing mode */
  int            viewMask  /* => one bit per view */
);
```

Description The `mdlElmdscr_displayInSelectedViews` function is identical to `mdlElmdscr_display`, except that the *viewMask* argument determines the views in which element descriptor is displayed (see `mdlElement_displayInSelectedViews` for a discussion of *viewMask*).

The `mdlElmdscr_displayFromFileViews` function is identical to `mdlElmdscr_displayFromFile`, except that the *viewMask* argument determines the views in which element descriptor is displayed (see `mdlElement_displayInSelectedViews` for a discussion of *viewMask*).

Returns `mdlElmdscr_displayInSelectedViews` is of type `void`, it does not return a value.
`mdlElmdscr_displayFromFileViews` returns `SUCCESS` if the element is displayed and `ERROR` if *filePos* is invalid.

See Also `mdlElmdscr_display`, `mdlElmdscr_displayFromFile`.

mdlElmdscr_displaySingle

```
void mdlElmdscr_displaySingle
(
  MSElementDescr *elemDescr, /* => element to display */
  int             file,        /* => file number */
  int             drawMode     /* => display mode */
);
```

Description The mdlElmdscr_displaySingle function is identical to mdlElmdscr_display except that it displays a single element descriptor and will not display the elements pointed to by *elemDescr->next*.

See Also mdlElmdscr_display.

mdlElmdscr_stroke

```
#include <mselems.h>

int mdlElmdscr_stroke
(
    Dpoint3d          **points,          /* <= vectors for elmDscrP */
    int               *numPoints,        /* <= number of pts in points */
    MSElementDescr   *elmDscrP,         /* => elements */
    double            tolerance          /* => stroking tolerance */
);
```

Description The mdlElmdscr_stroke function strokes the element descriptor pointed to by *elmDscrP* into vectors in an array pointed to by *points*. (This array should be the address of a pointer to a Dpoint3d). MicroStation allocates memory for the vectors and returns the starting address in *points*.

The application programmer must free this memory when finished with it. The number of points in the array is returned in *numPoints*.

tolerance is the maximum distance between the actual curve and the approximating vectors for curved elements.

Returns The mdlElmdscr_stroke function returns SUCCESS if the *elmDscrP* is stroked and *points* and *numPoints* are valid. It returns ERROR if *elmDscrP* is not valid. Valid element types are LINE_ELM, LINE_STRING_ELM, SHAPE_ELM, ARC_ELM, ELLIPSE_ELM, CURVE_ELM, TEXT_ELM, CMPLX_STRING_ELM and CMPLX_SHAPE_ELM.

See Also mdlElement_stroke.

mdlElmdscr_operation

```
#include <mselems.h>

int mdlElmdscr_operation
(
    MSElementDescr   *elmDscrP,         /* => element descriptor */
    MdlFunctionP      elmFunc,           /* => function to call */
    void              *params,           /* => user defined */
    int               opFlags            /* => option flags */
);
```

Description Element descriptors commonly operate on each element in a complex element. Since many complex elements in MicroStation can be nested, this operation requires a recursive subroutine of the following form:

```
int myFunction(pointer to element descriptor)
```

```

{
    while (more elements are in the descriptor)
    {
        perform required operation
        if (this is a header element)
        {
            myFunction(first component element)
        }
        step to next component element
    }
}

```

While this procedure is basically straightforward, it can sometimes be inconvenient. The `mdlElmdscr_operation` function is an implementation of the above algorithm that calls your function, *elmFunc*, for each element in the descriptor.

The `mdlElmdscr_operation` function does not use *params* directly, but *params* is passed as an argument to *elmFunc* each time it is called. *params* is used for a pointer to a structure containing information needed by *elmFunc*. However, it can also be used as a pointer to a returned status or another value that will fit in an `int`.

opFlags determines the times at which *elmFunc* is called for the various elements in an element descriptor. The following values are possible:

opFlag	Meaning
ELMD_PRE_HDR	For outermost header elements, call <i>elmFunc</i> for this element before it is called for component elements.
ELMD_PRE_NESTEDHDR	For nested header elements, call <i>elmFunc</i> for this element before it is called for component elements.
ELMD_POST_HDR	For outermost header elements call <i>elmFunc</i> for this element after it is called for all component elements.
ELMD_POST_NESTEDHDR	For nested header elements, call <i>elmFunc</i> for this element after it is called for all component elements.
ELMD_ELEMENT	Call <i>elmFunc</i> for component elements.
ELMD_ALL_ONCE	Call <i>elmFunc</i> only once for each element. (defined as (ELMD_ELEMENT ELMD_PRE_HDR ELMD_PRE_NESTEDHDR)).
ELMD_HDRS_ONCE	Call <i>elmFunc</i> once for header elements only (defined as (ELMD_PRE_HDR ELMD_PRE_NESTEDHDR)).

elmFunc should expect the following parameters:

```
int elmFunc
(
MSElement      *element,      /* <=> element to act upon */
void            *params,       /* <=> passed from orig call */
int             operation,     /* => why you were called */
ULong          offset,        /* => offset from header */
MSElementDescr *elemDscrP     /* => element descr */
);
```

element points to the current element.

params is passed unchanged from the original call to `mdlElmdscr_operation`.

operation indicates why *elmFunc* was called (one of the values of *opFlag* above).

offset is the file position of the current element relative to the outermost header element. This argument is useful for identifying particular component elements.

elmDscrP points to the element descriptor containing the current element. This argument is rarely needed.



`mdlElmdscr_operation` extracts information from an element descriptor. *elmFunc* should not modify the elements passed to it. *elmFunc* can modify elements through `mdlModify_elementDescr`.

Returns If the *elmFunc* function returns a value other than `SUCCESS`, `mdlElmdscr_operation` aborts and returns that value to its caller.

See Also `mdlModify_elementDescr`.

mdlElmdscr_igdsSize

```
#include <mselems.h>

ULong mdlElmdscr_igdsSize
(
MSElementDescr *elmDscrP      /* => element descriptor */
);
```

Description The `mdlElmdscr_igdsSize` function returns the total file size of the elements contained in the element descriptor pointed to by *elmDscrP*.

The `mdlElmdscr_igdsSize` function is a perfect application of the `mdlElmdscr_operation` function. The actual implementation of `mdlElmdscr_igdsSize` is as follows:

```
Public ULong mdlElmdscr_igdsSize(elmDscrP)
MSElementDescr *elmDscrP;
```

```

{
    ULONG totalSize=0L;

    mdlElmdscr_operation(elmDscrP, accumulateSize, &totalSize,
                        ELMD_ALL_ONCE);

    return totalSize;
}

Private int accumulateSize(element, totalSize)
MSElement *element;
ULONG *totalSize;
{
    *totalSize += mdlElement_igdsSize(element, 0);
    return SUCCESS;
}

```

Returns mdlElmdscr_igdsSize returns the total size of the elements in *elmDscrP*.

See Also mdlElement_igdsSize, mdlElmdscr_operation.

mdlElmdscr_singleIgdsSize

```

ULONG mdlElmdscr_singleIgdsSize
(
    MSElementDescr    *elemDescr,
    int                file
);

```

Description The mdlElmdscr_singleIgdsSize function is identical to mdlElmdscr_igdsSize except that it returns the size of a single element descriptor and will not include the size of elements pointed to by *elemDescr->next*.

See Also mdlElmdscr_igdsSize.

mdlElmdscr_markElement

```

#include <mselems.h>

void mdlElmdscr_markElement
(
    MSElementDescr    *elmDscrP, /* <=> element descriptor */
    int                fileNum,    /* => file number */
    long               value,      /* => value to put in userData */
    ULONG              offset,     /* => offset of desired element */
);

```

Description The `mdlElmdscr_markElement` function marks an element in an element descriptor for later processing. It sets the `userData1` field in the header structure for the element at offset *offset* from the first element to the value specified by *value*.

fileNum is the file number with which the element descriptor is associated. It is required so MicroStation can determine whether the file is 2D or 3D and, therefore, derive the element sizes.

This function is typically used when an element's offset from its header (*tcb->componentOffset*) is known and will be marked for a later operation.

Returns The `mdlElmdscr_markElement` function returns no value.

mdlElmdscr_duplicate, mdlElmdscr_duplicateSingle

```
#include <mselems.h>

int mdlElmdscr_duplicate
(
  MSElementDescr    **newDscrPP,    /* <= new element descriptor */
  MSElementDescr    *oldDscrP      /* => original element descr */
);

int mdlElmdscr_duplicateSingle
(
  MSElementDescr    **newDscrPP,    /* <= new element descriptor */
  MSElementDescr    *oldDscrP      /* => original element descr */
);
```

Description Since MicroStation must always allocate element descriptors, the `mdlElmdscr_duplicate` function creates a copy of an element descriptor. This function copies the element descriptor pointed to by *oldDscrP*, allocates a new element descriptor, and returns the new descriptor's address in *newDscrPP*.

The `mdlElmdscr_duplicateSingle` function is identical to `mdlElmdscr_duplicate` except that it duplicates a single element descriptor and will not duplicate the elements pointed to by *oldDscr->next*.

Both the `userData1` and `userData2` fields in the element descriptor and the user data at the end of the elements are copied to the new element descriptor.



Element descriptors should be duplicated with discretion since they can require significant amounts of memory. Applications must remember to free both the old and new descriptors when finished with them.

Returns `mdlElmdscr_duplicate` returns `SUCCESS` if the element descriptor is copied. It returns `MDLERR_INSMEMORY` if there is not enough memory to allocate a new descriptor.

See Also `mdlElmdscr_new`.

mdlElmdscr_validate

```
# include <mselems.h>

void mdlElmdscr_validate
(
MSElementDescr    *elmDscrP,      /* <=> element descriptor */
int                 fileNum        /* => file number */
);
```

Description The `mdlElmdscr_validate` function sets the information in the complex header elements contained in *elmDscrP* based on the component elements in the descriptor. It sets the following fields:

Element types	Validated fields
CMPLX_STRING_ELM CMPLX_SHAPE_ELM SURFACE_ELM SOLID_ELM	Header range, number of elements, total words in description.
TEXT_NODE_ELM	Header range, number of text strings, maximum length of text strings, total words in description.
CELL_HEADER_ELM CELL_LIB_ELM SHAREDCELL_DEF_ELM	Header range, class mask, level mask, total words in description.
BSPLINE_CURVE_ELM	Header range, number of poles, total words in description.
BSPLINE_SURFACE_ELM	Header range, total words in description.
SHARED_CELL_ELM	Range, level mask.
DIMENSION_ELM MULTILINE_ELM	Range.



All MDL built-in functions that display element descriptors or add element descriptors to the design file call `mdlElmdscr_validate` internally. Applications rarely need to call this function directly.

An *isValid* flag is in the element descriptor structure header. `mdlElmdscr_validate` sets this flag to `TRUE` when the function is complete. The function does nothing if the flag is `TRUE` before the function is called. When an application modifies component information directly, it must set *isValid* to `FALSE`.

Returns The `mdlElmdscr_validate` function returns no value.

mdlElmdscr_reverse

```
int mdlElmdscr_reverse
(
MSElementDescr    **outEdPP,      /* <= reversed element */
MSElementDescr    *inEdP,        /* => element to be reversed */
int                 fileNum        /* => file number */
);
```

Description The `mdlElmdscr_reverse` function reverses the direction of *inEdP* and returns the reversed element in *outEdPP*. Valid element types include lines, line strings, shapes, arcs, curves, B-spline curves, complex chains and complex shapes.

fileNum is a the file number for the element. This is used only to resolve point associations and in most cases 0 (MASTERFILE) can be used for this parameter.

Returns `mdlElmdscr_reverse` returns `SUCCESS` if the element is successfully reversed and `MDLERR_BADTYPE` if the element type is invalid.



This function is included in the object library `mdl.lib.ml`. This library must be linked into any application calling this function.

See Also `mdlElmdscr_reverseNormal`.

mdlElmdscr_reverseNormal

```
int mdlElmdscr_reverseNormal
(
MSElementDescr    **outEdPP,      /* <= reversed element */
MSElementDescr    *inEdP,        /* => element to reverse */
int                 fileNum        /* => file number */
);
```

Description The `mdlElmdscr_reverseNormal` function reverses the normal direction for a surface, solid or closed element.

outEdPP points to the address of the reversed element.

inEdP is the address of the input element descriptor.

fileNum is a the file number for the element. This is used only to resolve point associations and in most cases 0 (MASTERFILE) can be used for this parameter.

Returns `mdlElmdscr_reverseNormal` returns `SUCCESS` if the element is reversed successfully and an appropriate error code otherwise.

See Also `mdlElmdscr_reverse`.

mdlElmdscr_partialDelete

```

int mdlElmdscr_partialDelete
(
MSElementDescr    **outEdPP1,    /* <= 1st partial elm (or NULL) */
MSElementDescr    **outEdPP2,    /* <= 2nd partial elm (or NULL) */
MSElementDescr    *inEdP,        /* => input element */
Dpoint3d           *point1,        /* => first point */
Dpoint3d           *point2,        /* => 2nd point (or NULL) */
Dpoint3d           *point3,        /* => 3rd point */
int                 view            /* => view number for projection */
);

```

Description The `mdlElmdscr_partialDelete` function returns the portions of *inEdP* that are not between the projection of *point1* and *point3* on the element. This is the same process performed by MicroStation's DELETE PARTIAL command. If *inEdP* is closed, *point2* is used to determine the portion of the element to be deleted. For closed elements, or open elements where the deleted portion includes the beginning or end of *inEdP*, only a single element is returned and *outEdPP2* is set to NULL. If the entire input element is deleted, *outEdPP1* is set to NULL as well.

view is used to perform the projection of the input points to the element.

Returns `mdlElmdscr_partialDelete` returns SUCCESS if the element is successfully partially deleted.

mdlElmdscr_extractEndPoints

```

int mdlElmdscr_extractEndPoints
(
Dpoint3d           *startP,        /* <= element start point */
Dpoint3d           *startTangentP, /* <= start tangent */
Dpoint3d           *endP,          /* <= element endpoint */
Dpoint3d           *endTangentP,   /* <= end tangent */
MSElementDescr    *edP,          /* => element descr. */
int                 fileNum        /* => file number */
);

```

Description `mdlElmdscr_extractEndPoints` returns the start and end points (in *start* and *end*) for the open element, *edP*. Valid element types include lines, linestrings, arcs, curves, B-spline curves, multilines, complex chains and complex shapes.

startTangentP and *endTangentP* are the starting and ending tangents of the curve. Pass NULL for these arguments if you do not need or want these values.

fileNum is used only if the input element is a multiline.

Returns `mdlElmdscr_extractEndPoints` returns SUCCESS if the element is successfully reversed and MDLERR_BADTYPE if the element type is invalid.



This function is included in the object library mdl.lib.ml. This library must be linked into any application calling this function.

mdlElmdscr_new

```
#include <mselems.h>

int mdlElmdscr_new
(
  MSElementDescr    **elmDscrPP,    /* <= pointer to element descr */
  MSElementDescr    *hdrElmDscrP,   /* => ptr to header elm descr */
  MSElement          *element       /* => existing element */
);
```

Description The mdlElmdscr_new function allocates a new element descriptor from an existing MicroStation element, *element*. It returns the element descriptor address in *elmDscrPP*.

hdrElmDscrP points to a parent element descriptor, to which this element descriptor is attached. If *hdrElmDscrP* is non-NULL, mdlElmdscr_new sets the myHeader field in the new element descriptor structure to that value. It also sets the complex bit in the copy of *element*. *hdrElmDscrP* should be set to NULL unless the MDL functions that add or insert elements in an element descriptor cannot be used.

Returns The mdlElmdscr_new function returns SUCCESS if a new element descriptor is allocated and *elmDscrPP* is valid. It returns MDLERR_BADELEMENT if *element* is not a valid MicroStation element. It returns MDLERR_INSFMEMORY if it cannot allocate memory for the element descriptor.

See Also mdlElmdscr_appendElement, mdlElmdscr_insertElement, mdlElmdscr_replaceElement.

mdlElmdscr_appendElement, mdlElmdscr_insertElement

```
#include <mselems.h>

MSElementDescr *mdlElmdscr_appendElement
(
  MSElementDescr    *existingElmDscrP, /* <=> element descr */
  MSElement          *element          /* => elm to append */
);

int mdlElmdscr_insertElement
(
  MSElementDescr    *existingElmDscrP, /* <=> element descr */
  MSElement          *element          /* => elm to insert */
);
```

Description `mdlElmdscr_appendElement` allocates a new element descriptor from an existing MicroStation element, *element*. It appends the new descriptor to the end of an existing element descriptor, *existingElmDscrP*.

`mdlElmdscr_insertElement` allocates a new element descriptor from an existing MicroStation element, *element*. It inserts the new descriptor in an existing element descriptor chain before the element descriptor pointed to by *existingElmDscrP*.

Returns `mdlElmdscr_appendElement` returns a pointer to the new element descriptor allocated for *element*. If a failure occurs, the pointer is set to NULL.

`mdlElmdscr_insertElement` returns SUCCESS if the element is inserted in the chain and MDLERR_NOTCMPLXHDR if *existingElmDscrP* does not contain a complex header.

See Also `mdlElmdscr_removeElement`, `mdlElmdscr_replaceElement`, `mdlElmdscr_new`, `mdlElmdscr_addToChain`, `mdlElmdscr_appendDscr`.

mdlElmdscr_removeElement, mdlElmdscr_replaceElement

```
#include <mselems.h>

MSElementDescr *mdlElmdscr_removeElement
(
    MSElementDescr    *existingElmDscrP      /* <=> element descr */
);

int mdlElmdscr_replaceElement
(
    MSElementDescr    **existingElmDscrPP,    /* <=> element descr */
    MSElement          *element               /* => elm to replace */
);
```

Description `mdlElmdscr_removeElement` removes an element from an element descriptor chain. It frees the memory allocated for the element and returns a pointer to the next element descriptor in the chain.

`mdlElmdscr_replaceElement` allocates a new element descriptor from an existing MicroStation element, *element*. It replaces an existing element descriptor in an element descriptor chain.

Returns `mdlElmdscr_removeElement` returns a pointer to the next element descriptor in the chain.

`mdlElmdscr_replaceElement` returns SUCCESS if the element is replaced in the chain and MDLERR_INSFMEMORY if a new element descriptor cannot be allocated.

See Also `mdlElmdscr_appendElement`, `mdlElmdscr_insertElement`, `mdlElmdscr_new`, `mdlElmdscr_addToChain`, `mdlElmdscr_appendDscr`, `mdlElmdscr_replaceDscr`.

mdlElmdscr_addToChain, mdlElmdscr_appendDscr

```
#include <mselems.h>

void mdlElmdscr_addToChain
(
MSElementDescr    *chainDscrP,    /* <=> element descr */
MSElementDescr    *elmDscrP      /* => element descr to add */
);

int mdlElmdscr_appendDscr
(
MSElementDescr    *existingElmDscrP, /* <=> element descr */
MSElementDescr    *elmDscrP        /* => element descr to add */
);
```

Description The `mdlElmdscr_addToChain` function adds the element descriptor chain pointed to by *elmDscrP* to the end of the element descriptor pointed to by *chainDscrP*. It can create a chain of non-complex elements in *chainDscrP*.

The `mdlElmdscr_appendDscr` function adds the element descriptor chain pointed to by *elmDscrP* to the end of the element descriptor pointed to by *existingElmDscrP*. The difference between `mdlElmdscr_appendDscr` and `mdlElmdscr_addToChain` is that `mdlElmdscr_appendDscr` requires the element descriptor to hold a single complex element and `mdlElmdscr_addToChain` does not.



These functions essentially merge the two element descriptors passed to them. Therefore, *elmDscrP* should not be referenced after either of these functions is called. (Thus, it should not be freed).

Returns The `mdlElmdscr_addToChain` function returns no value.

The `mdlElmdscr_appendDscr` function returns `SUCCESS` if *newDscrP* is appended to *existingElmDscrP*. It returns `MDLERR_NOTCMPLXHDR` if the *existingElmDscrP* does not hold a complex header.

See Also `mdlElmdscr_appendElement`, `mdlElmdscr_replaceDscr`.

mdlElmdscr_replaceDscr

```
void mdlElmdscr_replaceDscr
(
MSElementDescr    **existingDescrPP, /* <=> element descr */
MSElementDescr    *newDscrP        /* => element descr to replace with */
);
```

Description `mdlElmdscr_replaceDscr` replaces the element descriptor pointed to by the *existingDescrPP* parameter with the new element descriptor pointed to by *newDscrP*. If the element descriptor identified by *existingDescrPP* is part of a complex element, the new descriptor becomes part of that complex element. The

memory allocated to the element descriptor being replaced is freed by MicroStation and the memory allocated to *newDescrP* is transferred to the element descriptor which contained the one being replaced. The application program should not free the memory associated with *newDescrP* after this function is called.

existingDescrPP is a pointer to the address of the element descriptor to be replaced by the element descriptor pointed to by the *newDescrP* field.

Returns `mdlElmdscr_replaceDscr` is of type `void` and does not return a status.

See Also `mdlElmdscr_replaceElement`, `mdlElmdscr_appendDscr`.

mdlElmdscr_createFromVertices [mdl.lib.mli]

```
int mdlElmdscr_createFromVertices
(
  MSElementDescr  **edPP,    /* <= line string or complex chain */
  MSElement       *in,       /* => template element (or NULL) */
  Dpoint3d         *pointP,   /* => vertices */
  int              numPoints,  /* => number of vertices */
  boolean          closed,     /* => TRUE for closed */
  boolean          fillMode    /* => fill mode (if closed) */
);
```

Description The `mdlElmdscr_createFromVertices` function creates a line string or complex chain if *closed* is zero and a shape or complex shape if *closed* is non-zero. A linestring or shape is created if *numPoints* is less than 101, otherwise the appropriate complex element is created. This routine is convenient for creating elements when a the number of points may exceed 101.

edPP points to the address of an element descriptor containing the created element.

If *in* is `NULL`, the display parameters for the created element are taken from the active MicroStation parameters when the function is called. Otherwise the display parameter from *in* are used. All attribute information from *in* is retained in *out*.

pointP points to an array of *numPoints* vertices.

If *closed* is non-zero, *fillMode* determines whether the created shape is filled. See `mdlShape_create` for possible values for *fillmode*.

Returns `mdlElmdscr_createFromVertices` returns `SUCCESS` if the element is created successfully and an appropriate error code otherwise.

See Also `mdlLineString_create`, `mdlShape_create`.

mdlElmdscr_generatePartial

```
int mdlElmdscr_generatePartial
(
MSElementDescr  **outEdPP,      /* <= output (partial) element */
MSElementDescr  *elmDescr,      /* => input element */
double           t1,              /* => start parameter */
double           t2,              /* => end parameter */
boolean          splineParameters /* => t1 and t2 are B-spline params */
);
```

Description The `mdlElmdscr_generatePartial` function generates the portion of the element descriptor *elmDescr* that is between *t1* and *t2* and returns this portion in *outEdPP*. The parameters correspond to the parameterization of the element after conversion to a B-spline curve entity with the `mdlBspline_convertToCurve` function. The *splineParameters* argument should always be set to `TRUE`.

Returns `mdlElmdscr_generatePartial` returns `SUCCESS` if the partial element is generated successfully and an appropriate error status otherwise.

See Also `mdlBspline_convertToCurve`.

mdlElmdscr_createShapeWithHoles

```
int mdlElmdscr_createShapeWithHoles
(
MSElementDescr  **outEdPP, /* <= shape with holes */
MSElementDescr  *solidEdP, /* => outer shape (included in *outEdPP) */
MSElementDescr  *holeEdP  /* => hole edP (chain) */
);
```

Description The `mdlElmdscr_createShapeWithHoles` function creates an orphan cell entity, *outEdPP*, that associates the solid element, *solidEdP* with the hole entity(s) in *holeEdP*. The hole elements should be planar with and contained within the solid. The solid and hole descriptors become a part of the output element descriptor, they are not copied.

The holes will be recognized as being associated with the solid by MicroStation's rendering, hidden line removal, filled element display and translators.

Returns `mdlElmdscr_createShapeWithHoles` returns `SUCCESS` if a grouped hole is created and an appropriate error code otherwise.

See Also `mdlElmdscr_isGroupedHole`.

mdlElmdscr_hiddenLineRemoval

```

int mdlElmdscr_hiddenLineRemoval
(
    MdlFunctionP preFunction,          /* => called before processing elems */
    MdlFunctionP visibleFunction,      /* => called for visible elms */
    MdlFunctionP invisibleFunction,    /* => called for invisible elms */
    MSElementDescr *inEdP,           /* => input elements */
    int viewNumber,                   /* => view (0-7) */
    double tolerance,                 /* => tolerance */
    boolean outputTo3D,               /* => TRUE=create 3D outputfile */
    boolean ruleLines,                /* => TRUE=include ruleLines */
    boolean calculateIntersections    /* => TRUE=surf-surf intersects */
);

```

Description The `mdlElmdscr_hiddenLineRemoval` function is identical to the `mdlView_hiddenLineRemoval` routine except that it processes the element(s) in the element descriptor *inEdP* rather than reading a design file(s).

Returns `mdlElmdscr_hiddenLineRemoval` returns `SUCCESS` if processing is completed successfully and an appropriate error code otherwise.

See Also `mdlView_hiddenLineRemoval`, Hidden Line Removal Functions

mdlElmdscr_addFill, mdlElmdscr_stripFill

```

int mdlElmdscr_addFill
(
    MSElementDescr **edPP             /* <=> element to modify */
);

int mdlElmdscr_stripFill
(
    MSElementDescr **edPP             /* <=> element to modify */
);

```

Description The `mdlElmdscr_addFill` function adds an attribute linkage to a closed element that causes the element to be displayed filled. The fill color is set to the same color as the elements color, but can be changed with the `mdlElement_setFillColor` function.

The `mdlElmdscr_stripFill` function removes the fill attribute linkage from an element.

edPP points to the address of the element descriptor to be modified. This must be a closed element (shape, complex shape, ellipse, grouped hole orphan cell or closed B-spline curve).

Returns `mdlElmdscr_addFill` returns `SUCCESS` if the element is modified successfully and an appropriate error code otherwise.

See Also `mdlElement_setFillColor`.

mdlElmdscr_freeAll

```
#include <mselems.h>

void mdlElmdscr_freeAll
(
  MSElementDescr    **elmDscrPP    /* <=> element descr to free */
);
```

Description The mdlElmdscr_freeAll function frees all memory allocated for the element descriptor chain pointed to by *elmDscrPP*. This function should be called only once for every element descriptor.



The mdlElmdscr_freeAll function sets *elmDscrPP* to NULL so that it cannot accidentally be freed twice.

Returns The mdlElmdscr_freeAll function returns no value.

mdlElmdscr_convertTo2D

```
#include <mselems.h>

int mdlElmdscr_convertTo2D
(
  MSElementDescr    **newElemDscr, /* <= new element descr */
  MSElementDescr    *oldElemDscr,  /* => original element descr */
  int                 view,          /* => view number */
  Transform           *trans,        /* => transformation matrix */
  int                 fileNum        /* File # (slot) of .dgn or cell lib. contain. elm */
);
```

Description The mdlElmdscr_convertTo2D function converts the 3D element descriptor pointed to by *oldElemDscr* to a 2D element descriptor. A pointer to the new 2D element descriptor is returned in *newElemDscr*.

The element's orientation in the resulting 2D design plane is defined by *view* and *trans*. *view* defines which view (0 to 7, where 0 = view 1, and 1 = view 2) establishes the final orientation of the converted elements. A value of -1 indicates that the top view is used.

trans allows an additional transformation to be applied to the element descriptor before the transformation defined by the chosen *view* parameter is applied.

Applications must free memory pointed to by *newElemDscr* using mdlElmdscr_freeAll when they are finished with it.

The following list of element types are not converted to 2D elements and are removed from the output element list:

- B-spline surface element headers
- Surface element headers

- Solid element headers
- Cone elements
- Reference file attachment definitions

Returns The `mdlElemDscr_convertTo2D` function returns `SUCCESS` when the elements convert successfully. If an error occurs, `ERROR` is returned and `mdlErrno` contains the reason for failure.

See Also `mdlElemDscr_convertTo3D`, `mdlElemDscr_freeAll`, `mdlElemDscr_new`.

mdlElemDscr_convertTo3D

```
#include <mselems.h>

int mdlElemDscr_convertTo3D
(
  MSElementDscr  **newElemDscr, /* <=  new element descr */
  MSElementDscr  *oldElemDscr, /* => original element descr */
  int             depthType,     /* => depth type */
  long            depth,         /* => depth */
  Transform       *trans,       /* => transformation matrix */
  int             fileNum       /* => File# (slot) of .dgn or cell lib contain elm */
);
```

Description The `mdlElemDscr_convertTo3D` function converts the 2D element descriptor pointed to by *oldElemDscr* to a 3D element descriptor. A pointer to the new 3D element descriptor is returned in *newElemDscr*.

The element depth (Z value) in the resulting 3D design plane is defined by the *depthType* and *depth* parameters. The *depthType* parameter defines the type of value to be used in defining each element's Z depth. The following values are allowed:

Value	Description
FIXEDDEPTH	All elements have the fixed Z depth specified by the <i>depth</i> parameter.
ELEMHIGH	Each element has the Z depth defined by the high value of the Z range in the element definition.
ELEMLOW	Each element has the Z depth defined by the low value of the Z range in the element definition.

The *trans* parameter allows a transformation to be applied to the element descriptor before each element is converted from 2D to 3D.

Applications must free the memory pointed to by *newElemDscr* using `mdlElemDscr_freeAll` when they are finished with it.

Reference file attachment definitions are not converted to 3D elements and are removed from the output element list.

Returns The mdlElmdscr_convertTo3D function returns SUCCESS when elements are converted successfully. If an error occurs, ERROR is returned and mdlErrno contains the reason for failure.

See Also mdlElmdscr_convertTo2D, mdlElmdscr_freeAll, mdlElmdscr_new.

mdlElmdscr_fromCompoundElement

```
int mdlElmdscr_fromCompoundElement
(
  MSElementDescr  **edPP,    /* <= output element(s) */
  MSElement       *elemP,    /* => input compound element */
  int              fileName,  /* => file number for element */
  ULong            graphicGroup, /* => graphic group for *edPP */
  boolean          transformToWorld, /* => for shared cell instances */
  boolean          expandNested /* => expand nested sh. cells */
);
```

Description The mdlElmdscr_fromCompoundElement function converts a compound element (multi-line, dimension, shared cell) to a set of simple (pre-MicroStation 4.0 compatible) elements.

edPP points to the address of an element descriptor that contains the simple elements representing the compound element, *elemP*.

fileName is the file number for *elemP*. It is used to resolve associative points.

graphicGroup is a graphic group number for the output elements. If this is nonzero, the elements will all receive this graphic group number and therefore will be a part of the same graphic group.

If the element is a shared cell and *transformToWorld* is TRUE, shared cell definitions are transformed to world coordinates. In most cases, this variable should be TRUE.

If the element is a shared cell instance and *expandNested* is TRUE, shared cell instances are replace with their definitions.

Returns mdlElmdscr_fromCompoundElement returns SUCCESS if the compound element is expanded successfully and MDLERR_BADELEMENT if the input element is not a compound element.

mdlElmdscr_isClosed, mdlElmdscr_isOpen, mdlElmdscr_isGroupedHole

```

boolean mdlElmdscr_isClosed
(
  MSElementDescr  *edP      /* => element to test */
);
boolean mdlElmdscr_isOpen
(
  MSElementDescr  *edP      /* => element to test */
);
boolean mdlElmdscr_isGroupedHole
(
  MSElementDescr  *edP      /* => element to test */
);

```

Description mdlElmdscr_isClosed queries if an element descriptor is a closed element; i.e., a shape, complex shape, ellipse or closed B-spline curve.

mdlElmdscr_isOpen queries if an element descriptor is an open element; i.e., a line, line string, curve, open B-Spline curve or complex chain.

mdlElmdscr_isGroupedHole queries if an element descriptor is a grouped hole element; i.e., an element that has a hole punched in it (such as can be created with mdlElmdscr_createShapeWithHoles).

edP is the element descriptor to query.

Returns mdlElmdscr_isClosed returns TRUE if the element descriptor is a closed element.
mdlElmdscr_isOpen returns TRUE if the element descriptor, *edP* is an open element.
mdlElmdscr_isGroupedHole returns TRUE if the element descriptor is a grouped hole element.

See Also mdlElmdscr_open, mdlElmdscr_close, mdlElmdscr_createShapeWithHoles.

mdlElmdscr_distanceAtPoint, mdlElmdscr_pointAtDistance

```

#include <mdl.h>
#include <mselems.h>

int mdlElmdscr_distanceAtPoint
(
    double          *distance,      /* <= distance along elm */
    Dpoint3d        *position,      /* <= point on element */
    Dpoint3d        *tangent,       /* <= tangent direction */
    MSElementDescr *edP,           /* => element descriptor */
    Dpoint3d        *inputPoint,    /* => input point */
    double          tolerance       /* => stroking tolerance */
);

int mdlElmdscr_pointAtDistance
(
    Dpoint3d        *position,      /* <= point on element */
    Dpoint3d        *tangent,       /* <= tangent direction */
    double          *distance,      /* => distance along elm */
    MSElementDescr *edP,           /* => element */
    double          tolerance       /* => stroking tolerance */
);

```

Description The mdlElmdscr_distanceAtPoint routine returns the distance along an element from the beginning of the element to the projection of *inputPoint* on the element. The position and tangent vector are also returned. The mdlElmdscr_pointAtDistance function returns the point and tangent vector at *distance* along the element. All input parameters are specified in the current coordinate system.

Returns The mdlElmdscr_pointAtDistance and mdlElmdscr_distanceAtPoint return SUCCESS if they function correctly and a non-zero error status otherwise.

mdlElmdscr_open, mdlElmdscr_close

```

int mdlElmdscr_open
(
    MSElementDescr **outEdPP,      /* <= open element descriptor */
    MSElementDescr *inEdP,         /* => element descriptor to open */
    int              fileNumber     /* => file number */
);

int mdlElmdscr_close
(
    MSElementDescr **outEdPP,      /* <= closed element descriptor */
    MSElementDescr *inEdP,         /* => element descriptor to close */
    int              fileNumber     /* => file number */
);

```

Description The `mdlElmdscr_open` function converts the closed element *inEdP* to an open element, *outEdPP*. This function will convert a shape to a line string, an ellipse to an arc, a complex shape to a complex chain etc.

The `mdlElmdscr_close` function converts the open element *inEdP* to a closed element, *outEdPP*. If the beginning and endpoints of the element are not identical, a segment connecting them is inserted. This function will convert a complex chain into a complex shape, a linstring into a shape, etc.

fileNumber is used only to resolve associative points, in most cases 0 (MASTERFILE) can be passed for this parameter.

Returns `mdlElmdscr_open` and `mdlElmdscr_close` return `SUCCESS` if the element is opened or closed successfully and an appropriate error code otherwise.

See Also `mdlElmdscr_isOpen`, `mdlElmdscr_isClosed`.

mdlElmdscr_stripAttributes

```
#include <mselems.h>

void mdlElmdscr_stripAttributes
(
  MSElementDescr    **elmdscrPP    /* <=> element descriptor to process */
);
```

Description The `mdlElmdscr_stripAttributes` function removes *all* of the attribute data on the elements contained in descriptor *elmdscrPP*. If there are complex elements contained in the element descriptor, only the headers of the complex elements are processed.

Returns `mdlElmdscr_stripAttributes` is of type `void`. It returns no value.

See Also `mdlElement_extractAttributes`, `mdlElmdscr_stripAttributes`, `mdlElement_appendAttributes`, `mdlElmdscr_extractAttributes`, `mdlElmdscr_appendAttributes`.

mdlElmdscr_appendAttributes

```
void mdlElmdscr_appendAttributes
(
  MSElementDescr    **elmdscrPP,    /* <=> descr to receive attributes */
  int                 length,          /* => length (words) of attr. data */
  short               *attributes      /* => attr. data to be appended */
);
```

Description The mdlElmdscr_appendAttributes function appends the attribute information contained in the *attributes* buffer to the end of any existing attributes attached to the elements in the descriptor *elmdscrPP*.

length specifies the size of the attribute data in words. If complex elements are present in the element descriptor *elmdscrPP*, only the header of the complex element receives the attributes.



No validity checking is performed on the content of the attribute information in *attributes*. Care should be exercised to make sure that you adhere to all of the rules for attributes.

Returns The mdlElmdscr_appendAttributes function is of type `void`. It returns no value.

See Also mdlElement_extractAttributes, mdlElmdscr_stripAttributes, mdlElement_appendAttributes, mdlElmdscr_extractAttributes, mdlElmdscr_appendAttributes.

mdlElmdscr_extractAttributes

```
#include <mselems.h>

void mdlElmdscr_extractAttributes
(
    int          *length,          /* <= length (words) of attr. data */
    short        *attributes,      /* <= attribute data extracted */
    MSElementDescr *elmdscrP /* => element descriptor to process */
);
```

Description The mdlElmdscr_extractAttributes function extracts the attribute information from the first element contained in the element descriptor *elmdscrP*. The attributes are copied into the *attributes* buffer and *length* is set to the size of the attribute data in words.

If more than one element is contained in the element descriptor, only the first element is processed. If the first element is a complex element, attributes are extracted from the header only.

The *attributes* array should be large enough to hold `MAX_ATTRIBSIZE` words.

Returns The mdlElmdscr_extractAttributes function is of type `void`. It returns no value.

See Also mdlElement_extractAttributes, mdlElement_stripAttributes, mdlElement_appendAttributes, mdlElmdscr_stripAttributes, mdlElmdscr_appendAttributes.

mdlElmdscr_setProperties

```
void mdlElmdscr_setProperties
(
MSElementDescr *edP,      /* <=> element to set properties */
int *level,               /* => level (or NULL) */
int *ggNum,               /* => graphic group (or NULL) */
int *class,               /* => class (or NULL) */
int *locked,              /* => locked (or NULL) */
int *new,                 /* => new (or NULL) */
int *modified,            /* => modified (or NULL) */
int *viewIndepend,        /* => view independent (or NULL) */
int *solidHole            /* => solid/hole (or NULL) */
);
```

Description `mdlElmdscr_setProperties` is identical to `mdlElement_setProperties` except it sets the properties of the elements in element descriptor *edP* rather than a single element.

Returns `mdlElmdscr_setProperties` is of type `void`; it returns no value.

See Also `mdlElement_setProperties`.

mdlElmdscr_extractNormal

```
int mdlElmdscr_extractNormal
(
Dpoint3d *normal,          /* <= normal to element */
Dpoint3d *point,           /* <= point on element */
MSElementDescr *edP,      /* => element */
Dpoint3d *inputDefaultNormal /* => default normal */
);
```

Description The `mdlElmdscr_extractNormal` function returns the normal vector to a planar element.

normal points to the normal vector for the element.

point points to a point on the element.

edP points to an element descriptor containing the planar element.

inputDefaultNormal is a the default normal. It is used only if the element normal is ambiguous, as is the case for a line element. In this case, the element normal that is closest to *defaultInputNormal* is returned.

Returns `mdlElmdscr_extractNormal` returns `SUCCESS` if the normal is successfully extracted and `MDLERR_BADELEMENT` if the element is not planar.

mdlElmdscr_displayToWindow

```
int mdlElmdscr_displayToWindow
(
    MSWindow      *windowP,      /* => window */
    BSIRect       *rectP,        /* => rectangle in window (local) */
    Viewflags     *viewFlags,    /* => view flags to use */
    MSElementDescr *edP,        /* => element descriptor */
    RotMatrix     *rotMatrixP,   /* => rotation matrix */
    Dpoint3d      *originP,      /* => origin in element coords */
    Dpoint3d      *rangeP,       /* => range in element coords */
    int           threeD,        /* => TRUE if 3D transform/element */
    int           menuColor      /* => Menu Index/ -1 for elm colors */
);
```

Description `mdlElmdscr_displayToWindow` is used to display the graphics element designated by *edP* in the window designated by *windowP*. Please refer to the Graphics Functions section of Chapter 13 for a discussion of MicroStation windows.

viewFlags is used to determine the element display mode. The typedef for the *viewFlags* structure is included in *mstypes.h*. The display of line weights, fast curve display etc., are controlled by the bitfields within this structure. If *NULL* is passed, *edP* is displayed with slow curves and text, and line weight, pattern, text node and, enter data field display is enabled.

If *threeD* is set to zero, both *edP* and *rotMatrixP* should be in two dimensional, otherwise they must be three dimensional.

rectP, *originP*, *rotMatrixP* and *rangeP* determine the location of the element within the window. These parameters are exactly analogous to the way that MicroStation views are specified in the TCB. *originP* designates the coordinate (in UORs) that will be mapped to the lower left corner of *rectP*. *rangeP* designates the dimensions of the display rectangle (volume in 3D) in UORs. *rotMatrixP* designates the rotation from world to view coordinates. A point in design file coordinates is, therefore, mapped to window coordinates in the following manner:

1. Subtract *originP*
2. Rotate by *rotMatrixP*
3. Scale to screen coordinates

$$scale.x = (rectP->corner.x - rectP->origin.x) / rangeP->x$$

$$scale.y = (rectP->corner.y - rectP->origin.y) / rangeP->y$$
4. Add lower left corner of *rectP*



The ratio between *rangeP->x* and *rangeP->y* should match the aspect ratio of *rectP*. If it doesn't, the display of *edP* will be stretched to reflect the difference.

If *menuColor* is non-negative, it designates the menu color index for all of the elements. For example, if `BLACK_INDEX` is used, all elements are displayed in black. If a negative *menuColor* is used, the elements are displayed in their normal MicroStation colors.

Returns `mdlElmdscr_displayToWindow` returns `SUCCESS` if the element is displayed successfully and an appropriate non-zero error code otherwise.

See Also `mdlElmdscr_extendedDisplayToWindow`, `mdlWindow_...` functions.

mdlElmdscr_extendedDisplayToWindow

```
int mdlElmdscr_extendedDisplayToWindow
(
  MWindow      *windowP,      /* => window */
  BSIRect      *rectP,        /* => rect in window (local coords) */
  Viewflags    *viewFlags,    /* => view flags to use */
  MElementDescr *elmDP,       /* => element descriptor */
  RotMatrix    *rotMatrixP,   /* => rotation matrix */
  Dpoint3d     *originP,      /* => origin in element coordinates */
  Dpoint3d     *rangeP,       /* => range in element coordinates */
  int          threeD,        /* => TRUE = 3D tranform/element */
  int          menuColor,     /* => Menu Index or -1 = elm colors */
  void         *colormap,     /* => NULL=master dgn colormap */
  Ext_viewflags *extViewFlags, /* => Extended view flags */
  boolean      clearFirst,     /* => clear window/zBuffer first */
  byte         *backgroundImageP /* => backgrnd RGB image or NULL */
);
```

Description The `mdlElmdscr_extendedDisplayToWindow` function provides some extensions to the `mdlElmdscr_displayToWindow` function that allow the display of rendered images within a window. See the `mdlElmdscr_displayToWindow` function for information on the first nine common arguments.

colormap is a pointer to an array of 256 bytes that contain the mapping from element colors to display indices. If `NULL` is passed, the master design file color map is used.

extViewFlags is a pointer to an extended *viewFlags* structure. This structure contains the rendering mode and flags for several additional viewing parameters.

If *clearFirst* is non-zero, the background is cleared before displaying the element.

backgroundImageP is a pointer to an RGB image buffer. If `NULL` is passed, no background image is displayed. The size of the image must match *rectP*.

Returns `mdlElmdscr_extendedDisplayToWindow` returns `SUCCESS` if the element is displayed successfully and an appropriate error code otherwise.

See Also mdlElmdscr_displayToWindow.

mdlElmdscr_copyParallel

```
int mdlElmdscr_copyParallel
(
  MSElementDescr  **outDscrPP,  /* <= output element */
  MSElementDescr  *inDscrP,     /* => input element */
  Dpoint3d         *point,       /* => direction/distance point */
  double           distance,     /* => distance to copy (or 0.0) */
  Dpoint3d         *normal       /* => perp. of offset plane or NULL */
);
```



This function will not function properly if used in versions of MicroStation prior to version 4.4.

Description mdlElmdscr_copyParallel creates a copy of the element descriptor *outDscrPP* specified by *inDscrP* that is offset parallel from the original element. This is the same result as obtained by the MicroStation COPY PARALLEL... key-ins. If *distance* is zero, the offset distance and direction is obtained from *point*. If *distance* is non-zero, it specifies the offset distance and the direction only is obtained from *point*.

**normal* defines the plane in which the geometry will be offset. If it is set to NULL the unit vector in the z direction will be used. To obtain an offset in a particular view, supply the third row of the view rotation matrix.

Returns mdlElmdscr_copyParallel returns SUCCESS if the element is successfully copied and an appropriate non-zero error code otherwise.

mdlProject_tangent

```
#include <mdl.h>
#include <mselems.h>

int mdlProject_tangent
(
  Dpoint3d         *position,     /* <= position of tangent end */
  Dpoint3d         *tangent,     /* <= tangent direction */
  Dpoint3d         *perpendicular, /* <= perpendicular direction */
  MSElementDescr  *edP,        /* => element */
  int              fileNum,      /* => (for compounds only) */
  Dpoint3d         *point,       /* => input point */
  RotMatrix        *rotMatrix,   /* => rotation matrix */
  Dpoint3d         *closePoint,  /* => closest point to tangent */
  double           tolerance     /* => tolerance */
);
```

Description The `mdlProject_tangent` function projects a point to an element along an element tangent. The position of the tangent's endpoint, direction, and the element perpendicular are returned in *position*, *tangent* and *perpendicular* respectively. If more than one tangent is possible, the tangent closest to *closePoint* is selected.

Returns `mdlProject_tangent` returns `SUCCESS` if the projection is successful and an error status otherwise.

mdlProject_perpendicular

```
#include <mdl.h>
#include <mselems.h>

int mdlProject_perpendicular
(
  Dpoint3d    *position,      /* <= position of perp. end */
  Dpoint3d    *tangent,      /* <= tangent direction */
  Dpoint3d    *perpendicular, /* <= perpendicular direction */
  MSElementDescr *edP,      /* => element */
  int         fileName,      /* => (for compounds only) */
  Dpoint3d    *point,        /* => point to project */
  RotMatrix   *rotMatrix,    /* => rotation matrix */
  double      tolerance      /* => tolerance */
);
```

Description The `mdlProject_perpendicular` function projects a point to an element along an element perpendicular. The position of the perpendicular endpoint, direction, and the element perpendicular are returned in *position*, *tangent* and *perpendicular*, respectively.

Returns `mdlProject_perpendicular` returns `SUCCESS` if the projection is successful and a non-zero error status otherwise.

Boolean Functions

The boolean functions operate on the intersections of, unions of, and differences between two closed, planar elements—shapes, ellipses, circles and complex shapes.

The following table lists the boolean functions:

Function	Used to
mdlElmdscr_unionShapes	construct complex shape(s) representing the union of two closed elements.
mdlElmdscr_intersectShapes	construct complex shape(s) representing the intersection of two closed shapes.
mdlElmdscr_differenceShapes	construct complex shape(s) representing the difference between two shapes.
mdlRegion_floodFill	compute the smallest region bounded by an element descriptor.

mdlElmdscr_unionShapes

```
#include <mselems.h>

int mdlElmdscr_unionShapes
(
  MSElementDescr    **insideEdPP, /* <= inside element(s) */
  MSElementDescr    *obsoleteEdPP, /* <= in 4.0 this was outputEdPP */
  MSElementDescr    *shape1,      /* => first shape to union */
  MSElementDescr    *shape2,      /* => second shape to union */
  double              tolerance     /* => tolerance for curved elms */
);
```

Description mdlElmdscr_unionShapes constructs a single complex shape or series of complex shapes that enclose the area represented by the union of the two input elements. Input elements must be coplanar, closed elements (shapes, ellipses, complex shapes). The union area is represented by one or more complex shapes with disjoint enclosing shapes returned as an element descriptor chain in **insideEdPP* and islands within the enclosing regions returned as an element descriptor chain in **obsoleteEdPP*.

tolerance represents the maximum allowable error when calculating intersections between curved elements. A small tolerance will result in high accuracy at the expense of increased processing time.

Returns mdlElmdscr_unionShapes returns SUCCESS if the union is constructed successfully. Possible error values include:

Value	Description
MDLERR_NONCOPLANARSHAPES	The input shapes are not coplanar.

See Also mdlElmdscr_intersectShapes, mdlElmdscr_differenceShapes.

mdlElmdscr_intersectShapes

```
#include <mselems.h>

int mdlElmdscr_intersectShapes
(
  MSElementDescr    **insideEdPP, /* <= inside element(s) */
  MSElementDescr    *obsoleteEdPP, /* <= in 4.0 this was outputEdPP */
  MSElementDescr    *shape1,      /* => first shape to intersect */
  MSElementDescr    *shape2,      /* => second shape to intersect */
  double              tolerance     /* => tolerance for curved elms */
);
```

Description The mdlElmdscr_intersectShapes function constructs a single complex shape or series of complex shapes that enclose the area represented by the intersection of the two input elements. Input elements must be coplanar, closed elements (shapes, ellipses, complex shapes). The intersection area is represented by one or more complex shapes with disjoint enclosing shapes returned as an element descriptor chain in **insideEdPP* and islands within the enclosing regions returned as an element descriptor chain in **obsoleteEdPP*.

tolerance represents the maximum allowable error when calculating intersections between curved elements. A small tolerance will result in high accuracy at the expense of increased processing time.

Returns mdlElmdscr_intersectShapes returns SUCCESS if the intersect is constructed successfully. Possible error values include:

Value	Description
MDLERR_NONCOPLANARSHAPES	The input shapes are not coplanar.
MDLERR_NULLSOLUTION	The input elements are disjoint and therefore have no intersection.

Returns mdlElmdscr_unionShapes, mdlElmdscr_differenceShapes.

mdlElmdscr_differenceShapes

```
#include <mselems.h>

int mdlElmdscr_differenceShapes
(
  MSElementDescr    **insideEdPP, /* <= inside element(s) */
  MSElementDescr    *obsoleteEdPP, /* <= in 4.0 this was outputEdPP */
  MSElementDescr    *shape1,      /* => subtrahend shape */
  MSElementDescr    *shape2,      /* => minuend shape */
  double              tolerance     /* => tolerance for curved elms */
);
```

Description The mdlElmdscr_differenceShapes function constructs a single complex shape or series of complex shapes that enclose the area represented by the difference between the two input elements. Input elements must be coplanar, closed elements (shapes, ellipses, complex shapes). The difference area is represented by one or more complex shapes with disjoint enclosing shapes returned as an element descriptor chain in **insideEdPP* and islands within the enclosing regions returned in **obsoleteEdPP*.

tolerance represents the maximum allowable error when calculating intersections between curved elements. A small tolerance will result in high accuracy at the expense of increased processing time.

Returns mdlElmdscr_differenceShapes returns SUCCESS if the difference is constructed successfully. Possible error values include:

Value	Description
MDLERR_NONCOPLANARSHAPES	The input shapes are not coplanar.
MDLERR_NULLSOLUTION	The difference area is empty.
MDLERR_UNBOUNDEDSOLUTION	The difference area is unbounded.

See Also mdlElmdscr_intersectShapes, mdlElmdscr_unionShapes.

mdlRegion_floodFill

```
int mdlRegion_floodFill
(
  MSElementDescr    **outEdPP,           /* <= elm bounding region */
  MSBsplineCurve     *outCurveP,         /* <= curve bounding region */
  MSElementDescr    *inEdP,             /* => seed element */
  Dpoint3d           *normalP,           /* => normal to plane */
  Dpoint3d           *seedPointP,        /* => seed point */
  double              closureTolerance,   /* => closure tolerance */
  MdlFunctionP       stopFunction        /* => stop func. (or NULL) */
);
```

Description The mdlRegion_floodFill function computes the smallest region bounded by the elements in the element descriptor *inEdP* containing the seed point, *seedPointP*. This operation is referred to as flood fill as it mimics the behavior of a raster flood fill operation common to pixel-oriented paint programs.

outEdPP points to an element descriptor address. This closed element descriptor encloses the flood fill region. NULL may be passed for *outEdPP* if the enclosing element is not required.

outCurveP is a pointer to a B-spline curve enclosing the flood fill region. NULL may be passed for *outCurveP* if the enclosing curve is not required.

inEdP points to an element descriptor that contains the potential boundary elements.

normalP points to a unit vector that determines (in 3D only) the viewing direction for the flood region. For a MicroStation view, this vector can be determined by extracting the third row of the view rotation matrix:

```
mdlRMMatrix_getRowVector(&normal, &vuMatrix, 2)
```

seedPointP points to the flood region seed point.

closureTolerance is the maximum allowable gap between adjacent boundary elements.

stopFunction is a pointer to a function that is called periodically as the flood region is computed. If this function returns a non-zero value, the processing is aborted. If a large number of potential boundaries are to be searched, the processing time can be significant and *stopFunction* is useful for allowing the user to abort the operation. NULL may be specified for *stopFunction* if no stop function is required.

Returns mdlRegion_floodFill returns SUCCESS if an enclosing region is successfully computed. MDLERR_USERABORT is returned if processing is terminated by a non-zero return status from stopFunc. ERROR is returned if no enclosing region is found.

See Also mdlRMMatrix_getRowVector.

Measurement Functions

The measurement functions provide methods of calculating the area of a closed element or polygon, and the distance between two elements.

The following table lists the area calculation functions:

Function	Used to
mdlMeasure_elmDscrArea	measure of area of closed element.
mdlMeasure_polygonArea	measure area of polygon.
mdlMinDist_betweenElms	measure the minimum distance between two elements.
mdlMeasure_areaProperties	measure basic area properties and principal moments and axes of a closed element.
mdlMeasure_linearProperties	measure linear properties of a planar element.
mdlMeasure_polygonProperties	measure basic area properties of a closed polygon.
mdlMeasure_volumeProperties	measure volume properties and principal moments and axes of a closed element.
mdlMeasure_surfaceProperties	measure surface properties and principal moments and axes of an element.

mdlMeasure_elmDscrArea

```
#include <mdl.h>
#include <mselems.h>

int mdlMeasure_elmDscrArea
(
double          *area,          /* <= area of element */
double          *perimeter,     /* <= perimeter of element */
MSElementDescr *elemDscrP     /* => element to measure */
);
```

Description The mdlMeasure_elmDscrArea function returns the area and perimeter length of the closed element described by *elemDscrP*. Valid closed elements include shapes, ellipses, complex shapes and closed B-spline curves. If a nonplanar element is measured, the area of the elements projection to a plane is returned.

Returns mdlMeasure_elmDscrArea returns SUCCESS if the element is successfully measured and MDLERR_NONCLOSEDELM if the element is not closed.

mdlMeasure_polygonArea

```
#include <mdl.h>

int mdlMeasure_polygonArea
(
double          *area,          /* <= area of polygon */
double          *perimeter,     /* <= perimeter of polygon */
Dpoint3d        *points,        /* => polygon points */
int             numPoints       /* => number of points */
);
```

Description mdlMeasure_polygonArea returns the area and perimeter length of the polygon described by *points* and *numPoints*. The polygon must be closed (first and last points identical). If the polygon is nonplanar, the area of the polygon's projection to a plane is calculated.

Returns mdlMeasure_polygonArea returns SUCCESS if the polygon area is successfully calculated and MDLERR_NONCLOSEDELM if the polygon is not closed.

mdlMinDist_betweenElms

```

#include <mdl.h>
#include <mselems.h>
int mdlMinDist_betweenElms
(
    Dpoint3d      *minPnt1,      /* <= minimum point on elm 1 */
    Dpoint3d      *minPnt2,      /* <= minimum point on elm 2 */
    double        *distance,      /* <= minimum distance */
    MSElementDescr *elemDscrP1, /* => first element */
    MSElementDescr *elemDscrP2, /* => second element */
    Dpoint3d      *closePoint,    /* => point close to minima */
    double        tolerance       /* => stroking tolerance */
);

```

Description The `mdlMinDist_betweenElms` function sets *minPnt1* and *minPnt2* to the closest points between two elements described by *elemDscrP1* and *elemDscrP2*.

distance is set to the distance between these points. If the elements intersect, the intersection points are returned and *distance* is set to zero. If there is more than one minima, or there are multiple intersections, the points closest to *closePoint* are selected. *tolerance* designates to the maximum allowable error. The minimum distance calculation for curved elements is an iterative process that requires additional processing to converge to small tolerances. Setting a small tolerance value will therefore cause a more accurate solution at the expense of increased processing time.



This function was previously named `mdlMeasure_minimumDistance`. Only the name has changed; the functionality remains intact.

Returns `mdlMinDist_betweenElms` returns `SUCCESS` if the minima is successfully calculated.

See Also `mdlBspline_minimumDistanceToCurve`, `mdlBspline_minimumDistanceToSurface`.

mdlMeasure_areaProperties

```
#include <mdl.h>

int mdlMeasure_areaProperties
(
double      *perimeterP,          /* <= perimeter of element */
double      *areaP,              /* <= area of element */
Dpoint3d    *normalP,            /* <= normal to plane of element*/
Dpoint3d    *centroidP,          /* <= centroid of enclosed area */
Dpoint3d    *momentP,            /* <= second moments of area */
double      *iXYP,               /* <= product of inertia (xy) */
double      *iXZP,               /* <= product of inertia (xz) */
double      *iYZP,               /* <= product of inertia (yz) */
Dpoint3d    *principalMomentsP, /* <= principal moments */
Dpoint3d    *principalDirectionsP, /* <= principal axes */
MSElementDescr *edP,           /* => element to measure */
double      tolerance            /* => stroking tolerance */
);
```

Description *mdlMeasure_areaProperties* returns the basic area properties, along with the principal moments and principal axes, of the area enclosed by the element described by *edP*. The element is assumed to be closed, but this is not checked. Area properties are undefined if the element is not closed. The element is assumed to be planar, but this is not checked and the element is not projected to a plane if nonplanar. Area properties are not defined if the element is not planar. See *mdlMeasure_surfaceProperties*. Area properties are returned in the specified output parameters. If a particular property is not of interest, then pass NULL for the corresponding output parameter.

perimeterP is the perimeter of the area enclosed by the element in UORs.

areaP is the area enclosed by the element, measured in UORs squared.

normalP is a unit vector normal to the plane of the element.

centroidP is the (x, y, z) location of the centroid of the area, in UORs in the world coordinate system.

momentP holds the second moments of area, {Ixx, Iyy, Izz}, about axes through the centroid and parallel to the world x, y and z axes. The fundamental units are master units. *iXYP*, *iXZP* and *iYZP* are the are the Ixy, Ixz and Iyz products of inertia with respect to these axes.

principalMomentsP holds the principal second moments. Principal moments are defined as the extreme values of moments, about the principal axes of the area.

principalDirectionsP is an array of three direction vectors which define the principal axes of the area in the world coordinate system, relative to the centroid. By definition, the principal moments are defined about the principal axes. The principal axes will define a plane which is parallel to

the plane of the element (and possibly rotated about the element's normal).

edP describes the element to measure.

tolerance is the stroke tolerance. For curved elements, area properties are computed from a polygonal approximation to the original. *tolerance* is the maximum distance between the actual curve and the approximating vectors.

Returns `mdlMeasure_areaProperties` returns `SUCCESS` if the area properties are successfully calculated, `MDLERR_BADELEMENT` if the element cannot be stroked, and `MDLERR_INSMEMORY` if there is insufficient memory to stroke the element.

See Also `mdlMeasure_polygonProperties`, `mdlMeasure_linearProperties`, `mdlMeasure_elmDscrArea`, `mdlMeasure_surfaceProperties`.

mdlMeasure_linearProperties

```
#include <mdl.h>

int mdlMeasure_linearProperties
(
    double      *lengthP,           /* <= length of element */
    Dpoint3d    *centroidP,         /* <= centroid of curve */
    Dpoint3d    *momentP,           /* <= second moments of curve */
    double      *iXYP,              /* <= product of inertia (xy) */
    double      *iXZP,              /* <= product of inertia (xz) */
    double      *iYZP,              /* <= product of inertia (yz) */
    Dpoint3d    *principalMomentsP, /* <= principal moments */
    Dpoint3d    *principalDirectionsP, /* <= principal axes */
    MSElementDescr *edP,           /* => element to measure */
    double      tolerance            /* => stroking tolerance */
);
```

Description `mdlMeasure_linearProperties` returns the basic linear properties of the element described by *edP*. The element is assumed to be non-closed, but is not required to be. *edP* is not required to be planar. Linear properties are returned in the specified output parameters. If a particular property is not of interest, then pass `NULL` for the corresponding output parameter.

lengthP is the length of the element in UORs.

centroidP is the (x,y,z) location of the centroid of the element, in UORs in the world coordinate system. The centroid will not generally be on the element.

momentP holds the second moments of the element, {Ixx, Iyy, Izz}, about axes through the centroid and parallel to the world x, y and z axes. The fundamental units are UORs. *iXYP*, *iXZP* and *iYZP* are the Ixy, Ixz and Iyz products of inertia with respect to these axes.

principalMomentsP holds the principal second moments. Principal moments are defined as the extreme values of moments, about the principal axes of the curve.

principalDirectionsP is an array of three direction vectors which define the principal axes of the element in the world coordinate system, relative to the centroid. By definition, the principal moments are defined about the principal axes.

edP describes the element to measure.

tolerance is the stroke tolerance. For curved elements, linear properties are computed from a stroked approximation to the original. *tolerance* is the maximum distance between the actual curve and the approximating vectors.

Returns mdlMeasure_linearProperties returns SUCCESS if the area properties are successfully calculated, MDLERR_BADELEMENT if the element cannot be stroked, or MDLERR_INSFMEMORY if there is insufficient memory to stroke a curved element.

See Also mdlMeasure_areaProperties.

mdlMeasure_polygonProperties

```
#include <mdl.h>

int mdlMeasure_polygonProperties
(
    double      *perimeterP,    /* <= perimeter of polygon */
    double      *areaP,         /* <= enclosed area */
    Dpoint3d    *normalP,       /* <= normal to plane of polygon*/
    Dpoint3d    *centroidP,     /* <= centroid of enclosed area */
    Dpoint3d    *momentP,       /* <= second moments of area */
    double      *iXYP,          /* <= product of inertia (xy) */
    double      *iXZP,          /* <= product of inertia (xz) */
    double      *iYZP,          /* <= product of inertia (yz) */
    Dpoint3d    *points,        /* => points defining polygon */
    int          nPoint          /* => number of points */
);
```

Description mdlMeasure_polygonProperties returns the basic area properties of the area enclosed by the polygon defined by *points*. The polygon is assumed to be closed, but this is not checked. Area properties are undefined if the polygon is not closed. The polygon is assumed to be planar, but this is not checked and the polygon is not projected to a plane if nonplanar. Area properties are not defined if the polygon is not planar. The polygon is assumed to be planar but is not checked or projected if not. Area properties are undefined in this case. Area properties are returned in the

specified output parameters. If a particular property is not of interest, then pass NULL for the corresponding output parameter.

perimeterP is the perimeter of the area enclosed by the polygon in master units.

areaP is the area enclosed by the polygon, measured in master units squared.

normalP is a unit vector normal to the plane of the polygon.

centroidP is the (x, y, z) location of the centroid of the area, in master units in the world coordinate system.

momentP holds the second moments of area, {Ixx, Iyy, Izz}, about axes through the centroid and parallel to the world x, y and z axes. The fundamental units are master units. The *iXYP*, *iXZP* and *iYZP* parameters are the Ixy, Ixz and Iyz products of inertia with respect to these axes.

points and *nPoints* define the closed polygon to be measured.

Returns mdlMeasure_polygonProperties returns SUCCESS if the area properties are successfully calculated and MDLERR_NONCLOSEDELM if the polygon is not closed.

mdlMeasure_volumeProperties

```
#include <mdl.h>

int mdlMeasure_volumeProperties
(
    double      *volumeP,           /* <= volume of element */
    double      *areaP,             /* <= area of element */
    double      *closureErrorP,     /* <= closure error */
    Dpoint3d    *centroidP,         /* <= centroid of encl'd volume */
    Dpoint3d    *momentP,           /* <= second moments of volume */
    double      *iXYP,              /* <= product of inertia (xy) */
    double      *iXZP,              /* <= product of inertia (xz) */
    double      *iYZP,              /* <= product of inertia (yz) */
    Dpoint3d    *principalMomentsP, /* <= principal moments */
    Dpoint3d    *principalDirectionsP, /* <= principal axes */
    MSElementDescr *edP,           /* => element to measure */
    double      tolerance,          /* => stroking tolerance */
    mdlFunctionP stopFunc           /* => meshing stop func or NULL */
);
```

Description mdlMeasure_volumeProperties returns the basic volume properties along with the principal moments and principal axes of the volume enclosed by the element described by *edP*. Volume properties are returned in the specified output parameters. Pass NULL for any output parameter which is not of interest.



Volume properties are not defined if the element is not closed. In most cases, this function returns an error status and does not return any output

parameters if the element is not closed. If the returned status is success, check the value returned in *closureErrorP* for a confidence measure. Note that a “flat” (planar) element is not closed, so that

mdlMeasure_volumeProperties is mutually exclusive with
mdlMeasure_areaProperties.

volumeP returns the volume enclosed by the element, measured in master units cubed.

areaP returns the surface area of the volume enclosed by the element in master units squared.

closureErrorP is set to a quantitative measurement ranging from zero to one of element closure error and is essentially a “no-confidence” statistic for the output parameters. Loosely defined, *closureError* is the proportion of the volume’s total surface area which is perforated by gaps or openings. A zero or insignificant closure *error* indicates that the element should be considered closed, while a significant closure error indicates that the element is not closed and no volume properties can be calculated for it. Specifically, if the closure error is less than or equal to 0.1, the function will return the output parameters and report success; otherwise, if the closure error is greater than 0.1, the function will return an error status of MDLERR_NONCLOSEDELM and will not return the output parameters. As a rule of thumb, even when the function returns success, a closure error greater than 0.01 suggests that the element is not closed and suggests low confidence in the returned volume properties.

centroidP returns the (x, y, z) location of the centroid of the volume, in master units in the world coordinate system. The centroid will not necessarily be inside the volume.

momentP returns the second moments, {Ixx, Iyy, Izz}, about axes through the centroid and parallel to the world x, y and z axes. The fundamental units are master units. The *iXYP*, *iXZP* and *iYZP* parameters are the Ixy, Ixz and Iyz products of inertia with respect to these axes. Together, these six parameters define the element’s inertia tensor (which is symmetric).

principalMomentsP returns the principal second moments. Principal moments are defined as the extreme values of the second moments about the principal axes of the volume. By definition, the products of inertia become zero about the principal axes. *principalDirectionsP* is an array of three direction vectors which define the principal axes of the volume in the world coordinate system, relative to the centroid. The principal axes are mutually orthogonal and describe how the inertia tensor is rotated to its diagonalized, extreme form.

edP describes the element to measure.

tolerance is the meshing tolerance. For curved elements, volume properties are computed from a polygonal approximation to the original. *tolerance* is the maximum distance between the actual surface and the

approximating vectors. The accuracy of the volume properties therefore improves as the tolerance becomes smaller.

stopfunc allows the caller to supply an MDL function to monitor and potentially limit the meshing sub-division process. If no *stopfunc* is supplied, the element is broken down to the granularity required to satisfy the stated meshing tolerance. A *stopfunc* can stop the sub-division process at any point before that.

Returns `mdlMeasure_volumeProperties` returns `SUCCESS` if the volume properties are successfully calculated, `MDLERR_BADELEMENT` if the element cannot be meshed, `MDLERR_INSMEMORY` if there is insufficient memory to mesh the element, or `MDLERR_NONCLOSEDELM` if the element is not closed.

See Also `mdlMeasure_surfaceProperties`.

mdlMeasure_surfaceProperties

```
#include <mdl.h>

int mdlMeasure_surfaceProperties
(
    double      *areaP,           /* <= area of element */
    Dpoint3d    *centroidP,       /* <= centroid of encl. volume */
    Dpoint3d    *momentP,        /* <= second moments of volume */
    double      *iXYP,           /* <= product of inertia (xy) */
    double      *iXZP,           /* <= product of inertia (xz) */
    double      *iYZP,           /* <= product of inertia (yz) */
    Dpoint3d    *principalMomentsP, /* <= principal moments */
    Dpoint3d    *principalDirectionsP, /* <= principal axes */
    MSElementDescr *edP,        /* => element to measure */
    double      tolerance,        /* => stroking tolerance */
    mdlFunctionP stopFunc        /* => meshing stop function */
);
```

Description `mdlMeasure_surfaceProperties` returns the basic surface area properties along with the principal moments and principal axes of the element described by *edP*. The element need not be planar (but it may be) and need not be closed. Surface area properties are returned in the specified output parameters. Pass `NULL` for any output parameter which is not of interest.

areaP returns the surface area of the element in master units squared.

centroidP returns the (x, y, z) location of the centroid of the surface, in master units in the world coordinate system. The centroid will not generally be on the surface.

momentP returns the second moments, {Ixx, Iyy, Izz}, about axes through the centroid and parallel to the world x, y and z axes. The fundamental units are master units. The *iXYP*, *iXZP* and *iYZP* parameters are the Ixy, Ixz

and Iyz products of inertia with respect to these axes. Together, these six parameters define the element's inertia tensor (which is symmetric).

principalMomentsP returns the principal second moments. Principal moments are defined as the extreme values of the second moments about the principal axes of the surface. By definition, the products of inertia become zero about the principal axes. *principalDirectionsP* is an array of three direction vectors which define the principal axes of the volume in the world coordinate system, relative to the centroid. The principal axes are mutually orthogonal and describe how the inertia tensor is rotated to its diagonalized, extreme form.

edP describes the element to measure.

tolerance is the meshing tolerance. For curved elements, surface area properties are computed from a polygonal approximation to the original. *tolerance* is the maximum distance between the actual surface and the approximating vectors. The accuracy of the area properties therefore improves as the tolerance becomes smaller.

stopFunc allows the caller to supply an MDL function to monitor and potentially limit the meshing sub-division process. If no *stopFunc* is supplied, the element is broken down to the granularity required to satisfy the stated meshing tolerance. A *stopFunc* can stop the sub-division process at any point before that.

Returns `mdlMeasure_surfaceProperties` returns `SUCCESS` if the area properties are successfully calculated, `MDLERR_BADELEMENT` if the element cannot be meshed, or `MDLERR_INSFMEMORY` if there is insufficient memory to mesh the element.

See Also `mdlMeasure_volumeProperties`.

Element Intersection Functions

Routines are provided to calculate the intersection between two elements. All of these routines compute the apparent as opposed to the true intersections. In two dimensions the true and apparent intersections are identical. In three dimensions the apparent intersection is the intersection as viewed along the Z-Axis of the specified rotation matrix. If a true intersection is found, an intersection point on each element is returned. If the elements are two dimensional, the two intersection points will be identical.

The following table lists the element intersection functions:

Function	Used to
<code>mdlIntersect_allBetweenElms</code>	get all intersections between two elements.
<code>mdlIntersect_allBetweenExtendedElms</code>	get all intersections between two (possibly extended) elements.

Function	Used to
<code>mdlIntersect_closestBetweenElms</code>	get closest intersection between two elements to a given point.
<code>mdlIntersect_betweenElmsByIndex</code>	get intersection between two elements by index.



B-spline specific intersection functions can be found in the B-spline chapter.

`mdlIntersect_allBetweenElms`, `mdlIntersect_allBetweenExtendedElms`

```
#include <mdl.h>
#include <mselems.h>

int mdlIntersect_allBetweenElms
(
    Dpoint3d    *isPnt1,        /* <= intersection(s) on elm 1*/
    Dpoint3d    *isPnt2,        /* <= intersection(s) on elm 2*/
    int         isPntSize,      /* => size of *isPnt1 & *isPnt2 */
    MSElementDescr *edP1,      /* => element 1 */
    MSElementDescr *edP2,      /* => element 2 */
    RotMatrix    *rotMatrixP,   /* => rotation matrix (3D only) */
    double       tolerance      /* => convergence tolerance */
);

int mdlIntersect_allBetweenExtendedElms
(
    Dpoint3d    *isPnt1,        /* <= intersection(s) on elm 1*/
    Dpoint3d    *isPnt2,        /* <= intersection(s) on elm 2*/
    int         isPntSize,      /* => size of *isPnt1 & *isPnt2 */
    MSElementDescr *edP1,      /* => element 1 */
    MSElementDescr *edP2,      /* => element 2 */
    RotMatrix    *rotMatrixP,   /* => rotation matrix (3D only) */
    double       tolerance,      /* => convergence tolerance */
    Dpoint3d    *idPnt1,        /* => identification Point 1 */
    Dpoint3d    *idPnt2        /* => identification Point 2 */
);
```

Description More than one intersection may exist between two elements. The `mdlIntersect_allBetweenElms` function returns all of the intersections.

The `mdlIntersect_allBetweenExtendedElms` differs only in its handling of linear (line, line string and shape) elements. For these elements, the intersection between a the infinite extension of single line segment is computed rather than the actual element. If a linear element contains more than one segment (linestrings and shapes), the closest segments to *idPnt1* and *idPnt2* are used (for *edP1* and *edP2* respectively).

isPnt1 and *isPnt2* are pointers to the location for the intersection point(s) on the first and second elements, respectively. For `mdlIntersect_allBetweenElms` these are arrays of type *isPntSize* `Dpoint3ds`.

edP1 and *edP2* are pointers to element descriptors describing the elements to be tested for intersections. Valid element types include lines, line strings, shapes, curves, arcs, ellipses and B-spline curves. For three dimensional elements, the `mdlIntersect_...` routines return apparent intersections, ignoring differences along the Z-axis.

rotMatrixP is a pointer to the rotation matrix describing the coordinate system in which the apparent intersections are calculated. Because all rotation matrices will produce the same result when working in two dimensions, *rotMatrixP* has no meaning in such cases and `NULL` should be specified. If `NULL` is specified in 3D, the world coordinate system is used.

tolerance designates the maximum allowable error. The intersection calculation for certain curved elements is an iterative process that requires additional processing to converge to small tolerances. Setting a small tolerance value will, therefore, cause a more accurate solution at the expense of increased processing time.

Returns `mdlIntersect_allBetweenElms` returns the number of intersections found.

See Also `mdlIntersect_closestBetweenElms`, `mdlIntersect_betweenElmsByIndex`, `mdlElmdscr_read`.

mdlIntersect_closestBetweenElms, mdlIntersect_betweenElmsByIndex

```
#include <mdl.h>
#include <mselems.h>

int mdlIntersect_closestBetweenElms
(
    Dpoint3d      *isPnt1,          /* <= intersection on elm 1 */
    Dpoint3d      *isPnt2,          /* <= intersection on elm 2 */
    int           *index,           /* <= index of intersection */
    MSElementDescr *edP1,          /* => element 1 */
    MSElementDescr *edP2,          /* => element 2 */
    RotMatrix     *rotMatrixP,      /* => rotation matrix (3D only) */
    Dpoint3d      *closePoint,      /* => pt close to intersection */

```

```
double      tolerance      /* => convergence tolerance */
);

int mdlIntersect_betweenElmsByIndex
(
Dpoint3d    *isPnt1,      /* <= intersection on element 1*/
Dpoint3d    *isPnt2,      /* <= intersection on element 2*/
int          index,       /* => index of intersection */
MSElementDescr *edP1,    /* => element 1 */
MSElementDescr *edP2,    /* => element 2 */
RotMatrix    *rotMatrixP, /* => rotation matrix (3D only) */
double      tolerance     /* => convergence tolerance */
);
```

Description More than one intersection may exist between two elements. `mdlIntersect_closestBetweenElms` returns the closest intersection to **closePoint* and the index of the intersection point. This index can then be used as input to the `mdlAssoc_createIntersection` function. `mdlIntersect_betweenElmsByIndex` returns the intersection specified by index.

isPnt1 and *isPnt2* are pointers to the location for the intersection point(s) on the first and second elements, respectively. For `mdlIntersect_closestBetweenElms` and `mdlIntersect_betweenElmsByIndex`, these are pointers to a single `Dpoint3d`.

edP1 and *edP2* are pointers to element descriptors describing the elements to be tested for intersections. Valid element types include lines, line strings, shapes, curves, arcs, ellipses and B-spline curves. For three dimensional elements, the `mdlIntersect_...` routines return apparent intersections, ignoring differences along the Z-axis. *rotMatrixP* is a pointer to the rotation matrix describing the coordinate system in which the apparent intersections are calculated. A pointer to a view rotation matrix will result in the intersections, as seen in that view.

In two dimensions, this argument is not used and `NULL` may be specified. If `NULL` is specified in 3D, the world coordinate system is used.

tolerance designates the maximum allowable error. The intersection calculation for certain curved elements is an iterative process that requires additional processing to converge to small tolerances. Setting a small tolerance value will, therefore, cause a more accurate solution at the expense of increased processing time.

Returns These functions return `SUCCESS` if an intersection is successfully found and a non-zero error status otherwise.

See Also `mdlIntersect_allBetweenElms`, `mdlElmdscr_read`.

View Functions

MicroStation displays the master design file and its reference files in screen windows called **views**. Each view has settings that determine the displayed image's scale and orientation and certain design file display characteristics. These display characteristics include information such as the levels that display, whether line weight display is on or off, whether text nodes display, whether dimension and construction class elements display, and fast text display settings.

View functions let an MDL program adjust the scale and orientation of the displayed image and the display characteristics. Some functions repaint one or more views, save and retrieve named and standard views, determine whether a view is visible, and notify an MDL program every a view is being repainted. Most functions require a *viewNumber* argument. This is the view affected by the function, and its range must be greater than or equal to 0 and less than the constant `MAX_VIEWS`, as defined in `msdefs.h`. On the screen, the views are labeled 1 through `MAX_VIEWS`, since most users are more comfortable with using 1 as the index origin.



Some of these functions are included in the MDL object library `mdlilib.mli`. A new version of this library must be linked into an application that calls these functions.

The following table lists view functions:

Function	Used to
<code>mdlView_turnOn</code>	open a view.
<code>mdlView_turnOff</code>	close a view.
<code>mdlView_fit</code>	fit all displayable elements in a view.
<code>mdlView_setArea</code>	set the view's viewing area.
<code>mdlView_zoom</code>	change the size of the viewing extents by a given scale factor.
<code>mdlView_setDisplayDepth</code> <code>mdlView_setDisplayDepthPoints</code>	set the view's front and back clipping planes.
<code>mdlView_setActiveDepth</code> <code>mdlView_setActiveDepthPoint</code>	set the view's active depth.
<code>mdlView_rotateToRMatrix</code>	rotate the view to a given rotation matrix.
<code>mdlView_getParameters</code>	extract the view settings from a view.
<code>mdlView_getCamera</code>	extract the camera settings from a view.
<code>mdlView_getStandardCameraLens</code>	get lens parameters for standard lens.
<code>mdlView_setCameraLens</code>	set camera lens for a view.
<code>mdlView_getCameraParameters [mdlilib.mli]</code>	get the camera parameters for an existing MicroStation view.
<code>mdlView_setCameraParameters [mdlilib.mli]</code>	set the camera parameters for a view.

Function	Used to
<code>mdlView_cameraLensLengthFromAngle</code>	get the camera lens length, in millimeters, that corresponds to a field of vision angle.
<code>mdlView_cameraLensAngleFromLength</code>	get the camera lens angle, in radians, that corresponds to a camera lens length
<code>mdlView_updateSingle</code>	repaint a view using the current settings.
<code>mdlView_updateMulti</code>	repaint multiple views simultaneously, using their current settings.
<code>mdlView_updateMultiExtended</code>	similar to <code>mdlView_updateMulti</code> , with the additional ability to modify the scan list that MicroStation uses for the update.
<code>mdlView_renderSingle</code>	render a view.
<code>mdlView_setRenderMode</code>	set the rendering mode for a view.
<code>mdlView_hiddenLineRemoval</code>	perform hidden line removal on a view.
Hidden Line Removal Functions	user functions which can be set using <code>mdlView_hiddenLineRemoval</code> to remove hidden lines in any custom manner desired.
<code>mdlView_getStandard</code>	set the <i>RotMatrix</i> to the rotation matrix for the standard view.
<code>mdlView_isStandard</code>	indicate whether a rotation matrix matches a standard view transformation matrices.
<code>mdlView_applyNamed</code>	apply the information returned from <code>mdlView_findNamed</code> to a view.
<code>mdlView_findNamed</code>	search the design file for a saved view.
<code>mdlView_attachNamed</code>	attach the named view to a displayed view.
<code>mdlView_saveNamed</code>	save the current view orientation, scale, and display characteristics to a named view.
<code>mdlView_deleteNamed</code>	delete a named view from the master file.
<code>mdlView_getContentCorners</code>	return the corners of a view rectangle.
<code>mdlView_createSavedViewElement [mdlLib.mli]</code>	create a type 5 saved view element.
<code>mdlView_extractSavedView [mdlLib.mli]</code>	read a type 5 saved view element.
<code>mdlView_getLevels</code>	retrieve the currently displayed levels.
<code>mdlView_setLevels</code>	set the currently displayed levels.
<code>mdlView_getDisplayControl</code>	retrieve display control flags.
<code>mdlView_setDisplayControl</code>	set display control flags.
<code>mdlView_defaultCursor</code>	cancel any special cursor behavior for views and return to the default crosshair cursor.

Function	Used to
<code>mdlView_refreshGrid</code>	redraw or erase the grid that appears in views for which grid display is turned on.
<code>mdlView_isActive</code>	determine whether view is active.
<code>mdlView_isVisible</code>	test for view visibility.
<code>mdlView_synchWithTCB</code>	synchronize the private view information maintained separately for each view with the “public” view information maintained in the TCB.
<code>mdlView_queuePartialUpdate</code>	update one or more subparts of a MicroStation view.
<code>mdlView_newWindowCenter</code>	set a new window center without changing the orientation or scale of the view.
<code>mdlView_pointToScreen</code>	calculate a screen position for an input point.
<code>mdlView_screenToPoint</code>	calculate the design file coordinates given a point on the screen.
<code>mdlView_setPopupMenu</code>	sets the menu that pops up when the user presses Shift-Reset inside a MicroStation view.
<code>mdlView_blitDirtyAreas (Macintosh Only)</code>	blits all “dirty” portions of the Macintosh offscreen bitmap to the screen.
<code>mdlView_setFunction</code>	designate an MDL function to be called when a significant event happens in one of the MicroStation view windows.
<code>mdlView_clipToContent</code>	adjust the graphics clipping to the full content rectangle.
<code>mdlView_clipToViewRect</code>	adjust the graphics clipping to the view rectangle.
<code>mdlView_getViewRectangle</code>	retrieve the screen area taken up by the view.
<code>mdlView_indexFromWindow</code>	retrieve the view number from a window pointer.

The following table lists view control user functions. MDL calls these user-supplied functions when certain events occur within MicroStation. The programmer determines the user function name. (The names below are for illustration only). These functions are designated to MDL through function pointer arguments to MDL routines.

Function	MicroStation calls when
<code>userView_update</code>	some or all views are being updated.
<code>userView_motion</code>	the pointer (cursor) moves in a MicroStation view window.

Function	MicroStation calls when
<i>userView_noMotion</i>	the pointer (cursor) stops moving in a MicroStation view window.
<i>userView_updateEachElement</i>	drawing elements to a view or to a plot.
<i>userView_plot</i>	before and after a view is plotted.
<i>userView_drawCursor</i>	draw vector cursor to a view.

Example

See `view.mc`.

mdlView_turnOn

```
#include <mdl.h>

int mdlView_turnOn
(
    int      viewNumber      /* => view to turn on */
);
```

Description The `mdlView_turnOn` function turns on a view given by *viewNumber* using the current view parameters. The function does not return until the view is initially painted.

Returns The `mdlView_turnOn` function returns `SUCCESS` if the view is turned on (or already on) and `MDLERR_BADVIEWNUMBER` if *viewNumber* is invalid.

mdlView_turnOff

```
#include <mdl.h>

int mdlView_turnOff
(
    int      viewNumber      /* => view to turn off */
);
```

Description The `mdlView_turnOff` function hides the view given by *viewNumber*.

Returns The `mdlView_turnOff` function returns `SUCCESS` if the view is turned off, `MDLERR_BADVIEWNUMBER` if *viewNumber* is invalid, and `MDLERR_VIEWNOTDISPLAYED` if the view is not currently displayed.

mdlView_fit

```
#include <mdl.h>

int mdlView_fit
(
    int     viewNumber,    /* => view to fit */
    ULONG   *fileMask     /* => files to include in fit */
);
```

Description The `mdlView_fit` function scans the files indicated by the file mask pointed to by *fileMask*. It then sets the display extents of view *viewNumber* to fit all elements found.

viewNumber designates which view to fit (starting with 0 for the view labeled View 1).

fileMask points to an array of eight long integers. Bits control elements as follows:

- The lowest bit (bit 0) of the first long integer controls whether elements from the master file are included in the fit.
- Bit 1 controls whether elements from the reference file with attachment number 1 are included.
- Bit 0 of the second long integer controls whether elements from the reference file with attachment number 32 are included.
- The highest bit (bit 31) of the eighth long integer controls whether elements from the reference file with attachment number 255 are included.



After calling `mdlView_fit`, you must update the view to see the changed view extents using the `mdlView_updateSingle` function.

Returns The `mdlView_fit` function returns `SUCCESS` if the view extents were modified, `MDLERR_BADVIEWNUMBER` if *viewNumber* is invalid, and `ERROR` if no elements were found and the view extents were not modified.

See Also `mdlView_updateSingle`.

mdlView_setArea

```
#include <mdl.h>

int mdlView_setArea
(
    int          viewNumber,      /* => view to set area for */
    Dpoint3d     *viewRange,      /* => new range for view */
    Dpoint3d     *zOrigin,        /* => origin for view */
    double       zDelta,          /* => depth of displayed portion */
    double       actZDepth,        /* => active depth */
    RotMatrix    *rMatrix         /* => view rotation matrix */
);
```

Description *mdlView_setArea* sets the viewing area for view *viewNumber*. All parameters other than *viewNumber* are transformed by the current transform if one exists.

viewRange is an array of two points that, combined with the rotation matrix *rMatrix*, define the viewing rectangle. MicroStation adjusts the rectangle to match the view's actual aspect ratio.

zOrigin is a point that lies on the view's back clipping plane. If *zOrigin* is NULL, MicroStation uses (0, 0, 0) in the current coordinate system.

zDelta is the distance from the front clipping plane to the front clipping plane. *zDelta* must be greater than zero.

actZdepth is the distance from the back clipping plane to the view's active depth. *actZDepth* must be greater than zero and less than *zDelta*.

rMatrix defines the view's rotation. If *rMatrix* is NULL, MicroStation uses the identity matrix (top view).



After calling *mdlView_setArea*, you need to update the view to see the changed view extents using the *mdlView_updateSingle* function.

Returns The *mdlView_setArea* function returns **SUCCESS** if the viewing area is set, **MDLERR_BADVIEWNUMBER** if *viewNumber* is invalid, and **MDLERR_VIEWNOTDISPLAYED** if the view is not currently displayed.

mdlView_zoom

```
#include <mdl.h>

int mdlView_zoom
(
    int          outViewNumber, /* => view to change parameters for */
    int          inViewNumber,  /* => view to get parameters from */
    DPoint3d     *centerPoint,  /* => point to zoom about */
    double       zoomFactor     /* => zoom ratio */
);
```

Description The `mdlView_zoom` function changes the view's viewing extents indicated by *inViewNumber* by a scale factor given by *zoomFactor*. It copies the resulting view into *outViewNumber*. The center for the new view is the point given by *centerPoint* in the current coordinate system.

If *zoomFactor* is greater than 1.0, the effect is to zoom out. If *zoomFactor* is less than 1.0, the effect is to zoom in. A *zoomFactor* of 1.0 can be used to center the window.



inViewNumber and *outViewNumber* can be the same.

After calling `mdlView_zoom`, you need to update the view to see the changed view extents using the `mdlView_updateSingle` function.

Returns The `mdlView_zoom` function returns `SUCCESS` if the view extents are changed and `MDLERR_BADVIEWNUMBER` or `MDLERR_VIEWNOTDISPLAYED` if *inViewNumber* or *outViewNumber* is invalid or not displayed.

See Also `mdlView_updateSingle`.

mdlView_setDisplayDepth, mdlView_setDisplayDepthPoints

```
#include <mdl.h>

int mdlView_setDisplayDepth
(
    int    viewNumber,    /* => view to set display depth for */
    double depth1,        /* => front display depth */
    double depth2         /* => back display depth */
);

int mdlView_setDisplayDepthPoints
(
    int    viewNumber,
    Dpoint3d*points
);
```

Description The `mdlView_setDisplayDepth` and `mdlView_setDisplayDepthPoints` functions change the front and back clipping planes for view *viewNumber*. All parameters are given in the current coordinate system.

The `mdlView_setDisplayDepth` function uses the values *depth1* and *depth2* to define the positions of the front and back clipping planes. These values are signed distances from the current front clipping plane.

The `mdlView_setDisplayDepthPoints` function uses an array of two points that define the front and back clipping planes. The address of the array is given by *points*.



After `mdlView_setDisplayDepth` or `mdlView_setDisplayDepthPoints` is called, the view needs to be updated with the `mdlView_updateSingle` function so the changed clipping planes can be seen.

Returns The `mdlView_setDisplayDepth` and `mdlView_setDisplayDepthPoints` functions return `SUCCESS` if the view clipping planes are changed, `MDLERR_BADVIEWNUMBER` if *viewNumber* is invalid, and `MDLERR_VIEWNOTDISPLAYED` if the view is not currently displayed. If the resulting display depth is less than one UOR, the functions return -1.

See Also `mdlView_updateSingle`.

mdlView_setActiveDepth, mdlView_setActiveDepthPoint

```
#include <mdl.h>

int mdlView_setActiveDepth
(
    int      viewNumber,    /* => view to set active depth for */
    double   depth         /* => active depth distance */
);

int mdlView_setActiveDepthPoint
(
    int      viewNumber,    /* => view to set active depth for */
    Dpoint3d *point        /* => point on active depth plane */
);
```

Description The `mdlView_setActiveDepth` and `mdlView_setActiveDepthPoint` functions change the active depth for view *viewNumber*. The parameters are given in the current coordinate system.

The `mdlView_setActiveDepth` function uses the value *depth* to define the position of the active depth plane. This value is a distance from the front clipping plane. *depth* must be greater than zero and less than the current display depth.

The `mdlView_setActiveDepthPoint` function uses the point *point* to define the active depth plane.

Returns The `mdlView_setActiveDepth` and `mdlView_setActiveDepthPoint` functions return `SUCCESS` if the active depth plane is changed, `MDLERR_BADVIEWNUMBER` if *viewNumber* is invalid, and `MDLERR_VIEWNOTDISPLAYED` if the view is not currently displayed.

See Also `mdlView_setDisplayDepth`.

mdlView_rotateToRMMatrix

```
#include <mdl.h>

int mdlView_rotateToRMMatrix
(
    RotMatrix    *rMatrix,      /* => rotation matrix */
    int          viewNumber     /* => view to rotate */
);
```

Description The mdlView_rotateToRMMatrix function rotates the view *viewNumber* to the rotation matrix given by *rMatrix*.

The view is rotated around its center point and the view extents are unchanged.



After mdlView_rotateToRMMatrix is called, the view needs to be updated with the mdlView_updateSingle function so the changed view extents can be seen.

Returns The mdlView_rotateToRMMatrix function returns SUCCESS if the view is changed, MDLERR_BADVIEWNUMBER if *viewNumber* is invalid, and MDLERR_VIEWNOTDISPLAYED if the view is not currently displayed.

See Also mdlView_updateSingle.

mdlView_getParameters

```
#include <mdl.h>

int mdlView_getParameters
(
    Dpoint3d    *origin,        /* <= view org (lower left back) */
    Dpoint3d    *center,        /* <= view center point */
    Dpoint3d    *delta,         /* <= view extents */
    RotMatrix    *rMatrix,      /* <= rotation matrix */
    double      *activeZ,       /* <= active Z */
    int          viewNumber     /* => view to return info for */
);
```

Description mdlView_getParameters extracts the view settings from view *viewNumber*.

All settings other than *viewNumber* are returned in the current coordinate system. If any settings are NULL, MicroStation does not attempt to fill them in.

origin points to a Dpoint3d to receive the view's origin. The view origin is the point on the back clipping plane's lower left corner.

center points to a Dpoint3d to receive the view's center, which is the center of the volume contained by the front and back clipping planes.

delta points to a `Dpoint3d` to receive the size of the view in the view's X, Y and Z directions.

rMatrix points to a `RotMatrix` to receive the view's rotation matrix.

activeZ points to a `double` to receive the active depth. Active depth is defined as the positive distance from the back clipping plane to the active depth plane.

Returns The `mdlView_getParameters` function returns `SUCCESS` if the information requested is valid. `MDLERR_BADVIEWNUMBER` is returned if *viewNumber* is invalid (not in the range 0-7). `MDLERR_VIEWNOTDISPLAYED` if *viewNumber* is valid, but the view is not currently displayed.

mdlView_getCamera

```
#include <mdl.h>

int mdlView_getCamera
(
    Dpoint3d    *position,      /* <= camera position */
    Dpoint3d    *target,       /* <= camera target */
    double      *angle,        /* <= camera angle */
    double      *focalLength,  /* <= camera focal length in mm */
    int         *lensNumber,    /* <= camera lens */
    int         viewNumber     /* => view to extract settings from */
);
```

Description The `mdlView_getCamera` function extracts camera settings from view *viewNumber*:

All settings other than *viewNumber* are returned in the current coordinate system. If any settings are `NULL`, MicroStation does not attempt to fill them in.

position points to a `Dpoint3d` to receive the camera position.

target points to a `Dpoint3d` to receive the camera target.

angle points to a `double` to contain the camera field of vision angle in degrees.

focalLength and *lensNumber* are alternative ways to express the camera angle in parameters that correspond to the standard 35 millimeter [mm] camera. *focalLength* points to a `double` to contain the focal length in millimeters and *lensNumber* points to an integer specifying standard 35 mm lenses from the following table:

Lens Number	Name	Angle	FocalLength [mm]
0	FishEye	93.3	20
1	ExtraWide	74.3	28

Lens Number	Name	Angle	FocalLength [mm]
2	Wide	62.4	35
3	Normal	46.0	50
4	Portrait	28.0	85
5	Telephoto	12.1	200
6	Telescopic	2.4	1000
7	Custom	n/a	n/a

MicroStation allows 3D design files to be viewed using either a parallel (orthographic) or perspective projection. The perspective projection mimics a physical camera in which light rays pass through a lens, converge at a focal point and focus on an image plane. The distance from the camera to the image plane is referred to as the focal length.

In orthographic (non-camera) views, the view is specified by a transformation matrix to the view coordinate system and a rectangular clipping volume. In camera views, the view volume is a pyramid with the apex at the camera and the cross section increasing with distance from the camera. In specifying a camera view, it is more convenient to think in terms of the camera location, orientation, focal length and fore and aft clipping planes. Camera views are derived from the view settings, the camera position, and the camera focal length in the following manner: The view transformation matrix specifies the camera orientation; the virtual camera points along the negative view Z-axis; the camera X and Y axis coincide with view axes. The front and back clipping planes are identical to the clipping planes for an orthographic view, and the camera image plane is parallel to the view and is offset from the camera position by the focal length. The portion of the image displayed is determined by the intersection of the orthographic view volume with the camera image plane.

Several important aspects of MicroStation's camera views should be noted:

Obviously, it is not possible to display items that are behind the camera, and items that are very close to the camera also cause problems as they end up being magnified excessively. MicroStation therefore truncates views at a fixed fraction of the focal length (approximately 1/50), effectively ignoring the tip of the viewing pyramid.

Unlike most conventional cameras, the camera location is not necessarily centered in the view. This makes it possible to set the camera position and then "pan" within the image. MicroStation's window, pan and zoom commands do not alter the camera settings, instead altering only the view settings to control the portion of the image plane being displayed. This is analogous to taking a picture and then panning, zooming or windowing within the photograph. This makes it possible to simulate the output of a

bellows camera by selecting a camera position that produces the desired perspective and then windowing to the desired portion of the image plane.

If the camera position has never been set, the camera information is invalidated by initializing the focal length to -1. If a negative focal length is encountered when a camera view is updated, a camera position centered in the view with the image plane centered between the front and back clipping planes is calculated automatically. When the FIT command is executed, the current camera position is invalidated by negating the focal length and the update automatically re-centers the camera.

The camera viewing volume does not coincide with orthographic viewing volume. The viewing area coincides at the image plane only, with the camera viewing area smaller than the orthographic view area at depths in front of the image plane and larger at depths beyond the image plane.

The camera lens angle is stored for each view, but is not necessary for the camera specification and is never used during an update. The angle is used for automatically calculating the focal length from the view extents when a camera view is first updated or the SET CAMERA command is executed.

Returns The `mdlView_getCamera` function returns `SUCCESS` if the requested information is valid, `MDLERR_BADVIEWNUMBER` if *viewNumber* is invalid, and `MDLERR_VIEWNOTDISPLAYED` if the view is not displayed.

mdlView_getStandardCameraLens

```
int mdlView_getStandardCameraLens
(
double *angle,          /* <= lens angle in radians */
double *focalLength,    /* <= focal length of lens (mm) */
int     lensNumber      /* => standard lens number */
);
```


Description The mdlView_getStandardCameraLens function returns the camera angle and focal length for the standard camera lenses supported by MicroStation. The standard lens is specified by *lensNumber* in the following manner:

Lens Number	FocalLength [mm]	Name
0	20	FishEye
1	28	ExtraWide
2	35	Wide
3	50	Normal
4	85	Portrait
5	200	Telephoto
6	1000	Telescopic

Returns mdlView_getStandardCameraLens returns SUCCESS if a valid *lensNumber* is specified and ERROR otherwise.

See Also mdlView_setCameraLens, mdlView_getCamera.

mdlView_setCameraLens

```
int mdlView_setCameraLens
(
double angle,          /* => lens angle in radians */
int viewNumber         /* => view number */
);
```

Description The mdlView_setCameraLens function sets the camera lens angle for the view specified by *viewNumber* to the angle in radians specified by *angle*.

Returns mdlView_setCameraLens returns SUCCESS if the lens angle is set successfully and an appropriate error status otherwise.

See Also mdlView_getStandardCameraLens, mdlView_getCamera.

mdlView_getCameraParameters [mdl.lib.mli]

```

#include <mdl.h>
#include <mdl.lib.fdf>

int mdlView_getCameraParameters
(
    DPoint3d      *positionP,          /* <= camera position */
    DPoint3d      *targetP,           /* <= camera target */
    DPoint3d      *upVectorP,         /* <= up vector */
    double        *angleP,            /* <= field of view angle */
    Dvector2d     *screenWindowP,     /* <= screen window */
    double        *frontClipDistanceP, /* <= front clipping distance */
    double        *backClipDistanceP, /* <= back clipping distance */
    int           viewNumber          /* => view index */
);

```

Description The `mdlView_getCameraParameters` function returns the camera parameters for the MicroStation camera specified by *viewNumber*. If any of the parameters are not required, NULL may be specified for the parameter pointer.

The camera position is returned in *positionP* and the camera target is returned in *targetP*. The camera direction is always from *positionP* towards *targetP*, which coincides with the negative Z-Axis for the view.

MicroStation's image plane is always coincident with the camera target. Therefore, the target appears the same size in a camera (perspective) or non-camera (orthographic) view.

The view rotation about the camera axis is determined by *upVectorP*. This is a normalized direction vector that defines the y-axis for the view. *upVectorP* is always perpendicular to the camera axis.

The camera field of view angle (in radians) is returned in *angleP*. The camera angle can be derived from the maximum of the view width and height and the distance from the camera to the target in the following manner:

```
cameraAngle = 2 * arctan(maxDimension/(2*targetDistance));
```

The screen window defines the portion of the image plane that is displayed. The screen window coordinates are defined in an image plane coordinate system with an origin at the camera target. For a centered view, the screen window coordinates will be (-width/2, -height/2) (width/2, height/2). This corresponds the design of most modern cameras as the viewing direction is perpendicular to the camera plane. In an uncentered window, the image plane is tilted in relation to the viewing direction. This corresponds to a view, or bellows camera.

frontClipDistance and *backClipDistance* represent the distance from the camera to the front and back clipping planes respectively.

Returns `mdlView_getCameraParameters` returns SUCCESS if the view is successfully defined. Possible error codes include:

MDLERR_BADVIEWNUMBER

See Also mdlView_setCameraParameters [mdl.lib.mdl].

mdlView_setCameraParameters [mdl.lib.mdl]

```
#include <mdl.h>
#include <mdl.lib.fdf>

int mdlView_setCameraParameters
(
    DPoint3d      *positionP,          /* <= camera position */
    DPoint3d      *targetP,           /* <= camera target */
    DPoint3d      *upVectorP,         /* <= up vector */
    double        *angleP,            /* <= camera angle (radians) */
    double        *aspectRatioP,      /* <= view aspect ratio */
    Dvector2d     *screenWindowP,     /* <= screen window */
    double        *frontClipDistanceP, /* <= front clipping distance */
    double        *backClipDistanceP, /* <= back clipping distance */
    int           viewNumber          /* => view number */
);
```

Description mdlView_setCameraParameters sets the camera parameters for the view specified by *viewNumber*. As the camera parameters are not independent, it is possible to omit many of the camera parameter arguments. If NULL is passed for an argument, it is defaulted to the existing value or calculated from the other parameters as described below.

The camera position is specified by *positionP*, the camera target is specified by *targetP*. If NULL is passed for either of these parameters, the camera position or target is unchanged. The camera direction is always from *positionP* towards *targetP*, this coincides with the negative Z-Axis for the view. MicroStation's image plane is always coincident with the camera target, therefore, the target appears the same size in a camera (perspective) or non-camera (orthographic) view.

The view rotation about the camera axis is determined by *upVectorP*, this is a normalized direction vector that defines the y-axis for the view. It is always perpendicular to the camera axis. If NULL is passed for *upVectorP*, the camera will be oriented such that the camera horizontal is parallel to the view X-Y plane.

The camera field of view angle (in radians) is specified by *angleP*. If NULL is passed, the camera angle is unchanged. The camera angle can be derived from the maximum of the view width and height and the distance from the camera to the target in the following manner:

$$\text{cameraAngle} = 2 * \arctan(\text{maxDimension} / (2 * \text{targetDistance}));$$

The camera field of view angle is specified by *angleP*. The camera angle can be derived from the maximum of the view width and height and the

distance from the camera to the image plane. If `NULL` is passed, the camera angle is unchanged.

aspectRatioP specifies the view aspect ratio. This represents the view height divided by the view width. If `NULL` is passed, the aspect ratio of the view window is used.

The screen window coordinates are specified by *screenWindowP*, they define the portion of the image plane that is displayed. The screen window coordinates are defined in an image plane coordinate system with an origin at the camera target. For a centered view, the screen window coordinates will be $(-width/2, -height/2)$ $(width/2, height/2)$. This corresponds the design of most modern cameras as the viewing direction is perpendicular to the camera plane. In an uncentered window, the image plane is tilted in relation to the viewing direction. This corresponds to a view, or bellows camera. If `NULL` is defined, a centered window is calculated with the size derived from the view angle and distance to the image plane.

frontClipDistance and *backClipDistance* represent the distance from the camera to the front and back clipping planes respectively. If `NULL` is passed for *frontClipDistanceP*, the front clipping distance is arbitrarily set to the target distance divided by 300. If `NULL` is passed for *backClipDistanceP*, the back clipping distance is set to the ten times the target distance.

Returns `mdlView_setCameraParameters` returns `SUCCESS` if the view is successfully defined. Possible error codes include:

`MDLERR_BADVIEWNUMBER`

See Also `mdlView_setCameraParameters` [`mdl1lib.m1`].

mdlView_cameraLensLengthFromAngle, mdlView_cameraLensAngleFromLength **[mdl1lib.m1]**

```
double mdlView_cameraLensLengthFromAngle
(
double angle          /* => Angle (radians) */
);

double mdlView_cameraLensAngleFromLength
(
double lensLength     /* => length (mm) */
);
```

Description The `mdlView_cameraLensLengthFromAngle` function returns the length in millimeters of a camera lens that corresponds to the field of vision angle, *angle* radians. This length is useful to users who relate to lens lengths rather than angles.

Returns The `mdlView_cameraLensAngleFromLength` function returns the field of vision angle in radians that corresponds to a camera lens of length, *lensLength* millimeters.

See Also `mdlView_setCameraParameters` [`mdl1lib.m1`].

mdlView_updateSingle

```
#include <mdl.h>

int mdlView_updateSingle
(
    int    viewNumber    /* => number of view to update */
);
```

Description The `mdlView_updateSingle` function repaints the view referenced by *viewNumber*, using the current settings. This function returns only after the entire view is displayed.

`mdlView_updateSingle` often needs to be called after a view's settings are changed.

Returns `mdlView_updateSingle` returns `SUCCESS` if the view is updated, `MDLERR_BADVIEWNUMBER` if *viewNumber* is invalid, and `MDLERR_VIEWNOTDISPLAYED` if the view is not currently displayed.

See Also `mdlView_updateMulti`, `mdlView_renderSingle`.

mdlView_updateMulti

```
#include <mdl.h>

void mdlView_updateMulti
(
    short  viewList[],    /* => array of view draw flags */
    boolean incremental,  /* => flag for incremental update */
    int    drawMode,      /* => drawing mode */
    ULONG  *fileMask,     /* => files to update */
    boolean startEndMsg   /* => display start and end messages */
);
```

Description The `mdlView_updateMulti` function repaints the views indicated by a non-zero entry in the *viewList* array. The *viewList* array is dimensioned to `MAX_VIEWS`, and *viewList[i]* controls whether view *i* is drawn. If *incremental* is set to `TRUE`, the views are not cleared before the update, and the portion of the master file before the working window designated by *tcb->wwsect*, *tcb->wwbyte* is not drawn.

Possible values for *drawMode* are `NORMALDRAW`, `ERASE` and `HILITE`. These values draw the elements normally, erase elements from the screen, and highlight elements.

fileMask points to an array of eight long integers. Bits control elements as follows:

- The lowest bit (bit 0) of the first long integer controls whether elements from the master file are included in the fit.
- Bit 1 controls whether elements from the reference file with attachment number 1 are included.

- Bit 0 of the second long integer controls whether elements from the reference file with attachment number 32 are included.
- The highest bit (bit 31) of the eighth long integer controls whether elements from the reference file with attachment number 255 are included.

If *startEndMsg* is *TRUE*, MicroStation displays the “Update in progress” and “Display complete” messages before and after the update. This function returns only after the entire view displays.

Returns *mdlView_updateMulti* is of type *void*. It returns no value.

mdlView_updateMultiExtended

```
#include <update.h>
#include <scanner.h>

void mdlView_updateMultiExtended
(
    short          viewList[],    /* => array of view draw flags */
    boolean        incremental,   /* => flag for incr update */
    int            drawMode,      /* => drawing mode */
    ULONG          *fileMask,     /* => file mask */
    boolean        startEndMsg,   /* => display start & end msgs */
    Update_scanmod *scanMods     /* => more selection criteria */
);
```

Description *mdlView_updateMultiExtended* provides all the flexibility of *mdlView_updateMulti*, with the additional ability to modify the scan list that MicroStation uses for the update. The additional parameter, *scanMods*, points to a structure that defines the scan list modifications. This structure is declared in *update.h* as follows:

```
/* structure for scanlist modification in update */
typedef struct
{
    short          scantype;      /* scantype or'ed with std */
    short          propval;       /* properties value to set */
    short          propmsk;       /* properties mask to set */
    unsigned short grgroup;       /* graphics group to scan */
    unsigned short sector;        /* sector at start */
    short          offset;        /* byte at start */
    unsigned short entity;        /* entity value */
};
```

```

short      setclasmk;          /* class mask to set */
short      clrclasmk;          /* class mask to clear */
short      setlevmask[8];      /* level masks to set */
short      clrlevmask[8];      /* level masks to clear */
short      settypmask[8];      /* type masks to set */
short      clrtypmask[8];      /* type masks to clear */
long       occurance;          /* occurrence number to set */
unsigned shortstopsector;      /* stop on this sector */
} Update_scanmod;

```

The *scantype* member is OR'ed with the *scantype*, in the scanlist, that the update would normally use. The other members of the structure are described above. An example usage would be to update only level 5. To do so, initialize the scanMod structure to all zeros, and then set *scantype* to LEVELS, *clrlevmask* to all 1's, and *setlevmask*[0] to 0x0010. MicroStation will clear all levels it would ordinarily update, and then set level 5 to be updated.

Returns mdlView_updateMultiExtended is of type void. It does not return a value.

mdlView_renderSingle

```

int mdlView_renderSingle
(
int          viewNumber,          /* => view number */
int          renderMode,          /* => render mode */
boolean      antiAlias,           /* => TRUE = antialiasing */
boolean      stereo,              /* => TRUE = stereo display */
boolean      disableRangeChecking /* => TRUE = disable range */
);

```

Description The mdlView_renderSingle function is used to render a single view, *viewNumber*.

renderMode is the rendering mode. The rendering modes (SMOOTH, PHONG, CONSTANT, etc.) are defined in msdefs.h.

If *antialias* is non-zero, the view is antialiased. If *stereo* is non-zero, the view is rendered in stereo mode.

If *disableRangeChecking* is non-zero, all design file geometry is rendered even if it does not overlap the view. This should be set to FALSE for most applications.

Returns mdlView_renderSingle returns SUCCESS if the view is rendered successfully. MDLERR_BADVIEWNUMBER is returned if the view number is invalid and MDLERR_INSUFFICIENTMEMORY is returned if a memory error occurs.

See Also mdlView_setRenderMode, mdlView_updateSingle.

mdlView_setRenderMode

```
void mdlView_setRenderMode
(
  int      viewNumber,    /* => view number */
  int      renderMode,    /* => render mode */
  boolean   textures,     /* => TRUE = enable texture mapping */
  boolean   shadows,      /* => TRUE = enable shadows (phong only) */
  boolean   haze,         /* => TRUE = enable haze */
  boolean   transparency, /* => TRUE = enable transparency */
  boolean   slowDither    /* => TRUE = enable slow dithering */
);
```

Description The `mdlView_setRenderMode` function sets the rendering mode for the view specified by *viewNumber*. `mdlView_setRenderMode` can be used to change the rendering mode temporarily to display rendered elements in a wireframe view.

renderMode is the rendering mode. The rendering modes (SMOOTH, PHONG, CONSTANT etc.) are defined in `msdefs.h`.

If the *textures*, *shadows*, *haze*, *transparency* or *slowDither* parameters are non-zero, the texture mapping, shadow mapping, haze, transparency and delayed display attributes are enabled for the view.

Returns `mdlView_setRenderMode` returns `SUCCESS` if the render mode is successfully changed. `MDLERR_BADVIEWNUMBER` is returned if the view number is invalid and `MDLERR_INSUFFICIENTMEMORY` is returned if a memory error occurs.

See Also `mdlView_renderSingle`.

mdlView_hiddenLineRemoval

```
int mdlView_hiddenLineRemoval
(
  MdlFunctionP preFunction,    /* => called before processing elems */
  MdlFunctionP visibleFunction, /* => called for visible elms */
  MdlFunctionP invisibleFunction, /* => called for invisible elms */
  int      viewNumber,        /* => view (0-7) */
  ULong    *fileMaskP,        /* => file mask */
  double    tolerance,        /* => tolerance */
  boolean   outputTo3D,        /* => TRUE=create 3D output file */
  boolean   ruleLines,         /* => TRUE=include ruleLines */
  boolean   calculateIntersects, /* => TRUE=surf.-surf. intersects */
  boolean   viewOnly,          /* => TRUE=process view content only */
  boolean   fenceOnly          /* => TRUE=process fenced area only */
);
```

Description The `mdlView_hiddenLineRemoval` function performs hidden line removal on the view specified by *viewNumber*. The *visibleFunction* is called with all visible geometry and the *invisibleFunction* is called for geometry that is not visible. Before each element is processed the *preFunction* is called to allow the calling application

to preprocess or reject processing for elements. The prototypes for these functions are listed below.

fileMaskP points to an array of eight long integers. Bit 0 of the first long controls the processing of the master file. Bit 1 controls the first reference file. If *NULL* is passed, all reference files are processed.

tolerance determines the maximum error for intersection calculations. In most cases it is best to set this parameter to zero.

outputTo3D controls whether the elements are converted to 2D before being passed to the *visibleFunction* or *invisibleFunction*.

ruleLines controls whether rule lines are processed for surfaces. If *ruleLines* is zero, only surface silhouettes and edges are processed.

calculateIntersects controls whether intersections between surfaces are calculated. Enabling intersection calculations can significantly increase processing time.

viewOnly controls the entire design file is processed or elements are clipped to the view boundaries.

fenceOnly controls whether the fence contents are processed. If this is non-zero, there must be a fence active in the view specified by *viewNumber*.

Returns mdlView_hiddenLineRemoval returns *SUCCESS* if hidden line removal is completed successfully. An appropriate error status is returned otherwise.

See Also mdlElmdscr_hiddenLineRemoval, Hidden Line Removal User Functions, Exact Hidden Line Removal Functions, mdlView_hiddenLineRemoval2, mdlElmdscr_hiddenLineRemoval2.

Hidden Line Removal User Functions

The hidden line removal user functions, *preFunction*, *visibleFunction* and *invisibleFunction* must *not* add elements to a file or display elements. This will cause infinite processing to occur. If elements are to be added to a file or displayed, they should be added to an element descriptor and displayed or added to the file after the hidden line removal is complete.

preFunction

```
#include <userfnc.h>

int preFunction
(
MSElementDescr    *edP,           /* => element to be processed */
ExtraElementInfo    *elementInfoP /* => information on element */
);
```

edP points to an element descriptor containing the element that is about to be processed.

elementInfoP points to a structure containing the file position and file number of the element. The definition of this structure is in `userfnc.h`.

Returns Returning a non-zero status causes the element to be ignored.

visibleFunction, invisibleFunction

```
int visibleFunction
(
  MSElementDescr *edP3d,      /* => 3D element */
  MSElementDescr *edP2d,      /* => 2D element (if not outputTo3d) */
  MSBsplineCurve *curveP,      /* => B-spline curve */
  double          startParam,   /* => start parameter on curveP */
  double          endParam,     /* => end parameter on curveP */
  int             fileNumber    /* => file number */
);

int invisibleFunction
(
  MSElementDescr *edP3d,      /* => 3D element */
  MSElementDescr *edP2d,      /* => 2D element (if not outputTo3d) */
  MSBsplineCurve *curveP,      /* => B-spline curve */
  double          startParam,   /* => start parameter on curveP */
  double          endParam,     /* => end parameter on curveP */
  int             fileNumber    /* => file number */
);
```

edP3d represents the visible (or invisible) element. If *outputTo3D* is non-zero, *edP2d* will point to a two dimensional representation of the same element, otherwise it will be the same as *edP3d*.

curveP points to a B-spline curve representation of the element. The visible (or invisible) portion of the element is between *startParam* and *endParam*.

fileNumber contains the file number for the element being processed.

Returns The return status from *visibleFunction* and *invisibleFunction* is ignored.

Exact Hidden Line Removal Functions

The exact hidden line removal code has been completely redeveloped for MicroStation SE. The new version contains significant improvements in both quality and processing speed.

- Automatic removal of coincident or “stacked” edges. If one or more edges are on top of each other, only the one closest to the eye is exported. This greatly reduces output file size and improves output quality. This filtering can be performed separately (without hidden line removal) by selecting the “Trace Edges Only” method from the main dialog box.
- Optimized handling of completely hidden geometry. Geometry that is completely hidden is now detected and processed efficiently. Small detail geometry that is completely obscured (i.e., the coat hangers in the closet of a large building) has very little effect on total processing time.
- Improved handling of simple geometry. Exact solutions are now used for many of the simpler geometric forms (i.e., cones, cylinders, torii, etc.) that are typically created by extruding or revolving arcs and lines. This results in much higher quality and significantly faster processing.

Using Enhanced Hidden Line Removal in MDL Applications

Whereas in MicroStation95, the hidden line removal code resided in the MicroStation kernel, the exact hidden line removal code now exists in its own dynamic link module (truehide.dl*). This makes it possible to replace or update this component without changing the main MicroStation executable. It also makes it possible to deliver the new capabilities to MicroStation 95 users, (the SE dynamic link module is MicroStation 95 compatible). In order to link use functions from the truehide DLM, you will need to link with truehide.dlo (delivered).

There are two functions in this DLM that are enhanced versions of the hidden line removal built-in functions (mdlView_hiddenLineRemoval and mdlElmdscr_hiddenLineRemoval) available for MicroStation 95. These functions (mdlView_hiddenLineRemoval2 and mdlElmdscr_hiddenLineRemoval2) are prototyped in the new include file, mshline.fdf and have additional arguments that control the enhanced features of the new code.

curveHiding: Set *curveHiding* to TRUE to remove duplicate, stacked curves. In most cases this should be set to TRUE.

surfaceHiding: Set this to TRUE to have perform hidden line removal. If only duplicate edge removal (Trace) is desired, this would be FALSE, but in most instances this should be set to TRUE.

zBufferCulling: In the new algorithm a ZBuffer is used to automatically reject hidden geometry. This is much faster when there is a significant amount of geometry, but is unnecessary overhead if the amount of geometry is small.

zBufferResolution: The resolution in pixels of the ZBuffer if *zBufferCulling* is TRUE. A good resolution value is 1024.

mdlView_hiddenLineRemoval2

```

nativeCode StatusInt mdlView_hiddenLineRemoval2
(
MdlFunctionP    preFunction, /* => func called before processing elems */
MdlFunctionP    visibleFunction, /* => func called for visible elems */
MdlFunctionP    invisibleFunction, /* => func called for invis. elems */
int             viewNumber,      /* => view (0-7) */
ULong           *fileMaskP,      /* => file mask */
double          tolerance,       /* => tolerance */
BoolInt         outputTo3D,      /* => TRUE - create a 3D output file */
BoolInt         ruleLines,       /* => TRUE - include surf ruleLines */
BoolInt         calculateIntersections, /* => TRUE for surf-surf intrscts */
BoolInt         viewOnly,        /* => TRUE - process view contents only */
BoolInt         fenceOnly,       /* => TRUE - process fenced area only */
BoolInt         curveHiding,     /* => TRUE for curve hiding */
BoolInt         surfaceHiding,   /* => TRUE for surface hiding */
BoolInt         zBufferCulling,  /* => TRUE - use ZBuffer for culling */
BoolInt         zBufferResolution /* => Z-Buffer resolution (for
                                occlusion culling) */
);

```

mdlElmdscr_hiddenLineRemoval2

```

nativeCode int mdlElmdscr_hiddenLineRemoval2
(
MdlFunctionP    preFunction, /* => func called before processing elems */
MdlFunctionP    visibleFunction, /* => func called for visible elems */
MdlFunctionP    hiddenFunction, /* => func called for hidden elems */
MSElementDescr *inEdP,        /* => input elements */
int             viewNumber,     /* => view (0-7) */
double          tolerance,      /* => tolerance */
BoolInt         outputTo3D,     /* => TRUE - create 3D output file */
BoolInt         ruleLines,      /* => TRUE - include surf ruleLines */
BoolInt         calculateIntersections, /* => TRUE - surf-surf intersects */
BoolInt         curveHiding,    /* => TRUE for curve hiding */
BoolInt         surfaceHiding,  /* => TRUE for surface hiding */
BoolInt         zBufferCulling, /* => TRUE - use ZBuffer for culling */
BoolInt         zBufferResolution /* => Z-Buffer resolution (for
                                occlusion culling) */
);

```

mdlView_getStandard, mdlView_isStandard

```
#include <mdl.h>

void mdlView_getStandard
(
    RotMatrix    *rMatrix,      /* => rotation matrix for given view */
    int          stdViewType    /* => standard view number */
);
int mdlView_isStandard
(
    RotMatrix    *rMatrix      /* => rotation matrix to test */
);
```

Description mdlView_getStandard sets the RotMatrix pointed to by *rMatrix* to the rotation matrix for the standard view indicated by *stdViewType*. Possible values for *stdViewType* are defined in mdl.h and include STDVIEW_TOP, STDVIEW_BOTTOM, STDVIEW_LEFT, STDVIEW_RIGHT, STDVIEW_FRONT, STDVIEW_BACK and STDVIEW_ISO.

The mdlView_isStandard function indicates whether a rotation matrix matches a standard view rotation matrix. This function helps identify whether a view is currently displaying a standard view.

Returns The mdlView_getStandard function is of type void. It returns no value. The mdlView_isStandard function returns the standard view number defined in mdl.h or -1 if *rMatrix* does not match a standard view.

mdlView_applyNamed

```
int mdlView_applyNamed
(
    ViewInfo      *viewInfo,      /* => view information */
    ExendedViewInfo *extViewInfo, /* => extended view info */
    int           viewNumber      /* => view to set */
);
```

Description mdlView_applyNamed is used to apply information returned from mdlView_findNamed to the indicated view. A common usage is to retrieve named view information, make some modification to it, and then apply the modified version to a view.

Returns mdlView_applyNamed returns SUCCESS if the operation completes successfully. It returns MDLERR_BADVIEWNUMBER if the view number is out of range (0-7 are valid), MDLERR_VIEWNOTDISPLAYED if the view designated is not currently displayed.

mdlView_findNamed, mdlView_attachNamed

```

#include <mdl.h>
#include <tcb.h>

int mdlView_findNamed
(
    ViewInfo          *viewInfo,      /* <= view information */
    ExtendedViewInfo *extViewInfo,    /* <= extended view information */
    ULong             *filePos,       /* <= file position */
    char              *viewName       /* => saved view name */
);
void mdlView_attachNamed
(
    char    *viewName,    /* => view information */
    int     viewNumber    /* => view to set */
);

```

Description mdlView_findNamed searches the design file for a saved view with the name *viewName*. It fills the ViewInfo structure pointed to by *viewInfo* with the viewing information from the saved view. It then sets *filePos* to the saved view's file position.

Views that are saved with MicroStation version 4.0 or later contain extended view information that includes an extended view flags word and the information for generating camera views (3D only). If this data is present, it is placed in the ExtendedViewInfo structure pointed to by *extViewInfo*.

extViewInfo is optional; NULL can be passed to ignore the extended view information.

The mdlView_attachNamed function attaches the named view specified by *viewName* to view *viewNumber*.

This function matches the aspect ratio to the view's aspect ratio, ensures that the active level is turned on, and clears the active fence in 3D files, if necessary.

Returns The mdlView_findNamed function returns SUCCESS if a saved view is located. Otherwise, it returns MDLERR_VIEWNOTFOUND.

The mdlView_attachNamed function returns SUCCESS if the view is successfully attached and MDLERR_VIEWNOTFOUND if the view cannot be located.

mdlView_saveNamed, mdlView_deleteNamed

```
#include <mdl.h>

boolean mdlView_saveNamed
(
    char    *viewName,      /* => name for saved view */
    char    *description,   /* => named view description */
    int     *viewNumber     /* => view to get information from */
);
int mdlView_deleteNamed
(
    char    *viewName       /* => name of saved view to delete */
);
```

Description The `mdlView_saveNamed` function saves the information from the view specified by *viewNumber* to the design file in a saved view with the name *viewName*. The description of the view contents specified by *description* is saved with the saved view information. The *viewName* is limited to six characters, and the *description* is limited to 27 characters. If a saved view of the same name already exists, it is overwritten with the new definition. It is possible (but usually not advised) to override MicroStation's standard view names (TOP, BOTTOM, LEFT, RIGHT, FRONT, BACK and ISO).

The `mdlView_deleteNamed` function attempts to locate the saved view specified by *viewName*. If the saved view is found, it is deleted.

Returns The `mdlView_saveNamed` function returns `FALSE` if it replaced a previous definition or `TRUE` if there was no saved view with the same name before the call.

The `mdlView_deleteNamed` function returns `SUCCESS` if it could delete the specified named view. Otherwise, it returns `MDLERR_VIEWNOTFOUND`.

mdlView_getContentCorners

```
void mdlView_getContentCorners
(
    Point3d    *ptLow,      /* <= upper left back */
    Point3d    *ptHigh,     /* <= lower right front */
    MSWindow   *window,     /* => view window */
    int        reserved     /* => Must be 0! */
);
```

Description `mdlView_getContentCorners` returns the corners of the view rectangle within a window in global (screen) coordinates. The area occupied by the view rectangle may be a subset of the window's content region. If the window specified by *window* is not a view, then the window's content rectangle corners are returned. The *reserved* argument must be set to zero.

Returns This function does not return a value.

See Also `mdlWindow_viewWindowGet`, `mdlWindow_contentRectGetGlobal`.

mdlView_createSavedViewElement [mdl.lib.mli]

```
int mdlView_createSavedViewElement
(
short      nvElement[],          /* <= saved view element */
char       *name,                /* => view name */
char       *description,         /* => view description */
Dpoint3d   *originP,             /* => view origin */
Dpoint3d   *deltaP,              /* => view delta */
RotMatrix   *rotMatrixP,         /* => view rotation matrix */
double     *activeZP,            /* => active depth */
Viewflags   *flagsP,             /* => view flags */
Ext_viewflags *extFlagsP,        /* => extended view flags */
short      *levelsP,             /* => levels */
Dpoint3d   *cameraPositionP,     /* => camera position */
double     *focalLengthP,        /* => focal length */
double     *cameraAngleP,        /* => camera angle */
);
```

Description The `mdlView_createSavedViewElement` function creates a type 5 saved view element.

nvElement points to the element to be created. This buffer should be at least 768 words long.

name points to the view name, with a maximum of six characters. As view names are stored internally in Radix 50 format, the view name must contain only upper case alphanumeric characters.

description points to the view description.

originP points to the view origin in UORS.

deltaP points to the view extents in UORS.

rotMatrixP points to a the view rotation matrix.

activeZP is the view active depth in UORS.

tflagsP points to the view flags. This structure contains view attributes such as line weights and fast text display. If `NULL` is passed, a default set of flags is used.

extFlagsP points to the extended view flags. This structure contains view attributes such as camera and rendering modes.

levelsP points to a bit map for the view level display.

cameraPositionP points to the camera origin. `NULL` may be passed if a camera view is not being saved.

focalLengthP points to the camera focal length (distance from camera to image plane). `NULL` may be passed if a camera view is not being saved.

cameraAngleP points to the camera angle. NULL may be passed if a camera view is not being saved.

Returns mdlView_createSavedViewElement returns SUCCESS if the saved view element is created successfully.

See Also mdlView_extractSavedView.

mdlView_extractSavedView [mdl.lib.mll]

```
int mdlView_extractSavedView
(
    char          *name,           /* <= name */
    char          *description,    /* <= description */
    Dpoint3d      *origin,         /* <= origin */
    Dpoint3d      *delta,          /* <= delta */
    ULong         *activeZ,        /* <= active Z */
    Viewflags     *flagsP,         /* <= flags */
    RotMatrix     *rotMatrixP,     /* <= rotMatrix */
    short         *levelsP,        /* <= levels */
    boolean       *extendedFoundP, /* <= TRUE=extended info present */
    Ext_viewflags *extendedFlagsP, /* <= extended view flags */
    CameraInfo    cameraInfoP,     /* <= camera info */
    short         nvElement[]      /* => element */
);
```

Description The mdlView_extractSavedView function reads a type 5 saved view element.

name points to the view name, with a maximum of six characters (name buffer should be at least 7 characters to accomodate a NULL terminator).

description points to the view description.

originP points to the view origin in UORS.

deltaP points to the view extents in UORS.

activeZP is the view active depth in UORS.

flagsP points to the view flags. This structure contains view attributes such as line weights and fast text display. If NULL is passed, a default set of flags is used.

rotMatrixP points to a the view rotation matrix.

levelsP points to an array of 4 short integers for the view level bitmap.

extendedFoundP points to an integer that is set to TRUE if the saved view information is present in the saved view element. If this flag is not TRUE, the *extendedFlagsP* and *cameraInfoP* are not set.

extendedFlagsP is a pointer to the extended view flags. This structure contains view attributes such as the camera and rendering mode for the view.

cameraInfoP points to a structure containing the camera information for the view. This data is only valid if *extendedFlagsP->camera* is TRUE and *extendedFoundP* is TRUE.

nvElement is points to a type 5 saved view element.

Returns mdlView_extractSavedView returns SUCCESS if the view information is extracted successfully and MDLERR_BADELEMENT if the element is not a saved view type 5.

See Also mdlView_createSavedViewElement.

mdlView_getLevels, mdlView_setLevels

```
#include <mdl.h>

boolean mdlView_getLevels
(
    short   levelMask[], /* <= levels currently on */
    int     viewNumber   /* => view to get information from */
);

void mdlView_setLevels
(
    boolean onOffFlag, /* => turn on or off */
    short   levelMask[], /* => levels to turn on or off */
    short   viewList[], /* => list of views to operate on */
    boolean doUpdate    /* => update affected views */
);
```

Description The mdlView_getLevels function returns information about the levels that are on for the master design file in the view specified by *viewNumber*.

The level information is returned in an array, *levelMask*, of short integers with one bit for each level. The *levelMask* argument should be dimensioned to 4. The lowest bit of *levelMask*[0] is set to 1 if level 1 is on in the given view and to 0 if level 1 is off. Similarly, bit 1 of *levelMask*[0] is set to the state of level 2. The highest bit (15) of *levelMask*[3] represents the state of the special level reserved for cell headers. This bit is always on.

The mdlView_setLevels function turns on or off the levels for the master file in one or more views. The *onOffFlag* argument is set to TRUE to turn levels on and FALSE to turn levels off. The affected levels are specified in the four-element short integer array *levelMask*. The views for which the level information will be changed are specified in the *viewList* argument, which has the same meaning as mdlView_updateMulti. If *doUpdate* is TRUE, the results of changing the levels display immediately.



In releases of MicroStation prior to version 5, mdlView_setLevels contained a bug that was activated when the last argument, *doUpdate*, was set to TRUE. The workaround was to set *doUpdate* to FALSE and use

mdlView_updateSingle or mdlView_updateMulti. This bug has been remedied in MicroStation 5 and later.

Returns The mdlView_getLevels function returns SUCCESS if the level information for the specified view is successfully retrieved, MDLERR_BADVIEWNUMBER if *viewNumber* is invalid, and MDLERR_VIEWNOTDISPLAYED if the view is not currently displayed.

The mdlView_setLevels function is of type void. It returns no value.

mdlView_getDisplayControl, mdlView_setDisplayControl

```
#include <mdl.h>

boolean mdlView_getDisplayControl
(
  int      paramNum,      /* => number of view attribute to get */
  int      viewNumber     /* => view to get view attribute for */
);

void mdlView_setDisplayControl
(
  int      paramNum,      /* => number of view attribute to get */
  int      viewNumber,    /* => view to get view attribute for */
  int      value          /* => value to set view attribute to */
);
```

Description mdlView_getDisplayControl returns information about a view attribute. *viewNumber* specifies the view, and *paramNum* specifies which view attribute is required. Possible values for *paramNum* are defined in mdl.h and are provided with an explanation in the following table:

<i>paramNum</i> value	Description (equivalent MicroStation key-in)	Range
VIEWCONTROL_FAST_CURVE	fast curves (SET CURVE FAST)	TRUE or FALSE
VIEWCONTROL_FAST_TEXT	text display off/on (SET TEXT OFF)	TRUE or FALSE
VIEWCONTROL_LINE_WEIGHTS	line weights (SET WEIGHT ON)	TRUE or FALSE
VIEWCONTROL_PATTERNS	patterns (SET PATTERN ON)	TRUE or FALSE
VIEWCONTROL_TEXT_NODES	text nodes (SET NODES ON)	TRUE or FALSE
VIEWCONTROL_ED_FIELDS	enter data fields (SET ED ON)	TRUE or FALSE
VIEWCONTROL_GRID	grid (SET GRID ON)	TRUE or FALSE
VIEWCONTROL_LEV_SYMB	level symbology (SET LVLSYMB ON)	TRUE or FALSE
VIEWCONTROL_POINTS	display point elements (SET POINT ON)	TRUE or FALSE
VIEWCONTROL_CONSTRUCTION	constructions (SET CONSTRUCTION ON)	TRUE or FALSE

<i>paramNum</i> value	Description (equivalent MicroStation key-in)	Range
VIEWCONTROL_DIMENSIONS	dimensions (SET DIMENSION ON)	TRUE or FALSE
VIEWCONTROL_AREA_FILL	area fill (SET FILL ON)	TRUE or FALSE
VIEWCONTROL_RASTER_TEXT	enable raster text (SET RASTERTEXT ON)	TRUE or FALSE
VIEWCONTROL_AUX_DISPLAY	ACS triad (SET ACSDISPLAY ON)	TRUE or FALSE
VIEWCONTROL_CAMERA	camera (perspective display) (SET CAMERA ON)	TRUE or FALSE
VIEWCONTROL_RENDERMODE	display mode (SET VIEW <i>type</i>)	See below.
VIEWCONTROL_BACKGROUND	display view background (SET BACKGROUND ON)	TRUE or FALSE
VIEWCONTROL_REF_BOUND	reference file boundaries (SET REFBOUND ON)	TRUE or FALSE
VIEWCONTROL_FAST_BOUND_CLIP	fast reference file clipping (SET REFCLIP FAST)	TRUE or FALSE
VIEWCONTROL_DEPTH_CUE	depth cueing (SET DEPTHCUE ON)	TRUE or FALSE
VIEWCONTROL_NO_DYNAMICS	dynamics (SET DYNAMICS OFF)	TRUE or FALSE

For the VIEWCONTROL_RENDERMODE parameter, possible values are defined in mdl.h. These values include VIEWMODE_WIREFRAME, VIEWMODE_CROSSECTION, VIEWMODE_WIREMESH, VIEWMODE_HIDDENLINE, VIEWMODE_FILLEDHLINE, VIEWMODE_CONSTANTSHADE, VIEWMODE_SMOOTHSHADE and VIEWMODE_PHONGSHADE.

The mdlView_setDisplayControl function sets any view attribute listed above. *viewNumber* specifies the view to affect, *paramNum* specifies which view attribute to set, and *value* specifies the new value.

Returns The mdlView_getDisplayControl function returns the requested view attribute if the view number is valid and the view is displayed. Otherwise it returns MDLERR_BADVIEWNUMBER if *viewNumber* is invalid, and MDLERR_VIEWNOTDISPLAYED if the view is not currently displayed. The error conditions return negative numbers.

The mdlView_setDisplayControl function returns SUCCESS if *viewNumber* is valid, MDLERR_BADVIEWNUMBER if *viewNumber* is invalid, and MDLERR_VIEWNOTDISPLAYED if the view is not currently displayed.

mdlView_defaultCursor

```
void mdlView_defaultCursor
(
void
);
```

Description mdlView_defaultCursor is used to cancel any special cursor behavior for MicroStation views and return to the default crosshair cursor. This allows the programmer to cancel screen dynamics without manipulating or clearing state functions.

Returns The mdlView_defaultCursor function is of type void. It returns no value.

See Also mdlState_clear, mdlState_startModifyCommand.

mdlView_refreshGrid

```
void mdlView_refreshGrid
(
  boolean draw      /* => TRUE to draw, FALSE to erase */
);
```

Description The mdlView_refreshGrid function is used to redraw or erase the grid that appears in MicroStation views for which grid display is turned on. The *draw* parameter is set to TRUE to draw the grids and FALSE to erase the grids. Every view that is displayed that has grids displayed is affected.

Returns mdlView_refreshGrid is of type void. It returns no value.

mdlView_isActive, mdlView_isVisible

```
boolean mdlView_isActive
(
  int    viewNumber    /* => view to turn on */
);

boolean mdlView_isVisible
(
  int    viewNumber    /* => view to turn on */
);
```

Description The mdlView_isActive function returns TRUE if the specified view is “active” (turned on). The mdlView_isVisible function returns TRUE if the specified view is “visible” (can be drawn to). On the PC, a view can be active but not visible if it is turned on, but is on the invisible virtual screen of a single screen configuration. On the Intergraph workstation version, views can be drawn to even if they are on the invisible virtual screen, so the two functions will always return the same value.

Returns The mdlView_isActive and mdlView_isVisible functions return the state of the view if *viewNumber* is valid. If *viewNumber* is invalid, they always return FALSE.

mdlView_synchWithTCB

```
void mdlView_synchWithTCB
(
  int    viewNumber    /* => view to synchronize with TCB info */
);
```

Description The `mdlView_synchWithTCB` function synchronizes the private view information that MicroStation maintains separately for each view with the “public” view information that is maintained in the TCB. The reason that the private information is separated from the TCB information is that the private information reflects what is currently drawn on the screen. Elements will be undrawn in the same mode in which they were originally drawn, regardless of the current drawing mode.

For example, suppose that the state of the Fast Text Display is on when an element is originally drawn, and the user subsequently turns Fast Text Display off for a particular view and does not immediately update. Those text elements will be undrawn in Fast display mode until the next update. An application can override this normal behavior using `mdlView_synchWithTCB`. This allows an application to specify a draw mode in the TCB, and force MicroStation to immediately adopt this draw mode for the given view.

The *viewNumber* parameter specifies the view for which the synchronization with the TCB information is to occur.

Returns `mdlView_synchWithTCB` is of type `void`. It returns no value.

mdlView_queuePartialUpdate

```
int mdlView_queuePartialUpdate
(
    int      viewNumber,      /* => view to update */
    BSIRect *rectsP,         /* => list of update rectangles */
    int      numRects         /* => number in list of rectangles */
);
```

Description `mdlView_queuePartialUpdate` allows an application to update one or more subparts of a MicroStation view.

viewNumber specifies the view for which portions are to be updated.

rectsP is the address of an array of rectangles for which updates are to be performed. They must be in global coordinates (i.e., relative to the upper left of the screen).

numRects specifies the number of rectangles in the *rectsP* array. The maximum number of rectangles that can be queued at one time is 15.

If the application wants the updates to happen immediately, the `mdlWindow_windowEventsProcessAll` function should be called following `mdlView_queuePartialUpdate`.

Returns `mdlView_queuePartialUpdate` returns `SUCCESS` if the input parameters are valid, `MDLERR_BADVIEWNUMBER` if *viewIndex* is not between 0 and `MAX_VIEWS`, or `MDLERR_VIEWNOTDISPLAYED` if the view is not currently displayed.

See Also `mdlView_updateSingle`, `mdlView_updateMulti`, `mdlView_updateMultiExtended`.

mdlView_newWindowCenter

```
int mdlView_newWindowCenter
(
    int          viewNumber,      /* => view to update */
    DPoint3d     *newCenterP     /* => new view center, in curr. coords */
);
```

Description mdlView_newWindowCenter provides an easy way for an application to set a new window center without changing the orientation or scale of the view.

viewNumber specifies the view for which the new view center is to be applied.

newCenterP specifies the new view center (in the application's current coordinate system).

Returns mdlView_newWindowCenter returns SUCCESS if the input parameters are valid, MDLERR_BADVIEWNUMBER if *viewIndex* is not between 0 and MAX_VIEWS, or MDLERR_VIEWNOTDISPLAYED if the view is not currently displayed.

See Also mdlView_setArea, mdlView_zoom, mdlView_rotateToRMatrix.

mdlView_pointToScreen

```
int mdlView_pointToScreen
(
    Point2d      *screenPointP, /* <= output pt in screen or window coords */
    DPoint3d     *inPointP,     /* => input pnt in curr. coords */
    int          viewNumber,     /* => view number */
    int          coordinateSystem /* => _INLOCALCOORDS or _INGLOBALCOORDS */
);
```

Description mdlView_pointToScreen calculates a screen position for an input point. It takes as input a point in the current coordinate system (i.e., transformed by the current transformation), a view index, and a flag indicating what screen coordinate system is desired.

screenPointP points to a Point2d structure that will hold the output point on return from the function.



The point returned may be outside the range that can be displayed in the indicated view, depending on the input point and the area visible in the view.

inPointP is the address of the point (in the current coordinate system for the application) for which screen coordinates are desired.

viewNumber indicates the view for which the *screenPointP* is desired.

coordinateSystem must be set to either VIEW_INLOCALCOORDS or VIEW_INGLOBALCOORDS. If it is set to VIEW_INLOCALCOORDS, *screenPointP* is returned relative to the upper left corner of the window, and if it is set to

VIEW_INGLOBALCOORDS, *screenPointP* is returned relative to the upper left corner of the screen. In both cases x increases to the right and y increases down the window.

Returns mdlView_pointToScreen returns SUCCESS if the input parameters are valid, MDLERR_BADVIEWNUMBER if *viewIndex* is not between 0 and MAX_VIEWS, or MDLERR_VIEWNOTDISPLAYED if the view is not currently displayed.

See Also mdlView_screenToPoint, *userView_drawCursor*.

mdlView_screenToPoint

```
int mdlView_screenToPoint
(
DPoint3d      *outPointP,    /* <= output pnt in curr. coords */
Point2d       *screenPointP, /* => input pt in screen or window coords */
int           viewNumber,    /* => view number */
int           coordinateSystem /* => _INWINDOWCOORDS or _INGLOBALCOORDS */
);
```

Description The mdlView_screenToPoint function calculates the design file coordinates given a point on the screen. The output is in the current coordinate system for the MDL application (i.e., transformed by the current transformation). The inputs are the screen point, a view index, and a flag indicating what screen coordinate system is desired.

The *outPointP* parameter points to a DPoint3d structure that will hold the output point (in the current coordinate system) on return from the function.

The *screenPointP* parameter is the address of the point for which the design file coordinates are desired.

The *viewNumber* parameter indicates the view for which the *screenPointP* is desired.

The *coordinateSystem* parameter must be set to either VIEW_INWINDOWCOORDS or VIEW_INGLOBALCOORDS. If it is set to VIEW_INWINDOWCOORDS, *screenPointP* is specified relative to the upper left corner of the window, with x increasing to the right and y increasing down the window. If it is set to VIEW_INGLOBALCOORDS, *screenPointP* is specified relative to the upper left corner of the screen.

Returns mdlView_screenToPoint returns SUCCESS if the input parameters are valid, MDLERR_BADVIEWNUMBER if *viewIndex* is not between 0 and MAX_VIEWS, or MDLERR_VIEWNOTDISPLAYED if the view is not currently displayed.

See Also mdlView_pointToScreen.

mdlView_setPopupMenu

```
DItem_PulldownMenu *mdlView_setPopupMenu/* <= menu ptr or NULL */
(
    ULONG    menuType,      /* => type of menu to set as the view popup */
    long     menuId,        /* => id of menu to insert into view popup */
    void     *ownerMD       /* => usually NULL */
);
```

Description The `mdlView_setPopupMenu` function sets the menu that pops up when the user presses <shift-reset> inside a MicroStation view. The view popup can be used for quick access to command-specific options or parameters. The view popup is automatically reset to the default MicroStation popup whenever `mdlState_startPrimitive` is called.

menuType is the resource type for the menu. If *menuType* is 0 it defaults to `RTYPE_PulldownMenu`.

menuId is the resource ID for the popup menu.

The MDL task is specified by *ownerMD*. If *ownerMD* is `NULL`, as is usually the case, the current task is the owner.

Returns `mdlView_setPopupMenu` returns a pointer to the installed popup menu, or `NULL` if it fails to find the desired menu.

See Also `mdlState_startPrimitiveAndSetPopupMenu`.

mdlView_blitDirtyAreas (Macintosh Only)

```
void mdlView_blitDirtyAreas
(
    void
);
```

Description On the Macintosh only, an offscreen bitmap is maintained for each view. MicroStation elements are displayed to this bitmap and periodically during an update any portion of the bitmap that has been changed (dirty) is copied (blitted) to the screen. The `mdlView_blitDirtyAreas` function blits all “dirty” portions of the offscreen bit map to the screen.

Returns `mdlView_blitDirtyAreas` is of type `void`.

mdlView_setFunction

```
#include <userfnc.h>

MdlFunctionP mdlView_setFunction
(
    int         type,      /* => UPDATE_PRE,UPDATE_POST, etc. */
    MdlFunctionP function /* => address of MDL function */
);
```

Description `mdlView_setFunction` sets a function to handle events for MicroStation view windows. These events include:

<i>type</i>	<i>functionP</i> called
UPDATE_PRE	before every update of a view window. (See <i>userView_update</i>)
UPDATE_POST	after every update of a view window. (See <i>userView_update</i>)
VIEW_MOTION	whenever the cursor moves inside a view window. (See <i>userView_motion</i>)
VIEW_NOMOTION	whenever the cursor stops moving inside a view window. (See <i>userView_noMotion</i>)
UPDATE_EACH_ELEMENT	for each element MicroStation draws during a view update. (See <i>userView_updateEachElement</i>)
PLOTUPDATE_PRE	after MicroStation calls the plot driver initialization function, draws any border and raster data, but before any other drawing instructions are written. (See <i>userView_plot</i>)
PLOTUPDATE_POST	after MicroStation completes the last drawing calls but before calling the plot driver termination function. (See <i>userView_plot</i>)
PLOTUPDATE_FINISHED	after Plot has been generated and plotfile is closed. (See <i>userView_plot</i>)
VIEW_DRAWCURSOR	when MicroStation is drawing a vector cursor in a view.

function must be a valid pointer to an MDL function or NULL. If this argument is NULL, no more events of type *type* are sent to your MDL application.

Returns `mdlView_setFunction` returns a pointer to the user function (of the same type) that was previously set using `mdlView_setFunction`. If *type* is invalid, `mdlView_setFunction` returns -1.

mdlView_clipToContent, mdlView_clipToViewRect

```
#include <msview.fdf>

void mdlView_clipToContent
(
    int viewNumber    /* => view to set clipping for */
);

void mdlView_clipToViewRect
(
    int viewNumber    /* => view to set clipping for */
);
```

Description Starting with MicroStation 95, views can optionally have scrollbars and view control icons along the bottom and right edges. The content rectangle includes the scrollbars and icons, but the “view rectangle” includes only the actual view. The `mdlView_clipToContent` and `mdlView_clipToViewRect` functions are provided to give MDL programs a way of adjusting the graphics clipping to either the full content rectangle or the view rectangle. These functions are needed only in unusual circumstances where an MDL program is drawing directly to view windows. In other situations, such as updating a view, drawing dynamics, or displaying elements or element descriptors to a view, MicroStation sets the clip rectangle appropriately.

viewNumber designates the view to set the clipping for (starting with 0 for the view labelled View 1).



These functions were implemented in MicroStation 95.

Returns `mdlView_clipToContent` and `mdlView_clipToViewRect` are of type `void`. They return no values.

mdlView_getViewRectangle

```
#include <msdefs.h>
#include <basetype.h>
#include <msdefs.h>
#include <msview.fdf>

void mdlView_getViewRectangle
(
    BSIRect      *rectP,           /* <= view rectangle */
    MSWindow     *windowP,        /* => view rectangle */
    int          coordinateSystem /* coordinate system */
);
```

Description Starting with MicroStation 95, views can optionally have scrollbars and view control icons along the bottom and right edges. Before MicroStation 95, it was possible to get the screen area taken up by a view by calling `mdlWindow_contentRectGetLocal` function. Since the content rectangle includes the scrollbars and icons, but the “view rectangle” includes only the actual view, the `mdlView_getViewRectangle` function is now the correct method of retrieving the screen area taken up by the view.

rectP points to a `BSIRect` structure that receives the view rectangle. `BSIRect` is declared in `basetype.h`.

windowP designates the view window. To get an `MSWindow` from a view number, use `mdlWindow_viewWindowGet`. `MSWindow` is declared in `mdl.h`.

coordinateSystem designates the coordinate system desired. Currently, the only valid argument is `VIEW_INGLOBALCOORDS`, and you must pass that value. This define is in `msdefs.h`.



This function was implemented in MicroStation 95.

Returns mdlView_getViewRectangle is of type void. It returns no value.

See Also mdlWindow_contentRectGetLocal, mdlWindow_viewWindowGet.

mdlView_indexFromWindow

```
#include <msview.fdf>
#include <mdl.h>

int mdlView_indexFromWindow
(
MSWindow      *windowP      /* => window to get view index for */
);
```

Description The mdlView_indexFromWindow function retrieves the view number from a pointer to the MSWindow of the view.

windowP is a pointer to a view of type MSWindow.



This function was implemented in MicroStation 95.

Returns mdlView_indexFromWindow returns a view Number from 0 and MAX_VIEWS if *windowP* is an MSWindow pointer that points to a view window, and -1 otherwise.

See Also mdlWindow_viewWindowGet.

userView_update

```
#include <userfnc.h>
#include <mstypes.h>

int userView_update
(
int          preUpdate,      /* => TRUE if called before update */
int          eraseMode,      /* => display mode */
long         *fileMask,      /* => files involved in update */
int          numberRegions,  /* => number of regions */
Asynch_update_view regions[], /* => region descriptions */
BSIRect      *coverLists[],  /* => pointers to cover lists */
int          numCovers[],    /* => cover list sizes */
MSDisplayDescr *displayDescr[] /* => display descriptors */
);
```

Description If an MDL application designates an update function, *userView_update* is called before or after view updates as specified in a call to mdlView_setFunction. The application programmer determines the function name; *userView_update* is used

merely as an example. When MDL calls the user function, the arguments are set as follows:

preUpdate is `TRUE` if the call is before an update. Otherwise, it is `FALSE`.

eraseMode is set to `NORMALDRAW`, `ERASE` or `HILITE`, and indicates the mode with which MicroStation is drawing elements on the screen.

fileMask is a bitmap of files involved in the update. See the `mdlView_fit` function for a description.

The *regions*, *coverLists*, *numCovers*, and *displayDescr* arguments are parallel arrays. All information required on a given region is contained in the parallel entries for a given region. For example, to obtain all information on region *i*, access *regions[i]*, *coverLists[i]*, *numCovers[i]* and *displayDescr[i]*.

numberRegions gives the number of regions being updated. The regions being updated do not necessarily correspond to the views, although they will all be contained in views.

coverLists is an array of pointers to `Rectangle` structures. Each rectangle designates the covered portion of the region. For each entry in the array that *coverLists* points to, there is an entry in the array *numCovers*. *numCovers[i]* tells how many entries are in the array of `Rectangle` structures pointed to by *coverLists[i]*. Applications seldom need to pay attention to the cover lists. This information is provided only for the PC and is used only by applications that will optimize their display rather than using the clipping capabilities provided by window functions.

regions is an array of `ASYNCH_UPDATE_VIEW` structures describing the regions being updated. The elements in this array give all view-related information required for each region.

displayDescr is an array of pointers to structures of display descriptors, with one structure (i.e., descriptor) for each region to be updated.

Returns If *userView_update* is called for a pre-update, it must return a zero so MicroStation will continue the update. Or it must be a non-zero value to stop the update. If *userView_update* is called for a post-update, MicroStation ignores the return value.

userView_motion, userView_noMotion

```

#include <userfnc.h>
#include <mstypes.h>

void userView_motion
(
MSWindow      *windowP,      /* window in which motion occurred */
int           xCoord,        /* current x coordinate of cursor */
int           yCoord         /* current y coordinate of cursor */
);

void userView_noMotion
(
MSWindow      *windowP,      /* window in which motion occurred */
int           xCoord,        /* current x coordinate of cursor */
int           yCoord         /* current y coordinate of cursor */
);

```

Description MDL applications typically show the user what is happening as the cursor moves using the dynamic functions (`mdlState_dynamicUpdate` and `mdlState_setFunction` with `STATE_COMPLEX_DYNAMICS`). However, applications that need to show another indicator can use the cursor motion and no-motion functions. The precision input dialog box uses the no-motion function.

Cursor motion functions differ from dynamic update functions as follows:

- They can turn the cursor off. This means that they can draw to the screen using `NORMALDRAW` mode rather than `TEMPDRAW` and `TEMPERASE`.
- They are not affected by windowing commands. MicroStation automatically halts the dynamic functions when window commands temporarily push the current command state. The cursor motion functions are still called.
- The no-motion function is called only when the cursor stops moving for a finite period of time.

If a cursor motion or no-motion function is specified in `mdlView_setFunction` in your MDL application, MicroStation calls that MDL user function when the cursor is in a MicroStation view window.



You cannot have both a cursor motion function and a cursor no-motion function operating concurrently. If a program attempts this, the function that is implemented last takes precedence.

The application programmer determines the function name; `userView_motion` and `userView_noMotion` are used merely as examples. When MDL calls the user functions, the arguments are set as follows:

windowP points to the window containing the cursor.

xCoord and *yCoord* are the cursor coordinates in global coordinates. Call `mdlWindow_pointToLocal` to get the coordinates in local units. The current cursor location in world coordinates is stored in `MSStateData.current.uors`.



The cursor is on when the motion or no-motion function is called. To draw to the window in these functions, first turn the cursor off using `mdlWindow_cursorTurnOff`.

Returns The return value for *userView_motion* and *userView_noMotion* is ignored.

userView_updateEachElement

```
#include <userfnc.h>

int userView_updateEachElement
(
  MSElement *elementP,      /* <=> current element to be drawn */
  int drawMode,             /* => drawing mode */
  MSWindow *windowP,        /* => pointer to view window */
  ExtraElementInfo *extraInfoP, /* => additional information */
  int viewNum,              /* => view number to draw into */
  int *priorityP,           /* <= priority if elm is deferred */
  MSElementDescr *edP      /* => elmdscr containing elementP */
  int allowDefer            /* => TRUE = ..._STATUS_DEFER legal */
);
```

Description As MicroStation draws elements to a view or to a plot, it calls all active *userView_updateEachElement* functions to allow MDL applications to affect the way the drawing appears. *userView_updateEachElement* functions are passed a pointer to the current element, *elementP*. They can modify the element if necessary, but should not change its size. *userView_updateEachElement* can be used, for example, to change the symbology of elements based on some external criteria.

drawMode will either be `NORMALDRAW`, `HILITE` or `ERASE`.

windowP is an opaque pointer to the view window.

extraInfoP is a pointer to a structure of information about the element being drawn. It is of type:

```
typedef struct extraElementInfo
{
  int      compoundIndex; /* index into compound elements; 0=none */
  ULong    filePos;      /* file position of element */
  int      drawMode;     /* drawing mode (also in drawMode arg) */
  int      file;          /* file number */
} ExtraElementInfo;
```

allowDefer indicates to the *userView_updateEachElement* function whether or not the current drawing operation may be deferred. If

allowDefer is FALSE, a return status of UPDATE_ELM_STATUS_DEFER is ignored.

Returns Your *userView_updateEachElement* function must return one of the following:

value	Description
UPDATE_ELM_STATUS_CONTINUE	continue processing the element normally
UPDATE_ELM_STATUS_BLOCK	do not draw this element
UPDATE_ELM_STATUS_DEFER	defer drawing this element (see below).

After the all elements have been processed, MicroStation will sort any elements deferred by the *userView_updateEachElement* in ascending order of the priorities returned by the filter function. It then displays them in that order.



userView_updateEachElement functions are also called during plotting updates and element highlights.

userView_plot

```
#include <userfnc.h>

int userView_plot
(
    int          type,          /* => type of event */
    MWindow      *windowP      /* => pointer to plot window */
);
```

Description If an MDL application designates a plot update function, the *userView_plot* function is called before and after the view is plotted, as well as after the entire plot has completed, as specified in a call to *mdlView_setFunction*. When MDL calls the user function, the arguments are set as follows:

<i>type</i>	Called when
PLOTUPDATE_PRE	MicroStation has written initialization commands to the plotfile, possibly drawn border and raster components but has yet to output ordinary view drawing commands. Any plot driver entry point may be called at this time. The plotfile is open.
PLOTUPDATE_POST	MicroStation has completed writing drawing commands to the plotfile but has not yet written termination commands. Any plot driver entry point may be called at this time. The plotfile is open.
PLOTUPDATE_FINISHED	The plot has completed and the plotfile has been closed. This hook is useful if you want to do something to the plotfile from an external program.

windowP is an opaque pointer to the plot view window.

Returns If *userView_plot* returns anything other than SUCCESS, MicroStation aborts the plot.

userView_drawCursor

```
#include <global.h>
#include <userfnc.h>

int userView_drawCursor
(
  CursorInfo  *ciP,          /* => cursor position information */
  int         cursorType     /* => cursor type */
);
```

Description If an MDL application designates a view cursor drawing function, the *userView_drawCursor* function is called whenever MicroStation is drawing a cursor to a view.

The *userView_drawCursor* function is called with two arguments. The *ciP* argument is a pointer to a structure that contains information about where the cursor is in the view. The *cursorInfo* structure is defined in *global.h* as:

```
typedef struct
{
  Point3d   rawUors;         /* => design file coordinates */
  Point3d   uors;           /* => coordinates adjusted for locks */
  Uspoint3d input;          /* => input device coordinates */
  Spoint2d  screen;         /* => screen coordinates */
  MSWindow  *windowP        /* => window */
  short     region;         /* => region identifier */
  short     screenNum;      /* => screen cursor is on */
  short     view;           /* => view index */
  short     buttonTrans;    /* => for button CursorInfo, transition */
  short     qualifierMask; /* => for button CursorInfo, key state */
} CursorInfo;
```

Of these fields, usually only *rawUors*, *uors*, *screen*, *screenNum*, and *windowP* are important to drawing a cursor. The screen coordinates supplied are in global coordinates (i.e., relative to the upper left of the screen).

The *cursorType* argument can be one of the following values defined in *userfnc.h*:

<i>cursorType</i> argument	Description
VIEWCURSOR_DEFAULT	MicroStation's default crosshair cursor.
VIEWCURSOR_DYNAMICS	MicroStation's dynamic "x" cursor.
VIEWCURSOR_SPECIAL	Isometric or full-screen cursor.

In addition, the `VIEWCURSOR_DEFAULT` and `VIEWCURSOR_SPECIAL` cursors may sometimes have the `VIEWCURSOR_LOCATE` logical ORed with their value to indicate that MicroStation would be drawing the locate circle as well as the cursor. The locate cursor portion is generally drawn at the position specified by `ciP->uors`, while the rest of the cursor is drawn at the position specified by `ciP->rawUors`, which is translated into screen coordinates in the `ciP->screen` member. To go from uors to screen position, the `mdlView_pointToScreen` function will be useful.

Returns If `userView_drawCursor` returns anything other than `SUCCESS`, MicroStation uses its own internal code to draw the cursor.

See Also `mdlView_pointToScreen`.

Sectioning and Hidden Line Viewing Functions

The following table lists the sectioning and hidden line viewing functions:

Function	Used to
<code>mdlHview_clearView</code>	erase the view.
<code>mdlHview_closeContext</code>	finish use of the context.
<code>mdlHview_eraseAndRedrawView</code>	draw to a view in hidden line mode.
<code>mdlHview_message</code>	set various advanced features.
<code>mdlHview_openContext</code>	create a context to be used for advanced features.
<code>mdlHview_openFile</code>	open a work file (.hln) to receive outputs on subsequent calls to <code>mdlHview_processView</code> .
<code>mdlHview_processView</code>	write all geometry from the indicated view into the hidden line view. This may be invoked many times within an <code>openContext</code> ... <code>closeContext</code> pair.
<code>mdlHview_setSectionPlanes</code>	define section planes.

`mdlHview_clearView`

```
#include <mdlhview.h>
#include <mdlhview.fdf>

void mdlHview_clearView
(
    HviewContextP      contextP
);
```

Description The mdlHview_clearView function erases the view addressed by the context pointer.

The *contextP* parameter points to the hidden line context created by a prior call to mdlHview_openContext.



This function was implemented in MicroStation 95.

Returns mdlHview_clearView is of type void. It returns no value.

See Also mdlHview_openContext, mdlHview_processView.

mdlHview_closeContext

```
#include <mdlhview.h>
#include <mdlhview.fdf>

void mdlHview_closeContext
(
HviewContextP    contextP
);
```

Description The mdlHview_openContext discards the context created by a prior call to mdlHview_openContext; if a file was opened.

The *contextP* parameter points to the context to be closed.



This function was implemented in MicroStation 95.

Returns mdlHview_closeContext is of type void. It returns no value.

See Also mdlHview_openContext, mdlHview_processView, mdlHview_openFile.

mdlHview_eraseAndRedrawView

```
#include <mdlhview.h>
#include <mdlhview.fdf>

int mdlHview_eraseAndRedrawView
(
int    view    /* => zero-based view number to be redrawn */
);
```

Description The mdlHview_eraseAndRedrawView function clears the screen in the given view and redraws in hidden line mode. The picture is identical to what would occur from the Export...Visible Edges dialog with all default options.

The *view* parameter identifies the view to be drawn.

Use mdlHview_openContext, mdlHview_processView, mdlHview_closeContext and related functions to apply advanced features such as section planes, output to a file, and symbology changes.



This function was implemented in MicroStation 95.

Returns mdlHview_eraseAndRedrawView returns SUCCESS or !SUCCESS.

See Also mdlHview_openContext, mdlHview_processView and mdlHview_closeContext.

mdlHview_message

```
#include <mdlhview.h>
#include <mstypes.h>
#include <mdlhview.fdf>

void mdlHview_message
(
  HviewContextP    contextP,      /* <=> context previously created */
  int    message, /* => message code determines meaning of later args */
  int    ival,    /* => int param, meaning depends on message */
  void    *pointer /* => pointer, meaning depends on message */
);
```

Description The mdlHview_message function executes one of a variety of setup operations on a hidden line and sectioning context.

The *contextP* parameter points to the context created by a prior call to mdlHview_openContext.

The *message* parameter is an integer the indicates the operation. See the table below for the recognized values of the message parameter and the meaning attached to the remaining parameters for each message.

The *ival* and *pointer* parameters are an integer and pointer value whose meaning depends on the message.

ival = integer argument, meaning depends on message

pointer = pointer argument, meaning depends on message

default: description of settings if the message is never sent

The various messages are as follows:

Message	Description
MDLHVIEW_MSG_SET_VISIBLE_SYMBLOGY	<p><i>ival</i> =0 <i>pointer</i> =pointer to HLineSymbology structure to be applied to visible edges. <i>default</i>: all symbology as on original geometry.</p>
MDLHVIEW_MSG_SET_HIDDEN_SYMBLOGY	<p><i>ival</i> =0 <i>pointer</i> =pointer to HLineSymbology structure to be used for hidden edges. Note that this does NOT enable output of hidden edges — it just sets the symbology for if they are enabled. Send MDLHVIEW_MSG_INCLUDE_HIDDEN to enable the hidden edges. <i>default</i>: level, color and weight as on original geometry. Style dashed (i.e, the usual hidden line style on drawings).</p>
MDLHVIEW_MSG_SET_INCLUDE_HIDDEN	<p><i>ival</i> =TRUE/FALSE flag indicating whether hidden lines are to be shown, (TRUE) or not (FALSE) <i>pointer</i> = NULL <i>default</i>: FALSE</p>
MDLHVIEW_MSG_SET_NPC_WINDOW	<p><i>ival</i> =0 <i>pointer</i> =pointer to a Dvector2d which defines the active rectangle within the view. Coordinate 00 is at the lower left, 11 at the upper right. All subsequent resolution changes are applied in this rectangle rather than the full view. If the NPC window is to be changed, it must be done PRIOR TO setting resolutions. <i>default</i>: The entire window (00 to 11) is active.</p>
MDLHVIEW_MSG_SET_INCLUDE_INTERSECTIONS	<p><i>ival</i> =TRUE/FALSE indicating whether surface-surface intersections are to be computed. <i>pointer</i> =NULL <i>default</i>: FALSE NOTE: Surface-surface intersections add significantly to compute time and memory use.</p>

Message	Description
MDLHVIEW_MSG_SET_INCLUDE_RULES	<p><i>ival</i> =TRUE/FALSE flag indicating whether surface rule lines are to be output (TRUE) or not (FALSE)</p> <p><i>pointer</i> =NULL</p> <p><i>default:</i> FALSE</p> <p><i>default:</i> MDLHVIEW_ANNOTATION_HIDE</p>
MDLHVIEW_MSG_SET_INCLUDE_ANNOTATION	<p><i>ival</i> =One of the following choices of handling of text, dimensions and other annotation geometry:</p> <p>MDLHVIEW_ANNOTATION_HIDE — reduce the annotation to line strokes and process them through the hidden line algorithm, i.e., subject to hiding as with any other line strokes. MDLHVIEW_ANNOTATION_ALWAYS — output the annotation without testing for visibility.</p> <p>MDLHVIEW_ANNOTATION_NEVER — ignore annotation.</p> <p><i>pointer</i> =NULL</p>
MDLHVIEW_MSG_SET_OUTPUT_TO_MASTER	<p><i>ival</i> =TRUE/FALSE to enable/disable output to the master file.</p> <p><i>pointer</i> =NULL</p> <p><i>default:</i> FALSE</p>
MDLHVIEW_MSG_SET_WORKING_MEMORY	<p><i>ival</i> =maximum size of working memory to use for the hidden line process, in KB. E.g., <i>ival</i>=1024, sets working memory to 1024 KB = 1 MB.</p> <p><i>pointer</i> =NULL</p> <p><i>default:</i> 4096, i.e. 4 MB. The default of 4096 KB produces default resolution images (1000 by 1000) in a single pass. If the working memory is reduced or resolution increased, hidden line images are generated in multiple passes, each appearing as a horizontal band on screen.</p>
MDLHVIEW_MSG_SET_XRESOLUTION MDLHVIEW_MSG_SET_YRESOLUTION	<p><i>ival</i> =resolution to apply on the respective axis.</p> <p><i>pointer</i> =NULL</p> <p><i>default:</i> Both resolutions default to 1000.</p>
MDLHVIEW_MSG_SET_RESOLUTION_TO_SCREEN	<p><i>ival</i> =0</p> <p><i>pointer</i> =NULL Sets both x and y resolution to match the screen in the view.</p>

Message	Description
MDLHVIEW_MSG_SET_RESOLUTION	<i>ival</i> = resolution to apply on longer axis of view. Shorter side is proportionally smaller. <i>pointer</i> = NULL (If MDLHVIEW_MSG_SET_NPC_WINDOW is being used to restrict attention to a subset of the view, only that rectangle is considered by MDLHVIEW_MSG_SET_RESOLUTION_TO_SCREEN and MDLHVIEW_MSG_SET_RESOLUTION).
MDLHVIEW_MSG_CLEAR_MASK_POLYGONS	<i>ival</i> = 0 <i>pointer</i> = NULL The saved set of mask polygons is cleared.
MDLHVIEW_MSG_SET_INSIDE_POLYGON MDLHVIEW_MSG_SET_OUTSIDE_POLYGON	<i>ival</i> = number of points <i>pointer</i> = pointer to an array of Dpoint3d structures defining the (3D) polygon. When the mask is applied, points in the polygon are marked as visible (resp. hidden).
MDLHVIEW_MSG_SET_FENCEMODE	<i>ival</i> = 0 — ignore the MicroStation fence. 1 — compute visible edges inside the MicroStation fence. 2 — compute visible edges outside the MicroStation fence <i>pointer</i> = NULL. <i>default</i> : The MicroStation fence is ignored.
MDLHVIEW_MSG_SET_PROGRESS_FUNCTION	<i>ival</i> = 0 <i>pointer</i> = pointer to a function to be called with updates of the progress. The function arguments are: func(char * <i>messageP</i> , double <i>percentComplete</i>) where <i>messageP</i> = string describing the progress. <i>percentComplete</i> = percent of computation completed. <i>default</i> : Progress messages are issued to a dialog box with a completion bar.
MDLHVIEW_MSG_SET_PRE_FILE_FUNCTION	<i>ival</i> = 0 <i>pointer</i> = pointer to a function to be called each time the master file or a reference file is opened for scanning. The parameter list is preFileFunc(<i>iFile</i>) where <i>iFile</i> is the file number. <i>default</i> : No pre-file function is called.

Message	Description
MDLHVIEW_MSG_SET_FILE_HIDE_MASK MDLHVIEW_MSG_SET_FILE_INCLUDE_HIDDEN_MASK	<p><i>ival</i> =0 <i>pointer</i> =pointer to an array of bit indicating whether the corresponding files (master=0, 1...255=reference files) are to be processed. The INCLUDE_HIDDEN mask allows hidden lines to be included in selected files. (Turning on MDLHVIEW_MSG_SET_INCLUDE_HIDDEN is equivalent to setting all bits in this array to 1). The default for the INCLUDE_HIDDEN mask is all bits 0. The FILE_HIDE mask allows selected files to be skipped completely. A 1 bit in this array means the file is to be included in the scan, and a zero means skip. The <i>default</i> for FILE_HIDE mask is all bits. 1, i.e., scan all files.</p>
MDLHVIEW_MSG_SET_EDGE_DEPTH_CHANGE	<p><i>ival</i> =number of units of depth to subtract to edge coordinates to make them closer to the eye for depth comparisons. <i>pointer</i> =NULL <i>default</i>: zero (0). It is strongly recommended that this value NOT be changed.</p>
MDLHVIEW_MSG_SET_EDGE_WIDTH	<p><i>ival</i> =Extra width, in pixels of resolution, to add to each side of edges as they are tested for visibility. <i>pointer</i> =NULL <i>default</i>: 1. Reducing this to zero may cause some 'speckling' or 'stitching' effects, i.e. a solid edge appears to be dashed. It may also reduce undesirable double drawing.</p>
MDLHVIEW_MSG_SET_INCLUDE_RAW_MESH	<p><i>ival</i> =TRUE/FALSE to enable/disable output of raw mesh on surfaces. <i>pointer</i> =NULL <i>default</i>: FALSE</p>

The hLineSymbology structure is declared as, (from mstypes.h):


```
typedef struct hLineSymbology
{
    int      level;
    int      color;
    int      style;
    int      weight;
    unsigned levelOverride:1;
    unsigned colorOverride:1;
    unsigned styleOverride:1;
    unsigned weightOverride:1;
} HLineSymbology;
```

Each field (*level*, *color*, *style*, *weight*) provides a replacement for the respective value. Each replacement value is applied only if the corresponding override bit (*levelOverride*, *colorOverride*, *styleOverride*, *weightOverride*) is set. Hidden geometry is not created at all (regardless of the HLineSymbology settings) unless the application passes the MDLHVIEW_MSG_SET_INCLUDE_HIDDEN message with *ival* of 1.



This function was implemented in MicroStation 95.

Returns mdlHview_message is of type void. It returns no value.

See Also mdlHview_openContext, mdlHview_processView.

mdlHview_openContext

```
#include <mdlhview.h>
#include <mdlhview.fdf>

HviewContextP mdlHview_openContext
(
    int      view
);
```

Description The mdlHview_openContext creates a default context for the indicated view. mdlHview_closeContext discards the context.

The *view* parameter is the (zero based) view number to be drawn in subsequent calls to mdlHview_processView.



This function was implemented in MicroStation 95.

Returns mdlHview_openContext returns a pointer to the context structure. The calling program must retain this pointer to be passed to later functions, finally discarding it by the call to mdlHview_closeContext.

mdlHview_openFile

```
#include <mdlhview.h>
#include <mdlhview.fdf>

int mdlHview_openFile
(
    HviewContextP    contextP,
    int              fileType, /* => selector for geometry data in file */
    char             *nameP    /* => file name buffer */
);
```

Description mdlHview_openFile conditionally creates a workfile to receive subsequent geometry.

The *contextP* parameter points to the context created by a prior call to mdlHview_openContext.

The *fileType* parameter indicates the type of file to create. The valid values are:

MDLHVIEW_FILE_OFF — no output file.
 MDLHVIEW_FILE_2D — create a 2D file.
 MDLHVIEW_FILE_3D — create a 3D file.

The *nameP* parameter is a buffer for the file name. If a name is provided by the caller, it is used without change. If the caller provides an empty string or a NULL pointer, the a file creation dialog is presented to the user.



This function was implemented in MicroStation 95.

Returns mdlHview_openFile returns SUCCESS if the file was opened or if the *fileType* parameter is MDLHVIEW_FILE_OFF. It returns ERROR if the file was not opened. (E.g. the user canceled from the dialog.)

See Also mdlHview_openContext, mdlHview_processView.

mdlHview_processView

```
#include <mdlhview.h>
#include <mdlhview.fdf>

int mdlHview_processView
(
    HviewContextP    contextP
);
```

Description The mdlHview_processView function generates a hidden line drawing using all parameters previously set in the context. This may be called several times (with

different settings) within an mdlHview_openContext...mdlHview_closeContext block.

The *contextP* parameter is the context with section plane, symbology, files and other advanced settings.



This function was implemented in MicroStation 95.

Returns mdlHview_processView returns SUCCESS if the hidden line drawing is successfully built.

See Also mdlHview_openContext, mdlHview_closeContext, mdlHview_openFile, mdlHview_message, mdlHview_setSectionPlanes, mdlHview_clearView.

mdlHview_setSectionPlanes

```
#include <mdlhview.h>
#include <mdlhview.fdf>

void mdlHview_setSectionPlanes
(
    HviewContextP    contextP,
    Dpoint3d         *originP,      /* => origin points of planes */
    Dpoint3d         *normalP,      /* => plane normals */
    int               nPlane        /* => number of planes */
);
```

Description The mdlHview_setSectionPlanes function defines section planes to be applied by the subsequent call to mdlHview_processView. These planes replace any previously defined planes.

The *contextP* parameter points to the context created by a prior call to mdlHview_openContext.

The *originP* parameter points to an array containing an origin point on each section plane.

The *normalP* parameter points to an array containing section plane normals. The normal points to the 'outside' (clipped away) half space bounded by the plane.

The *nPlanes* parameter is the number of planes being passed.



This function was implemented in MicroStation 95.

Returns mdlHview_setSectionPlanes is of type void. It returns no value.

See Also mdlHview_openContext, mdlHview_processView.

Auxiliary Coordinate System Functions

An auxiliary coordinate system (ACS) is a coordinate system with an orientation and, usually, an origin, different from those of the design cube coordinate system. Although ACS is not exclusively a 3D concept, ACS is most useful in 3D design.

The ACS Functions allow you to attach, delete, and save an ACS and/or manipulate ACS settings.

The following table lists MDL auxiliary coordinate system (ACS) functions:

Function	Used to
<code>mdlACS_attachNamed</code>	attach a named ACS.
<code>mdlACS_deleteNamed</code>	delete a named ACS.
<code>mdlACS_getCurrent</code>	get the current ACS settings.
<code>mdlACS_saveNamed</code>	save a named ACS.
<code>mdlACS_setCurrent</code>	set the current ACS settings.
<code>mdlACS_createElement</code>	create an saved ACS element.
<code>mdlACS_extractElement</code>	extract ACS from saved ACS element.

mdlACS_attachNamed

```
int mdlACS_attachNamed
(
    char    *name           /* => name of the coordinate system */
);
```

Description The `mdlACS_attachNamed` function attaches the named coordinate system specified by *name*. The user can save named ACSs in the design file. Named ACSs can also be saved through the `mdlACS_saveNamed` function.

Returns `mdlACS_attachNamed` returns `SUCCESS` if the coordinate system is successfully attached and `MDLERR_ACSNOTFOUND` if the specified coordinate system is not found.

See Also `mdlACS_deleteNamed`, `mdlACS_saveNamed`.

mdlACS_deleteNamed

```
int mdlACS_deleteNamed
(
    char    *name           /* => name of the coordinate system */
);
```

Description The mdlACS_deleteNamed function deletes the named coordinate system specified by *name*. The user can save named ACSs in the design. Or ACSs can be saved through the mdlACS_saveNamed function.

Returns mdlACS_deleteNamed returns SUCCESS if the coordinate system is successfully deleted and MDLERR_ACSNOTFOUND if the specified coordinate system is not found.

See Also mdlACS_attachNamed, mdlACS_saveNamed.

mdlACS_getCurrent

```
int mdlACS_getCurrent
(
    Dpoint3d    *origin,      /* <= current ACS origin */
    RotMatrix   *rotMatrix,   /* <= current ACS orientation */
    int         *type         /* <= current ACS Type */
);
```

Description The mdlACS_getCurrent function returns the current ACS's origin, rotation matrix, and type. The coordinate system type is specified by *type* as defined in msdefs.h:

```
#define ACS_RECTANGULAR  1
#define ACS_CYLINDRICAL 2
#define ACS_SPHERICAL    3
```

Returns The mdlACS_getCurrent function returns SUCCESS if an ACS is active and MDLERR_NOACSDDEFINED if no system is currently defined.

See Also mdlACS_setCurrent.

mdlACS_saveNamed

```
int mdlACS_saveNamed
(
    char    *name,          /* => name of the coordinate system */
    char    *description    /* => description--27 char. max */
);
```

Description The mdlACS_saveNamed function saves the current coordinate system with the specified *name* and *description*. The name must consist of one to six alphanumeric characters, and the description can contain up to 27 alphanumeric characters. The user or an MDL application can recall saved coordinate systems with the mdlACS_attachNamed function.

Returns mdlACS_saveNamed returns SUCCESS if the coordinate system is successfully saved. If the coordinate system *name* already exists, MDLERR_ACSREPLACED is returned and the existing ACS is overwritten. If no ACS is active, MDLERR_NOACSDDEFINED is returned.

See Also mdlACS_attachNamed, mdlACS_deleteNamed.

mdlACS_setCurrent

```
#include <msdefs.h>

int mdlACS_setCurrent
(
  Dpoint3d    *origin,      /* => ACS Origin */
  RotMatrix   *rotMatrix,   /* => ACS Orientation */
  int         type          /* => ACS Type */
);
```

Description The `mdlACS_setCurrent` function defines the current ACS. The coordinate system's origin and rotation matrix are specified in *origin* and *rotMatrix*. The coordinate system type is specified by *type* as defined in `msdefs.h`:

```
#define ACS_RECTANGULAR 1
#define ACS_CYLINDRICAL 2
#define ACS_SPHERICAL 3
```

Returns The `mdlACS_setCurrent` function returns `SUCCESS` if the coordinate system is successfully defined.

See Also `mdlACS_getCurrent`.

mdlACS_createElement, mdlACS_extractElement

```
#include <mselems.h>

int mdlACS_createElement
(
  MElement    *elementP,    /* <= element */
  char        *name,        /* => ACS Name */
  char        *description, /* => ACS Description */
  Dpoint3d    *originP,     /* => ACS Origin */
  RotMatrix   *rotMatrixP,  /* => ACS RotMatrix */
  int         acsType       /* => ACS Type */
);

int mdlACS_extractElement
(
  Dpoint3d    *originP,     /* <= origin */
  RotMatrix   *rotMatrixP,  /* <= rotation matrix */
  int         *type,        /* <= ACS type */
  char        *nameP,       /* <= name */
  MElement    *elementP     /* => element */
);
```

Description `mdlACS_createElement` creates a saved ACS element. The ACS described by *name*, *description*, *originP* and *rotMatrixP* is returned in *elementP*.

`mdlACS_extractElement` extracts the auxiliary coordinate system from the saved ACS element in *elementP*.

Returns `mdlACS_createElement` and `mdlACS_extractElement` return `SUCCESS` or an appropriate error code (see `mdlerrs.h`).

Current Transformation Functions

The **current transformation** is a transformation matrix that MicroStation maintains for every MDL application. When MDL applications pass coordinates (x, y, z) or distances to MicroStation or when MicroStation returns this information to an MDL application, the application's current transformation matrix transforms this data. By default, the current transformation is set to the identity matrix and coordinates are passed through unchanged.

Applications can rotate, scale, or translate (change the origin of) their current transform. In addition, MDL provides functions to nest transformations so programmers can temporarily shift coordinate systems and subsequently return to the current one. For example, suppose an application is modeling an assembly. An application would first set the current transform to correspond to the assembly's coordinate system. As an application models each assembly part, it can **push** (add to the end of the transformation stack) the current transform and then shift, scale and rotate it to align it with the part's axes. When an application is finished with the part, it can **pop** (remove from the end of the transformation stack) the assembly's coordinate system.

MicroStation's 32-bit integer coordinate system uses units called **UORs**. However, most applications prefer to use more natural units like meters, feet or inches. One application for the `mdlCurrTrans_...` functions lets MDL applications change units.

The following table lists the current transformation functions:

Function	Used to
<code>mdlCurrTrans_begin</code>	push a copy of the current transformation matrix.
<code>mdlCurrTrans_end</code>	pop the last pushed transformation matrix.
<code>mdlCurrTrans_clear</code>	pop all pushed transformation matrices from the stack.
<code>mdlCurrTrans_identity</code>	replace the current transformation matrix with the identity matrix.
<code>mdlCurrTrans_masterUnitsIdentity</code>	replace the current transformation matrix with the identity matrix and set the scales by the number of UORs per master unit.
<code>mdlCurrTrans_rotateByAngles</code>	rotate the current transformation about its origin by the angles specified in <i>xAngle</i> , <i>yAngle</i> , and <i>zAngle</i> in radians.

Function	Used to
<code>mdlCurrTrans_rotateByRMatrix</code>	rotate the current transformation about its origin by the rotation matrix that <i>rMatrix</i> points to.
<code>mdlCurrTrans_rotateByView</code>	rotate the current transformation about its origin by a view's rotation matrix.
<code>mdlCurrTrans_translateOrigin</code>	translate the current transformation matrix's origin by a distance given in units in the current coordinate system's scale.
<code>mdlCurrTrans_translateOriginWorld</code>	translate the current transformation matrix's origin by a distance given in units in UORs.
<code>mdlCurrTrans_scale</code>	scale the current transformation matrix by scale factors <i>xScale</i> , <i>yScale</i> and <i>zScale</i> .
<code>mdlCurrTrans_scaleDoubleArray</code>	scales an array of double precision values from the current coordinate system into design file coordinates.
<code>mdlCurrTrans_invScaleDoubleArray</code>	scales an array of double precision values from design file coordinates into the current coordinate system.
<code>mdlCurrTrans_transformPointArray</code>	transform an array of points from the current coordinate system into design file coordinates.
<code>mdlCurrTrans_invtransPointArray</code>	transform an array of points from the design file coordinates into the current coordinate system.
<code>mdlCurrTrans_transformRMatrix</code>	transform a rotation matrix from the current coordinate system into design file coordinates.
<code>mdlCurrTrans_invtransRMatrix</code>	transform a rotation matrix from the design file coordinate system into the current coordinate system.
<code>mdlCurrTrans_fromACS</code>	sets the current coordinate system from the current ACS.
<code>mdlCurrTrans_toACS</code>	sets the current ACS from the current coordinate system.
<code>mdlCurrTrans_getAddresses</code>	return the address of the current forward transform and the address of the current inverse transform.

Example

See the example in the mdl\examples\trumpet directory.

mdlCurrTrans_begin, mdlCurrTrans_end, mdlCurrTrans_clear

```
#include <mdl.h>

int mdlCurrTrans_begin();
int mdlCurrTrans_end();
int mdlCurrTrans_clear();
```

Description mdlCurrTrans_begin pushes a copy of the current transformation matrix and the mdlCurrTrans_end function pops the last pushed transformation matrix.

mdlCurrTrans_clear pops all pushed transformation matrices from the stack. After the mdlCurrTrans_clear has been used, the current transformation matrix is the identity matrix. This is useful for recovering from error conditions in a program.

Calls to mdlCurrTrans_begin can be nested to any depth, but should always be matched with calls to mdlCurrTrans_end or mdlCurrTrans_clear.



mdlCurrTrans_begin must be called before calls can be made to other mdlCurrTrans_... functions.

Returns mdlCurrTrans_begin returns SUCCESS if the transformation is pushed. Otherwise, it returns MDLERR_INSFMEMORY.

mdlCurrTrans_end returns SUCCESS if a transformation is popped. Otherwise, it returns MDLERR_NOTTRANSFORM if no transformation was pushed.

mdlCurrTrans_clear returns the number of transformation matrices it popped.

mdlCurrTrans_identity, mdlCurrTrans_masterUnitsIdentity

```
#include <mdl.h>

int mdlCurrTrans_identity();
int mdlCurrTrans_masterUnitsIdentity
(
    int    translateToGlobalOrigin /* => make 0,0,0 global origin? */
);
```

Description The `mdlCurrTrans_identity` function replaces the current transformation matrix with the identity matrix. This function is useful for returning the current transformation to a known state.

`mdlCurrTrans_masterUnitsIdentity` is identical to `mdlCurrTrans_identity`, except that it sets the scales of the transformation matrix to the number of UORs per master unit. After `mdlCurrTrans_masterUnitsIdentity` is called, all coordinates are in master units. If *translateToGlobalOrigin* is `TRUE`, the current transformation is translated so that its (0, 0, 0) point is at the global origin. Otherwise, (0, 0, 0) is at the center of the design plane.



`mdlCurrTrans_identity` does not need to be called after the first call to `mdlCurrTrans_begin` since the default transformation is the identity matrix.

Returns `mdlCurrTrans_identity` and `mdlCurrTrans_masterUnitsIdentity` return either `SUCCESS` if successful, or `MDLERR_NOTTRANSFORM` if there is no current transformation.

`mdlCurrTrans_rotateByAngles`, `mdlCurrTrans_rotateByRMatrix`, `mdlCurrTrans_rotateByView`

```
#include <mdl.h>

int mdlCurrTrans_rotateByAngles
(
    double  xAngle,
    double  yAngle,
    double  zAngle
);

int mdlCurrTrans_rotateByRMatrix
(
    RotMatrix *rMatrix,
    int       reverse
);

int mdlCurrTrans_rotateByView
(
    int      viewNumber
);
```

Description `mdlCurrTrans_rotateByAngles` rotates the current transformation about its origin by the angles specified in *xAngle*, *yAngle*, and *zAngle* in radians. In 3D files, the rotations are applied first about the Z-axis, then about the Y-axis, and finally about

the X-axis. If this is not the desired order of rotation, make separate calls to `mdlCurrTrans_rotateByAngles`. In 2D files, only *zAngle* is used.

`mdlCurrTrans_rotateByRMatrix` rotates the current transformation about its origin by the rotation matrix pointed to by *rMatrix*. If *reverse* is `TRUE`, the current transformation matrix is rotated by the inverse of *rMatrix*.

`mdlCurrTrans_rotateByView` rotates the current transformation about its origin by the rotation matrix of view *viewNumber*. For the rotation of the current transformation matrix to match the rotation of a view, verify that the current transformation is set to the identity matrix before calling `mdlCurrTrans_rotateByView`.

Returns `mdlCurrTrans_rotateByAngles`, `mdlCurrTrans_rotateByRMatrix` and `mdlCurrTrans_rotateByView` return `MDLERR_NOTTRANSFORM` if there is no current transformation. Otherwise, they return `SUCCESS`.

mdlCurrTrans_translateOrigin, mdlCurrTrans_translateOriginWorld

```
#include <mdl.h>

int mdlCurrTrans_translateOrigin
(
    Dpoint3d    *delta
);

int mdlCurrTrans_translateOriginWorld
(
    Dpoint3d    *delta
);
```

Description The `mdlCurrTrans_translateOrigin` function translates the origin of the current transformation matrix by the distance specified by **delta*. The units for *delta* are in the scale of the current coordinate system.

The `mdlCurrTrans_translateOriginWorld` function translates the origin of the current transformation matrix by the distance specified by **delta*. The units for *delta* are in UORs. This function can be used to set the origin to a known point in the design plane.



The `mdlCurrTrans_translateOrigin` and `mdlCurrTrans_translateOriginWorld` functions always translate the origin from the current position. To set the origin to an absolute coordinate, call `mdlCurrTrans_identity` first.

Returns The `mdlCurrTrans_translateOrigin` and `mdlCurrTrans_translateOriginWorld` functions return `MDLERR_NOTTRANSFORM` if there is no current transformation. Otherwise, they return `SUCCESS`.

mdlCurrTrans_scale

```
#include <mdl.h>

int mdlCurrTrans_scale
(
    double  xScale,
    double  yScale,
    double  zScale
);
```

Description The `mdlCurrTrans_scale` function scales the current transformation matrix by the scale factors *xScale*, *yScale*, and *zScale*. The axes refer to the axes of the current coordinate system. In 2D files, *zScale* is ignored.

Returns The `mdlCurrTrans_scale` function returns `MDLERR_NOTTRANSFORM` if there is no current transformation. Otherwise, it returns `SUCCESS`.

mdlCurrTrans_scaleDoubleArray, mdlCurrTrans_invScaleDoubleArray

```
void mdlCurrTrans_scaleDoubleArray
(
    double *out,
    double *in,
    int     numValues
);

void mdlCurrTrans_invScaleDoubleArray
(
    double *out,
    double *in,
    int     numValues
);
```

Description The `mdlCurrTrans_scaleDoubleArray` function scales an array of *numValues* double precision values pointed to by *in* from the current coordinate system into design file coordinates. The resulting values are stored in the array pointed to by *out*.

The `mdlCurrTrans_invScaleDoubleArray` function scales an array of *numValues* double precision values pointed to by *in* from design file coordinates into the current coordinate system. The resulting values are stored in the array pointed to by *out*.

Both `mdlCurrTrans_scaleDoubleArray` and `mdlCurrTrans_invScaleDoubleArray` will produce incorrect results if the current coordinate system transformation contains nonuniform scaling (different scale factors in the X, Y or Z directions).

Returns `mdlCurrTrans_scaleDoubleArray` and `mdlCurrTrans_invScaleDoubleArray` are of type `void`. They return no value.

See Also mdlCurrTrans_transformPointArray, mdlCurrTrans_invtransPointArray, mdlCurrTrans_transformRMatrix, mdlCurrTrans_invtransRMatrix.

mdlCurrTrans_transformPointArray, mdlCurrTrans_invtransPointArray

```
#include <mdl.h>

void mdlCurrTrans_transformPointArray
(
    Dpoint3d    *out,
    Dpoint3d    *in,
    int         numPoints
);

void mdlCurrTrans_invtransPointArray
(
    Dpoint3d    *out,
    Dpoint3d    *in,
    int         numPoints
);
```

Description The mdlCurrTrans_transformPointArray function transforms an array of *numPoints* points, pointed to by *in*, from the current coordinate system into design file coordinates. The resulting points are stored in the array pointed to by *out*.

The mdlCurrTrans_invtransPointArray function transforms an array of *numPoints* points, pointed to by *in*, from design file coordinates into the current coordinate system. The resulting points are stored in the array pointed to by *out*.



in and *out* can point to the same array.

Returns mdlCurrTrans_transformPointArray and mdlCurrTrans_invtransPointArray are of type void; they return no value.

mdlCurrTrans_transformRMatrix, mdlCurrTrans_invtransRMatrix

```
void mdlCurrTrans_transformRMatrix
(
    RotMatrix    *out,      /* <= dgn coordinates */
    RotMatrix    *in        /* => current coordinate */
);

void mdlCurrTrans_invtransRMatrix
(
    RotMatrix    *out,      /* <= current coordinate */
    RotMatrix    *in        /* => dgn coordinates */
);
```

Description `mdlCurrTrans_transformRMatrix` transforms a rotation matrix pointed to by *in* from the current coordinate system into design file coordinates. The result is stored in the rotation matrix pointed to by *out*.

`mdlCurrTrans_invtransRMatrix` transforms a rotation matrix pointed to by *in* from design file coordinates into the current coordinate system. The result is stored in the rotation matrix pointed to by *out*.

Returns `mdlCurrTrans_transformRMatrix` and `mdlCurrTrans_invtransRMatrix` are of type `void`. They return no value.

See Also `mdlCurrTrans_transformPointArray`, `mdlCurrTrans_invtransPointArray`, `mdlCurrTrans_scaleDoubleArray`, `mdlCurrTrans_invScaleDoubleArray`.

mdlCurrTrans_fromACS, mdlCurrTrans_toACS

```
int mdlCurrTrans_fromACS
(
void
);

int mdlCurrTrans_toACS
(
int      auxType      /* 1=rectangular, 2=cylindrical, 3=spherical */
);
```

Description The `mdlCurrTrans_fromACS` function sets the current coordinate system from the current Auxilliary Coordinate System (ACS).

`mdlCurrTrans_toACS` sets the current auxilliary coordinate system (ACS) from the current coordinate system. The type of ACS is specified by *auxType*, values of 1, 2 and 3 specify rectangular, cylindrical and spherical systems respectively.

Returns `mdlCurrTrans_fromACS` returns `SUCCESS` if the current coordinate system is defined successfully and `MDLERR_NOACSDDEFINED` if an auxiliary coordinate system is not currently defined.

`mdlCurrTrans_toACS` is of type `void`; it returns no value.

mdlCurrTrans_getAddresses

```
#include <mdl.h>

int mdlCurrTrans_getAddresses
(
Transform    **fwd,          /* <= current forward transform */
Transform    **inv          /* <= current inverse transform */
);
```

Description The `mdlCurrTrans_getAddresses` function returns the address of the current forward transform in **fwd* and the address of the current inverse transform in **inv*.

These addresses can be used to perform transform operations that MDL does not support. If you change the forward transform, you must also change the inverse transform so that the product of the two is always the identity matrix.

The addresses returned are valid only for the current transformation. Any calls to `mdlCurrTrans_begin` or `mdlCurrTrans_end` cause these addresses to be invalid.

Returns The `mdlCurrTrans_getAddresses` function returns `MDLERR_NOTTRANSFORM` if there is no current transformation. Otherwise, it returns `SUCCESS`.

Transient Element Functions

The transient element functions are used to specify elements that are displayed, but do not exist in a design file. Transient element display is more persistent than the standard element display functions (`mdlElement_display`, `mdlElemDescr_display`) as the transient elements are displayed during each view update, and are therefore not cleared by viewing commands.

Transient elements are grouped into data structures that are referred to as transient descriptors. Transient descriptors hold more than one transient element by chaining them in a manner similar to element descriptors. The exact format of these structures is not published and is not required, as the descriptors are always referred to by (void) pointers.



A general purpose transient descriptor is defined by the built-in global variable, `msTransientElmP`. This transient descriptor is provided for used by primitive commands. It is automatically freed and erased whenever `mdlState_startPrimitive` is called. It can therefore be used by commands which wish to display graphics while the command is being executed, but a clean up function is not required to clear the graphics if the command is aborted or interrupted.

The following table lists transient element functions:.

Function	Used to
<code>mdlTransient_addElement</code>	add a transient element.
<code>mdlTransient_addElemDescr</code>	add a transient element descriptor.
<code>mdlTransient_replaceElement</code>	replace a transient element.
<code>mdlTransient_replaceElemDescr</code>	replace a transient element descriptor.

Function	Used to
<code>mdlTransient_returnElemDescr</code>	get the address of the element descriptor contained in a transient descriptor.
<code>mdlTransient_free</code>	free a transient element descriptor.

mdlTransient_addElement, mdlTransient_addElemDescr

```
#include <mselems.h>

void *mdlTransient_addElement
(
    void          *tEdP,          /* => TED to attach to or NULL for new */
    MSElement    *elementP,      /* => element to add to TED */
    int           snappable,      /* => TRUE=make it possible to snap to */
    long          viewMask,       /* => views to be drawn into */
    int           displayMode,     /* => mode: NORMALDRAW, ERASE, XORDRAW */
    boolean       displayFirst,    /* => TRUE=display for design graphics */
    boolean       atHead,         /* => TRUE=put at head of list */
    boolean       initialDisplay /* => TRUE=initial display */
);

void *mdlTransient_addElemDescr
(
    void          *tEdP,          /* => TED to attach to or NULL for new */
    MSElementDescr *edP,        /* => element descriptor to add to TED */
    int           snappable,      /* => TRUE=make it possible to snap to */
    long          viewMask,       /* => views to be drawn into */
    int           displayMode,     /* => mode: NORMALDRAW, ERASE, XORDRAW */
    boolean       displayFirst,    /* => TRUE=display before dgn */
    boolean       atHead,         /* => TRUE=put at head of list */
    boolean       initialDisplay /* => TRUE=initial display */
);
```

Description `mdlTransient_addElement` and `mdlTransient_addElemDescr` add an element or element descriptor (*elementP* or *edP*) to an existing **transient descriptor** (*tEdP*), or create a new transient descriptor if *tEdP* is NULL. The transient descriptor format is not published or required, as transient descriptors are always referenced by (void) pointers.

If *snappable* is non-zero, the transient element will be snappable.



This is not currently supported; the snappable argument is currently ignored.

viewMask controls the views in which the transient element is displayed. The least significant bit controls the display in view 1, the next bit controls view 2 etc. A view mask of 0x00ff causes the element to be displayed in all views, a mask of 0x0008 causes display in view 4 only.

displayMode controls the element display mode. Valid display modes are defined in `msdefs.h` and include `NORMALDRAW`, `HILITE`, `ERASE` and `XORDRAW`.

If *displayFirst* is non-zero, the elements in this transient descriptor are displayed before design file graphics, otherwise the elements are displayed after the design file graphics.

If *atHead* is non-zero, the elements in this transient descriptor are displayed before those in other transient descriptors. Otherwise, all transient elements are displayed in the order they were created.

initialDisplay controls the initial display of the element. If *initialDisplay* is zero, the element is added to the transient descriptor but is not displayed.

Returns `mdlTransient_addElement` and `mdlTransient_addElemDescr` return a pointer to a transient descriptor or `NULL` if an error occurs. If *tEdP* is not `NULL` and the element is successfully added, the functions return *tEdP*.

See Also `mdlTransient_replaceElement`, `mdlTransient_replaceElemDescr`, `mdlTransient_returnElemDescr`, `mdlTransient_free`.

mdlTransient_replaceElement, mdlTransient_replaceElemDescr

```
void *mdlTransient_replaceElement
(
    void          *tEdP,          /* => TED to replace or NULL for new */
    MSElement    *elementP,      /* => element to put in TED */
    int           snappable,      /* => TRUE=make it possible to snap to */
    long          viewMask,       /* => views to be drawn into */
    int           displayMode,     /* => mode: NORMALDRAW, ERASE, XORDRAW */
    boolean       displayFirst,   /* => TRUE=display for design graphics */
    boolean       atHead,         /* => TRUE to put at head of list */
    boolean       initialDisplay /* => TRUE for initial display */
);

void *mdlTransient_replaceElemDescr
(
    void          *tEdP,          /* => TED to replace or NULL for new */
    MSElementDescr *edP,        /* => element descriptor to put in TED */
    int           snappable,      /* => TRUE=make it possible to snap to */
    long          viewMask,       /* => views to be drawn into */
    int           displayMode,     /* => mode: NORMALDRAW, ERASE, XORDRAW */
    boolean       displayFirst,   /* => TRUE=display for design graphics */
    boolean       atHead,         /* => TRUE=put at head of list */
    boolean       initialDisplay /* => TRUE=initial display */
);
```

Description `mdlTransient_replaceElement` and `mdlTransient_replaceElemDescr` replace the entire contents of the existing transient descriptor *tEdP* with the element or element descriptor described by *elementP* or *edP*. The transient descriptor pointer *tEdP* must

be returned by a previous call to `mdlTransient_addElement` or `mdlTransient_addElemDescr`.



Be aware that `mdlTransient_addElement` and `mdlTransient_addElemDescr` return `NULL` if they encounter an error, yet this same value will cause `mdlTransient_replaceElement` and `mdlTransient_replaceElemDescr` to create a new descriptor. If you pass *tEdP* directly from one set of functions to the other without checking its value, you may inadvertently create a new transient descriptor.

If *snappable* is non-zero, the transient element will be snappable.



This is not currently supported; the snappable argument is currently ignored.

viewMask controls the views in which the transient element is displayed, the least significant bit controls the display in view 1, the next bit controls view 2 etc. A view mask of `0x00ff` causes the element to be displayed in all views, a mask of `0x0008` causes display in view 4 only.

displayMode controls the element display mode. Valid display modes are defined in `msdefs.h` and include `NORMALDRAW`, `HILITE`, `ERASE` and `XORDRAW`.

If *displayFirst* is non-zero, the elements in this transient descriptor are displayed before design file graphics, otherwise the elements are displayed after the design file graphics.

If *atHead* is non-zero, the elements in this transient descriptor are displayed before those in other transient descriptors. Otherwise, all transient elements are displayed in the order they were created.

initialDisplay controls the initial display of the element. If *initialDisplay* is zero, the element is added to the transient descriptor but is not displayed.

Returns `mdlTransient_replaceElement` and `mdlTransient_replaceElemDescr` return a pointer to a transient descriptor or `NULL` if an error occurs. If *tEdP* is not `NULL` and the element is successfully replaced, the functions return *tEdP*.

Returns `mdlTransient_addElement`, `mdlTransient_addElemDescr`, `mdlTransient_returnElemDescr`.

mdlTransient_returnElemDescr

```
int *mdlTransient_returnElemDescr
(
MSElementDescr    **edPP,           /* <= element descriptor */
void               *tEdP             /* => TED */
);
```

Description mdlTransient_returnElemDescr returns a pointer to the address of the element descriptor (*edPP*) associated with a transient element descriptor (*tEdP*).

Returns mdlTransient_returnElemDescr returns SUCCESS if the element descriptor is successfully found.

See Also mdlTransient_addElemDescr, mdlTransient_replaceElemDescr.

mdlTransient_free

```
void mdlTransient_free
(
void    **freeTedPP,    /* <=> Transient element to free */
boolean eraseDisplay    /* => TRUE = erase display */
);
```

Description mdlTransient_free frees the transient descriptor pointer *freeTedPP*. The pointer is then set to NULL. The transient descriptor must be returned by a previous call to mdlTransient_addElement or mdlTransient_addElemDescr.

A general purpose transient descriptor is defined by the built-in global variable, msTransientElmP. This transient descriptor is provided for use by primitive commands. It is automatically freed and erased whenever mdlState_startPrimitive is called. It can therefore be used by commands which wish to display graphics while the command is being executed, but a clean up function is not required to clear the graphics if the command is aborted or interrupted.

Returns mdlTransient_free is of type void; it returns no value.

See Also mdlTransient_replaceElemDescr.

6

Element Linkage Functions

The element linkage functions are used to extract, delete or append element attribute data (linkages) from/to elements.

Their purpose is similar to that of `mdlElement_appendAttributes`, `mdlElement_extractAttributes` and `mdlElement_stripAttributes`, but they simplify linkage manipulation in the following ways:

- The extract and delete linkage functions allow the caller to process multiple linkages in an element.
- All functions described in this section can convert the specified element linkages to or from file format (if needed).
- `mdlLinkage_appendToElement` and `mdlLinkage_appendUsingDescr` will automatically set the size of the linkage in the linkage header (if needed).
- The `mdlLinkage_...` functions that use element descriptors provide more flexibility when dealing with complex elements.

Automatic data conversion using data definitions

The element linkage functions described in this section all take a *dataDefinition* resource ID parameter as a means of automatically converting data for the caller. Therefore, the linkage functions require at least one step in preparation for their use:

Structures to be converted to or from file format must be described by a data definition resource. To generate a data definition resource file, list each of the structures to be converted in your application's **type file** (.mt extension). Each structure must appear once in a `createDataDef` statement with an associated resource ID. The .mt file is compiled with the RSCTYPE compiler producing a resource source (.r) file. This generated source file contains the data definitions which describe your structures. Compile this .r file with the resource compiler (rcomp) and merge the subsequent .rsc file into your application's .ma file.

The following table lists the element attribute linkage functions:

Function	Used to
<code>mdlLinkage_extractFromElement</code>	extract one or more linkages of a given type from an element.
<code>mdlLinkage_deleteFromElement</code>	delete one or more linkages of a given type from an element.
<code>mdlLinkage_appendToElement</code>	append a linkage to an element.
<code>mdlLinkage_extractUsingDescr</code>	extract one or more linkages of a given type from one or more elements pointed to by an element descriptor.
<code>mdlLinkage_deleteUsingDescr</code>	delete one or more linkages of a given type from one or more elements pointed to by an element descriptor.
<code>mdlLinkage_appendUsingDescr</code>	append a linkage to one or more elements pointed to by an element descriptor.

mdlLinkage_extractFromElement

```
#include <mselems.h>
#include <mslinkge.fdf>

void *mdlLinkage_extractFromElement /* <= last linkage processed */
(
    void          *inLinkBufP, /* <= save linkage here if non-NULL */
    MSElement    *elemP,      /* => input element */
    int           reqID,        /* => requested link signature */
    ULong         ddbID,        /* => data definition block RscID */
    void          **convRulesPP, /* <=> data conversion rules */
    MdlFunctionP   linkFunc,     /* => function to process links */
    void          *paramsP      /* => parameters to linkFunc() */
);
```

Description The `mdlLinkage_extractFromElement` function is used to extract one or more linkages from the element pointed to by *elemP*.

The *inLinkBufP* parameter points to a buffer where the extracted linkage will be stored. The first 2 words of *inLinkBufP* represents the linkage header and is always converted to internal format for the caller. The remaining portion of *inLinkBufP* is also converted to internal format if the *ddbID* parameter is provided, or if **convRulesPP* points to a previously generated set of conversion rules. If the caller needs to process multiple linkages in the element, *inLinkBufP* is set to `NULL` and a *linkFunc* function pointer is provided instead.

The *elemP* parameter points to the element from which the linkage is extracted. The element is not affected by this function.

The *reqID* parameter is the requested linkage ID. A *reqID* of 0 indicates a DMRS database linkage.

The *ddbID* parameter is the resource ID of a data definition block which describes the structure to be converted. The data definition block is used to convert the linkage data (starting after the 2 word linkage header) from file format to internal format. If *ddbID* is 0 and *convRulesPP* is NULL or points to NULL, then the data portion of the linkage is left in file format. For information on generating data definition resources, see the note at the top of this section.

The *convRulesPP* parameter can be set in one of 3 ways. If it is set to NULL, it indicates that after *ddbID* is used to load the data definition resource, the resulting conversion rules produced can be thrown away once the conversion is complete. Set *convRulesPP* to NULL if you don't intend to convert this type of data very often. Secondly, if *convRulesPP* is a pointer to NULL, then a pointer to the conversion rules produced from the data definition resource are saved in *convRulesPP*. Third, *convRulesPP*, can point to a previously generated set of conversion rules (as described in the second case). In this case, the *ddbID* parameter is ignored.

The *linkFunc* parameter is a pointer to a function that will be called for each linkage encountered with an ID of *reqID*.

The *paramsP* parameter is passed in to *linkFunc* unchanged.

The *linkFunc* should expect the following parameters:

```
int linkFunc /* <= Return 0 to keep processing, 1 to stop. */
(
void *linkageP, /* => extracted linkage. */
void *paramsP /* <=> passed from original call */
);
```

The extracted linkage will always have its 2 word header converted to internal format for the convenience of *linkFunc*. The data portion will also be converted if *ddbID* was specified in the original call.

Returns *mdlLinkage_extractFromElement* returns a pointer to the last element linkage processed in *elemP* or NULL if no linkage was found. The linkage pointer returned points to the raw file format linkage.

See Also *mdlLinkage_extractUsingDescr*, *mdlCnv_bufferFromFileFormat*.

mdlLinkage_deleteFromElement

```

#include <mselems.h>
#include <mslinkge.fdf>

int mdlLinkage_deleteFromElement /* <= # of linkages deleted */
(
MSElement    *elemP,          /* <=> element w/ linkage(s) */
int           reqID,           /* => requested link signature */
ULong         ddbID,           /* => data definition block RscID */
void          **convRulesPP,   /* <=> data conversion rules */
MdlFunctionP  linkFunc,        /* => function to process links */
void          *paramsP         /* => parameters to linkFunc () */
);

```

Description The `mdlLinkage_deleteFromElement` function is used to delete linkage(s) from the element pointed to by *elemP*.

The *elemP* parameter points to the element from which linkages are (potentially) deleted.

The *reqID* parameter is the requested linkage ID. A *reqID* of 0 indicates a DMRS database linkage.

The *ddbID* parameter is the resource ID of a data definition block which describes the structure to be converted. The data definition block is used to convert the linkage data (starting after the 2 word linkage header) from file format to internal format. If *ddbID* is 0 and *convRulesPP* is `NULL` or points to `NULL`, then the data portion of the linkage is left in file format. You would use the *ddbID* parameter to make it easier for *linkFunc* to inspect the linkages.

The *convRulesPP* parameter can be set in one of 3 ways. If it is set to `NULL`, it indicates that after *ddbID* is used to load the data definition resource, the resulting conversion rules produced can be thrown away once the conversion is complete. Set *convRulesPP* to `NULL` if you don't intend to convert this type of data very often. Secondly, if *convRulesPP* is a pointer to `NULL`, then a pointer to the conversion rules produced from the data definition resource are saved in *convRulesPP*. Third, *convRulesPP* can point to a previously generated set of conversion rules (as described in the second case). In this case, the *ddbID* parameter is ignored.

The *linkFunc* parameter is a pointer to a function that will be called for each linkage encountered with an ID of *reqID*. The usual purpose of the *linkFunc* is to decide which linkages should actually be deleted from the element.

The *paramsP* parameter is passed in to *linkFunc* unchanged.

The *linkFunc* should expect the following parameters:

```
int linkFunc /* <= return 0 to keep linkage, 1 to delete it */
(
void *linkageP, /* => linkage to inspect for deletion */
void *paramsP /* <=> passed from original call */
);
```

The extracted linkage will always have its 2 word header converted to internal format for the convenience of *linkFunc*. The data portion will also be converted if *ddbID* was specified in the original call.

Returns *mdlLinkage_deleteFromElement* returns the number of linkages that were deleted.

See Also *mdlLinkage_deleteUsingDescr*, *mdlCnv_bufferFromFileFormat*.

mdlLinkage_appendToElement

```
#include <mselems.h>
#include <mslinkge.fdf>

int mdlLinkage_appendToElement /* <= SUCCESS or ERROR */
(
MSElement *elemP, /* <=> element to receive linkage */
LinkageHeader *linkHdrP, /* => header filled in by caller */
void *linkDataP, /* => data to follow linkhdr */
ULong ddbID, /* => data definition block RscID */
void **convRulesPP /* <=> data conversion rules */
);
```

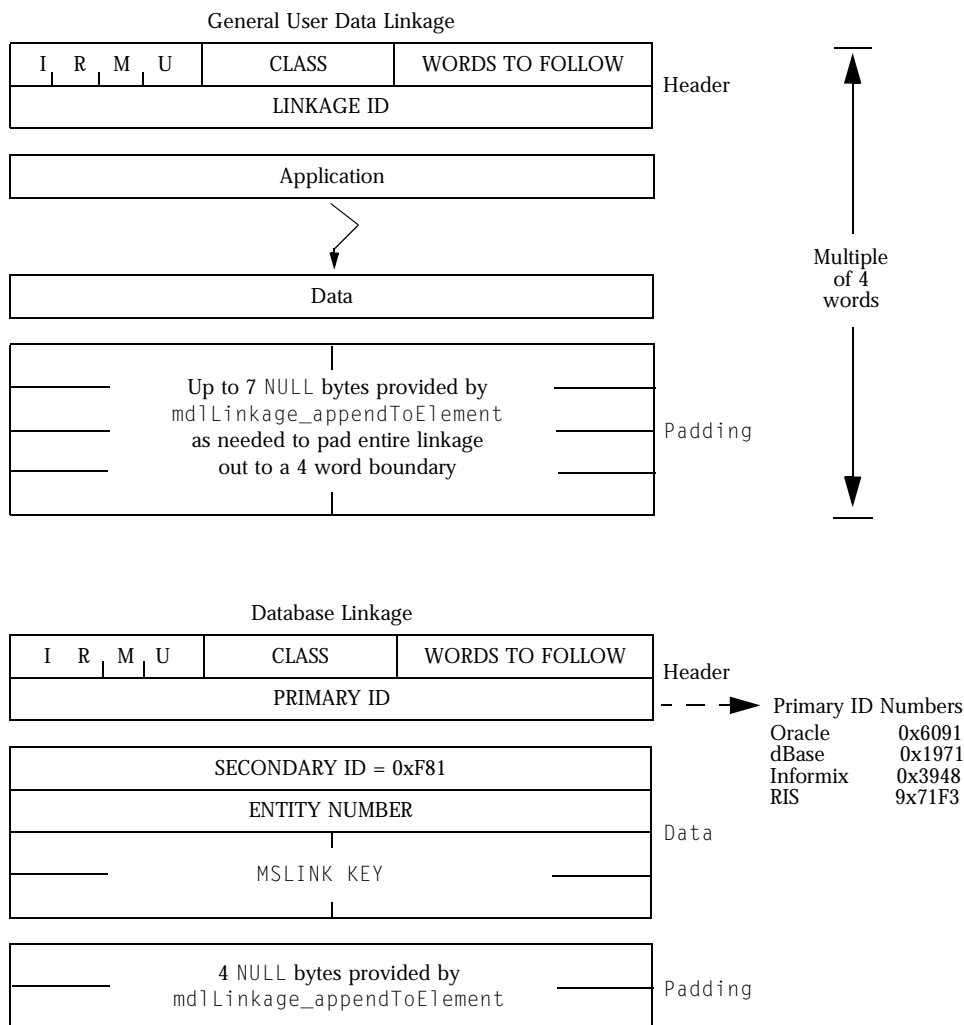
Description The *mdlLinkage_appendToElement* function is used to append a user attribute linkage to the element pointed to by *elemP*.

The *elemP* parameter points to the element that will receive the linkage.

The *linkHdrP* parameter points to a 2 word linkage header that has been filled in by the caller. If the *ddbID* parameter is provided, then the caller does not have to set the size of the linkage in the linkage header. The *mdlLinkage_appendToElement* function will always convert the linkage header to file format for the caller.

The *linkDataP* parameter points to the remaining data of the attribute linkage. This will be appended to the linkage header and added to the element. If the caller supplies the *ddbID* parameter, the data pointed to by *linkDataP* will first be converted to file format.

Below are examples of linkages broken down into their header and data components.



The *ddbID* parameter is the resource ID of a data definition block which describes the structure to be converted. The data definition block is used to convert the linkage data located at *linkDataP* from internal format to file format. Furthermore, if *ddbID* is provided, *mdlLinkage_appendToElement* will automatically pad the linkage out to the necessary 4 word boundary and set the size of the linkage in the linkage header. If *ddbID* is 0 and

convRulesPP is NULL or points to NULL, then it is the caller's responsibility to convert the data and set the linkage size in the linkage header.

The *convRulesPP* parameter can be set in one of 3 ways. If it is set to NULL, it indicates that after *ddbID* is used to load the data definition resource, the resulting conversion rules produced can be thrown away once the conversion is complete. Set *convRulesPP* to NULL if you don't intend to convert this type of data very often. Secondly, if *convRulesPP* is a pointer to NULL, then a pointer to the conversion rules produced from the data definition resource are saved in *convRulesPP*. Third, *convRulesPP*, can point to a previously generated set of conversion rules (as described in the second case). In this case, the *ddbID* parameter is ignored.

Returns mdlLinkage_appendToElement returns SUCCESS if the data was successfully appended.

See Also mdlLinkage_appendUsingDescr, mdlCnv_bufferToFileFormat.

mdlLinkage_extractUsingDescr

```
#include <mselems.h>

#include <mslinkge.fdf>

void *mdlLinkage_extractUsingDescr /* <= last linkage processed */
(
    void          *inLinkBufP,          /* <= linkage if !NULL */
    MSElementDescr *elmDscrP,          /* => input element dscr */
    int            reqID,                /* => requested link signature */
    ULong          ddbID,               /* => data def block RscID */
    void          **convRulesPP,        /* <=> data conversion rules */
    MdlFunctionP   linkFunc,            /* => function to process links */
    void          *paramsP,             /* => parameters to linkFunc () */
    boolean        complexProcessing    /* => proc. complex components? */
);
```

Description The mdlLinkage_extractUsingDescr function is used to extract one or more linkages from one or more elements in the descriptor *elmDscrP*.

The parameters *inLinkBufP*, *reqID*, *ddbID*, *convRulesPP*, *linkFunc* and *paramsP* are the same as those for mdlLinkage_extractFromElement.

The *complexProcessing* flag only affects descriptors containing complex elements. If the flag is set to TRUE, all elements of the descriptor containing linkages matching *reqID* will be passed to *linkFunc*. If the flag is set to FALSE, only outermost headers of complex elements are eligible.

If this function is called with *linkFunc* set to NULL, the effect is to ignore all elements in the descriptor except the last one containing a linkage matching *reqID*.

Returns `mdlLinkage_extractUsingDescr` returns a pointer to the last element linkage processed from the descriptor or `NULL` if no linkage was found. The linkage pointer returned points to the raw file format linkage.

See Also `mdlLinkage_extractFromElement`, `mdlCnv_bufferFromFileFormat`.

mdlLinkage_deleteUsingDescr

```
#include <mselems.h>
#include <mslinkge.fdf>

int mdlLinkage_deleteUsingDescr /* <= number of linkages deleted */
(
  MSElementDescr **elmDscrPP, /* <=> linkages descriptor */
  int reqID, /* => requested link signature */
  ULong ddbID, /* => data definition block RscID */
  void **convRulesPP, /* <=> data conversion rules */
  MdlFunctionP linkFunc, /* => func to process links */
  void *paramsP, /* => parameters to linkFunc() */
  boolean complexProcessing /* => proc complex components? */
);
```

Description The `mdlLinkage_deleteUsingDescr` function is used to delete one or more linkages from one or more elements contained in the descriptor *elmDscrPP*.

The parameters *reqID*, *ddbID*, *convRulesPP* and *paramsP* are the same as those for `mdlLinkage_deleteFromElement`.

The *complexProcessing* flag only affects descriptors containing complex elements. If the flag is set to `TRUE`, all elements of the descriptor containing linkages matching *reqID* will be passed to *linkFunc*. If the flag is set to `FALSE`, only outermost headers of complex elements are eligible.

If this function is called with *linkFunc* set to `NULL`, the effect is to ignore all elements in the descriptor except the last one containing a linkage matching *reqID*.

Returns `mdlLinkage_deleteUsingDescr` returns the number of linkages that were deleted.

See Also `mdlLinkage_deleteFromElement`, `mdlCnv_bufferFromFileFormat`.

mdlLinkage_appendUsingDescr

```
#include <mselems.h>
#include <mslinkge.fdf>

int mdlLinkage_appendUsingDescr /* <= SUCCESS or ERROR */
(
  MSElementDescr    **elmDscrPP,          /* <=> elm descr to rcv linkage */
  LinkageHeader *linkHdrP,                 /* => linkage header */
  void *linkDataP,          /* => data to follow linkhdr */
  ULong ddbID,              /* => data def block RscID */
  void **convRulesPP,       /* <=> data conversion rules */
  boolean complexProcessing /* => proc complex components? */
);
```

Description The `mdlLinkage_appendUsingDescr` function is used to append a user attribute linkage to one or more of the elements contained in the descriptor *elmDscrPP*.

The parameters *linkHdrP*, *linkDataP*, *ddbID* and *convRulesPP* are the same as those for `mdlLinkage_appendToElement`.

The *complexProcessing* flag only affects descriptors containing complex elements. If the flag is set to `TRUE`, all elements of the descriptor will be appended with the linkage. If the flag is set to `FALSE`, only outermost headers of complex elements will receive the linkage.

Returns `mdlLinkage_appendUsingDescr` returns `SUCCESS` if the data was successfully appended.

See Also `mdlLinkage_appendToElement`, `mdlCnv_bufferToFileFormat`.

7

Element Location and Manipulation

This chapter discusses Scan functions, Element location functions, Element modification functions, Selection set processing functions and Fence processing functions.

In addition, this discussion covers Surface creation functions (3D only), Complex chain creation functions, Element clipping functions and Dynamic buffer functions.

Scan Functions

The following table lists the scan functions:

Function	Used to
<code>mdlScan_initScanlist</code>	initialize a scan list.
<code>mdlScan_setDrawnElements</code>	set the type mask member so that the scan list returns all displayable MicroStation elements.
<code>mdlScan_noRangeCheck</code>	set the range members of the scan list so that the scan list encompasses the entire design plane.
<code>mdlScan_singleViewClass</code>	set the class mask member of the scan list so that only element classes that are currently displayed are returned.
<code>mdlScan_viewRange</code>	set the scan range to find all elements in a view.
<code>mdlScan_initialize</code>	load a scan list into the MicroStation design file scanner.
<code>mdlScan_initOpenedFile</code>	load a scan list into the MicroStation design file scanner for a fopened file.
<code>mdlScan_file</code>	scan the design file according to previously established criteria.

Function	Used to
<code>mdlScan_extended</code>	scan the design file according to previously established criteria and gain control after each matching element is found.
<code>mdlScan_saveContext</code>	temporarily save the state of the scanner when a program will need to continue later.
<code>mdlScan_restoreContext</code>	restore the state of the scanner saved previously by <code>mdlScan_saveContext</code> .

mdlScan_initScanlist

```
#include <scanner.h>

void mdlScan_initScanlist
(
  ScanList      *scanlist      /* => scan list */
);
```

Description The `mdlScan_initScanlist` function initializes a MicroStation scan list pointed to by *scanlist* for subsequent calls to `mdlScan_file`. The scan list is zeroed and the scan list length member is set to the proper size.

Returns The `mdlScan_initScanlist` function is of type `void` and returns no value.

See Also `mdlScan_setDrawnElements`, `mdlScan_noRangeCheck`, `mdlScan_singleViewClass`, `mdlScan_initialize`, `mdlScan_file`.

mdlScan_setDrawnElements, mdlScan_noRangeCheck, mdlScan_singleViewClass, mdlScan_viewRange

```
#include <scanner.h>

void mdlScan_setDrawnElements
(
  Scanlist      *scanlist      /* => scan list */
);

void mdlScan_noRangeCheck
(
  Scanlist      *scanlist      /* => scan list */
);

void mdlScan_singleViewClass
(
  Scanlist      *scanlist,      /* => scan list */
```



```
int          viewNumber      /* => view to scan */
);

void mdlScan_viewRange
(
Scanlist     *scanlist,      /* => scan list */
int          viewNumber,     /* => view to scan */
int          fileNumber      /* => file to scan */
);
```

Description The MicroStation design file scanning utility `mdlScan_file` is used for several purposes. `mdlScan_setDrawnElements`, `mdlScan_singleViewClass`, `mdlScan_noRangeCheck` and `mdlScan_viewRange` ease the task of establishing scan criteria.

`mdlScan_setDrawnElements` sets the type mask member *scanlist->typmask* so the scan list returns displayable elements when used in subsequent calls to `mdlScan_file`. The `ELEMENTYPE` bit in *scanlist->scantype* must be set to cause the scanner to check element types.

`mdlScan_noRangeCheck` sets the range members of the scan list pointed to by *scanlist* (*scanlist->xlowlim*, *scanlist->xhighlim*, *scanlist->ylowlim*, *scanlist->yhighlim*, *scanlist->zlowlim*, *scanlist->zhighlim*) so that the scan list encompasses the entire design cube. When this scan list is used in subsequent calls to `mdlScan_file`, no elements are rejected based on range criteria. Since the scanner always checks element ranges, make sure you call `mdlScan_noRangeCheck` if you do want to accept elements regardless of their range.

`mdlScan_singleViewClass` sets the class mask member of the scan list pointed to by *scanlist* (*scanlist->clasmask*) so that subsequent calls to `mdlScan_file` return only the element classes that currently display in view *viewNumber*. The `PROPCLAS` bit in *scanlist->scantype* must be set to cause the scanner to check element class.

`mdlScan_viewRange` sets the range members of the scan list pointed to by *scanlist* (*scanlist->xlowlim*, *scanlist->xhighlim*, *scanlist->ylowlim*, *scanlist->yhighlim*, *scanlist->zlowlim*, *scanlist->zhighlim*) to find only elements in view *viewNumber* and from file *fileNumber*.

Returns `mdlScan_setDrawnElements`, `mdlScan_noRangeCheck`, `mdlScan_singleViewClass` and `mdlScan_viewRange` return no values.

See Also `mdlScan_initScanlist`, `mdlScan_initialize`, `mdlScan_file`.

mdlScan_initialize

```
#include <scanner.h>

int mdlScan_initialize
(
    int          fileNumber,      /* => file to scan */
    Scanlist     *scanlist       /* => scan list */
);
```

Description The `mdlScan_initialize` function loads a scan list to the MicroStation design file scanner for subsequent calls to `mdlScan_file`.

The scan list pointed to by *scanlist* defines the scan criteria to be used. *fileNumber* specifies the file to be scanned. File 0 is the master design file, 1 through `MAX_REFS` are the attached reference files, and the `CELL_LIB` file is the attached cell library. `MAX_REFS` and `CELL_LIB` are defined in `msdefs.h`.

Scanlist is a structure defined in `scanner.h`. It consists of members that determine elements that are returned in subsequent calls to `mdlScan_file`. The meanings of the members are as follows:

Member name	Meaning
<code>sllen</code>	This member defines the length of the scan list. For compatibility with the IGDS scan list, bit 12 can be set (as it must be on the VAX). The easiest way to set it properly is to call <code>mdlScan_initScanlist</code> before setting other <code>scanlist</code> members.
<code>scantype</code>	This member sets various parameters for the scan, enabling the search criteria and controlling the type of information that the scan returns. Each bit in the <code>scantype</code> member enables a scan criterion or controls the information returned. The bits may be ORed to enable several criteria. The bit values are defined in <code>scanner.h</code> , and are named: <code>ELEMDATA</code> , <code>NESTCELL</code> , <code>PICKCELL</code> , <code>PROPCLAS</code> , <code>GRPHGRP</code> , <code>MULTI</code> , <code>BOTH</code> , <code>ONEELEM</code> , <code>ATTRENT</code> , <code>ATTROCC</code> , <code>STOPSECT</code> , <code>LEVELS</code> and <code>ELEMTYPE</code> . The <code>ELEMDATA</code> and <code>BOTH</code> bits control whether the scanner returns file positions, actual elements, or <code>BOTH</code> . When <code>BOTH</code> is set, both file positions and the actual elements are set regardless of the setting of <code>ELEMDATA</code> . When neither <code>BOTH</code> or <code>ELEMDATA</code> is set, file positions only are returned. When the <code>ONEELEM</code> bit is set, the scanner finds one element only and then returns <code>BUFF_FULL(11)</code> . The effect of the other bits are discussed below.
<code>tplval</code> <code>tplmsk</code>	The <code>tplval</code> and <code>tplmsk</code> members are used together to cause the scanner to return only elements of a given type or given level (or both). The scanner takes the first word of each element, ANDs it with <code>tplmsk</code> , and compares it to <code>tplval</code> . The element is accepted only if these values are equal. The scanner always performs this check; to disable the check, set <code>tplval</code> and <code>tplmsk</code> to zero.

Member name	Meaning
xlowlim ylowlim zlowlim xhighlim yhighlim zhighlim	<p>These members define a cube in the design file. If the element range intersects this cube, the element is returned. Otherwise, it is rejected.</p> <p>For historical reasons, the range values in the scan list are stored in an unusual format (see mdlCnv_toScanFormat). The range check is always performed. To effectively disable this check, set each dimension's range using values from -2,147,483,648 (0x80000000) to +2,147,483,647 (0x7fffffff). The easiest way to set these values is to use mdlScan_noRangeCheck.</p>
pcl.propval pch.propmsk pcl.cell0 pch.cell1	<p>The structure's two union members specify element properties and a cell name, depending on whether the PROPCLAS or PICKCELL bit is set. If the PROPCLAS bit is set, the scanner checks the element's properties and class. The scanner checks properties by joining pch.propmsk with the element's properties bits using the AND operator and then comparing the result with pcl.propval. If these values do not match, the element is rejected. The scanner checks the class by checking the bit corresponding to the element class in the clasmask member. If the bit is not set, the element is rejected.</p> <p>If the PICKCELL bit is set, the pcl.cell0 and pch.cell1 members are compared to the low and high words of the cell name stored in element types 1 and 2. If no exact match is found, the element is rejected.</p>
grgroup	<p>This member is set to the graphic group number for which data is needed. If the GRPHGRP bit in scantype is set, elements that do not match the graphic group member are rejected.</p>
sector offset	<p>These members generate a file position where the scan will start. Use the DGN_BLOCK and DGN_OFFSET members to convert from a file position to sector and offset.</p>
entity occurance	<p>These members selects elements according to the attribute data attached to them. If the ATTRENT or ATTROCC bit is set in scantype, only elements with attribute data will be returned. The entity value and/or occurrence value must match. If they do not, the elements will be rejected. These checks are only valid for DMRS type linkages and are of very limited use. They are included for compatilby with legacy applications only.</p>
levmask	<p>This member determines the levels of elements that will be returned. Bit 0 of levmask[0] must be set for element on level 1 to be returned. Bit 14 of levmask[3] must be set for elements on level 63 to be returned. To check for elements on level 64 (stored as level 0 in the design file), set bit 15 of levmask[3]. To enable level checking, set the LEVELS bit in scantype.</p>

Member name	Meaning
typmask	This member determines the element types that will be returned. Bit 0 of <code>typmask[0]</code> must be set for element type 1 to be returned. Bit 14 of <code>typemask[8]</code> must be set for element type 127 to be returned. The MDL scan list contains eight words of type masks rather than the four words in IGDS.
stopsector	This member defines the sector where the scan will stop. This member is effective only if the <code>STOPSECT</code> bit is set in <code>scantype</code> . This feature is rarely used.
extendedType	Certain bits of this member enable some additional features of the scanner. These bits may be individually set by ORing together the values <code>RETURN3D</code> , <code>FILEPOS</code> , <code>EXTATTR</code> and <code>ITERATEFUNC</code> defined in <code>scanner.h</code> . When the <code>RETURN3D</code> bit is set, the elements returned from a 2D file will be returned in the 3D format. When the <code>FILEPOS</code> bit is set, values that would normally be returned as <code>unsigned short sector, offset</code> pairs are returned in a single <code>unsigned long file position</code> instead. When the <code>EXTATTR</code> bit is set, the <i>extAttrBuf</i> member of the scan list is used. The <code>ITERATEFUNC</code> member is used in conjunction with the <code>mdlScan_extended</code> function.

Member name	Meaning
exrange	This member is an array of extended range blocks that can be used if more than one range cube in the design plane is needed (as it frequently is in multi-view updates). If the <code>MULTI</code> bit is set in <code>scantype</code> and the <code>sllen</code> member is large enough to contain one or more extended range blocks, the scanner returns all elements that pass all other enabled criteria and meet any enabled scan ranges. The scanner steps through the extended range blocks until it passes the end of the enable ranges (as determined from <code>sllen</code>) or encounters a range with <code>xlowlim</code> set to a positive number and <code>xhighlim</code> set to a negative number.
extAttrBuf	<p>This member points to a structure of type <code>ExtendedAttrBuf</code>, and provides a flexible mechanism for selecting elements based on their nongraphical attributes. The <code>EXTATTR</code> bit of the <i>extendedType</i> member must be set to enable this test. The <code>ExtendedAttrBuf</code> structure is defined in <code>scanner.h</code> as:</p> <pre>typedef struct { short numWords; Ushort extAttData[32]; } ExtendedAttrBuf;</pre> <p>The <i>numWords</i> member of this structure is set to the number of words in the attribute data that are to be considered at the beginning of each attribute linkage. The <i>extAttData</i> array specifies how the scanner is to test those <i>numWords</i>, which it does as follows: The first word of each attribute linkage is ANDed with <i>extAttData</i>[0] and the result compared with <i>extAttData</i>[<i>numWords</i>]. If the value matches, the check continues by ANDing the second word of the linkage with <i>extAttData</i>[1] and comparing with <i>extAttData</i>[<i>numWords</i>+1], etc. If the values for all <i>numWords</i> of any attribute linkage match, the element is accepted. Otherwise the element is rejected.</p>



The `PROPCLAS` and `PICKCELL` bits in `scantype` are mutually exclusive, and the `BOTH` bit overrides the value of the `ELEMDATA` bit. The effect of `NESTCELL` bit is as follows:

If `NESTCELL` is set and a complex header is rejected, all of the complex element's component elements are rejected also. Similarly, all component elements of the complex element are returned when the complex header is accepted. When only element data is being returned, the scanner is always in nest mode. When file positions are being returned (`BOTH` set or `ELEMDATA` clear) in nest mode, the scanner will return only complex header information and never any of the component elements. When file positions are being returned in unnest mode, the scanner can return elements of a complex even if the complex header is rejected. It can also

reject elements that are components of a complex element, even if the complex header is accepted.

Returns The `mdlScan_initialize` function returns `SUCCESS` if the scanner accepts the scan list. It returns `MDLERR_BADFILENUMBER` if *fileNumber* is not in the 0 through `CELL_LIB` range; it returns `MDLERR_CANTOPENFILE` if the indicated reference file or cell library cannot be opened; and it returns `MDLERR_BADSCANLIST` if the size in *scanlist->sllen* is wrong.

See Also `mdlScan_setDrawnElements`, `mdlScan_noRangeCheck`, `mdlScan_singleViewClass`, `mdlScan_initScanlist`, `mdlScan_file`.

mdlScan_initOpenedFile

```
#include <scanner.h>

int mdlScan_initOpenedFile
(
    FILE          *fileP,          /* => file pointer of file */
    ExtScanList *scanListP        /* => scan list */
);
```

Description `mdlScan_initOpenedFile` loads the scan list pointed to by *scanListP* to the file pointed to by *fileP* for subsequent calls to `mdlScan_file`. The scan list is zeroed and the scan list length member is set to the proper size.

Returns The `mdlScan_initOpenedFile` returns `SUCCESS` upon successful initialization of the scan list. It returns `MDLERR_BADFILENUMBER` if *fileP* parameter does not point to a valid design file opened with the `fopen` routine, and it returns `MDLERR_BADSCANLIST` if the size in *scanlist->sllen* is wrong.

See Also `mdlScan_initScanlist`, `mdlScan_initialize`, `mdlScan_file`.

mdlScan_file

```
#include <scanner.h>

int mdlScan_file
(
    char    *scanBuffer,    /* <= accepted data */
    int     *acceptedSize,  /* <= amount of information in scanBuffer */
    int     bufferSize,    /* => size of scanBuffer */
    ULong   *filePos        /* <= ending file position */
);
```

Description `mdlScan_file` scans the design file according to the criteria established in the last call to `mdlScan_initialize`. When the scan completes, **filepos* is set to the start of the next element in the file.

The accepted elements (and/or element addresses) are copied to the output buffer pointed to by *scanBuffer*. *bufferSize* should be set to the size

in bytes of *scanBuffer*. MicroStation sets the integer pointed to by *acceptedSize* to the number of words returned in *scanBuffer*.

Returns mdlScan_file returns one of the following values:

Value	Meaning
10	End of file reached, (END_OF_DGN).
11	<i>scanBuffer</i> full before end of file, (BUFF_FULL).
65	Bad file number.
66	Physical end of file encountered before MicroStation end of file mark.
67	Improper scan list.
68	Invalid element encountered.

The values END_OF_DGN and BUFF_FULL are encountered during normal scan operation. The other return values are returned when scan errors are encountered.

See Also mdlScan_extended, mdlScan_setDrawnElements, mdlScan_noRangeCheck, mdlScan_singleViewClass, mdlScan_initialize, mdlScan_initScanlist.

mdlScan_extended

```
int mdlScan_extended
(
void    *scanBuffer,          /* <= output buffer */
int     *scanSize,           /* <=> size of buffer, words returned */
int     *eofBlock,           /* <= block where we stopped scanning */
int     *eofByte,            /* <= byte where we stopped scanning */
int     (*iteratorFunc)(),    /* => func called for each accepted elm */
void    *iteratorArg         /* => argument passed to iteratorFunc */
);
```

Description mdlScan_extended provides the same functionality as the mdlScan_file function, but also provides the ability for the application program to receive control after each element which meets the scanning criteria instead of waiting for the scanner to return the information after each call. That is, the complete file can be scanned in a single call to this function.

When the *iteratorFunc* argument is a NULL pointer, the accepted elements (and/or element addresses) are copied to the output buffer pointed to by *scanBuffer*. *scanSize* should be set to the size in words of *scanBuffer*; contains the number of words returned in *scanBuffer* on return. *eofBlock* contains the block number and *eofByte* contains the byte position of the next element to be scanned upon return. These values can be used to calculate the file position of the next element to be processed by using the DGN_FILEPOS (contained in msdefs.h) macro as shown below:

```
filePos = DGN_FILEPOS(eofBlock, eofByte);
```

iteratorFunc is a pointer to a function which will be called by MicroStation for each element which meets the scanning criteria specified in the scanlist. If the value of this parameter is `NULL`, then element information is returned in the *scanBuffer* parameter. If this value is non-`NULL`, then the function specified is called once for each element which meets the scan criteria and no information is returned in *scanBuffer* and *scanSize*. The function pointed to by *iteratorFunc* is an integer function which is called with two parameters:

```
int iterator_func
(
  MSElementDescr *elmDscrP,      /* => elm meeting scan criteria */
  void             *iteratorArg    /* => user defined parameter */
);
```

This function should return either `SUCCESS` or `ERROR`. `SUCCESS` indicates that the scanning process can continue and `ERROR` causes the scanning process to be halted and a return code of `BUFF_FULL` to be returned to the calling application. To use the *iteratorFunc*, make sure the `ITERATEFUNC` bit is set in the `extendedType` member of the scan list.

Returns `mdlScan_extended` returns `SUCCESS` if the scan terminates normally. If an error occurs, `mdlScan_extended` returns one of the following error values:

Value	Reason
10	End of file reached (<code>END_OF_DGN</code>).
11	<i>scanBuffer</i> full before end of file, (<code>BUFF_FULL</code>).
65	Bad file number.
66	Physical end of file encountered before MicroStation end of file mark.
67	Improper scan list.
68	Invalid element encountered.

See Also `mdlScan_file`.

mdlScan_saveContext

```
#include <scanner.h>

void mdlScan_saveContext
(
  ScanContext *scanContextBuffer /* scan buffer */
);
```

Description The `mdlScan_saveContext` function is used to temporarily save the state of the scanner when a program needs to continue scanning where it left off after calling another function that also uses the scanner. For example, the `cellx` example

program scans a file for all cell header elements. It then reads each cell in as an element descriptor, checks to see if the element is already in the attached cell library, and if not, calls `mdlCell_addLibDescr` to add the cell. Both searching the library to see if it exists (`mdlCell_existsInLibrary`) and adding the cell (`mdlCell_addLibDescr`) use the scanner. Since the program needs to continue to scan the design file to find the cell headers, it must save the scan context before using the two functions mentioned above, and restore the saved context (using `mdlScan_restoreContext`) before continuing its scan. Other built-in functions that use the scanner include `mdlFence_process`, `mdlLocate_...` functions, `mdlCell_createFromFence`, `mdlCell_deleteInLibrary`, `mdlCell_getFilePosInLibrary`, `mdlCell_attachLibrary`, `mdlCell_rename`, `mdlCell_addLibElement`, `mdlCell_addLibDescr`, `mdlCell_generateLibIndex`, `mdlView_fit`, `mdlView_...Named` functions, `mdlView_updateMulti`, `mdlScan_initialize`, `mdlScan_initOpenedFile`, `mdlScan_file`, `mdlSelect_allElements`, `mdlWorkDgn_findEof` and `mdlPattern_...` functions.

scanContextBuffer points to a `ScanContext` structure that is used to hold the information the scanner needs to store to specify its current state.

Returns `mdlScan_saveContext` is of type `void` and has no return value.

See Also `mdlScan_restoreContext`.

mdlScan_restoreContext

```
#include <scanner.h>

void mdlScan_restoreContext
(
    ScanContext *scanContextBuffer
);
```

Description The `mdlScan_restoreContext` function is used to restore the state of the scanner that was saved by a previous call to `mdlScan_saveContext`. See `mdlScan_saveContext` for information concerning its usage.

scanContextBuffer points to a `ScanContext` structure that was previously filled by a call to `mdlScan_saveContext`.

Returns `mdlScan_restoreContext` is of type `void` and has no return value.

See Also `mdlScan_saveContext`.

Element Location Functions

Most MDL applications identify existing elements in the design file based on criteria that the developer can specify. The element location functions make identifying these elements as easy as possible for normal selection cases and are extendable for the unusual cases.

To locate elements, applications can use criteria such as element type, location, properties (color, weight, and style), level, file number, locked status and attribute information. MicroStation has a set of element location variables in the `TCB` structure. These variables define the current **search masks** to be used during location. The following table lists `tcb` variables:

Variable name	Type	Meaning during location
<code>tcb->propval</code> <code>tcb->propmsk</code>	short short	The properties word of the element header is joined with <code>tcb->propmsk</code> using the AND operator, and the result is tested for equality with <code>tcb->propval</code> . If the test is <code>TRUE</code> , the element is accepted.
<code>tcb->searchFile</code>	8 longs	Each bit corresponds to one file. The master file is bit 0. If the bit is set, the file is searched.
<code>tcb->searchType</code>	8 shorts	Each bit corresponds to an element type. Element type 1 is bit 0. If the bit is set, the element type is accepted.

Element location is most commonly used to find a single element to modify. The `mdlState_startModifyCommand` function sets up a data point function to be `mdlLocate_identifyElement`. It manages the normal element selection process for MicroStation element modification commands.

Typical element modification commands use `mdlState_startModifyCommand` for element location. The commands call the various element location functions to change the search masks to determine which elements are accepted. In some cases, however, MDL applications locate elements differently. To do so, they call `mdlLocate_findElement` or `mdlLocate_identifyElement` directly.

The following table lists element location functions:

Function	Used to
<code>mdlLocate_normal</code>	set the search masks to find all normal modifiable elements in the master file.
<code>mdlLocate_allowLocked</code>	set the search masks to find all displayable elements in all files.
<code>mdlLocate_noElemNoLocked</code>	set the search masks to find only unlocked elements in the master file. It clears the element type mask so that no elements are accepted.
<code>mdlLocate_noElemAllowLocked</code>	set the search masks to find elements in all files. It clears the element type mask so that no element types are accepted.
<code>mdlLocate_setElemSearchMask</code>	set the bits in the element search mask for the given element types.

Function	Used to
<code>mdlLocate_clearElemSearchMask</code>	clear the bits in the element search mask for the given element types.
<code>mdlLocate_identifyElement</code>	identify the next element in the locate range.
<code>mdlLocate_restart</code>	restart an element location command.
<code>mdlLocate_init</code>	reset the restarting locate flag so that the next element location process starts at the beginning of the file.
<code>mdlLocate_findElement</code>	locate an element.
<code>mdlLocate_getProjectedPoint</code>	get the information about the point on the accepted element that is closest to the selection point.
<code>mdlLocate_setCursor</code>	set the MicroStation cursor to the element location cursor.
<code>mdlLocate_setFunction</code>	designate one of the user functions below to be called during MicroStation's element location process.

The following table lists element location user functions. The programmer can change the user function names. Function pointers designate these functions to MDL, but MDL does not use the names.

Function	Used to
<code>userLocate_elementFilter</code>	examine those elements MicroStation examines while scanning.
<code>userLocate_globalFilter</code>	examine all elements.
<code>userLocate_selectCmd</code>	called whenever the data button is pressed while the selection tool is active.

Examples

See `locate.mc`.

`mdlLocate_normal`, `mdlLocate_allowLocked`, `mdlLocate_noElemNoLocked`, `mdlLocate_noElemAllowLocked`

```
void mdlLocate_normal(void);
void mdlLocate_allowLocked(void);
void mdlLocate_noElemNoLocked(void);
void mdlLocate_noElemAllowLocked(void);
```

Description These functions change the element location search masks used by the MicroStation element location logic. These functions should be called after `mdlState_startModifyCommand` or before `mdlLocate_findElement`.

The `mdlLocate_normal` function sets the search mask for element properties to locate only unlocked elements. It sets the search mask for files to locate only elements in the master file. It also sets the search mask for element types to find all displayable elements.

The `mdlLocate_allowLocked` function sets the search mask for element properties to locate all elements. It sets the search mask for files to locate elements from all files. It also sets the search mask for element types to find all displayable elements.

The `mdlLocate_noElemNoLocked` function sets the search mask for element properties to locate only unlocked elements. It also sets the search mask for files to locate only elements in the master file. It clears all bits in the search mask for element types. This function is useful for finding only a few element types. After calling this function, call `mdlLocate_setElemSearchMask` with the element types to be located.

The `mdlLocate_noElemAllowLocked` function sets the search mask for element properties to locate all elements. It sets the search mask for files to locate elements from all files. It clears all bits in the search mask for element types. This function is useful for finding only a few element types. After calling this function, call `mdlLocate_setElemSearchMask` with the element types to be located.

Returns The `mdlLocate_normal`, `mdlLocate_allowLocked`, `mdlLocate_noElemNoLocked`, and `mdlLocate_noElemAllowLocked` functions are of type `void`. They return no values.

See Also `mdlLocate_setElemSearchMask`, `mdlLocate_clearElemSearchMask`.

`mdlLocate_setElemSearchMask`, `mdlLocate_clearElemSearchMask`

```
void mdlLocate_setElemSearchMask
(
    int      numElems,          /* => num of entries in elementNumber array */
    int      *elementNumber /* => element number array */
);

void mdlLocate_clearElemSearchMask
(
    int      numElems,          /* => num of entries in elementNumber array */
    int      *elementNumber /* => element number array */
);
```

Description These functions set or clear bits corresponding to element types in the element type search mask. Bits are set or cleared for the element location routines. These

functions should be called after `mdlState_startModifyCommand` or before `mdlLocate_findElement`.

Each function accepts two arguments. The second argument, *elementNumber*, is an array of `ints`. Each `int` holds one element type to be turned on or off. The first argument, *numElems*, is the number of entries in *elementNumber*.

The `mdlLocate_setElemSearchMask` function turns on the bits in the element type search mask for each member of *elementNumber*.

The `mdlLocate_clearElemSearchMask` function turns off bits in the element type search mask for each member of *elementNumber*.

Returns The `mdlLocate_setElemSearchMask` and `mdlLocate_clearElemSearchMask` functions are of type `void`. They return no values.

See Also `mdlLocate_normal`, `mdlLocate_allowLocked`, `mdlLocate_noElemNoLocked`, `mdlLocate_noElemAllowLocked`.

mdlLocate_identifyElement

```
int mdlLocate_identifyElement
(
    int      allowComponents,    /* => allow component elements? */
    boolean notFoundMsg         /* => display "element not found" */
);
```

Description MicroStation uses the `mdlLocate_identifyElement` function internally to locate elements for modification. It restarts the element location process when the user rejects an element. It also displays the prompts, calls `show` and `clean` functions, and establishes the `accept` function as the data point function.

`mdlLocate_identifyElement` automatically becomes the data point function when `mdlState_startModifyCommand` is called. MDL applications that need non-standard element location can call `mdlLocate_identifyElement` directly.

The `mdlLocate_identifyElement` function uses the following logic:

```
{
    if (an element is currently highlighted)
        unhighlight the element;
    if (we are starting a new locate)
        save the data point and the view number;
    call mdlLocate_findElement(saved data point and view)
    if (we found an element)
    {
        display accept/reject prompt;
        put information about element in message field;
        assign data point function to user's accept function;
        assign reset function to internal function that calls
```

```

        user's clean function and then calls this function;
        load dynamics buffers;
        call user's show function;
        set the restarting locate flag;
    }
    else
    {
        if (want not found message)
            display element not found error message;
        set relerr to 21;
    }
}

```

allowComponents indicates how complex elements are treated during the location process. `mdlLocate_identifyElement` passes this parameter unchanged to `mdlLocate_findElement`. See `mdlLocate_findElement` for an explanation of the possible values for *allowComponents*.

notFoundMsg is a boolean flag indicating whether the “Element not found” error message displays.

Returns `mdlLocate_identifyElement` returns `SUCCESS` if an element was located and the current element information is valid. (See `mdlLocate_findElement` for further discussion of this information). It returns `ERROR` if no element was found.

See Also `mdlLocate_findElement`, `mdlState_startModifyCommand`.

mdlLocate_restart

```

void mdlLocate_restart
(
    int      restartFlag    /* => TRUE=start from beginning of file */
);

```

Description Commands use the `mdlLocate_restart` function to locate another element while they are locating and changing elements. This function first calls `mdlState_checkSingleShot`. If single-shot mode is not active, it then calls `mdlLocate_identifyElement`, setting the internal restart flag to *restartFlag*.

If *restartFlag* is `TRUE`, the search restarts from the beginning of the file. If `FALSE`, the search continues from the position in the file where it left off.

MicroStation uses `mdlLocate_restart` for restarting commands like `CHANGE COLOR` and `DROP COMPLEX`.

Returns The `mdlLocate_restart` function is of type `void`. It returns no value.

See Also `mdlState_checkSingleShot`, `mdlLocate_identifyElement`.

mdlLocate_init

```

void mdlLocate_init(void);

```

Description The mdlLocate_identifyElement function keeps an internal flag (*tcb->restartLocate*) that specifies whether the mdlLocate_findElement function starts at the beginning of the design file or continues from the last element located. The mdlLocate_init function sets this internal restart locate flag to FALSE to force locates to start at the beginning of the file.

Returns The mdlLocate_init function is of type void. It returns no value.

See Also mdlLocate_findElement, mdlLocate_identifyElement.

mdlLocate_findElement

```
ULong mdlLocate_findElement
(
    Dpoint3d    *inPoint,      /* => point to search about */
    int         viewNum,       /* => view number inPoint was in */
    int         restart,       /* => restart at next element */
    int         allowComponents, /* => complex element mode */
    boolean hilite             /* => hilite located element */
);
```

Description The mdlLocate_findElement function is the lowest-level routine in the element location scheme. The mdlLocate_identifyElement function calls this function automatically. This element location function scans the design file and all of its reference files searching for elements that are near the location point, *inPoint*, and meet the search criteria. If an element is found, it becomes the current element and mdlLocate_findElement stops.

For each element that is accepted for its proximity to *inPoint*, mdlLocate_findElement calls the application's established LOCATE_PRELOCATE and LOCATE_POSTLOCATE user functions. If the element is accepted, it becomes the current element. mdlLocate_findElement then puts a copy of it in dgnBuf and saves its address. Applications can retrieve the address of the current element and the address of the next element in the file by using the mdlElement_getFilePos function with the FILEPOS_CURRENT and FILEPOS_NEXT_ELE elements.

inPoint and *viewNum* are used together to determine a range for element acceptance. *viewNum* is needed because the location tolerance (set in the User Preference Dialog) is specified in pixels on the screen. Therefore, the view magnification must be known.

If *restart* is 1, the location process is started at the address defined by FILEPOS_NEXT_ELE. If *restart* is 0, mdlLocate_findElement starts at the address defined by FILEPOS_WORKING_WINDOW. In either case, if the user snapped to an element using a tentative point, the address of that element becomes the starting address for the search.

allowComponents is a flag that determines how complex element components are handled. The following table describes the possible values for *allowComponents*:

allowComponents	Meaning
0	Always set the current element to the address of the outermost header for complex elements.
1	Set current element to complex components.
2	Set current element to the innermost nested headers for nested elements.

If a complex header is returned, the address of the component element that accepted the complex element can be retrieved with `FILEPOS_COMPONENT` as the argument to `mdlElement_getFilePos`.

If *hilite* is `TRUE`, elements are highlighted when they are accepted.

Returns The `mdlLocate_findElement` function returns the current element's file position. It returns 0 if no element was found.

See Also `mdlLocate_init`, `mdlElement_getFilePos`, `mdlLocate_identifyElement`, `mdlState_startModifyCommand`, `mdlLocate_getProjectedPoint`.

mdlLocate_getProjectedPoint

```
void mdlLocate_getProjectedPoint
(
    Dpoint3d    *locatePoint, /* <= point on element from last locate */
    int         *segment,     /* <= segment number of last locate */
    int         *view         /* <= view number for last locate */
);
```

Description The `mdlLocate_getProjectedPoint` function returns information about the last call to `mdlLocate_findElement`. It must be called after a successful call to `mdlLocate_findElement` or `mdlLocate_identifyElement`, or from an accept function from `mdlState_startModifyCommand`.

locatePoint is the point on the current element that is closest to the locate point. It is returned in the current coordinate system.

If the current element is a line string, shape, or curve, *segment* is the segment number from which *locatePoint* was derived.

view is the view number used to locate the current element.

Returns The `mdlLocate_getProjectedPoint` function is of type `void`. It returns no value.

See Also `mdlLocate_findElement`, `mdlLocate_identifyElement`, `mdlState_startModifyCommand`.

mdlLocate_setCursor

```
void mdlLocate_setCursor(void);
```

Description The mdlLocate_setCursor function sets the MicroStation cursor to the element location cursor. This cursor is the arrow cursor with the location circle centered on the tip of the pointer.

The mdlLocate_setCursor function is needed when applications change the state functions directly without calling mdlState_startModifyCommand.

Returns The mdlLocate_setCursor function is of type void. It returns no value.

mdlLocate_setFunction

```
#include <userfnc.h>

MdlFunctionP mdlLocate_setFunction
(
    int          type,      /* => LOCATE_PRELOCATE */
    MdlFunctionP function /* => address of MDL function */
);
```

Description The mdlLocate_setFunction function designates user functions to be called during MicroStation's element location process.

type specifies the user filter function within the element location logic to be set. Possible values for *type* are as follows:

Filter Function	When Called
LOCATE_PRELOCATE	Before MicroStation decides to accept an element.
LOCATE_POSTLOCATE	After MicroStation decides to accept an element.
LOCATE_ACCEPTFUNC	User accepts the highlighted element. This function is also set with the <i>pAccept</i> argument to mdlState_startModifyCommand.
LOCATE_SHOWFUNC	To show the result of a selection. This function is also set with the <i>pShow</i> argument to mdlState_startModifyCommand.
LOCATE_CLEANFUNC	To clean up the result of a selection. This function is also set with the <i>pClean</i> argument to mdlState_startModifyCommand.
LOCATE_NOTFOUNDFUNC	Locate logic passes through all files without finding an element.
LOCATE_GLOBAL_LOCATE	While MicroStation is considering elements for processing at various times. This function is remains active, regardless of the active command.
LOCATE_SELECT_CMD	The data button is pressed while the selection tool is active.

function must be a valid MDL function pointer or NULL.

The LOCATE_PRELOCATE user function is called for all elements that MicroStation finds by scanning the file for an intersection of the identification point and the element range. This will typically find a large

number of elements whose actual geometry is not close enough to the identification point to be accepted (consider the range block of a 45° line). In addition, the information available from `mdlLocate_getProjectedPoint`, as well as the component offset information (available from `mdlElement_getFilePos`) is not valid for `LOCATE_PRELOCATE` functions. For these reasons, `LOCATE_POSTLOCATE` is almost always desired rather than `LOCATE_PRELOCATE`. `LOCATE_PRELOCATE` is provided to change the behavior of the location logic (i.e., to accept closed elements by identifying a point in their interior).

Whenever a primitive command is started, the locate user functions (with the exception of `LOCATE_GLOBAL_LOCATE` and `LOCATE_SELECT_CMD` functions, see below) are cleared. Therefore, if `mdlLocate_setFunction` is used with `mdlState_startModifyCommand`, `mdlState_startModifyCommand` must be called first.

However, there are cases where applications wish to establish filter functions that remain active on a “global” basis regardless of the active command. To establish a global locate filter, use `LOCATE_GLOBAL_LOCATE`. Global locate filters remain active until they are explicitly removed (by passing `NULL` to `mdlLocate_setFunction`). Global Locate filters are called for during element location, selection set processing, fence clipping, fence processing, and snapping. See `userLocate_globalFilter` for details.



The message that displays when the accept function is called is stored in the built-in variable `locateAcceptPrompt`. `locateAcceptPrompt` must be an index into a message list resource. The message list resource identifier must have been previously set by calling `mdlState_registerStringIds`. The `promptStringId` parameter is the resource identifier of the message list resource. It contains the prompt message, which is referenced by this index.

Returns `mdlLocate_setFunction` returns a pointer to the user function (of the same type) that was previously set using `mdlLocate_setFunction`.

See Also `userLocate_elementFilter`, `userLocate_globalFilter`, `userLocate_selectCmd`.

userLocate_elementFilter

```
#include <userfnc.h>

int userLocate_elementFilter
(
    int                preLocate,      /* => TRUE for prelocate.*/
    MSElementUnion    *elementP,      /* => element being examined */
    int                fileNumber,     /* => design or reference file */
    unsigned long       filePosition,  /* => element file position */
    Point3d            *pointP,       /* => point entered by user */
    int                viewNumber     /* => view containing point */
)
```

```
);
```

Description The *userLocate_elementFilter* function is called once for every element MicroStation examines during element location. This function is called either before or after MicroStation considers the element, depending on whether LOCATE_PRELOCATE or LOCATE_POSLOCATE is used. Examining elements *after* MicroStation considers the element (LOCATE_POSTLOCATE) is much more efficient, since MicroStation will have eliminated many elements that do not pass the location criteria.

Every time mdlState_startModifyCommand is called, all element location filters are cleared. Therefore, to use *userLocate_elementFilter* with mdlState_startModifyCommand, call mdlState_startModifyCommand first.

A *userLocate_elementFilter* function is designated to MicroStation when mdlLocate_setFunction is called with *type* equal to LOCATE_PRELOCATE or LOCATE_POSTLOCATE.

The argument *preLocate* is TRUE if *userLocate_elementFilter* is called before MicroStation applies its locate distance criteria and FALSE if it is called afterwards.

The argument *elementP* points to the current element.

fileNumber is 0 if the element is in the main design file. Values 1 through MAX_REFS are used for the reference files.

filePosition is the element's file position.

pointP is the location entered by the user (and adjusted for all applicable locks).

viewNumber is the number of the view in which the data point was entered.

Returns The *userLocate_elementFilter* function should return LOCATE_ELEMENT_REJECT to reject the element, LOCATE_ELEMENT_NEUTRAL to let MicroStation decide whether to accept or reject the element, and LOCATE_ELEMENT_ACCEPT to accept the element regardless of MicroStation's decision.

See Also userLocate_globalFilter.

userLocate_globalFilter

```
#include <userfnc.h>

int userLocate_globalFilter
(
    int          activity,      /* => one of GLOBAL_LOCATE_... */
    MSElement   *element,     /* => element under consideration */
    int          fileNum,      /* => file number of element */
    ULong        filePos,     /* => file position of element */
    Point3d      *point,      /* => applicable for _IDENTIFY and _SNAP */
    int          viewNum      /* => applicable for _IDENTIFY and _SNAP */
)
```

```
);
```

Description *userLocate_globalFilter* is directly analogous to *userLocate_elementFilter*, except that rather than filtering elements for one particular command, it filters elements for *all* commands.

In addition, multiple MDL applications can each have their own independent global location filters. The filters from multiple MDL applications are called sequentially, and elements are only considered eligible for manipulation if they pass all active filters. There is no way to specify the order in which the applications' filters are called.

Global Location functions are established by calling *mdlLocate_setFunction*. This establishes the function to be called whenever MicroStation considers an element for possible location or modification. The function remains active until it is either explicitly removed by passing *NULL* to *mdlLocate_setFunction* or your MDL process exits.

The parameter *activity* indicates what MicroStation is doing at the time this function is called (the reason for the call). Possible values are:

activity value	description
GLOBAL Locate IDENTIFY	user is attempting to locate an element.
GLOBAL Locate SELECTIONSET	element is in selection set and is about to be processed.
GLOBAL Locate FENCE	fence command is going to process an element.
GLOBAL Locate FENCECLIP	fence logic is going to clip an element.
GLOBAL Locate SNAP	user is attempting to snap to the element.

Returns *userLocate_globalFilter* must return one of the following status values, depending on the desired action:

```
LOCATE_ELEMENT_REJECT
LOCATE_ELEMENT_NEUTRAL
LOCATE_ELEMENT_ACCEPT
```

See Also *userLocate_elementFilter*.

userLocate_selectCmd

```
#include <userfnc.h>

int userLocate_selectCmd
(
    int      *action, /* => one of SELECT_STRETCH, ... */
    ULong    filePos, /* => element under consideration */
    int      fileNum  /* => file number of element */
);
```

Description The *userLocate_selectCmd* function is called whenever the data button is pressed while the selection tool is active. It is established by calling *mdlLocate_setFunction* and passing the value *LOCATE_SELECT_CMD* in the *type* parameter.

The parameter *action* indicates what MicroStation is doing at the time this function is called (the reason for the call).

action value	Description
SELECT_STRETCH	user is dragging on a handle of an element in the existing selection set.
SELECT_WINDOW	user is dragging away from any element.
SELECT_DRAG	user is dragging on an edge of an element in the existing selection set.
SELECT_DRAG_NEW	user is dragging on an edge of an element that was just added to the selection set.
SELECT_NEW	an element is to be added to the existing selection set (button-up).

The parameters *filePos* and *fileNum* are the file position and file number of the element currently under consideration (if applicable).

Returns If this function returns 0, MicroStation continues with its normal selection set processing. Otherwise, MicroStation assumes that the event has been handled and it does nothing further.

See Also *mdlLocate_setFunction*, *mdlSelect_isActive*.

Element Modification Functions

One of the most common operations that MDL applications perform is modifying an existing element or element group. Element modification typically entails modifying an existing element and then writing the new element, which replaces the original element in the file.

The following table lists element modification functions:

Function	Used to
<i>mdlModify_elementSingle</i>	modify a single (possibly complex) element given its file position.
<i>mdlModify_elementMulti</i>	modify a group of elements, operating on the selection set if one exists, a graphic group if Graphic Group lock is on, or a single element if neither of these conditions applies.

Function	Used to
<code>mdlModify_elementDescr</code>	modify an element descriptor.
<code>mdlModify_freeGGMap</code>	free memory allocated for graphic group map.

mdlModify_elementSingle

```
#include <mselems.h>

ULong mdlModify_elementSingle
(
    int          fileNum,          /* => file number to process */
    ULong        filePos,          /* => file position for element */
    int          componentFlag,    /* => flags for complex elements */
    int          modifyFlags,      /* => steps for modification */
    MdlFunctionP elementFunc,      /* => function called for each elem */
    void         *params,          /* => parameters for elementFunc */
    ULong        componentOffset   /* => used if componentFlag is set */
);
```

Description `mdlModify_elementSingle` reads one element from the design file and calls a function for each component of that element. This function is used when a single element has been identified for modification and an MDL application will change that element.

fileNum and *filePos* are the element's file number and file position. If *fileNum* is non-zero (indicating an element in a reference file), the `MODIFY_COPY` bit in *modifyFlags* must be set.

componentFlag indicates when to call *elementFunc* for complex elements. *componentFlag* can have one of the following values:

componentFlag	Meaning
<code>MODIFY_REQUEST_NOHEADERS</code>	Never call <i>elementFunc</i> for the headers of complex elements.
<code>MODIFY_REQUEST_HEADERS</code>	Call <i>elementFunc</i> for all components and headers of complex elements.
<code>MODIFY_REQUEST_ONLYONE</code>	Call <i>elementFunc</i> only once for the component element at offset <i>componentOffset</i> from the header. MDL applications typically use <code>mdlLocate_findElement</code> to find an element, and then get <i>componentOffset</i> by subtracting the header pos from the value returned by <code>mdlElement_getFilePos (FILEPOS_COMPONENT, NULL)</code> .

modifyFlags indicates the steps to use when modifying an element. These steps are represented as a series of constants that are joined using the

logical OR operator (|). *modifyFlags* indicates the action to be taken by mdlModify_elementSingle». The constants are as follows:

Constant for <i>modifyFlags</i>	Meaning
MODIFY_ORIG	Modify the original element. If the new element is larger than the old element, the old one is deleted and the new one is added to the end of the file.
MODIFY_COPY	Modify the element and add the new element to the file, leaving the original element unchanged. If a new element is created, it is given a new graphic group number and text node number (if it is a text node). The attributes are processed according to the current linkage generation mode, and the <i>new</i> and <i>not modified</i> bits are set in the header.
MODIFY_DONTREMOVEORIG	Do not erase the original element if MODIFY_ORIG is being used. Otherwise, the original element is erased from the screen.
MODIFY_DONTDRAWNEW	Do not draw the newly created element.
MODIFY_DRAWINHILITE	Draw the newly created element in the current hilite color. This is valid only if MODIFY_DONTDRAWNEW is not present.

elementFunc points to a function to be called once for each component of the element to be modified. *elementFunc* modifies (in memory) an element passed to it as an argument. mdlModify_elementSingle» automatically writes the modified element to the file according to the return status of *elementFunc*, which should have the following format:

```
int elementFunc(element, params, fileNum, elmDscrP, newDscrPP)
    MSElementUnion *element;      /* <=> element to be modified */
    void *params;                  /* => user parameter */
    int fileNum;                   /* => file # for current elem */
    MSElementDescr *elmDscrP;     /* => element descr for elem */
    MSElementDescr **newDscrPP; /* <= if replacing entire descr */
```

elementFunc should return a combination of the following values:

<i>elementFunc</i> return value	Meaning
MODIFY_STATUS_NOCHANGE	The element was not changed.
MODIFY_STATUS_REPLACE	Replace original element with the new element.
MODIFY_STATUS_DELETE	Delete the original element.
MODIFY_STATUS_ABORT	Stop processing component elements. MODIFY_STATUS_ABORT can be used with any of the other values.
MODIFY_STATUS_FAIL	An error occurred during element modification. Ignore all changes previously made.

<i>elementFunc</i> return value	Meaning
MODIFY_STATUS_REPLACEDSCR	Replace the current element descriptor with a new element descriptor. This is used to replace one or more elements with a (possibly) different number of elements.
MODIFY_STATUS_ERROR	MODIFY_STATUS_FAIL MODIFY_STATUS_ABORT

element points to the current element to be modified. This element should be modified in place so that when *elementFunc* returns, the new element is in *element*. `mdlModify_elementSingle` adjusts the appropriate values in the header element (if *element* is a component of a complex element).

params is the value that was passed to `mdlModify_elementSingle`. *params* typically points to a structure containing parameters for *elementFunc*.



If the `MODIFY_COPY` bit is set in *modifyFlags*, `mdlModify_elementSingle` expects that *params* points to a structure of type `MdlCopyParams`. The `MdlCopyParams` structure is used to maintain graphic grouping information across multiple calls to `mdlModify_elementSingle`. If you want to make sure that copied elements maintain the same graphic group association as original elements, you must allow MDL to maintain the information in the `MdlCopyParams` structure across your calls to `mdlModify_elementSingle`. It does this by allocating memory to hold a table of old/new `gg` numbers. When you are done with your operation, the call to `mdlModify_freeGMap` frees this memory.

There are two cases in MDL where `mdlModify_elementSingle` is called multiple times on behalf of your program: fence processing and the `mdlModify_elementMulti` function. In these cases you should call `mdlModify_freeGMap` immediately, since you won't be calling `mdlModify_elementSingle` again. You can tell if you've done things properly by issuing the MicroStation command `SHOW HEAP GGMP`. There should not be any entries of type `GGMP` in the heap if all goes well.

`mdlModify_elementSingle` uses the `MdlCopyParams` structure to track the mapping of old graphic group numbers to new graphic group numbers. MDL applications must call `mdlModify_freeGMap` after `mdlModify_elementSingle` to free this memory. If your *elementFunc* requires additional information, declare a structure containing the required information, with `MdlCopyParams` as the first member. Then pass a pointer to that structure for *params* as the following example illustrates:

```
typedef struct myModifyParams
{
    MdlCopyParams ggParams; /* mdlModify_elementSingle */
    long beamNumber;        /* beam being changed */
    int *classCode;         /* class code active */
    int project;            /* project type */
}
```



```
} MyModifyParams;
```

fileNum is *element*'s file number.

elmDscrP points to the element descriptor for *element*. *elementFuncs* should treat the element descriptor pointed to by *elmDscrP* as read-only and never modify it directly. *elmDscrP* rarely needs to be used, except when *elementFunc* needs to know the element structure (the header type).

newDscrPP points to an element descriptor. This descriptor replaces the current element descriptor if the `MODIFY_STATUS_REPLACEDSCR` bit is set in the return value of *elementFunc*. `mdlModify_elementSingle` automatically frees the memory associated with *newDscrPP*.

params points to a structure to be passed to your element function.

componentOffset is used in conjunction with *componentFlag* when `MODIFY_REQUEST_ONLYONE` is passed, and a specific component offset must be specified. See *componentFlag*, above.

Returns The `mdlModify_elementSingle` function adds a new element or replaces the old element that was modified by *elementFunc*. It returns the new element's file position.

See Also `mdlModify_freeGMap`, `mdlModify_elementMulti`, `mdlModify_elementDescr`.



modifyFlags, `mdlModify_elementSingle` expects that *params* points to a structure of type `MdlCopyParams`. The `MdlCopyParams` structure is used to maintain graphic grouping information across multiple calls to `mdlModify_elementSingle`. If you want to make sure that copied elements maintain the same graphic group association as original elements, you must allow MDL to maintain the information in the `MdlCopyParams` structure across your calls to `mdlModify_elementSingle`. It does this by allocating memory to hold a table of old/new `gg` numbers. When you are done with your operation, the call to `mdlModify_freeGMap` frees this memory.

mdlModify_elementMulti

```
#include <mselems.h>

ULong mdlModify_elementMulti
(
    int          fileNum,          /* => file number to process */
    ULong        filePos,          /* => file position for element */
    int          componentFlag,    /* => if TRUE, only do one element */
    int          modifyFlags,      /* => steps for modification */
    MdlFunctionP elementFunc,      /* => function called for each elem */
    void         *params,          /* => parameters for elementFunc */
    boolean      doGroups          /* => do graphic groups? */
);
```

Description `mdlModify_elementMulti` closely resembles `mdlModify_elementSingle`, except that this function also processes selection sets and graphic groups. It makes MDL applications function identically to MicroStation primitives for element modification. It uses the following algorithm:

```
if (there is a selection set active)
{
    call mdlModify_elementSingle for all other elements in selection set
}
else
{
    call mdlModify_elementSingle for the element at filePos
    if (doGroups is TRUE and element is a member of a graphic group)
    {
        call mdlModify_elementSingle for all elements in the graphic group
    }
}
```

fileNum, *filePos*, *componentFlag*, *modifyFlags*, *elementFunc* and *params* are identical to the arguments for `mdlModify_elementSingle`. See the function descriptions for their meanings.

If *doGroups* is TRUE, `mdlModify_elementMulti` processes graphic groups.

Returns `mdlModify_elementMulti` returns the file position of the new element. If more than one element is modified, such as when a selection set is active, it returns zero.

See Also `mdlModify_freeGMap`, `mdlModify_elementSingle`, `mdlModify_elementDescr`.

mdlModify_elementDescr

```
#include <mselems.h>

MdlFunctionP mdlModify_elementDescr
(
    MSElementDescr    **elmDscrPP,          /* <=> descriptor to modify */
    int                fileNum,              /* => file num to process */
    int                componentFlag,        /* => complex elem flags */
    MdlFunctionP       elementFunc,         /* => called for each elem */
    void               *params,             /* => parms for elementFunc */
    ULong              componentOffset      /* => if componentFlag set */
);
```

Description The `mdlModify_elementDescr` function modifies an element descriptor in memory. It is the lowest-level function in the `mdlModify_...` group. `mdlModify_elementSingle` reads an element descriptor from the design file and calls `mdlModify_elementDescr`

to modify that descriptor. `mdlModify_elementDescr` is useful when you want to modify an element descriptor but do not want to write the new elements to the file.

elmDscrPP points to a pointer to an element descriptor. Since this function can modify the element descriptor, the value of *elmDscrPP* can be different before and after the call.

The remaining arguments to `mdlModify_elementDescr` are the same as those for `mdlModify_elementSingle`.

Returns `mdlModify_elementDescr` returns the last status returned by *elementFunc*.

See Also `mdlModify_elementSingle`, `mdlModify_elementMulti`.

mdlModify_freeGMap

```
#include <mselems.h>

void mdlModify_freeGMap
(
MdlCopyParams    *params    /* => structure used by mdlModify_ */
);
```

Description `mdlModify_freeGMap` frees memory after element modification. When the `MODIFY_COPY` bit is set in the *modifyFlags* parameter for `mdlModify_elementSingle` or `mdlModify_elementMulti`, these functions duplicate elements before modifying them. One of the steps for duplicating an element is assigning it a new graphic group number. To ensure that all elements in the same graphic group are assigned the same new graphic group number, these functions must allocate memory to track the mapping from original graphic group numbers to new graphic group numbers. See the description in `mdlModify_elementSingle` for further information on this topic. When MDL applications complete their element modifications, they must call `mdlModify_freeGMap` once to free this memory.

params is the pointer used in the calls to `mdlModify_elementSingle` and `mdlModify_elementMulti`.

Returns The `mdlModify_freeGMap` function is of type `void`. It returns no value.

See Also `mdlModify_elementSingle`, `mdlModify_elementMulti`.

Selection Set Processing Functions

Selection sets are a convenient method for grouping elements for subsequent operations. The selection set processing functions provide a convenient means for MDL applications to modify the elements contained in selection sets. However, MDL applications sometimes need to be able to control the elements that are part of a selection set and to process the elements in the set. The `mdlSelect_...` functions provide techniques to do so.

The following table lists selection set processing functions:

Function	Used to
<code>mdlSelect_isActive</code>	determine whether a selection set is active.
<code>mdlSelect_returnPositions</code>	return the file positions of all elements in the selection set.
<code>mdlSelect_addElement</code>	add a single element to the selection set.
<code>mdlSelect_removeElement</code>	remove a single element from the selection set.
<code>mdlSelect_allElements</code>	add all elements in the design file to the selection set.
<code>mdlSelect_freeAll</code>	remove all elements from the selection set.
<code>mdlSelect_handleDrawAll</code>	draw the handles on all elements in the selection set.
<code>mdlSelect_handleDrawElement</code>	draw the handles on a single element.

Example

See `select.mc`.

mdlSelect_isActive

```
#include <mselems.h>

boolean mdlSelect_isActive(void);
```

Description The `mdlSelect_isActive` function indicates whether any elements are currently selected.



If you want to use selection sets as part of an element modification function, `mdlState_startModifyCommand` automatically determines whether there is a selection set active and calls your `accept` function accordingly.

Returns The mdlSelect_isActive function returns TRUE if a selection set is active. Otherwise, it returns FALSE.

See Also mdlSelect_allElements, mdlSelect_freeAll, userLocate_selectCmd.

mdlSelect_returnPositions

```
#include <mselems.h>

int mdlSelect_returnPositions
(
    ULong    **filePositions,    /* => array of file positions */
    int      **fileNums,        /* => file numbers of elements */
    int      *numSelected       /* => number of elements selected */
);
```

Description The mdlSelect_returnPositions function allocates two arrays in memory. These arrays list elements currently in the selection set, each of which has *numSelected* entries. The first array, *filePositions*, lists file positions, one for each element in the selection set; the second array contains the file numbers for the same.



MDL applications should pass a pointer to a pointer to a ULong for *filePositions*, and a pointer to a pointer to an int for *fileNums*. The MDL application must free the memory for both arrays when it is finished with them.

Returns The mdlSelect_returnPositions function returns SUCCESS if it allocated the memory for *filePositions* and *fileNum* and MDLERR_INSFMEMORY if it did not.

See Also mdlSelect_isActive.

mdlSelect_addElement, mdlSelect_removeElement

```
#include <mselems.h>

int mdlSelect_addElement
(
    ULong    filePos,          /* => file position of element */
    int      fileNumber,       /* => file number of element */
    MSElementUnion *element,   /* => element to add */
    boolean  drawHandles       /* => draw element handles */
);

int mdlSelect_removeElement
(
    ULong    filePos,          /* => file position of element */
    int      fileNumber,       /* => file number of element */
    boolean  drawHandles       /* => draw element handles */
);
```

Description The `mdlSelect_addElement` function adds an element to the current selection set. The address of the element is given by *filePos* and *fileNumber*. *element* points to a copy of the element. If *drawHandles* is `TRUE`, it draws the handles on the element as it is added to the selection set.

The `mdlSelect_removeElement` function removes the element at file position *filePos* and file number *fileNumber* from the selection set. If *drawHandles* is `TRUE`, the function erases the handles from the element as the element is removed from the selection set.



Only simple elements or complex headers should be added to selection sets; complex components should never be added.

Returns The `mdlSelect_addElement` function returns `SUCCESS` if the element is added to the selection set and `ERROR` if the element is invalid or unable to allocate memory for the selection set.

The `mdlSelect_removeElement` function returns `SUCCESS` if the element is removed from the selection set and `ERROR` if it cannot be found in the selection set.

See Also `mdlSelect_isActive`.

mdlSelect_allElements, mdlSelect_freeAll

```
#include <mselems.h>

void mdlSelect_allElements(void);
void mdlSelect_freeAll(void);
```

Description The `mdlSelect_allElements` function scans the design file and adds all of the displayable elements to the selection set. It ignores the levels or views *not* currently displayed. For this reason, this function's usefulness is limited.

The function `mdlSelect_freeAll` removes all elements from the selection set and frees memory associated with it. It erases the element handles displayed on the screen.

Returns The `mdlSelect_allElements` and `mdlSelect_freeAll` functions are of type `void`. They return no value.

See Also `mdlSelect_isActive`.

mdlSelect_handleDrawAll, mdlSelect_handleDrawElement

```
#include <select.h>

void mdlSelect_handleDrawAll
(
    int      handleDrawMode,      /* => HANDLE_DRAW, HANDLE_ERASE */
    void     *gwPList[],          /* => list of views or NULL */
    int      numWindows           /* => number of views in gwPList */
);

void mdlSelect_handleDrawElement
(
    MSElement *element,          /* => element to draw handles */
    int      handleDrawMode,      /* => HANDLE_DRAW, HANDLE_ERASE */
    int      fileNum,             /* => file number */
    void     *gwPList[],          /* => list of views or NULL */
    int      numWindows           /* => number of views in gwPList */
    int      handleColorIndex     /* => BLACK_INDEX or WHITE_INDEX */
);
```

Description The `mdlSelect_handleDrawAll` function draws the handles on all elements in the selection set in the specified views.

The `mdlSelect_handleDrawElement` function draws the handles on a single element in the specified views.

element is a pointer to an element for which the handles are drawn.

handleDrawMode is a flag that indicates the mode in which the handles are drawn. Possible values are: `HANDLE_DRAW`, `HANDLE_ERASE`, `LOCKED_HANDLE_DRAW` and `LOCKED_HANDLE_ERASE`.

fileNum is the file number of the element. This is needed to expand shared cells and dimensions.

gwPList is a pointer to an array of pointers to views, as returned by `mdlWindow_viewWindowGet`. To display the handles in all visible views (the most common situation), pass `NULL` for this parameter.

numWindows is the number of view pointers in *gwPList*.

handleColorIndex is the color used to draw the element handles. The valid values are `BLACK_INDEX` and `WHITE_INDEX`.

Returns Both `mdlSelect_handleDrawAll` and `mdlSelect_handleDrawElement` are of type `void`. They do not return a status.

Fence Processing Functions

Fences are a convenient way to process groups of elements that are close to one another. The fence processing functions let MDL applications use fences as MicroStation internal commands use them.

To use fences from an MDL application, the first step is to call `mdlState_startFenceCommand`. This function tells MicroStation to call functions in your MDL application when the user presses a data point or a reset button. It also establishes two of your functions to be called when MicroStation accepts an element as meeting the fence criteria and for operating on the fence.

The typical scenario is for an MDL application to call `mdlState_startFenceCommand` to establish the fence functions. When the application's data point function is called, it, in turn, calls `mdlFence_process` to process the elements inside the fence. `mdlFence_process` calls the accept function for each element in the fence and then calls the function to operate on the fence.

Although `mdlFence_process` is generally called from a data point function, it does not need to be. It can be called any time after the call to `mdlState_startFenceCommand`. Also, MicroStation can call `mdlFence_process` automatically if `NULL` is passed as the data point function to `mdlState_startFenceCommand`. (MicroStation establishes `mdlFence_process` as the data point function).

The following table lists fence processing functions:

Function	Used to
<code>mdlFence_process</code>	process all elements accepted by the fence using the current locks.
<code>mdlFence_fromShape</code>	create a new fence based on the given shape.
<code>mdlFence_toShape</code>	create a shape based on the current fence.
<code>mdlFence_clear</code>	clear the current fence.
<code>mdlFence_fromView</code>	create a fence from a view's extents.
<code>mdlFence_fromDesign</code>	create a fence from design file(s) extents.
<code>mdlFence_fromUniverse</code>	create a fence from the size of the design plane.
<code>mdlFence_fromElement</code>	create a fence from a closed element.

Example

See `fence.mc`.

mdlFence_process

```
#include <mselems.h>

void mdlFence_process
(
void    *arg        /* => argument passed to content routine */
);
```

Description The `mdlFence_process` function scans the design file and all reference files, calling the `contents` function for each accepted element, based on the current fence and the current fence locks. The argument *arg* (passed to `mdlFence_process`) is passed to the `contents` function; MicroStation does not use this argument. It is generally used to point to a structure that contains arguments and/or status information between the function calling `mdlFence_process` and the `contents` function.

When the fence processing is complete, `mdlFence_process` calls the current fence outline function to update the fence.

See the description of *userState_fenceContent* for further explanation of the `contents` function and *userState_fenceOutline* for further explanation of the fence outline function.



The content and outline functions are set with `mdlState_startFenceCommand`.

Returns The `mdlFence_process` function is of type `void`. It returns no value.

See Also `mdlState_startFenceCommand`, *userState_fenceContent*, *userState_fenceOutline*.

mdlFence_fromShape, mdlFence_toShape

```
#include <mselems.h>

void mdlFence_fromShape
(
MSElementUnion    *shape    /* => shape element converted fence */
);

void mdlFence_toShape
(
MSElementUnion    *shape    /* <= shape elem created from fence */
);
```

Description `mdlFence_fromShape` creates a new fence using the shape element *shape* to define the outline. `mdlFence_toShape` creates a MicroStation shape element in *shape* that matches the outline of the current fence.

When these two functions are used together, MDL applications can use them to create and modify the current fence.



Before calling `mdlFence_fromShape`, the application should set the variable *tcb* → *fenvw* to the view number (minus 1).



The application should make sure the fence exists before calling `mdlFence_toShape`.

See Also `mdlFence_fromShape` is of type `void`; it returns no value. `mdlFence_toShape` is of type `void`. It returns no value.

mdlFence_clear

```
#include <msmisc.fdf>

void mdlFence_clear
(
    boolean      erase      /* => TRUE to erase the current fence */
);
```

Description The `mdlFence_clear` function clears (turns off) the current fence. It also synchronizes or sets the enabled state of dialog items which depend on whether or not a fence is active.

The *erase* parameter specifies whether or not to erase the current fence.



This function was implemented in MicroStation 95.

Returns The `mdlFence_clear` function is of type `void`. It returns no value.

mdlFence_fromView

```
mdlFence_fromView(int      view);
```

Description `mdlFence_fromView` creates a fence from a view's extents.

mdlFence_fromDesign

```
mdlFence_fromDesign
(
    int      view,
    ULong    *fileMaskP
);
```

Description `mdlFence_fromDesign` creates a fence from design file(s) extents.

mdlFence_fromUniverse

```
mdlFence_fromUniverse(int      view);
```

Description mdlFence_fromUniverse creates a fence from the size of the design plane. That is, a fence large enough to contain all elements within the design plane.

mdlFence_fromElement

```
mdlFence_fromElement  
(  
  MSElementDescr      *edP;  
  int                   view;  
);
```

Description mdlFence_fromElement creates a fence from a closed element, i.e., shape, ellipse, complex shape, or closed B-spline curve. Non-linear segments of the closed element, if any, are stroked into multiple linear segments.

Surface Creation Functions (3D Only)

Surface elements represent surfaces in space in 3D files. The surface creation functions create these surfaces from existing elements.

The following table lists surface creation functions:

Function	Used to
mdlSurface_createHeader	create a complex surface header element.
mdlSurface_project	create a surface of projection from specified element(s).
mdlSurface_revolve	create a surface of revolution from specified element(s).

These functions can create both capped and uncapped versions of surface elements. To control which they create, set ACTIVEPARAM_CAPMODE using the mdlParams_setActive function.

Example

See trumpet.mc.

mdlSurface_createHeader

```
#include <mselems.h>

void mdlSurface_createHeader
(
MSElementUnion    *out,          /* <= new surface header */
int                surfaceType    /* => surface type */
);
```

Description The `mdlSurface_createHeader` function creates a complex surface header element. Complex surface elements are non-displayable elements that group other displayable elements. For this reason, this function does not have an input element as an argument. Complex surface header elements cannot be automatically added to the design file. Instead, they should be used with the Element Descriptor routines.

surfaceType indicates the type of surface to be created by `mdlSurface_createHeader`. Possible values are: 0 = surface of projection, 8 = surface of revolution.

MDL applications do not generally need `mdlSurface_createHeader`, since the `mdlSurface_project` and `mdlSurface_revolve` functions automatically create surface headers. Only advanced applications need this function to create special types of surfaces.

Returns The `mdlSurface_createHeader` function is of type `void`. It returns no value.

See Also `mdlSurface_project`, `mdlSurface_revolve`, `mdlParams_setActive`.

mdlSurface_project

```
#include <mselems.h>

int mdlSurface_project
(
MSElementDescr    **surfDscrPP, /* <= addr of new elm dscr */
MSElementDescr    *elmDscrP,    /* => element(s) to project */
Dpoint3D           *point1,       /* => anchor point */
Dpoint3D           *point2,       /* => dest point */
Transform           *transform     /* => rotation and scaling */
);
```

Description The `mdlSurface_project` function creates a surface of projection from the element descriptor pointed to by *elmDscrP*. It allocates a new element descriptor and returns its address in *surfDscrPP*. The element descriptor is transformed by the transformation matrix *transform* about *point2*. It is then projected by the distance from *point1* to *point2*. If *transform* is `NULL`, the element descriptor is projected without modification. The points are given in the current coordinate system.

Programmers should free the element descriptor pointed to by *surfDscrPP* when finished with it.

Returns The mdlSurface_project function returns SUCCESS if the element is projected and surfDscrPP is valid. It returns ERROR if the resulting surface is off the design plane. It returns MDLERR_FILE2SUB3 if the design file is a 2D file.

See Also mdlSurface_revolve, “Element Descriptor Functions” (mdlElmdscr...), “B-spline Functions” (mdlBSpline...), mdlParams_setActive.

mdlSurface_revolve

```
#include <mselems.h>

int mdlSurface_revolve
(
  MSElementDescr    **surfDscrPP, /* <= addr of new elm dscr */
  MSElementDescr    *elmDscrP,    /* => element(s) to revolve */
  Dpoint3D           *center,       /* => center of revolution */
  int                 axis,          /* => axis (0, 1, or 2) */
  double              angle          /* => angle of revolution */
);
```

Description The mdlSurface_revolve function creates a surface of revolution from the element descriptor pointed to by *elmdscrP*. It allocates a new element descriptor and returns its address in *surfDscrPP*. The element is revolved about the point given by *center*, or the (0, 0, 0) point for the current coordinate system if *center* is NULL.

The axis for the revolution is determined from the current coordinate system and *axis*. *axis* can be either 0 (X-axis), 1 (Y-axis) or 2 (Z-axis). This is normally the most convenient way to specify the axis of rotation.

The angle of revolution is given by *angle* in radians.

Programmers should free the element descriptor pointed to by *surfDscrPP* when finished with it.

Returns The mdlSurface_revolve function returns SUCCESS if the element is revolved and if surfDscrPP is valid. It returns ERROR if the resulting surface is off the design plane. It returns MDLERR_FILE2SUB3 if the design file is a 2D file.

See Also mdlSurface_project, Element Descriptor Functions, B-spline Functions, mdlParams_setActive.

Complex Chain Creation Functions

Complex chains group several open elements to form a single entity.

The following table lists complex chain creation functions:

Function	Used to
<code>mdlComplexChain_createHeader</code>	create a complex chain header element.
<code>mdlChain_begin</code>	create a chain header in the design file.
<code>mdlChain_end</code>	finish the chain header definition in the design file.

mdlComplexChain_createHeader

```
#include <mselems.h>

void mdlComplexChain_createHeader
(
  MSElementUnion  *out,          /* <= new chain header */
  int              chainType,     /* => chain type */
  int              fillMode      /* => filled or not, -1, 0 or 1 */
);
```

Description The `mdlComplexChain_createHeader` function creates a complex chain header element. Complex chain header elements are non-displayable elements used to group other displayable elements. For this reason, this function does not have an input element as an argument. Complex chain header elements cannot be added to the design file on their own. Instead, they are used with the element descriptor routines.

chainType indicates the type of chain `mdlComplexChain_createHeader` is to create. If *chainType* is `TRUE`, it creates a complex shape header. Otherwise, it creates a complex chain header.

If *chainType* is `TRUE`, *fillMode* for the `mdlComplexChain_createHeader` function determines whether the shape is filled or open. Possible values are as follows:

fillMode	Meaning
-1	Use the active <i>fillMode</i> from MicroStation.
0	The shape is not filled.
1	The shape is filled.

Returns The `mdlComplexChain_createHeader` function is of type `void`. It returns no value.

mdlChain_begin

```
#include <mselems.h>

ULong mdlChain_begin
(
    short  *attributes    /* => attributes to be put on header */
);
```

Description The mdlChain_begin function creates a complex chain header in the design file and returns the header's file position.

attributes points to an array of shorts containing any attribute information to be attached to the complex chain header. The first member of the array is the length of the attribute data. If there are no attributes to be appended, pass NULL for *attributes*.



Calls to mdlChain_begin should be matched with subsequent calls to mdlChain_end.

mdlChain_begin is provided mainly for compatibility with MicroCSL. Its use is discouraged. Rather, mdlComplexChain_createHeader and Element Descriptors should almost always be used for creating complex chains.

Returns The mdlChain_begin function returns the file position of the complex chain header created. If an error occurs, it returns zero and mdlErrno is set to the specific error cause. See mdlElement_add for the possible values of mdlErrno.

See Also mdlChain_end.

mdlChain_end

```
#include <mselems.h>

int mdlChain_end
(
    ULong  headerFilePos, /* => header address (from mdlChain_begin) */
    int    headerType     /* => chain type */
);
```

Description The mdlChain_end function finishes the definition of a complex started with a call to mdlChain_begin. *headerFilePos* is the file position of the complex chain header returned by mdlChain_begin. All elements between *headerFilePos* and the end of file should be part of the complex chain definition.

headerType defines the type of complex header to be created. Possible values are CMLPX_SHAPE_ELM and CMLPX_STRING_ELM.

The mdlChain_end function sets the range of the complex header to the union of the element ranges in the complex chain. It sets the number of elements field and the element's total size in the header.

`mdlChain_end` also verifies that the elements defining the complex shape or complex string are valid, reversing the order of the vertices if necessary.

Returns `mdlChain_end` returns `SUCCESS` if the operations described above are accomplished. It returns `ERROR` if *headerFilePos* does not point to a complex header.

See Also `mdlChain_begin`.

Element Clipping Functions

The element clipping functions may be used by an application to clip elements to a fence or reference file boundary. This can be useful when implementing specialized fence processing or translator applications that require only the portion of a reference file element that is within a fence or reference boundary.

The element clipping functions expect the clipping region to be described by a pointer to a data structure that is referred to as a “clipping descriptor.” Routines are provided to retrieve clipping descriptors for the current fence or the boundary of a reference file. These routines allocate memory that should be freed by calling `mdlClip_free` when the clipping descriptor is no longer required.

The following table lists the element clipping functions:

Function	Used to
<code>mdlClip_element</code>	clip an element.
<code>mdlClip_free</code>	free clipping descriptor.
<code>mdlClip_getFence</code>	get clipping descriptor representing current fence.
<code>mdlClip_getRefBoundary</code>	get a clipping descriptor that describes a reference file boundary.
<code>mdlClip_isElemInside</code>	returns a non-zero value if an element is within the clipping descriptor.

`mdlClip_element`

```
int mdlClip_element
(
  MSElementDescr  **insideEdPP, /* <= portion of elm inside */
  MSElementDescr  **outsideEdPP, /* <= portion of elm outside */
  MSElementDescr  *inputEdP,    /* => input element */
  int               fileNum,      /* => file num for input elm */
  void              *clipP,       /* => clipping descriptor */
  int               view          /* => view number */
);
```


Description The `mdlClip_element` function clips the element specified by *inputEdP* and *fileNum* to the region specified by *clipP* and *view*. *clipP* should be a pointer to a clipping descriptor retrieved by one of the `mdlClip_get...` routines.

The portion of the element that is inside the clipping descriptor is returned in *insideEdPP*. If the element is totally inside, **insideEdPP* is set to *inputEdP* (the element is not duplicated). If the element is totally outside, **insideEdPP* is set to `NULL`.

If the element overlaps the clipping descriptor, and *outsideEdPP* is not `NULL`, the portion of the element that is outside the descriptor is returned in **outsideEdPP*. If the portion of overlapping elements that is outside the clipping descriptor is not required, `NULL` should be passed for *outsideEdPP*.

Returns `mdlClip_element` returns `SUCCESS` if the element is successfully processed.

See Also `mdlClip_getFence`, `mdlClip_getRefBoundary`, `mdlClip_free`, `mdlClip_isElemInside`.

mdlClip_free

```
void mdlClip_free
(
void    *clipP          /* => clip descriptor to free */
);
```

Description The `mdlClip_free` function frees the memory allocated for the clipping descriptor, *clipP*. This should be a clipping descriptor obtained through one of the `mdlClip_get...` functions.

Returns `mdlClip_free` is of type `void`; it returns no value.

See Also `mdlClip_getFence`, `mdlClip_getRefBoundary`, `mdlClip_element`, `mdlClip_isElemInside`.

mdlClip_getFence

```
void mdlClip_getFence
(
void    **clipPP        /* <= clipping descriptor */
);
```

Description The `mdlClip_getFence` gets the clipping descriptor describing the current active fence. A fence must be defined prior to calling this function. If fence void lock is selected, the clipping region will include the area outside the fence, rather than the fence interior. The clipping descriptor may be passed to either `mdlClip_element` or `mdlClip_isElemInside` to clip elements to the fence or determine whether an element is within the fence. The clipping descriptor should be freed with `mdlClip_free` when it is no longer required.

Returns `mdlClip_getFence` returns `SUCCESS` if a fence is defined and `ERROR` if no fence is defined.

See Also `mdlClip_free`, `mdlClip_getRefBoundary`, `mdlClip_element`, `mdlClip_isElemInside`.

mdlClip_getRefBoundary

```
void mdlClip_getRefBoundary
(
void    **clipPP,      /* <= clipping descriptor */
int     refNumber,     /* => reference file number */
int     view           /* => view number */
);
```

Description The `mdlClip_getRefBoundary` function gets a clipping descriptor that describes a reference file boundary. The clipping descriptor may be passed to either `mdlClip_element` or `mdlClip_isElemInside` to clip elements to the reference file boundary or determine whether an element is within the boundary. The clipping descriptor should be freed with `mdlClip_free` when it is no longer required.

refNumber represents the reference file number with 1 representing the first reference file attachment. As reference file boundaries differ for views with different orientations, *view* is required to determine the correct orientation.

Returns `mdlClip_getRefBoundary` returns `SUCCESS` if the clipping descriptor is returned correctly, and an appropriate error status otherwise.

See Also `mdlClip_getFence`, `mdlClip_free`, `mdlClip_element`, `mdlClip_isElemInside`.

mdlClip_isElemInside

```
int mdlClip_isElemInside
(
int          *overlap,      /* <= TRUE if overlap */
MSElementDescr *edP,      /* => element to test */
void         *clipP,       /* => clip descriptor */
int          view,         /* => view number */
int          allowOverlap  /* => TRUE to allow overlaps */
);
```

Description `mdlClip_isElemInside` returns a non-zero value if the element *edP* is within the clipping descriptor, *clipP*. If *allowOverlap* is non-zero, overlapping elements are accepted as inside and *overlap* is set to `TRUE` if an element overlaps the clipping boundary.



Please note that “inside” can be relative, as the clipping descriptor may represent a fence void and, in this case, the clipping region would actually be the area outside the fence, rather than inside.

Returns `mdlClip_isElemInside` returns a non-zero value if the element is inside the clipping descriptor and zero otherwise.

See Also `mdlClip_getFence`, `mdlClip_getRefBoundary`, `mdlClip_free`, `mdlClip_element`.

Dynamic Buffer Functions

The dynamic buffer functions are used to load elements into MicroStation's internal buffers for processing by dynamic functions (the functions that are called repeatedly during cursor movement).

These functions are only rarely necessary, as MicroStation's element location logic normally loads these buffers automatically when elements are identified by the user (see logic flow diagram in the `mdlLocate_identifyElement` function).

MicroStation's dynamic buffer can either contain a single element or an element descriptor. If the dynamic buffer contains an element descriptor, the `userState_dynamicUpdate` function is called once for each element in the element descriptor.

The following table lists the dynamic buffer functions:

Function	Used to
<code>mdlDynamic_setElmDescr</code>	load an element descriptor into the dynamic buffer.
<code>mdlDynamic_loadElement</code>	load a single element into the dynamic buffer.

Example

See `dynamic.mc`.

`mdlDynamic_setElmDescr`

```
#include <mselems.h>

void mdlDynamic_setElmDescr
(
  MSElementDescr  *descrP  /* => ptr to element descr to load */
);
```

Description The `mdlDynamic_setElmDescr` function loads MicroStation's internal dynamic buffer with the element descriptor *descrP*. The element descriptor can either contain a single element, a complex element, or a list of unrelated (possibly complex) elements.

Ownership of the element descriptor pointed to by *descrP* is passed to MicroStation, and MicroStation will free it when it is finished with it. This is

one of the few cases in MDL where an application allocates an element descriptor and does not have to free it.

Returns `mdlDynamic_setElmDescr` is of type `void`. It returns no value.

See Also `mdlDynamic_loadElement`, `mdlLocate_identifyElement`, `userState_dynamicUpdate`, `userState_complexDynamicUpdate`.

mdlDynamic_loadElement

```
#include <mselems.h>

void mdlDynamic_loadElement
(
    MSElement    *eIP,          /* => pointer to element */
    int           fileNum,       /* => file number */
    ULONG         filePos       /* => file position */
);
```

Description The `mdlDynamic_loadElement` function loads MicroStation's internal dynamic buffer with the element pointed to by *eIP* in file number *fileNum* at file position *filePos*.

If a selection set is active, `mdlDynamic_loadElement` ignores all of its parameters and loads all of the elements in the selection set into the dynamic buffer.

Returns `mdlDynamic_loadElement` is of type `void`. It returns no value.

See Also `mdlDynamic_setElmDescr`, `mdlLocate_identifyElement`, `userState_dynamicUpdate`, `userState_complexDynamicUpdate`.

8

Text Utility Functions

Text utility functions load fonts, change text element fonts, manipulate text strings and perform other miscellaneous text manipulation functions.

MicroStation font libraries contain groups of vector text fonts. One of these fonts, usually a simple block font, is designated the fast font and is used during display operations when Fast Font is active in a view. MicroStation maintains the fast font and one other font in memory at a time. Fonts are loaded as needed during display and element creation functions.

The following table lists the text utility functions:

Function	Used to
<code>mdlText_getCurrentFont</code>	get the number of the font currently loaded in memory.
<code>mdlText_changeElementFont</code>	change the font of a text element.
<code>mdlText_extractFontStyle</code>	extract font information from an element.
<code>mdlText_loadFontStyle</code>	load a font from the font library.
<code>mdlText_fontExists</code>	determine whether font number exists in font library.
<code>mdlText_getFontInfo</code>	get properties of a font.
<code>mdlText_getFontName</code>	get descriptive name of text font.
<code>mdlText_extractStringsFromDscr</code>	get text strings from text node element descriptor
<code>mdlText_addStringsToNodeDscr</code>	add text strings to text node element descriptor.
<code>mdlText_expandString</code>	expand special characters in text string.
<code>mdlText_compressString</code>	compress special characters from text string.
<code>mdlText_expandTabs</code>	expand tabs to white space.
<code>mdlText_textFromNode</code>	get equivalent text element from text node element.

Function	Used to
<code>mdlText_nodeFromText</code>	get equivalent text node element from text element.
<code>mdlText_countBufferStrings</code>	count number of text strings in multi-line text buffer.
<code>mdlText_getStringArrayFromBuffer</code>	create array of text strings from multi-line text buffer.
<code>mdlText_freeStringArray</code>	free memory for string array allocated by <code>mdlText_getStringArrayFromBuffer</code> .
<code>mdlText_wordWrapBuffer</code>	word wrap text strings in multi-line text buffer.
<code>mdlText_getElementDescr</code>	generate an element descriptor whose components approximate the text item.

mdlText_getCurrentFont

```
#include <mdl.h>

int mdlText_getCurrentFont(void);
```

Description The `mdlText_getCurrentFont` function returns the number of the currently loaded font. MicroStation maintains the fast font and one slow font in memory at a time. The number of the currently loaded slow font is returned.

Returns `mdlText_getCurrentFont` returns the number of the currently loaded slow font.

See Also `mdlText_loadFontStyle`, `mdlText_getFontInfo`.

mdlText_changeElementFont

```
#include <mdl.h>

void mdlText_changeElementFont
(
  MSElementUnion  *elementP,      /* <= text element to be modified */
  TextStyleInfo    *textStyleP    /* => new font information */
);
```

Description The `mdlText_changeElementFont` function modifies a text element font.

elementP points to an element of type `TEXT_ELM`. No other element text properties, such as justification or text size, are modified.

textStyleP points to a structure of type `TextStyleInfo`. *textStyleP->font* contains the new font number of the text element. MicroStation loads this font into memory if necessary. The font does not need to be preloaded. The *textStyleP->style* member contains extended text attributes such as italics and bolding.

Returns The `mdlText_changeElementFont` function is of type `void`. It returns no value.

See Also `mdlText_extractFontStyle`.

mdlText_extractFontStyle

```
#include <mdl.h>

void mdlText_extractFontStyle
(
MSElementUnion    *elementP,      /* => text element */
TextStyleInfo      *textStyleP     /* <= font style information */
);
```

Description The `mdlText_extractFontStyle` function returns the font style information of a text element. *elementP* points to a text element of type `TEXT_ELM` or to a text node element of type `TEXT_NODE_ELM`.

The element's font number is returned in *textStyleP->font*. Element extended text style properties, such as italics and bolding, are returned in *textStyleP->style*. Normally *textStyleP->style* is zero.

Returns The `mdlText_extractFontStyle` function is of type `void`. It returns no value.

See Also `mdlText_loadFontStyle`, `mdlText_changeElementFont`.

mdlText_loadFontStyle

```
#include <mdl.h>

int mdlText_loadFontStyle
(
TextStyleInfo      *textStyleP     /* => font to be loaded */
);
```

Description The `mdlText_loadFontStyle` function loads a library font into memory. MicroStation keeps the fast font and one slow font in memory at a time. *textStyleP->font* contains the number of the library font to load into memory. *textStyleP->style* contains extended text properties such as italics and bolding. Normally *textStyleP->style* is zero.

Returns The `mdlText_loadFontStyle` function returns the number of the font loaded into memory. If the requested font could not be loaded, the fast font is loaded and this font number is returned.

See Also `mdlText_extractFontStyle`.

mdlText_fontExists

```
#include <mdl.h>

boolean mdlText_fontExists
(
    int      font      /* => font number */
);
```

Description The `mdlText_fontExists` function determines whether the font number *font* is present in the currently attached font library.



A text element can use a font not present in the font library if a different font library was in use at the time the element was created.

Returns The `mdlText_fontExists` function returns `TRUE` if the font is present in the library and `FALSE` if the font does not exist in the library.

See Also `mdlText_extractFontStyle`, `mdlText_loadFontStyle`.

mdlText_getFontInfo

```
#include <mdl.h>

int mdlText_getFontInfo
(
    TextFontInfo *fontInfoP,    /* <= font information */
    int          font          /* => font number */
);
```

Description The `mdlText_getFontInfo` function returns the properties of the font *font*. The `textFontInfo` structure is defined as follows:

```
typedef struct textFontInfo
{
    TextFlags lettersType;    /* upper, lower, fraction, int'l
                               characters are present in font */
    byte      charType;      /* symbol font or normal font */
    byte      vectorSize;    /* byte or word vectors used in def. */
    byte      graphicType;   /* raster or vector font */
    byte      charSize;      /* 8 or 16 bit representation of font*/
    short     tileSize;      /* size of character "tile" in def. */
    long      dataSize;      /* size of font def. incl. header */
    Point2d   origin;        /* origin of character in the "tile"*/
    TextScale scale;         /* scaling factor to be applied */
} TextFontInfo;
```



The `mdlText_getFontInfo` function returns all properties of a font except the descriptive name. This is designed like all font properties except the descriptive name are kept in memory. For efficiency, a separate function

`mdlText_getFontName` is provided to retrieve the descriptive font name as this operation requires a disk read.

Returns The `mdlText_getFontInfo` function returns `TRUE` if the font was found and `FALSE` if the font does not exist in the current font library.

See Also `mdlText_getFontName`.

`mdlText_getFontName`

```
#include <mdl.h>

boolean mdlText_getFontName
(
    char    *fontName,      /* <= font name */
    int     font            /* => font number */
);
```

Description Text fonts can have descriptive names associated with them in the font library. `mdlText_getFontName` copies the descriptive name of font number *font* to the character string *fontName*.

Returns The `mdlText_getFontName` function returns `TRUE` if the font was found in the font library or `FALSE` if it was not.

See Also `mdlText_getFontInfo`.

`mdlText_extractStringsFromDscr`

```
#include <mdl.h>

int mdlText_extractStringsFromDscr
(
    char        *buffer,      /* <= text buffer to receive strings*/
    int         bufferSize,    /* => maximum characters in buffer */
    MSElementDscr *elementDscrP /* => text node */
);
```

Description The `mdlText_extractStringsFromDscr` function returns the text strings associated with a text node element.

bufferSize contains the size (in characters) of the string buffer *buffer*.

elementDscrP points to an element descriptor for an element of type `TEXT_NODE_ELM`.

The strings are returned in the string buffer *buffer*. Multiple strings in the case of text node elements are separated by the new line character `textNLCharacter`. The entire buffer is `NULL`-terminated only at the end. This buffer's format is compatible with the format of the Dialog Manager for text editing.

The string buffer *buffer* can be NULL. In a NULL buffer, only the total length of the strings is returned in the function value
`mdlText_extractStringsFromDscr.`

Returns The `mdlText_extractStringsFromDscr` function returns the total number of characters placed in the user string buffer *buffer*.

See Also `mdlText_extractString`, `mdlText_addStringsToNodeDscr`.

mdlText_addStringsToNodeDscr

```
#include <mdl.h>

int mdlText_addStringsToNodeDscr
(
  MSElementDscr *elementDscrP, /* <= text node descriptor */
  char *buffer /* => text buffer with strings */
);
```

Description The `mdlText_addStringsToNodeDscr` function adds text strings to a text node element. *elementDscrP* points to an element descriptor for an element of type `TEXT_NODE_ELM`. The character strings are passed in the string buffer *buffer*. Multiple lines are delimited in *buffer* by the new line character `textNLCharacter`. The entire buffer should be NULL-terminated at the end.

Existing strings attached to the text node element are not disturbed.



In versions of MicroStation prior to 4.2, `\n` will not produce a new line. Instead, MDL provides the constant `textNLCharacter` which should be used in place of `\n`. For example, instead of the code:

```
sprintf(string, "%s", "line 1\nline 2\nline 3\n");
```

use:

```
sprintf(string, "%s%c%s%c%s%c", "line 1",textNLCharacter,
        "line 2",textNLCharacter, "line 3",textNLCharacter);
```

In versions 4.2 and later of MicroStation, `textNLCharacter` is equal to `\n`, so either may be used and backward compatibility will be maintained.



Every time this function is called, the same text origin is used. To avoid having many text items placed atop one-another, buffer all text beforehand and call this function a single time. This problem will be addressed in future releases.

Returns The `mdlText_addStringsToNodeDscr` function returns `ERROR` if the element descriptor *elementDscrP* does not point to an element of type `TEXT_NODE_ELM`.

See Also `mdlText_extractStringsFromDscr`.

mdlText_expandString

```
#include <mdl.h>

int mdlText_expandString
(
    char    *outBuffer,    /* <= buffer for converted string */
    char    *inBuffer,     /* => text string to be converted */
    int     bufferSize,    /* => size of outBuffer */
    int     conversionType /* => type of conversion */
);
```

Description The mdlText_expandString function expands strings containing special characters into a format suitable for editing.

The string buffer *inBuffer* is converted according to the conversion type *conversionType*.

It is then placed in the string buffer *outBuffer*.

bufferSize contains the size of the output buffer *outBuffer*.

conversionType consists of the constant FRACTIONS and/or INTERNATIONAL joined with the logical OR operator. The conversion expands special characters into their equivalent multi-character formats that are suitable for editing.

Returns The mdlText_expandString function returns the number of characters in the expanded string *outBuffer*.

See Also mdlText_compressString.

mdlText_compressString

```
#include <mdl.h>

int mdlText_compressString
(
    char    *outBuffer,    /* <= buffer for converted string */
    char    *inBuffer,     /* => text string to be converted */
    int     bufferSize,    /* => size of outBuffer */
    int     conversionType /* => type of conversion */
);
```

Description The mdlText_compressString function compresses strings containing multi-character sequences of special characters into their equivalent single-character representations.

The string buffer *inBuffer* is converted according to *conversionType*.

It is then placed in the string buffer *outBuffer*.

bufferSize contains the size of the output buffer *outBuffer*.

conversionType consists of the constants `FRACTIONS` and/or `CONTROL` joined with the logical OR operator. The conversion compresses special characters from their multi-character formats (normally used for editing) into their single-character representations.

Returns The `mdlText_compressString` function returns the number of characters in the compressed string *outBuffer*.

See Also `mdlText_expandString`.

mdlText_expandTabs

```
#include <mdl.h>

void mdlText_expandTabs
(
    char    *outBuffer,    /* <= buffer for converted string */
    char    *inBuffer,     /* => text string to be expanded */
    int     bufferSize,    /* => size of outBuffer */
    int     tabStops       /* => tab spacing */
);
```

Description `mdlText_expandTabs` expands tabs to their equivalent number of spaces.

Tabs embedded in the string buffer *inBuffer* are converted to spaces according to the tab stop spacing *tabStops*.

The tabs are then placed in the string buffer *outBuffer*. *bufferSize* contains the size of the output buffer *outBuffer*.

Returns The `mdlText_expandTabs` function is of type `void`. It returns no value.

See Also `mdlText_expandString`.

mdlText_textFromNode

```
#include <mdl.h>

int mdlText_textFromNode
(
    MSElementUnion *textP,        /* <= element based on textNodeP */
    MSElementUnion *textNodeP    /* => text node element */
);
```

Description The `mdlText_textFromNode` function creates an equivalent text element from a text node element. *textNodeP* points to a text node element of type `TEXT_NODE_ELM`. This element generates a text element of type `TEXT_ELM` from this text node element in the buffer *textP*.

Because a text node element's properties are a superset of text elements, the conversion is straightforward. The following text element parameters come directly from the text node: *level*, *graphic group*, *properties*, *symbology*, *justification*, *font*, *text size* and *rotation*.

No text strings are copied from the text node element to the text element.
The new text element *textP* contains no text.

Returns The mdlText_textFromNode function returns SUCCESS if it encounters no errors.

See Also mdlText_nodeFromText.

mdlText_nodeFromText

```
#include <mdl.h>

int mdlText_nodeFromText
(
  MSElementUnion  *textNodeP, /* <= text node element based on textP */
  MSElementUnion  *textP      /* => text element */
);
```

Description mdlText_nodeFromText creates an equivalent text node element from a text element.

textP points to a text element of type TEXT_ELM.

This element generates a text node element of type TEXT_NODE_ELM in the *textNodeP* buffer.

Text node element properties are a superset of text element properties. Properties of the text node element that has a corresponding text element property are copied directly. These properties are *level*, *graphic group*, *properties*, *symbology*, *justification*, *font* and *text size*. Properties that have no analog are taken from the active text parameters when the function is called. These properties are *node number*, *line length* and *line spacing*.

No text strings are copied from the text element to the text node element.
The new text node element *textNodeP* contains no text.

Returns The mdlText_nodeFromText function returns SUCCESS if it encounters no errors.

See Also mdlText_nodeFromText.

mdlText_countBufferStrings

```
#include <mdl.h>

int mdlText_countBufferStrings
(
  char  *buffer /* => multiline text buffer */
);
```

Description The mdlText_countBufferStrings function is used to count the number of text lines contained in a text buffer created by an RTYPE_MultilineText dialog item.

The text buffer *buffer* is NULL terminated. Individual lines of text are delimited in the buffer by the character defined by the MDL built-in variable textNLCharacter. Individual lines of text are *not* NULL terminated.

Returns `mdlText_countBufferStrings` returns the number of individual text lines in the buffer.

See Also `mdlText_getStringArrayFromBuffer`.

mdlText_getStringArrayFromBuffer

```
#include <mdl.h>

int mdlText_getStringArrayFromBuffer
(
  char    ***textStrings,    /* <= string array */
  char    *buffer,           /* => multiline text buffer */
  int     totalLines         /* => number of lines of text in buffer */
);
```

Description The `mdlText_getStringArrayFromBuffer` function is used to process a multi-line text buffer by extracting the individual text lines.

The argument *buffer* is a multi-line text buffer as created by an `RTYPE_MultilineText` dialog item. The argument *totalLines* is the number of strings present in the buffer, as returned by `mdlText_countBufferStrings`. The argument *textStrings* is a pointer to a variable of type pointer to a `char` pointer. MicroStation will allocate memory for an array of character pointers in *textStrings*. The resulting value of *textStrings* may be used as an array of character pointers, similar to the standard argument `argv` passed to every C main function. Typically, `mdlText_getStringArrayFromBuffer` may be used to prepare an array of strings to be passed to `mdlTextNode_createWithStrings`.

It is important that the memory allocated for *textStrings* be freed by the application after processing with a call to `mdlText_freeStringArray`.

Returns `mdlText_getStringArrayFromBuffer` returns `SUCCESS` if the string array was created successfully. Other return codes are listed in `mdlerrs.h`.

See Also `mdlText_freeStringArray`, `mdlTextNode_createWithStrings`.

mdlText_freeStringArray

```
#include <mdl.h>

int mdlText_freeStringArray
(
  char    **textStrings,    /* <=> array of text strings to be freed */
  int     totalLines        /* => number of lines of text */
);
```

Description The `mdlText_freeStringArray` function is used to free the memory MicroStation has allocated for an array of text string pointers in *textStrings* during a call to `mdlText_getStringArrayFromBuffer`. An MDL program processing multi-line text typically would use `mdlText_getStringArrayFromBuffer` to retrieve the text strings

from an `RTYPE_MultilineText` dialog item. After finishing with the text strings in *textStrings*, `mdlText_freeStringArray` should be called to release the memory.

Returns `mdlText_freeStringArray` returns `SUCCESS` if the memory was freed. Other error codes are listed in `mdlerrs.h`.

See Also `mdlText_getStringArrayFromBuffer`.

mdlText_wordWrapBuffer

```
#include <mdl.h>

int mdlText_wordWrapBuffer
(
    UChar    *buffer,          /* <=> multiline text buffer */
    int      bufferMax,        /* => size of buffer */
    int      lineLength        /* => maximum length of each line */
);
```

Description The `mdlText_wordWrapBuffer` function is used to word wrap a multi-line text buffer such that each individual line is less than *lineLength* characters.

The text buffer *buffer* is a NULL terminated buffer. Individual lines of text in the buffer are delimited with the character defined by the built-in variable `textNLCharacter`. This is the format of a buffer created by an `RTYPE_MultilineText` dialog item. The argument *bufferMax* is the size, in characters, of *buffer*. This is necessary as the word wrapping may cause the length of the text in *buffer* to grow due the addition of new `textNLCharacter` characters in *buffer*.

Returns `mdlText_wordWrapBuffer` returns `SUCCESS` if the text buffer was word wrapped successfully. Other error codes are listed in `mdlerrs.h`.

mdlText_getElementDescr

```
int mdlText_getElementDescr
(
    MSElementDescr  **outEdPP,      /* <= output element descriptor */
    Text_2d          *pTextElm,      /* => text element to convert */
    RotMatrix         *viewRMatrixP  /* => orientation if view ind. */
);
```

Description The `mdlText_getElementDescr` function returns an element descriptor, *outEdPP*, whose components represent the text element **pTextElm*. If the text element is view independent, *viewRMatrix* is used to determine the orientation.

Returns `mdlText_getElementDescr` returns `SUCCESS` if the element descriptor is processed successfully and an appropriate error code otherwise.

9

B-spline Functions

The `mdlBspline_...` functions deal specifically with B-spline topics. Uniform, non-uniform, rational and non-rational B-spline curves and surfaces are supported.

Unlike most other elements, B-splines are stored as several separate element types in a design file. To accommodate this, routines that need to create and manipulate B-splines in design files pass element descriptors containing all required element types. Other routines pass relevant information in data structures as defined in the `mdlbspln.h` file. These structures are described below.

All B-spline curves are defined over the parameter range 0.0 to 1.0. Similarly, surfaces are defined over the parameter range 0.0 to 1.0 in the U and V dimensions.

B-spline Structures

The `MSBsplineCurve` structure is defined as follows:

```
struct
{
    int          type;
    int          rational;
    BsplineDisplay display;
    BsplineParam  params;
    Dpoint3d      *poles;
    double        *knots;
    double        *weights;
} MSBsplineCurve;
```

The `MSBsplineSurface` structure is defined as follows:

```
struct
{
    int          type;
    int          rational;
    BsplineDisplay display;
```

```

    BsplineParam    uParams;
    BsplineParam    vParams;
    Dpoint3d        *poles;
    double           *uKnots;
    double           *vKnots;
    double           *weights;
    int              hole origin;
    int              numBounds;
    BsurfBoundary    *boundaries;
} MSBsplineSurface;

```

B-spline curves and surfaces can represent many graphic entities. The *type* integer specifies that the curve or surface represents one of the graphic entities listed below. These constants are defined in mdlbspln.h.

B-spline curve types:

B-spline curve type	Value
BSCURVE_GENERAL	0
BSCURVE_LINE	1
BSCURVE_CIRCULAR_ARC	2
BSCURVE_CIRCLE	3
BSCURVE_ELLIPTICAL_ARC	4
BSCURVE_ELLIPSE	5
BSCURVE_PARABOLIC_ARC	6
BSCURVE_HYPERBOLIC_ARC	7

B-spline surface types:

B-spline surface type	Value
BSSURF_GENERAL	0
BSSURF_PLANE	1
BSSURF_RIGHT_CYLINDER	2
BSSURF_CONE	3
BSSURF_SPHERE	4
BSSURF_TORUS	5
BSSURF_REVOLUTION	6
BSSURF_TAB_CYLINDER	7
BSSURF_RULED_SURFACE	8

The *rational* integer indicates that a rational B-spline is being described. If *rational* is TRUE, *weights* must point to an array that contains the same number of double-precision weights as there are poles. All weight values must be greater than zero and less than or equal to one. If *rational* is FALSE, the curve or surface is non-rational and the implied weight for each pole is one.

The `BsplineDisplay` structure is defined as follows:

```
struct
{
    int    curveDisplay;
    int    polygonDisplay;
    int    rulesByLength;
} BsplineDisplay;
```

The `BsplineDisplay` structure controls the display of a curve or surface. The integers *display->curveDisplay* and *display->polygonDisplay* control the display of the control polygon and B-spline curve or surface, respectively. If *display->rulesByLength* is TRUE, the rule lines used to display a surface will be spaced based on arc length along the surface. If *display->rulesByLength* is FALSE, the rule lines will be drawn at even intervals of the parameter range. This value is ignored when a B-spline curve is displayed.

The `BsplineParam` structure is defined as follows:

```
struct
{
    int    order;
    int    closed;
    int    numPoles;
    int    numKnots;
    int    numRules;
} BsplineParam;
```

The integer value of *params->order* indicates the B-spline order. MicroStation supports B-splines with a minimum order of 2 and a maximum order of 15.

A TRUE or FALSE value for *params->closed* indicates that the B-spline is closed (periodic) or open (non-periodic), respectively.

params->numPoles contains the number of poles in *poles*. The minimum number of poles is the same as the B-spline order. A single B-spline curve in a design file can contain a maximum of 101 poles. A single surface can contain a maximum of 101 rows of 101 poles each. If a curve or surface structure containing more poles is passed to the creation routines, the multiple curves or surfaces in the file will be chained together in the element descriptor returned from the routine.

params->numKnots contains the number of interior knot values. If zero knots are specified, a uniform knot vector is assumed. If a non-uniform knot vector is needed, *numKnots* must contain the number of interior knot values, *params->numPoles-params->order* for an open B-spline and *params->numPoles-1* for a closed B-spline. All interior knot values must be greater than or equal to zero and less than or equal to one.

For surfaces, separate structures determine the surface characteristics in each parameter direction, U and V. *uParams->numRules* and *vParams->numRules* contain the number of rule lines to be used in each parameter direction when the surface is displayed. Valid values are 2 through 256.

The *poles* array contains all the poles for a B-spline curve or surface stored as an array of *Dpoint3d* structures (as defined in the file *basetype.h*). For a surface the array contains *uParams->numPoles * vParams->numPoles* poles, with the U index varying fastest.

The poles are stored in homogeneous coordinates in the *poles* array. This means the value stored is the actual coordinates of the pole multiplied by its weight if the B-spline is rational. (A weight of 1.0 is used for non-rational B-splines). All routines described below that pass curve or surface structures as arguments assume that the *poles* array is appropriately weighted in the rational case. If the pole and weight arrays are passed separately, the poles are unweighted.

The number of boundaries on a surface is specified in the *numBounds* integer. If the surface contains boundaries, *holeOrigin* determines whether the area inside (if *holeOrigin* is *FALSE*) or outside (if *holeOrigin* is *TRUE*) of the boundaries is to be removed from the surface. Individual boundaries are stored in the array of *BsurfBoundary* structures, *boundaries*. The *BsurfBoundary* structure is defined as follows:

```
struct
{
    int          numPoints;
    Dpoint2d     *points;
} BsurfBoundary;
```

The number of points in each boundary is specified in *numPoints*, while *points* contains the actual boundary points in U-V space.

The array, *knots* for a curve or *uKnots* and *vKnots* for a surface, contains the complete knot vector for the B-spline. If the B-spline is open, these arrays contain *params->numPoles + params->order* doubles. If the B-spline is closed, the array contains *params->numPoles + 2* params->order -1* doubles.

B-spline Function Error Codes

If the routines described below do not complete successfully, they return one of the following error codes as defined in `mdlerrs.h`:

Error Code	Value
MDLERR_FILE2SUB3	-104
MDLERR_BADELEMENT	-105
MDLERR_INSMEMORY	-116
MDLERR_NOPOLES	-500
MDLERR_NOKNOTS	-501
MDLERR_NOWEIGHTS	-502
MDLERR_NOBOUNDS	-503
MDLERR_NONUMBOUNDS	-504
MDLERR_NOBSPHEADER	-505
MDLERR_TOOFEWPOLES	-506
MDLERR_TOOMANYPOLES	-507
MDLERR_BADBSPPELM	-508
MDLERR_BADPARAMETER	-509
MDLERR_BADORDER	-510
MDLERR_BADPERIODICITY	-511
MDLERR_BADPOLES	-512
MDLERR_BADKNOTS	-513
MDLERR_BADWEIGHTS	-514
MDLERR_BADSPIRALDEFINITION	-515

This chapter is comprised of the following sections

- B-spline Construction Functions
- B-spline Modification Functions
- B-spline Knot Functions
- B-spline Query Functions

B-spline Construction Functions

The following table lists B-spline construction functions:

Function	Used to
<code>mdlBspLine_createCurve</code>	create a B-spline curve element from specified input parameters.
<code>mdlBspLine_createSurface</code>	create a B-spline surface element from specified input parameters.
<code>mdlBspLine_copyCurve</code>	allocate memory for the <i>output</i> B-spline curve and copy data from input B-spline.
<code>mdlBspLine_copySurface</code>	allocate memory for the output B-spline surface and copy the data from the input B-spline.
<code>mdlBspLine_convertToCurve</code>	create a B-spline curve structure from a MicroStation element stored in an element descriptor.
<code>mdlBspLine_convertToSurface</code>	create a B-spline surface structure from a MicroStation element stored in an element descriptor.
<code>mdlBspLine_convertToEndcaps</code>	create an element descriptor containing closed B-spline curves that represent the end caps of a <code>SOLID_ELM</code> element.
<code>mdlBspLine_segmentCurve</code>	return a B-spline curve that is a portion of an initial curve.
<code>mdlBspLine_segmentSurface</code>	return a B-spline surface that is a part of an initial surface.
<code>mdlBspLine_appendCurves</code>	create a B-spline curve from the two initial B-spline curves.
<code>mdlBspLine_appendSurfaces</code>	create a B-spline surface from two initial B-spline surfaces that share a common edge.
<code>mdlBspLine_leastSquaresToCurve</code>	create the B-spline curve of specified parameters.
<code>mdlBspLine_leastSquaresToSurface</code>	create the B-spline surface of specified parameters.
<code>mdlBspLine_surfaceOfRevolution</code>	create a B-spline surface of revolution from the boundary curve and information.
<code>mdlBspLine_surfaceOfProjection</code>	create a B-spline surface of projection from the boundary curve and information.
<code>mdlBspLine_ruledSurface</code>	create a B-spline ruled surface between two B-spline curves.

Function	Used to
mdlBspline_coonsPatch	create a B-spline Coon's patch surface between four B-spline curves.
mdlBspline_skinPatch	create a B-spline skinned surface along the trace curve.
mdlBspline_extractFromCurve	create an element descriptor containing a MicroStation element from a B-spline curve.
mdlBspline_extractFromSurface	create an element descriptor containing a MicroStation SURFACE_ELM from a B-spline surface.
mdlBspline_extractIsoCurve	create an element descriptor containing B-spline curve(s) representing an iso-parametric curve of a B-spline surface.
mdlBspline_extractSilhouette	create a chain of silhouette curves that separate the visible and invisible parts of a B-spline surface.
mdlBspline_extractBoundary	create an element descriptor containing B-spline curve(s) representing the boundaries of a B-spline surface.
mdlBspline_extractProfile	create the B-spline curve that generates a B-spline surface of projection or surface of revolution.
mdlBspline_spiral	create a B-spline curve spiral from the supplied information.
mdlBspline_helix	create a B-spline curve helix from the supplied information.
mdlBspline_imposeBoundary	place a boundary in a surface that follows a curve in space.
mdlBspline_approximateCurve	create a mathematically defined curve.
userBspline_curveFunction	define a curve.
mdlBspline_approximateSurface	create a mathematically defined surface.
userBspline_surfaceFunction	define a surface.
mdlBspline_blendCurve	smoothly transition between two B-spline curves.
mdlBspline_blendSurface	smoothly transition between two B-spline surfaces.
mdlBspline_blendRails	create a B-spline blending surface.
mdlBspline_constantRadiusFillet	create a constant radius B-spline fillet between two B-spline surfaces.

Function	Used to
<code>mdlBspLine_variableRadiusFillet</code>	create a varying radius B-spline fillet between two B-spline surfaces.
<code>mdlBspLine_convertToPlanarSurface [mdlLib.m1]</code>	convert a shape to a planar B-spline surface.
<code>mdlBspLine_trimmedPlaneFromCurves [mdlLib.m1]</code>	trim a planar B-spline surface by B-spline curves.
<code>mdlBspLine_getCurveFromSurface</code>	create a B-spline curve from the poles in any row or column of the control net.
<code>mdlBspLine_getEdgeFromSurface</code>	create a B-spline curve from one of the four edges of a B-spline surface.
<code>mdlBspLine_offset</code>	create the offset of a B-spline curve.
<code>mdlBspLine_offsetSurface</code>	create the offset of a B-spline surface.
<code>mdlBspLine_catmullRomCurve</code>	create a B-spline curve element from interpolating a series of points.
<code>mdlBspLine_catmullRomSurface</code>	create a B-spline surface element from interpolating points.
<code>mdlBspLine_gordonSurface</code>	create a B-spline surface from a network of curves.
<code>mdlBspLine_crossSectionSurface</code>	create a B-spline surface from section curves.
<code>mdlBspLine_cubicInterpolation</code>	create a B-spline curve from interpolating a series of points.
<code>mdlBspLine_nSidedPatch</code>	create a B-spline surface from edges.
<code>mdlBspLine_createCurvesFromChain</code>	create an element descriptor containing B-spline curves from a curve chain.
<code>mdlBspLine_createSurfacesFromChain</code>	create an element descriptor containing B-spline surfaces from a surface chain.
<code>mdlBspLine_convertToCurveChain</code>	create a curve chain from an element descriptor.
<code>mdlBspLine_convertToSurfaceChain</code>	create a surface chain from an element descriptor.
<code>mdlBspLine_intersectSurfaceChains</code>	create a list of points representing the intersection points between two surface chains.
<code>mdlBspLine_arcLengthFromParameters</code>	compute arc length along a curve.
<code>mdlBspLine_computeEqualChordByLength</code>	evaluate curve to get equal chord length.
<code>mdlBspLine_cubicInterpolationExt</code>	create fourth order curve from set of points.
<code>mdlBspLine_parameterFromArcLength</code>	return parameter value corresponding to given arc length.

Example

See bspline.mc.

mdlBspline_createCurve

```

#include <mdlbsp1n.h>

#include <mselems.h>

int mdlBspline_createCurve
(
  MSElementDescr    **out,          /* <= curve elements created */
  MSElementUnion    *in,           /* => template element, or NULL */
  MSBsplineCurve     *curve,        /* => curve definition */
);

```

Description The `mdlBspline_createCurve` function creates an element descriptor containing a B-spline curve from a B-spline curve structure. As B-spline curves are represented by more than one design file element, `mdlBspline_createCurve` creates all component elements and returns them in the element descriptor, *out*.

If *in* is NULL, the display parameters for the created curve are taken from the active MicroStation settings at the time of the call; otherwise the display parameters from *in* are used. All attribute information from *in* will be retained in the B-spline curve header.

curve points to the structure containing the B-spline curve information. Individual arguments affect the created curve as discussed in the overview.

The pole points are transformed by the current transformation matrix if one exists. The maximum number of poles in a single B-spline curve in the design file is 101. If more poles are passed to `mdlBspline_createCurve`, the element descriptor returned will contain a complex chain containing multiple B-spline curves.

Returns The `mdlBspline_createCurve` function returns `SUCCESS` (zero) to indicate successful completion. Otherwise, it returns one of the following error codes:

```

#define MDLERR_NOPOLES      -500
#define MDLERR_NOKNOTS     -501
#define MDLERR_NOWEIGHTS   -502
#define MDLERR_TOOFEWPoles -506
#define MDLERR_TOOMANYPoles -507

```

See Also `mdlBspline_createSurface`.

mdlBspline_createSurface

```
#include <mdlbspln.h>
#include <mselems.h>

int mdlBspline_createSurface
(
  MSElementDescr    **out,          /* surface elements created */
  MSElementUnion    *in,           /* template element, or NULL */
  MSBsplineSurface   *surface       /* surface definition */
);
```

Description mdlBspline_createSurface creates an element descriptor containing a B-spline surface from a B-spline surface structure. As B-spline surfaces are represented by more than one design file element, mdlBspline_createSurface creates all of the component elements and returns them in the element descriptor, *out*.

If *in* is NULL, the display parameters for the created surface are taken from the active MicroStation settings at the time of the call; otherwise the display parameters from *in* are used. All attribute information from *in* will be retained in the B-spline surface header.

surface points to the structure containing the B-spline surface information. Individual arguments affect the created surface as discussed in the overview.

The pole points are transformed by the current transformation matrix, if one exists. The maximum number of poles in a single row of a B-spline surface in the design file is 101. The maximum number of rows in a B-spline surface is also 101.

Returns The mdlBspline_createSurface function returns SUCCESS (zero) to indicate successful completion. Otherwise, it returns one of the following error codes:

```
#define MDLERR_FILE2SUB3    -104
#define MDLERR_NOPOLES     -500
#define MDLERR_NOKNOTS     -501
#define MDLERR_NOWEIGHTS   -502
#define MDLERR_NOBOUNDS    -503
#define MDLERR_TOOFEWPOLY  -506
#define MDLERR_TOOMANYPOLY -507
```

See Also mdlBspline_createCurve.

mdlBspline_copyCurve

```
#include <mdlbspln.h>

int mdlBspline_copyCurve
(
  MSBsplineCurve    *output,        /* <= resulting curve */
  MSBsplineCurve    *input         /* => original curve */
);
```

Description The mdlBspine_copyCurve function allocates memory for the *output* B-spline curve and copies all data from the *input* B-spline. The values stored in the curve structure determine the amount of memory the function allocates.

Returns mdlBspine_copyCurve returns SUCCESS (zero) to indicate successful completion. It returns MDLERR_INSFMEMORY if there is not enough memory for the allocations.

See Also mdlBspine_copySurface.

mdlBspine_copySurface

```
#include <mdlbspin.h>

int mdlBspine_copySurface
(
    MSBspineSurface *output, /* <= resulting surface */
    MSBspineSurface *input  /* => original surface */
);
```

Description The mdlBspine_copySurface function allocates memory for the *output* B-spline surface and copies all data from the *input* B-spline. The values stored in the surface structure determine the amount of memory the function allocates.

Returns mdlBspine_copySurface returns SUCCESS (zero) to indicate successful completion, or MDLERR_INSFMEMORY if there is not enough memory for the allocations.

See Also mdlBspine_copyCurve.

mdlBspine_convertToCurve

```
#include <mdlbspin.h>
#include <mselems.h>

int mdlBspine_convertToCurve
(
    MSBspineCurve *curve, /* <= created curve structure */
    MSElementDescr *in    /* => defining element(s) */
);
```

Description mdlBspine_convertToCurve creates a B-spline curve structure from an element stored in an element descriptor. It accepts the following elements types (as defined in mselems.h): ARC_ELM, ELLIPSE_ELM, CURVE_ELM, LINE_ELM, LINE_STRING_ELM, SHAPE_ELM, CMLPX_STRING_ELM, CMLPX_SHAPE_ELM or BSPLINE_CURVE_ELM. This routine allocates the memory for the poles, knots and weights as necessary.

curve points to the B-spline curve structure returned.

in points to the element descriptor to be converted.

Returns mdlBspine_convertToCurve returns SUCCESS (zero) to indicate successful completion, or MDLERR_INSFMEMORY if there is not enough memory for the allocations. It returns MDLERR_BADELEMENT if the element descriptor does not contain an element of the correct type.

See Also mdlBspline_convertToSurface.

mdlBspline_convertToSurface

```
#include <mdlbspIn.h>
#include <mselems.h>

int mdlBspline_convertToSurface
(
    MSBsplineSurface *surface, /* <= created surface structure */
    MSElementDescr *in        /* => defining element(s) */
);
```

Description mdlBspline_convertToSurface creates a B-spline surface structure from a MicroStation element stored in an element descriptor. It accepts the following element types in the element descriptor (as defined in mselems.h): CONE_ELM, SURFACE_ELM, SOLID_ELM or BSPLINE_SURFACE_ELM. This routine allocates the memory for the poles, knots and weights as necessary.

surface points to the B-spline surface structure returned.

in points to the element descriptor to be converted.

Returns The mdlBspline_convertToSurface function returns SUCCESS (zero) to indicate successful completion, or MDLERR_INSMEMORY if there is not enough memory for the allocations. It returns MDLERR_BADELEMENT if the element descriptor does not contain an element of the correct type.

See Also mdlBspline_convertToEndcaps, mdlBspline_convertToCurve.

mdlBspline_convertToEndcaps

```
#include <mdlbspIn.h>
#include <mselems.h>

int mdlBspline_convertToEndcaps
(
    MSElementDescr **endCaps, /* <= created curves */
    MSElementDescr *in        /* => solid element */
);
```

Description mdlBspline_convertToEndcaps creates an element descriptor containing the closed B-spline curves that represent the end caps of a SOLID_ELM element. It allocates the memory for the element descriptor as needed.

endCaps points to the element descriptor containing the B-spline curves.

in points to the element descriptor to be converted.

Returns mdlBspline_convertToEndcaps returns SUCCESS (zero) to indicate successful completion or MDLERR_INSMEMORY if memory is insufficient for the allocations. It returns MDLERR_BADELEMENT if the element descriptor does not contain an element with the correct type (SOLID_ELM as defined in mselems.h).

See Also mdlBspline_convertToSurface, mdlBspline_convertToCurve.

mdlBspline_segmentCurve

```
#include <mdlbsp1n.h>

int mdlBspline_segmentCurve
(
    MSBsplineCurve  *output, /* <= created curve fragment */
    MSBsplineCurve  *input,  /* => original curve */
    double           uInitial, /* => initial parameter of fragment */
    double           uFinal   /* => final parameter of fragment */
);
```

Description mdlBspline_segmentCurve returns a B-spline curve that is a portion of an initial curve. If the input curve is open, the segment parameter range is intersected with the closed interval from zero to one. If the input curve is closed, the parameter range is treated periodically relative to the interval zero to one. This routine allocates memory for the output curve as necessary.

output points to the returned curve. It can point to the structure that *input* points to.

input points to the B-spline curve to segment.

uInitial and *uFinal* define the segment parameter range. If (*uFinal* < *uInitial*) the resulting segment will run in the opposite direction relative to the original curve. If (*uFinal* = *uInitial*), an error results and no curve is returned.

Returns mdlBspline_segmentCurve returns SUCCESS (zero) to indicate successful completion. It returns MDLERR_INSFMEMORY if there is not enough memory for the allocations. It returns MDLERR_BADPARAMETER if the segment parameter range is zero.

See Also mdlBspline_segmentSurface.

mdlBspline_segmentSurface

```
#include <mdlbsp1n.h>

int mdlBspline_segmentSurface
(
    MSBsplineSurface *output, /* <= created surface fragment */
    MSBsplineSurface *input,  /* => original surface */
    Dpoint2d         *initial /* => initial parameter of fragment */
    Dpoint2d         *final   /* => final parameter of fragment */
);
```

Description mdlBspline_segmentSurface returns a B-spline surface that is a portion of the initial surface. If the surface is open in a particular direction, the segment parameter range is intersected with the closed interval from zero to one in that parameter. If the surface is closed in a particular direction, then the range is treated periodically

relative to the interval zero to one in that direction. This routine allocates memory for the output surface as necessary.

output points to the returned surface. It can point to the structure that the *input* pointer points to.

input points to the B-spline surface to segment.

initial and *final* define the segment parameter range with their X coordinate corresponding to the U parameter and Y coordinate corresponding to the V parameter. If a parameter range of zero is specified in either direction, an error results and no surface is returned.

Returns `mdlBspline_segmentSurface` returns `SUCCESS` to indicate successful completion or `MDLERR_INSFMEMORY` if memory is insufficient for the allocations. It returns `MDLERR_BADPARAMETER` if the segment parameter range is zero in either direction.

See Also `mdlBspline_segmentCurve`.

mdlBspline_appendCurves

```
#include <mdlbspln.h>

int mdlBspline_appendCurves
(
    MSBsplineCurve  *output,  /* <= resulting curve */
    MSBsplineCurve  *input1,  /* => first curve */
    MSBsplineCurve  *input2   /* => second curve */
);
```

Description The `mdlBspline_appendCurves` function creates a B-spline curve from two initial B-spline curves. If the second curve does not start where the first curve ends, the resulting curve will have a discontinuity. This routine allocates memory for the output curve as necessary.

output points to the returned curve. It can point to the same structure that *input1* or *input2* points to.

input1 points to the first B-spline curve.

input2 points to the second B-spline curve.

Returns `mdlBspline_appendCurves` returns `SUCCESS` (zero) to indicate successful completion. It returns `MDLERR_INSFMEMORY` if there is not enough memory for the allocations.

See Also `mdlBspline_appendSurfaces`.

mdlBspline_appendSurfaces

```
#include <mdlbsp1n.h>

int mdlBspline_appendSurfaces
(
    MSBsplineSurface *output,          /* <= resulting surface */
    MSBsplineSurface *input1,         /* => first surface */
    MSBsplineSurface *input2,         /* => second surface */
    int edge,                          /* => direction flag */
);
```

Description `mdlBspline_appendSurfaces` creates a B-spline surface from two initial B-spline surfaces. The two surfaces must share a common edge or share common B-spline parameters along an edge; i.e., the surfaces must have the same order, number of poles, and knot vector along that edge. The two surfaces must also be either rational or non-rational. If the second surface does not start where the first surface ends, the resulting surface will have a discontinuity. This routine allocates memory for the output surface as necessary.

output points to the returned surface. It can point to the same structure that *input1* or *input2* points to.

input1 points to the first B-spline surface.

input2 points to the second B-spline surface.

edge contains a flag indicating the edge along which the surfaces will be joined. (The values `BSSURF_U` and `BSSURF_V` are defined in the file `mdlbsp1n.h`).

Returns `mdlBspline_appendSurfaces` returns `SUCCESS` (zero) to indicate successful completion, and `MDLERR_INSFMEMORY` if there is not enough memory for the allocations. It returns one of the following error codes if the surfaces are not compatible across the edge: `MDLERR_BADORDER`, `MDLERR_BADKNOTS`, `MDLERR_BADPOLES`, `MDLERR_BADWEIGHTS`.

See Also `mdlBspline_appendCurves`.

mdlBspline_leastSquaresToCurve

```
#include <mdlbsp1n.h>

int mdlBspline_leastSquaresToCurve
(
    MSBsplineCurve *curve,             /* <=> resulting curve */
    double *avgDistance,               /* <= average error, or NULL */
    double *maxDistance,               /* <= maximum error, or NULL */
    Dpoint3d *points,                  /* => data points */
    double *uValues,                   /* => parameter values or NULL */
    int numPoints                       /* => number of data points */
);
```

Description `mdlBspline_leastSquaresToCurve` creates the B-spline curve of specified parameters. The curve it creates is the best fit to a set of data points. If the returned curve is open, it starts at the first data point and ends at the last data point. The curve minimizes the sum of the square of the distances between each data point and a corresponding point on the curve. This corresponding point on the B-spline curve for each data point is the evaluated value at a specific parameter. This choice of parameter value for each data point can be supplied by the calling program or calculated within the routine. If *uValues* is `NULL`, the parameter values used will be the series of accumulated distances between data points along the line string defined by the data points. If *uValue* is not `NULL`, the values in this array will be used. This routine allocates memory for the returned curve as necessary.

curve points to the B-spline curve returned. The values passed in *curve->params* determine the order, number of poles, and periodicity of the returned curve. If a non-uniform curve is specified, the routine assumes that the knot vector is valid.

If *avgDistance* != `NULL`, this `double` will be set to the average of the distances between the data points and the B-spline curve.

If *maxDistance* != `NULL`, this `double` will be set to the maximum of the distances between the data points and the B-spline curve.

points is the array containing the data points.

If *uValues* != `NULL`, this array of parameter values will be used in the calculation.

numPoints is the number of data points supplied. This number must be greater than or equal to *curve->params.numPoles*.

Returns The `mdlBspline_leastSquaresToCurve` function returns `SUCCESS` (zero) to indicate successful completion. It returns `MDLERR_INSFMEMORY` if there is not enough memory for the allocations. It returns `MDLERR_INSFINFO`, `MDLERR_TOOFEWPOLES` or `MDLERR_NOKNOTS` if the function is not called with the correct information.

See Also `mdlBspline_leastSquaresToSurface`.

mdlBspline_leastSquaresToSurface

```
#include <mdlbsp1n.h>

int mdlBspline_leastSquaresToSurface
(
    MSBsplineCurve    *surface,      /* <=> resulting surface */
    double             *avgDistance, /* <= average error, or NULL */
    double             *maxDistance, /* <= maximum error, or NULL */
    Dpoint3d           *points,      /* => data points */
    Dpoint2d           *uvValues,     /* => parameter values or NULL */
    int                 *uNumPoints,  /* => number of points per row */
    int                 vNumPoints    /* => number of rows of points */
);
```


Description The `mdlBspline_leastSquaresToSurface` function creates the B-spline surface of specified parameters. The surface it creates is the best fit for a set of data points. If the returned surface is open, the function interpolates the data points along the surface edges. The surface minimizes the sum of the square of the distances between each data point and a corresponding point on the surface. This corresponding point on the B-spline surface for each data point is the evaluated value at a specific parameter. This choice of parameter value for each data point can be supplied by the calling program or calculated within the routine. If *uvValues* is `NULL`, the parameter values used will be an accumulation of distances between the data points along the line string the data points define in the U and V directions. If *uvValue* is not `NULL`, the values in this array will be used. Individual rows can have different numbers of data points in them, but the generated surface will have a rectangular array of poles. This routine allocates memory for the returned surface as necessary.

surface points to the B-spline surface returned. The values in *surface->uParams* and *surface->vParams* determine the order, number of poles, and periodicity of the returned surface in each direction. If a non-uniform surface is specified, the routine assumes that the knot vectors are valid.

If *avgDistance* != `NULL`, this `double` will be set to the average of the distance between the data points and the B-spline surface.

If *maxDistance* != `NULL`, this `double` will be set to the maximum of the distances between the data points and the B-spline surface.

points is the array containing the data points stored row after row.

If *uvValues* != `NULL`, this array of parameter values will be used in the calculation.

uNumPoints is an array of integers specifying the number of data points in each row. These values must be greater than or equal to *surface->uParams.numPoles*.

vNumPoints contains the number of rows of data points. This value must be greater than *surface->vParams.numPoles*.

Returns The `mdlBspline_leastSquaresToSurface` function returns `SUCCESS` (zero) to indicate successful completion. It returns `MDLERR_INSFMEMORY` if there is not enough memory for the allocations. It returns `MDLERR_INSFINFO`, `MDLERR_TOOFEWPOLES` or `MDLERR_NOKNOTS` if it is not called with the correct information.

See Also `mdlBspline_leastSquaresToCurve`.

mdlBspline_surfaceOfRevolution

```
#include <mdlbsp1n.h>

int mdlBspline_surfaceOfRevolution
(
    MSBsplineSurface *surface,      /* <= resulting surface */
    MSBsplineCurve   *boundary,     /* => defining curve */
    Dpoint3d         *center,       /* => center of revolution */
    Dpoint3d         *axis,         /* => axis of revolution */
    double            start,        /* => start of revolution */
    double            sweep         /* => extent of revolution */
);
```

Description The `mdlBspline_surfaceOfRevolution` function creates a B-spline surface of revolution from the boundary curve and information specifying start and sweep angles. It allocates memory for the surface as necessary.

surface points to the returned B-spline surface.

boundary is the curve used to generate the surface of revolution.

center and *axis* define the axis of revolution for the surface.

start defines the angle where the surface will start.

sweep defines how far the surface will be extended. It is defined relative to *start*.

Returns `mdlBspline_surfaceOfRevolution` returns `SUCCESS` (zero) to indicate successful completion. It returns `MDLERR_INSMEMORY` if there is not enough memory for the allocations.

See Also `mdlBspline_surfaceOfProjection`, `mdlBspline_extractProfile`.

mdlBspline_surfaceOfProjection

```
#include <mdlbsp1n.h>

int mdlBspline_surfaceOfProjection
(
    MSBsplineSurface *surface,      /* <= resulting surface */
    MSBsplineCurve   *boundary,     /* => defining curve */
    Dpoint3d         *delta         /* => projection definition */
);
```

Description `mdlBspline_surfaceOfProjection` creates a B-spline surface of projection from the boundary curve and information specifying the distance and direction to project the boundary curve. It allocates memory for the surface as necessary.

surface points to the returned B-spline surface.

boundary is the curve used to generate the surface of projection.

delta is a point that defines the distance and direction to project the boundary curve.

Returns The mdlBspline_surfaceOfProjection function returns SUCCESS (zero) to indicate successful completion. It returns MDLERR_INSFMEMORY if there is not enough memory for the allocations.

See Also mdlBspline_surfaceOfRevolution, mdlBspline_extractProfile.

mdlBspline_ruledSurface

```
#include <mdlbsp1n.h>

int mdlBspline_ruledSurface
(
    MSBsplineSurface *surface,      /* <= resulting surface */
    MSBsplineCurve   *curve1,      /* => first edge curve */
    MSBsplineCurve   *curve2,      /* => second edge curve */
);
```

Description The mdlBspline_ruledSurface function creates a B-spline ruled surface between two B-spline curves. It allocates memory for the surface as necessary.

surface points to the returned B-spline surface.

curve1 and *curve2* are the bounding curves of the B-spline surface. Their closest ends will be connected by the straight edges of the surface.

Returns The mdlBspline_ruledSurface function returns SUCCESS (zero) to indicate successful completion. It returns MDLERR_INSFMEMORY if there is not enough memory for the allocations.

See Also mdlBspline_coonsPatch.

mdlBspline_coonsPatch

```
#include <mdlbsp1n.h>

int mdlBspline_coonsPatch
(
    MSBsplineSurface *surface,      /* <= resulting surface */
    MSBsplineCurve   curves[4],    /* => edge curves */
);
```

Description The mdlBspline_coonsPatch function creates a B-spline Coon's patch surface bounded by the four B-spline curves. It allocates memory for the surface as necessary.

surface points to the returned B-spline surface.

curves is an array of the four bounding curves of the B-spline surface. For a valid surface to be returned, these four curves must meet at their ends to form a single closed area.

Returns `mdlBspline_coonsPatch` returns `SUCCESS` (zero) to indicate successful completion. It returns `MDLERR_INSMEMORY` if there is not enough memory for the allocations.

See Also `mdlBspline_ruledSurface`.

mdlBspline_skinPatch

```
#include <mdlbsp1n.h>

int mdlBspline_skinPatch
(
    MSBsplineSurface *surface,          /* <= resulting surface */
    MSBsplineCurve   *trace,           /* => trace curve */
    MSBsplineCurve   *section1,        /* => first section curve */
    MSBsplineCurve   *section2,        /* => second section curve */
    Dvector3d        *orientation1,    /* => first orientation */
    Dvector3d        *orientation2,    /* => second orientation */
    int               checkInflectionPt /* => inflection flag */
);
```

Description The `mdlBspline_skinPatch` function creates a B-spline skinned surface along the trace curve. This surface has two planar section curves as bounding edges at opposite ends of the trace. The alignment of these curves is defined by supplied vectors that lie in the same plane as their respective section curves. This function allocates memory for the surface as needed.

surface points to the returned B-spline surface.

trace points to the B-spline curve along which the surface will be created.

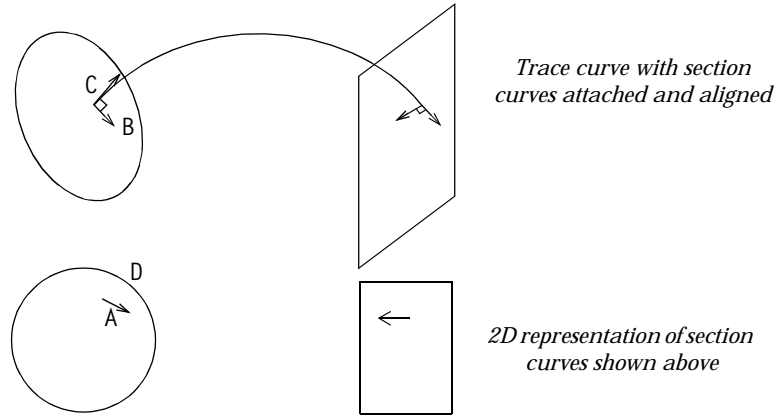
section1 and *section2* are the B-spline surface's bounding curves. They lie in the

$z=0$ plane; the Z coordinate of their poles is assumed to be zero. (If the Z coordinate is non-zero, it is ignored).

orientation1 and *orientation2* are planar vectors lying in the plane of the section curves. Both a translation and a rotation are needed to give the orientation vector (A) the same position and direction as the trace curve's normal vector (B). These same operations are performed on the section curve in order to align it relative to the trace curve before the surface is constructed. The trace curve's tangent vector (C) is perpendicular to, and thus defines, the plane containing the positioned section curve.

checkInflectionPt should usually be set to `TRUE` so that the cross sections of the resulting surface will remain on the same side of the trace when an inflection point is reached. This will prevent the "sausage link" effect caused when an inflection point flips 180°. If *checkInflectionPt* is `FALSE`, the routine will ignore the trace curve's inflection points.

Returns mdlBspline_skinPatch returns SUCCESS (zero) to indicate successful completion or. MDLERR_INSFMEMORY if memory is insufficient for the allocations.



A = Orientation vector, *B* = Trace curve's normal vector, *C* = Trace curve's tangent vector, *D* = The planar section curve in Z=Oplane

mdlBspline_extractFromCurve

```
#include <mdlbspln.h>
#include <mselems.h>

int mdlBspline_extractFromCurve
(
  MSElementDescr **out, /* <= resulting element(s) */
  MSBsplineCurve *curve /* => curve structure */
);
```

Description The mdlBspline_extractFromCurve function creates an element descriptor containing a MicroStation element from a B-spline curve. The element descriptor contains one of the following element types: LINE_ELM, LINE_STRING_ELM, SHAPE_ELM or ARC_ELM. This function succeeds only when *curve->type* equals BSCURVE_LINE, BSCURVE_CIRCULAR_ARC, BSCURVE_CIRCLE or BSCURVE_ELLIPSE. This function also allocates the memory needed for the element descriptor.

out points to the element descriptor created.

curve points to the B-spline curve that generates the element.

Returns mdlBspline_extractFromCurve returns SUCCESS (zero) to indicate successful completion or MDLERR_INSFMEMORY if memory is insufficient for the allocations. It returns MDLERR_BADELEMENT if the curve does not have the correct type.

See Also mdlBspline_extractFromSurface.

mdlBspline_extractFromSurface

```
#include <mdlbsp1n.h>
#include <mselems.h>

int mdlBspline_extractFromSurface
(
  MSElementDescr  **out,          /* <= resulting element(s) */
  MSBsplineSurface *surface       /* => surface structure */
);
```

Description The `mdlBspline_extractFromSurface` function creates an element descriptor containing a MicroStation `SURFACE_ELM` from a B-spline surface. This function succeeds only when *surface->type* equals `BSSURF_RIGHT_CYLINDER`, `BSSURF_CONE`, `BSSURF_REVOLUTION` or `BSSURF_TAB_CYLINDER`. This function also allocates memory needed for the element descriptor.

out points to the element descriptor created.

surface points to the B-spline surface that generates the element.

Returns `mdlBspline_extractFromSurface` returns `SUCCESS` (zero) to indicate successful completion or `MDLERR_INSFMEMORY` if memory is insufficient for the allocations. It returns `MDLERR_BADELEMENT` if the surface does not have the correct type.

See Also `mdlBspline_extractFromCurve`.

mdlBspline_extractIsoCurve

```
#include <mdlbsp1n.h>
#include <mselems.h>

int mdlBspline_extractIsoCurve
(
  MSElementDescr  **out,          /* <= resulting element(s) */
  MSBsplineSurface *surface,       /* => surface structure */
  double           uvValue,        /* => parameter value */
  int              direction       /* => direction flag */
);
```

Description The `mdlBspline_extractIsoCurve` function creates an element descriptor containing B-spline curve(s) representing an iso-parametric curve of a B-spline surface. An element descriptor is returned to enable the multiple pieces of the iso-parametric curve to be returned in case the surface contains boundaries. This routine allocates memory needed for the element descriptor.

out points to the returned element descriptor.

surface points to the B-spline surface from which the iso-curve will be extracted.

uvValue contains the parameter where the surface will be evaluated.

direction contains a flag indicating the requested iso-curve. (The BSSURF_U and BSSURF_V values are defined in mdlbspIn.h).

Returns mdlBspline_extractIsoCurve returns SUCCESS (zero) to indicate successful completion or MDLERR_INSMEMORY if memory is insufficient for the allocations.

See Also mdlBspline_extractSilhouette, mdlBspline_extractBoundary.

mdlBspline_extractSilhouette

```
#include <mdlbspIn.h>

int mdlBspline_extractSilhouette
(
    CurveChain      **chainPP,      /* <= resulting curves */
    MSBsplineSurface *surfaceP,     /* => input surface */
    double          tolerance,      /* => tolerance value */
    boolean          cubicFit,       /* => cubic curve or line str. */
    Dpoint3d         *cameraP       /* => camera position or NULL */
);
```

Description mdlBspline_extractSilhouette creates a chain of silhouette curves from the input surface. Silhouette curves, in general, are curves on a surface that separate the visible parts from the invisible parts of the surface. A surface may have more than one silhouette curve. This routine allocates memory for the resulting curves as necessary.

chainPP points to the address of the silhouette curve chain.

surface points to the input surface.

tolerance is the maximum allowed deviation from the true silhouette curve(s).

If *cubicFit* is TRUE, the silhouette curves are of B-spline curves of order 4. Otherwise, they are line strings.

cameraP points to the camera location if the camera view is on.



Previously documented versions of this function listed different parameters. Be sure to update your code to conform to the new function prototype.

Returns mdlBspline_extractSilhouette returns SUCCESS to indicate successful completion. It returns MDLERR_INSMEMORY if there is not enough memory for allocation.

See Also mdlBspline_extractIsoCurve, mdlBspline_extractBoundary.

mdlBspline_extractBoundary

```
#include <mdlbspln.h>
#include <mselems.h>
int mdlBspline_extractBoundary
(
  MSElementDescr  **out,          /* <= resulting element(s) */
  MSBsplineSurface *surface,      /* => surface structure */
  double           tolerance      /* => tolerance value */
);
```

Description mdlBspline_extractBoundary creates an element descriptor containing B-spline curve(s). These curves represent the boundaries of a B-spline surface. An element descriptor is returned so the multiple pieces of the boundary curve are returned in case the surface contains more than one boundary. The returned curves will remain within a specified tolerance of the surface boundaries. This function allocates the memory needed for the element descriptor.

out points to the returned element descriptor.

surface points to the B-spline surface where the boundary curve(s) will be extracted.

tolerance represents the max. allowed deviation from a straight line along the curve.

Returns mdlBspline_extractBoundary returns SUCCESS to indicate successful completion or MDLERR_INSMEMORY if memory is insufficient for the allocations. It returns MDLERR_NOBOUNDS if the surface contains no boundaries (*surface->numBounds* = 0).

See Also mdlBspline_imposeBoundary, mdlBspline_extractIsoCurve, mdlBspline_extractSilhouette.

mdlBspline_extractProfile

```
#include <mdlbspln.h>
int mdlBspline_extractProfile
(
  MSBsplineCurve  *curve,  /* <= resulting curve */
  MSBsplineSurface *surface /* => surface structure */
);
```

Description mdlBspline_extractProfile creates the B-spline curve that generates a B-spline surface of projection or surface of revolution. It allocates memory for the output curve as needed. This function succeeds only when *surface->type* is BSSURF_REVOLUTION or BSSURF_TAB_CYLINDER.

curve points to the returned B-spline curve.

surface points to the B-spline surface of projection or surface of revolution.

Returns mdlBspline_extractProfile returns SUCCESS (zero) to indicate successful completion or MDLERR_INSFMEMORY if memory is insufficient for the allocations. It returns MDLERR_BADELEMENT if the surface does not have the correct type.

See Also mdlBspline_surfaceOfProjection, mdlBspline_surfaceOfRevolution.

mdlBspline_spiral

```
#include <mdlbspn.h>

int mdlBspline_spiral
(
    MSBsplineCurve    *curve,           /* <= resulting curve */
    double             initialRadius,    /* => initial radius */
    double             finalRadius,      /* => final radius */
    double             sweepAngle,       /* => sweep angle */
    Dpoint3d           *startPt,         /* => spiral starting point */
    Dpoint3d           *tangentPt,       /* => spiral tangent point */
    Dpoint3d           *directionPt      /* => spiral direction pnt */
);
```

Description mdlBspline_spiral creates a B-spline curve spiral from the supplied information. This routine allocates memory for the curve as needed.

curve points to the returned curve.

initialRadius and *finalRadius* are the spiral radii at the beginning and end. If either value is less than zero, an infinite radius is assumed.

sweepAngle is the spiral's angle in radians.

startPt is the spiral's beginning position.

tangentPt, with *startPt*, defines a line along which the spiral's initial tangent lies.

directionPt, with the previous two points, defines the spiral plane and the directions in which the spiral curves.

Returns mdlBspline_spiral returns SUCCESS (zero) to indicate successful completion or MDLERR_INSFMEMORY if memory is insufficient for the allocations.

See Also mdlBspline_helix.

mdlBspline_helix

```
#include <mdlbspln.h>

int mdlBspline_helix
(
    MSBsplineCurve    *curve,          /* <= resulting curve */
    double             initialRadius,   /* => initial radius */
    double             finalRadius,     /* => final radius */
    double             pitchValue,      /* => pitch value */
    Dpoint3d           *startPt,        /* => helix start point */
    Dpoint3d           *axis1,          /* => first axis point */
    Dpoint3d           *axis2,          /* => second axis point */
    int                valueIsHeight    /* => pitch value flag */
);
```

Description `mdlBspline_helix` creates a B-spline curve helix from the supplied information. This function allocates memory for the curve as needed.

curve points to the returned curve.

initialRadius and *finalRadius* are the helix radii at the beginning and end. If either value is less than zero, an infinite radius is assumed.

pitchValue is the helix's pitch height (*valueIsHeight* is TRUE) or number of pitches (*valueIsHeight* is FALSE). If *pitchValue* > 0, the helix will have right-handed threads. Otherwise, it will have left-handed threads.

startPt is the helix's beginning position.

axis1, with *axis2*, defines the helix axis in space.

valueIsHeight defines how *pitchValue* will be interpreted.

Returns The `mdlBspline_helix` function returns SUCCESS (zero) to indicate successful completion or MDLERR_INSFMEMORY if memory is insufficient for the allocations.

See Also `mdlBspline_spiral`.

mdlBspline_imposeBoundary

```
int mdlBspline_imposeBoundary
(
    MSBsplineSurface *surface,          /* => surface structure */
    MSBsplineCurve   *curve,           /* => boundary curve */
    double            tolerance,        /* => tolerance value */
    Dvector3d         *direction,       /* => direction or NULL */
    Dpoint3d          **surfacePoints, /* <=> points on surface, or NULL */
    int               *numPoints        /* <= # of points returned */
);
```

Description `mdlBspline_imposeBoundary` creates a boundary (or hole) in a B-spline surface pointed to by *surface*. The shape of the boundary is defined by sample points from a closed curve in space, pointed to by *curve*. The boundary consists of those points

on the surface hit by rays that originate from the sample points and are projected onto the surface in the direction defined by the vector *direction*. The number of sample points on the space curve is determined by the *tolerance* argument.

surface points to the B-spline surface to which a boundary should be added.

curve points to the B-spline curve representing the desired boundary.

tolerance is the maximum allowed curve deviation from a straight line between sample points. The smaller the value of *tolerance*, the larger the number of sample points on the curve, and the longer the function takes to return.

direction is a vector indicating the direction to project the curve onto the surface. If no direction vector is supplied (*direction*==NULL), those points on the surface nearest to the sample points on the curve are used.

If neither *surfacePoints* nor *numPoints* are NULL then the points on the surface defining the boundary are returned in the array pointed to by *surfacePoints*. This routine allocates memory for this array and returns the number of points in the array in *numPoints*.

Returns mdlBspline_imposeBoundary returns SUCCESS (zero) to indicate successful completion or MDLERR_INSFMEMORY if memory is insufficient for the allocations. It returns MDLERR_BADPARAMETER if the tolerance value is too small.

See Also mdlBspline_extractBoundary.

mdlBspline_approximateCurve

```
#include <mdlbsp1n.h>

int mdlBspline_approximateCurve
(
    MSBsplineCurve    *curve,           /* <= B-spline curve returned */
    void              *curveFuncP,      /* => curve evaluator function */
    void              *userDataP,       /* => passed through to curveFunc */
    double            tolerance,        /* => accuracy of approximation */
    int               density,          /* => point density for error calc. */
    int               continuity        /* => degree of spline continuity */
);
```

Description mdlBspline_approximateCurve is used to create a B-spline curve that is mathematically defined by a user-specified function. It allocates memory for the curve as necessary. The routine use a recursive process and calls the user-specified function to evaluate the position and differential properties of the desired curve.

curve points to the B-spline curve returned.

curveFuncP specifies a function that is called to define the curve. The value of this argument must be a valid MDL function pointer.

userDataP is a pointer that gets passed through to the function indicated by *curveFuncP*. It can point to a data structure containing information needed by the evaluator function, or it can be `NULL`.

tolerance specifies the allowable deviation of the resulting B-spline curve from the mathematically defined curve.

density specifies how many data points are used to calculate the deviation of the B-spline from the mathematically defined curve.

continuity determines the degree of continuity of the B-spline curve at the knot values. This parameter may have one of the following values (defined in `mdlbspln.h`): `POSITION_CONTINUITY`, `TANGENT_CONTINUITY` or `CURVATURE_CONTINUITY`. The returned B-spline will be second, fourth or sixth order respectively, depending on this parameter's value.

Returns `mdlB spline_approximateCurve` returns `SUCCESS` (zero) to indicate successful completion or `MDLERR_INSFMEMORY` if memory is insufficient for the allocations. It returns `MDLERR_BADCONTINUITY` if *continuity* does not have one of the values described above.

See Also `userBspline_curveFunction`, `mdlB spline_approximateSurface`.

userBspline_curveFunction

```
#include <mdlbspln.h>

int userBspline_curveFunction
(
    Dpoint3d    *position,      /* <=> position of curve at parameter */
    Dpoint3d    *dPdU,         /* <=> tangent at parameter, or NULL */
    Dpoint3d    *dPdUU,        /* <=> second derivative, or NULL */
    double u,                /* => parameter value of curve */
    void    userDataP          /* => data passed through to function */
);
```

Description The curve definition function is designated to MicroStation when the curve function name is specified in a call to `mdlB spline_approximateCurve`.

userBspline_curveFunction does not need to be the function name. The actual function name is insignificant to MicroStation.

This function is used to define the position and, optionally, the first and second derivatives of a curve at a specified parameter value.

position points to a `Dpoint3d` structure that must contain the position of the desired curve at the parameter value specified by *u*.

dPdU, if different from `NULL`, points to a `Dpoint3d` structure that must contain the first derivative of the desired curve at the parameter value specified by *u*.

dPdUU, if different from NULL, points to a `Dpoint3d` structure that must contain the second derivative of the desired curve at the parameter value specified by *u*.

u specifies the parameter value at which to evaluate the curve. It is always lies in the range $(0.0 \leq u \leq 1.0)$.

userDataP is a pointer that gets passed through from `mdlBspline_approximateCurve`. It can be used to pass extra information necessary for the evaluation of the desired curve to this function.

Returns *userBspline_curveFunction* must return SUCCESS (zero) to continue processing. If it returns anything else the curve approximation process will be aborted.

See Also `mdlBspline_approximateCurve`, `mdlBspline_approximateSurface`.

mdlBspline_approximateSurface

```
#include <mdlbsp1n.h>

int mdlBspline_approximateSurface
(
    MSBsplineSurface*surface,      /* <= B-spline surface returned */
    void      *surfaceFuncP, /* => surface evaluator func. */
    void      *userDataP,    /* => passed thru to surfaceFunc */
    double    tolerance,     /* => accuracy of approximation */
    int       density,        /* => point density for error calc. */
    int       continuity      /* => degree of spline continuity */
);
```

Description `mdlBspline_approximateSurface` is used to create a b-spline surface that is mathematically defined by a user-specified function. It allocates memory for the surface as necessary. The routine use a recursive process and calls the user-specified function to evaluate the position and differential properties of the desired surface.

surface points to the B-spline surface returned.

surfaceFuncP specifies a function that is called to define the surface. The value of this argument must be a valid MDL function pointer.

userDataP is a pointer that gets passed through to the function indicated by *surfaceFuncP*. It can point to a data structure containing information needed by the evaluator function, or it can be NULL.

tolerance specifies the allowable deviation of the resulting B-spline surface from the mathematically defined surface.

density specifies how many data points are used to calculate the deviation of the B-spline from the mathematically defined surface.

continuity determines the degree of continuity of the B-spline surface at the knot values. This parameter may have one of the following values (defined in `mdlbsp1n.h`): `POSITION_CONTINUITY`, `TANGENT_CONTINUITY`. The

returned B-spline will be second or fourth order respectively, depending on this parameter's value.

Returns mdlBspline_approximateSurface returns SUCCESS (zero) to indicate successful completion or MDLERR_INSMEMORY if memory is insufficient for the allocations. It returns MDLERR_BADCONTINUITY if *continuity* does not have one of the values described above.

See Also userBspline_surfaceFunction, mdlBspline_approximateCurve.

userBspline_surfaceFunction

```
#include <mdlbspln.h>

int userBspline_surfaceFunction
(
    Dpoint3d      *position,      /* <=> position of the surface */
    Dpoint3d      *dPdU,          /* <=> 1st partial U, or NULL */
    Dpoint3d      *dPdV,          /* <=> 1st partial V, or NULL */
    Dpoint3d      *dPdUU,         /* <=> 2nd partial U, or NULL */
    Dpoint3d      *dPdVV,         /* <=> 2nd partial V, or NULL */
    Dpoint3d      *dPdUV,         /* <=> 2nd mixed partial, or NULL */
    double u,          /* => parameter value of surface */
    double v,          /* => parameter value of surface */
    void    userDataP /* => data passed thru to function */
);
```

Description The surface definition function is designated to MicroStation when the surface function name is specified in a call to mdlBspline_approximateSurface. *userBspline_surfaceFunction* does not need to be the function name. The actual function name is insignificant to MicroStation.

This function is used to define the position and, optionally, the first and second partial derivatives of a surface at a specified u-v parameter value pair.

position points to a Dpoint3d structure that must contain the position of the desired surface at the u-v parameter value pair specified by *u* and *v*.

dPdU, if not NULL, points to a Dpoint3d structure that must contain the first partial U derivative of the desired surface at the u-v parameter value pair specified by *u* and *v*.

dPdV, if not NULL, points to a Dpoint3d structure that must contain the first partial V derivative of the desired surface at the u-v parameter value pair specified by *u* and *v*.

dPdUU, if not NULL, points to a Dpoint3d structure that must contain the second partial U derivative of the desired surface at the u-v parameter value pair specified by *u* and *v*.

dPdVV, if not NULL, points to a `Dpoint3d` structure that must contain the second partial V derivative of the desired surface at the u-v parameter value pair specified by *u* and *v*.

dPdUV, if not NULL, points to a `Dpoint3d` structure that must contain the mixed second partial derivative of the desired surface at the u-v parameter value pair specified by *u* and *v*.

u specifies the parameter value at which to evaluate the surface. It is always lies in the range $(0.0 \leq u \leq 1.0)$.

v specifies the parameter value at which to evaluate the surface. It is always lies in the range $(0.0 \leq v \leq 1.0)$.

userDataP is a pointer that gets passed through from `mdlBspline_approximateSurface`. It can be used to pass extra information necessary for the evaluation of the desired surface to this function.

Returns *userBspline_surfaceFunction* must return `SUCCESS` (zero) to continue processing. If it returns anything else the surface approximation process will be aborted.

See Also `mdlBspline_approximateSurface`, `mdlBspline_approximateCurve`.

mdlBspline_blendCurve

```
#include <mdlbsp1n.h>

int mdlBspline_blendCurve
(
    MSBsplineCurve    *output,          /* <= resulting curve */
    MSBsplineCurve    *input1,         /* => curve to blend from */
    MSBsplineCurve    *input2,         /* => curve to blend to */
    double             param1,          /* => blend curve1 from here */
    double             param2,          /* => blend curve2 to here */
    int                degree,          /* => degree of continuity of blend */
    double             magnitude1,      /* => relative magnitude of blend */
    double             magnitude2       /* => tangent between 0.0 and 1.0 */
);
```

Description `mdlBspline_blendCurve` creates a B-spline curve that smoothly transitions from a specified point on one B-spline curve to a specified point on another. The blend curve matches the two original curves with a specified degree of continuity where they meet. This routine allocates memory for the returned curve as necessary.

output points to the returned curve.

input1 points to the curve from which the blend curve originates.

input2 points to the curve on which the blend curve terminates.

param1 is the parameter of the point on *input1* at which the blend curve starts.

param2 is the parameter of the point on *input2* at which the blend curve ends.

degree is the specified degree of continuity between the blend curve and the input curves at the origin and terminal of the blend curve. It may be between zero and 14.



POSITION_CONTINUITY, TANGENT_CONTINUITY and CURVATURE_CONTINUITY are already defined in mdlbspln.h. If the degree of continuity equals zero (POSITION_CONTINUITY), the blend curve will be a chamfer between *input1* at *param1* and *input2* at *param2*.

magnitude1 is the ratio of the magnitude of *output's* tangent at 0.0 to *input1's* tangent at *param1*.

magnitude2 is the ratio of the magnitude of *output's* tangent at 1.0 to *input2's* tangent at *param2*.

Together these two arguments affect the shape of the interior portion of the blend curve.

Returns mdlBspline_blendCurve returns SUCCESS (zero) to indicate successful completion or MDLERR_INSFMEMORY if memory is insufficient for the allocations. It returns MDLERR_BADORDER if *degree* does not have one of the values described above.

See Also mdlBspline_blendSurface, mdlBspline_blendRails.

mdlBspline_blendSurface

```
#include <mdlbspln.h>

int mdlBspline_blendSurface
(
    MSBsplineSurface *output,           /* <= resulting surface */
    MSBsplineSurface *input1,          /* => surface to blend from */
    MSBsplineSurface *input2,          /* => surface to blend to */
    double param1,                     /* => blend surface1 from here */
    double param2,                     /* => blend surface2 to here */
    int degree,                         /* => degree of continuity of blend */
    double magnitude1,                 /* => relative magnitude of blend */
    double magnitude2,                 /* => tangent between 0.0 and 1.0 */
    int edge                            /* => direction flag */
);
```

Description mdlBspline_blendSurface creates a B-spline surface that smoothly transitions from a specified iso-parametric curve on one B-spline surface to a specified iso-parametric curve on another. The blend surface matches the two original surfaces with a specified degree of continuity where they meet. This routine allocates memory for the returned surface as necessary.

output points to the returned surface.

input1 points to the surface from which the blend surface originates.

input2 points to the surface on which the blend surface terminates.

param1 is the parameter of the iso-parametric curve on *input1* at which the blend surface starts.

param2 is the parameter of the iso-parametric curve on *input2* at which the blend surface ends.

degree is the specified degree of continuity between the blend surface and the input surfaces at the origin and terminal of the blend surface. It may be between zero and 14.



POSITION_CONTINUITY, TANGENT_CONTINUITY and CURVATURE_CONTINUITY are already defined in mdlbspln.h. If the degree of continuity equals zero (POSITION_CONTINUITY), the blend surface will be a chamfer between *input1* at *param1* and *input2* at *param2*; i.e., a ruled surface.

magnitude1 is the ratio of the magnitude of *output*'s tangent at 0.0 to *input1*'s tangent at *param1*. *magnitude2* is the ratio of the magnitude of *output*'s tangent at 1.0 to *input2*'s tangent at *param2*. Together these two arguments affect the shape of the interior portion of the blend surface.

edge contains a flag indicating across which edge the input surfaces should be blended. The values BSSURF_U and BSSURF_V are defined in the file mdlbspln.h. If *edge* equals BSSURF_U, then *param1* and *param2* are the v-parameters of the iso-parametric curves of *input1* and *input2* that define the start and end of the blend surface.

Returns mdlBspline_blendSurface returns SUCCESS (zero) to indicate successful completion or MDLERR_INSFMEMORY if memory is insufficient for the allocations. It returns MDLERR_BADORDER if *degree* does not have one of the values described above.

See Also mdlBspline_blendCurve, mdlBspline_blendRails.

mdlBspline_blendRails

```
#include <mdlbspln.h>

int mdlBspline_blendRails
(
    MSBsplineSurface *outPut,          /* <= resulting surface */
    MSBsplineSurface *surface1,        /* => first surface */
    MSBsplineSurface *surface2,        /* => second surface */
    MSBsplineCurve *curve1,           /* => first rail curve */
    MSBsplineCurve *curve2,           /* => second rail curve */
    double tolerance,                 /* => tolerance value */
    boolean continuity                 /* => smoothness */
);
```

Description `mdlBspline_blendRails` creates a B-spline blending surface between two B-spline surfaces along their rail curves. A rail curve of a B-spline surface is any curve lying on the surface. If the continuity parameter is `FALSE`, a ruled surface is created. If it is `TRUE`, the resulting surface will blend them with tangent plane continuity. This routine allocates memory for the output surface as necessary.

output points to the B-spline surface being returned.

surface1 and *surface2* point to the first and second input surfaces, respectively.

curve1 and *curve2* point to the first and second rail curves, respectively.

continuity specifies the smoothness of the blending.

Returns `mdlBspline_blendRails` returns `SUCCESS` to indicate successful completion. It returns `MDLERR_INSFMEMORY` if there is not enough memory for allocation.

See Also `mdlBspline_blendCurve`, `mdlBspline_blendSurface`.

mdlBspline_constantRadiusFillet

```
int mdlBspline_constantRadiusFillet
(
    SurfaceChain    **chainPP,      /* <= resulting surface chain */
    MSBsplineSurface *surf0,        /* <=> first surface */
    MSBsplineSurface *surf1,        /* <=> second surface */
    double          radius0,        /* => initial radius */
    double          radius1,        /* => final radius */
    int             trim0,          /* => whether to trim first surf. */
    int             trim1,          /* => whether to trim second surf. */
    double          tolerance       /* => tolerance value */
);
```

Description `mdlBspline_constantRadiusFillet` creates a B-spline fillet (B-spline surface) between two B-spline surfaces. The result can be described as rolling a constant radius ball between two surfaces. One or both original surfaces can be trimmed or non-trimmed. This function allocates the memory for output surface chain as necessary.

chainPP points to the address of the surface chain.

surf0 points to the first surface and *surf1* points to the second surface.

radius0 is the initial radius and *radius1* is the same as *radius0*.

If *trim0* is `TRUE`, the first surface is trimmed along the edge of fillet. If *trim1* is `TRUE`, the second surface is trimmed along the edge of the fillet.

tolerance defines the number of points sampled to create the fillet.

Returns `mdlBspline_constantRadiusFillet` returns `SUCCESS` to indicate successful completion.

See Also `mdlBspline_variableRadiusFillet`.

mdlBspline_variableRadiusFillet

```
int mdlBspline_variableRadiusFillet
(
  SurfaceChain    **chainPP,      /* <= resulting surface chain */
  MSBsplineSurface *surf0,        /* <=> first surface */
  MSBsplineSurface *surf1,        /* <=> second surface */
  double          radius0,        /* => initial radius */
  double          radius1,        /* => final radius */
  int             trim0,          /* => whether to trim first surf. */
  int             trim1,          /* => whether to trim second surf. */
  double          tolerance,      /* => tolerance value */
  int             hermite         /* => TRUE=hermite varying radius */
);
```

Description mdlBspline_variableRadiusFillet creates a B-spline fillet (B-spline surface) between two B-spline surfaces. The result can be described as rolling a varying radius ball between two surfaces. One or both original surfaces can be trimmed or non-trimmed. This function allocates the memory for output surface chain as necessary.

chainPP points to the address of the surface chain.

surf0 points to the first surface and *surf1* points to the second surface.

radius0 is the initial radius and *radius1* is the final radius.

If *trim0* is TRUE, the first surface is trimmed along the edge of fillet. If *trim1* is TRUE, the second surface is trimmed along the edge of the fillet.

tolerance defines the number of points sampled to create the fillet.

If *hermite* is TRUE, the function will create a Hermite varying radius fillet. If FALSE, a normal linear fillet will be created.



Non-FALSE values for the hermite parameter are not currently supported. This feature will be implemented in future versions of MicroStation.

Returns mdlBspline_variableRadiusFillet returns SUCCESS to indicate successful completion.

See Also mdlBspline_constantRadiusFillet.

mdlBspline_convertToPlanarSurface [mdl.lib.ml]

```
#include <mdlbspn.h>
#include <mselems.h>

int mdlBspline_convertToPlanarSurface
(
    MSBsplineSurface *surface,      /* <= returned surface */
    MSElementDescr *edP,          /* => closed, planar, element descr. */
    double tolerance                /* => for boundary calculations */
);
```

Description mdlBspline_convertToPlanarSurface creates a planar B-spline surface from a planar, closed element descriptor. It allocates memory for the surface as necessary.

surface points to the B-spline surface returned.

edP points to the MSElementDescr structure that is used to generate the surface. This must contain closed, planar geometry; i.e., CMLX_SHAPE_ELM, SHAPE_ELM, ELLIPSE_ELM, or a group hole.

tolerance contains the value that is used in calculating trim boundaries. Smaller values yield more accurate boundaries, but at the cost of more points in them.

Returns mdlBspline_convertToPlanarSurface returns SUCCESS (zero) to indicate successful completion or MDLERR_INSFMEMORY if memory is insufficient for the allocations. It returns MDLERR_NONCLOSEDELM if the element descriptor does not contain closed geometry.

See Also mdlBspline_trimmedPlaneFromCurves [mdl.lib.ml],
mdlBspline_extractCapAsSurface.

mdlBspline_trimmedPlaneFromCurves [mdl.lib.ml]

```
#include <mdlbspn.h>

int mdlBspline_trimmedPlaneFromCurves
(
    MSBsplineSurface *surface,      /* <= resulting surface */
    MSBsplineCurve *curves,        /* => generating curves */
    int numCurves,                /* => number of curves */
    double tolerance                /* => for boundary calcs */
);
```

Description mdlBspline_convertToPlanarSurface creates a planar B-spline surface from a collection of closed, planar B-spline curves. It allocates memory for the surface as necessary.

surface points to the B-spline surface returned.

curves is an array of MSBsplineCurve structures that contain the generating geometry.

numCurves contains the number of entries in the *curves* array.

tolerance contains the value that is used in calculating trim boundaries. Smaller values yield more accurate boundaries, but at the cost of more points in them.

Returns mdlBspline_trimmedPlaneFromCurves returns SUCCESS (zero) to indicate successful completion or MDLERR_INSFMEMORY if memory is insufficient for the allocations.

See Also mdlBspline_convertToPlanarSurface [mdl1lib.m1],
mdlBspline_extractCapAsSurface.

mdlBspline_getCurveFromSurface

```
int mdlBspline_getCurveFromSurface
(
    MSBsplineCurve    *curve,          /* <= resulting curve */
    MSBsplineSurface  *surface,        /* => input surface */
    int                direction,       /* => U or V direction */
    int                which            /* => -1 mean last row/column */
);
```

Description mdlBspline_getCurveFromSurface creates a B-spline curve from the poles in any row or column of the control net. mdlBspline_getCurveFromSurface allocates the memory as necessary.

curve points to the resulting curve.

surface points to the input surface element.

direction is either BSSURF_U or BSSURF_V to represent whether the curve will be extracted from the U direction (row) or V direction (column).

which is the number of row or column to use to create the curve. A value of -1 means use the last row or column.

Returns mdlBspline_getCurveFromSurface returns SUCCESS to indicate successful completion. Otherwise, it returns MDLERR_INSFMEMORY if there is not enough memory for allocation.

See Also mdlBspline_extractIsoCurve, mdlBspline_getEdgeFromSurface.

mdlBspline_getEdgeFromSurface

```
int mdlBspline_getEdgeFromSurface
(
    MSBsplineCurve    *curve,          /* <= curve from edge of surface */
    MSBsplineSurface  *surf,          /* => input surface */
    int                edge            /* => edge code to extract */
);
```

Description `mdlBspline_getEdgeFromSurface` creates a B-spline curve from one of the four edges of a B-spline surface.

curve is the resulting curve.

surf is the input surface.

edge indicates from which edge to extract the curve. It can be `U0_EDGE`, `U1_EDGE`, `V0_EDGE` or `V1_EDGE` to represent different edges.

Returns `mdlBspline_getEdgeFromSurface` returns `SUCCESS` to indicate successful completion.

See Also `mdlBspline_getCurveFromSurface`, `mdlBspline_extractIsoCurve`.

mdlBspline_offset

```
#include <mdlbspln.h>

int mdlBspline_offset
(
    MSBsplineCurve  *output,          /* <= resulting curve */
    MSBsplineCurve  *input,           /* => original curve */
    double           distance,         /* => distance to offset */
    int              cuspTreatment,     /* => how to handle cusps */
    int              continuity,       /* => geometric continuity desired */
    RotMatrix        *rotMatrix,       /* => view to calc. offset in */
    double           tolerance         /* => used in calculation */
);
```

Description `mdlBspline_offset` calculates a B-spline curve that is the offset of a B-spline curve by a specified distance. Since an exact offset is not possible this routines creates an approximation to within a desired accuracy. This routine allocates memory for the output curve as necessary.

output points to the returned curve. It can point to the same structure as *input*.

input points to the B-spline curve that will be offset.

distance is the amount by which to offset the input curve.

cuspTreatment determines what happens at any corners of the input curve. Allowed values are defined in `mdlbspln.h` and yield the following results:

Value	Result
<code>OFFSET_JUMP_CUSP</code>	discontinuity in offset at cusp.
<code>OFFSET_CHAMFER_CUSP</code>	line segment in offset at cusp.
<code>OFFSET_POINT_CUSP</code>	offset extended to point intersection at cusp.

Value	Result
OFFSET_PARABOLA_CUSP	parabolic arc in offset at cusp.
OFFSET_ARC_CUSP	circular arc in offset at cusp.

continuity determines the degree of continuity of the offset curve at knot values. The allowed values, TANGENT_CONTINUITY and CURVATURE_CONTINUITY, are defined in mdlbspln.h. The returned B-spline will be fourth or sixth order respectively, depending on this parameter's value.

rotMatrix points to a view rotation matrix used to rotate and flatten the input curve if a planar offset is desired. If *rotMatrix* == NULL, the input curve will be offset by its normal vector in 3D space.

tolerance specifies the accuracy of the approximation process used to calculate the offset curve. Smaller values yield more accurate results but at the cost of greater number of poles, knots, etc. in the output curve.

Returns mdlBspline_offset returns SUCCESS (zero) to indicate successful completion. It returns MDLERR_INSFMEMORY if there is not enough memory for the allocations.

See Also mdlBspline_offsetSurface.

mdlBspline_offsetSurface

```
#include <mdlbspln.h>

int mdlBspline_offsetSurface
(
    MSBsplineSurface *output,      /* <= resulting surface */
    MSBsplineSurface *input,       /* => original surface */
    double            distance,     /* => offset distance */
    double            tolerance,    /* => tolerance value */
);
```

Description mdlBspline_offsetSurface creates a B-spline surface that is the offset of a B-spline surface by a specified distance in the surface normal direction. Since an exact offset is not possible in some cases, this routine creates an approximation to within a desired accuracy. This routine allocates memory for the output surface as necessary.

output points to the surface being returned.

input points to the original surface.

distance is the amount by which to offset the input surface.

tolerance represents the maximum allowed deviation from the original surface.

Returns mdlBspline_offsetSurface returns SUCCESS to indicate successful completion. It returns MDLERR_INSFMEMORY, if there is not enough memory for allocation.

See Also mdlBspline_offset.

mdlBspline_catmullRomCurve

```
#include <mdlbspln.h>

int mdlBspline_catmullRomCurve
(
    MSBsplineCurve *curve,          /* <= resulting curve */
    Dpoint3d *points,              /* => data points */
    double *uValues,              /* => parameter values or NULL */
    int numPoints                  /* => number of points */
);
```

Description mdlBspline_catmullRomCurve creates a B-spline curve that interpolates the given data points. The resulting curve is always of order four and open. If *uValues* is not NULL, the parameter values in the array will be used. Otherwise, the parameter values used will be the accumulations of distances between the data points. This routine allocates memory for the returned curve as necessary.

curve points to the B-spline curve being returned.

points is the array containing the data points.

If *uValues* != NULL, this array of parameter values will be used in the routine.

numPoints is the number of data points supplied, and must be greater than one.

Returns mdlBspline_catmullRomCurve returns SUCCESS to indicate successful completion. It returns MDLERR_INSFMEMORY, if there is not enough memory for allocation. It returns MDLERR_BADPOLES, if the function is not called with the correct *numPoints* information.

See Also mdlBspline_catmullRomSurface, mdlBspline_leastSquaresToCurve.

mdlBspline_catmullRomSurface

```
#include <mdlbspln.h>

int mdlBspline_catmullRomSurface
(
    MSBsplineSurface *surface,      /* <= resulting surface */
    Dpoint3d *points,              /* => data points */
    int uNumPoints,                /* => number of points per row */
    int vNumPoints,                /* => number of rows of points */
    Dpoint2d *uvValues             /* => parameter values or NULL */
);
```

Description mdlBspline_catmullRomSurface creates a B-spline surface which interpolates the given data points. The resulting surface is always of order four and open. If

uvValues is not `NULL`, the parameter values in the array will be used. Otherwise, the parameter values used will be the accumulations of distances between the data points. This routine allocates memory for the returned surface as necessary.

surface points to the B-spline surface being returned.

points is the array containing the data points.

uNumPoints is the number of data points in each row, and *vNumPoints* is the number of rows.

If *uvValues* != `NULL`, this array of parameter values will be used in the routine.

Returns `mdlBspline_catmullRomSurface` returns `SUCCESS` to indicate successful completion. It returns `MDLERR_INSFMEMORY`, if there is not enough memory for allocation. It returns `MDLERR_BADPOLES`, if the function is not called with the correct *uNumPoints* or *vNumPoints* information.

See Also `mdlBspline_catmullRomCurve`, `mdlBspline_leastSquaresToSurface`.

mdlBspline_gordonSurface

```
#include <mdlbsp1n.h>

int mdlBspline_gordonSurface
(
    MSBsplineSurface *surface,      /* <= resulting surface */
    MSBsplineCurve   *uCurves,    /* => curves in U */
    MSBsplineCurve   *vCurves,    /* => curves in V */
    int               numU,         /* => number of curves in U */
    int               numV,         /* => number of curves in V */
);
```

Description `mdlBspline_gordonSurface` creates a B-spline surface which interpolates a network of curves. This is a generalization of Coon's patch which interpolates four curves, two in each direction. Each curve in U should intersect all curves in V, and vice versa. The routine allocates memory for the output surface as necessary.

surface points to the B-spline surface being returned.

uCurves and *vCurves* point to the array of curves in U and V, respectively.

numU and *numV* are the number of curves in U and V, respectively.

Returns `mdlBspline_gordonSurface` returns `SUCCESS` to indicate successful completion. It returns `MDLERR_INSFMEMORY` if there is not enough memory for allocation.

See Also `mdlBspline_coonsPatch`, `mdlBspline_crossSectionSurface`.

mdlBspline_crossSectionSurface

```
#include <mdlbspIn.h>

int mdlBspline_crossSectionSurface
(
    MSBsplineSurface *surface,      /* <= resulting surface */
    MSBsplineCurve   *curves,      /* => section curves */
    int               numCurves    /* => number of curves */
);
```

Description mdlBspline_crossSectionSurface creates a B-spline surface from a set of section curves. The surface is of order four and open in V direction. The parameters in U direction depends on the parameters of the section curves. This routine allocates memory for the output surface as necessary.

surface points to the surface being returned.

curves points to the array of section curves.

numCurves is the number of section curves.

Returns mdlBspline_crossSectionSurface returns SUCCESS to indicate successful completion. It returns MDLERR_INSFMEMORY, if there is not enough memory for allocation.

See Also mdlBspline_gordonSurface.

mdlBspline_cubicInterpolation

```
#include <mdlbspIn.h>

int mdlBspline_cubicInterpolation
(
    MSBsplineCurve   *curve,      /* <= resulting curve */
    Dpoint3d         *points,     /* => data points */
    double            *uValues,   /* => parameter values or NULL */
    int               numPoints,  /* => number of points */
    boolean           remvData,   /* => TRUE=remove repeated points */
    double            tolerance   /* => tolerance if remvData = TRUE */
);
```

Description mdlBspline_cubicInterpolation creates a B-spline curve that interpolates the given data points. The resulting curve is always of order four and open. If *uValues* is not NULL, the parameter values in the array will be used. Otherwise, the parameter values used will be the accumulations of distances between the data points. The number of poles of the resulting curve is two more than the number of input points. The fitting curve is the smoothest interpolation possible compared to other fitting curve calculation methods. This routine allocates memory for the returned curve as necessary.

curve points to the B-spline curve being returned.

points is the array containing the data points.

If *uValues* != NULL, this array of parameter values will be used in the routine.

numPoints is the number of data points supplied. This number must be greater than one.

If *remvData* is TRUE, the repeated data points will be removed before the fitting curve in interpolated.

tolerance is the tolerance to remove the repeated data points.

Returns mdlBspline_cubicInterpolation returns SUCCESS to indicate successful completion. It returns MDLERR_INSFMEMORY, if there is not enough memory for allocation. It returns MDLERR_BADPOLES, if the function is not called with the correct *numPoints* information.

See Also mdlBspline_catmullRomCurve, mdlBspline_leastSquaresToCurve.

mdlBspline_nSidedPatch

```
#include <mdlbspn.h>
#include <mselems.h>

int mdlBspline_nSidedPatch
(
    ElementDescr    **surface,      /* <= surface element created */
    MSBsplineCurve  *curves,        /* => edge curves */
    int             numCurves,      /* => number of edge curves */
    Dpoint3d        *derivatives    /* => cross boundary derivatives or NULL */
);
```

Description mdlBspline_nSidedPatch creates an element descriptor containing B-spline surfaces created from edge curves. If *numCurves* is two, the output surface is a ruled surface. If *numCurves* is four, the output surface is a Coon's patch. If the curves are the boundary curves of other B-spline surfaces, the derivatives can be specified to achieve the cross boundary tangent plane continuities. For each edge, or *curves[i]*, *derivatives* contains *curves[i]->params.numPoles* points. If *derivatives* is NULL, the routine will use the cross boundary derivatives generated from the averages of the tangents of other edges of the surface. It allocates memory as necessary.

curves points to the array of edges curves.

numCurves is the number of edges curves.

derivatives points to the cross boundary derivatives or NULL.

Returns mdlBspline_nSidedPatch returns SUCCESS to indicate successful completion. It returns MDLERR_INSFMEMORY, if there is not enough memory for allocation.

See Also mdlBspline_coonsPatch, mdlBspline_ruledSurface, mdlBspline_gordonSurface.

mdlBspline_createCurvesFromChain

```
int mdlBspline_createCurvesFromChain
(
  MSElementDescr  **edPP,    /* <= curve elements created */
  MSElement       *in,       /* => template element, or NULL */
  CurveChain       *chainP    /* => curve chain */
);
```

Description `mdlBspline_createCurvesFromChain` creates an element descriptor containing B-spline curves from a curve chain. If *in* is `NULL`, the display parameters for the created curves are taken from the active MicroStation settings; otherwise the display parameters from *in* are used.

edPP points to the address of the output element descriptor.

in points to the template element.

chainP points to the B-spline curve chain.

Returns `mdlBspline_createCurvesFromChain` returns `SUCCESS` to indicate successful completion. Otherwise, it returns one of the following error codes:

```
MDLERR_NOPOLES      -500
MDLERR_NOKNOTS      -501
MDLERR_NOWEIGHTS    -502
MDLERR_TOOFEWPoles  -506
MDLERR_TOOMANYPOLES -507
```

See Also `mdlBspline_createSurfacesFromChain`.

mdlBspline_createSurfacesFromChain

```
int mdlBspline_createSurfacesFromChain
(
  MSElementDescr  **edPP,    /* <= surface elements */
  MSElement       *in,       /* => template element */
  SurfaceChain      *chainP    /* => surface chain */
);
```

Description `mdlBspline_createSurfacesFromChain` creates an element descriptor containing B-spline surfaces from a surface chain. If *in* is `NULL`, the display parameters for the created curves are taken from the active MicroStation settings; otherwise the display parameters from *in* are used.

edPP points to the address of the output element descriptor.

in points to the template element.

chainP points to the B-spline surface chain.

Returns mdlBspline_createSurfacesFromChain returns SUCCESS to indicate successful completion. Otherwise, it returns one of the following codes:

```
MDLERR_NOPOLES      -500
MDLERR_NOKNOTS      -501
MDLERR_NOWEIGHTS    -502
MDLERR_TOOFEWPoles  -506
MDLERR_TOOMANYPoles -507
```

See Also mdlBspline_createSurfacesFromChain.

mdlBspline_convertToCurveChain

```
int mdlBspline_convertToCurveChain
(
CurveChain **chainPP, /* <= linked list of Curves (userDataP=edP) */
MSElementDescr *edP   /* => element descriptor (chain) */
);
```

Description mdlBspline_convertToCurveChain creates a curve chain from an element descriptor. It allocates the memory as necessary.

chainPP points to the address of the curve chain.

edP points to the element descriptor.

Returns mdlBspline_convertToCurveChain returns SUCCESS to indicate successful completion.

See Also mdlBspline_createCurvesFromChain.

mdlBspline_convertToSurfaceChain

```
int mdlBspline_convertToSurfaceChain
(
SurfaceChain **chainPP, /* <= linkedList of surfs (userDataP=edP) */
MSElementDescr *edP,   /* => element descriptor (chain) */
double tolerance        /* => for trim boundaries */
boolean outwardSolidNorms /* => TRUE=orient solid normals outward */
);
```

Description mdlBspline_convertToSurfaceChain creates a surface chain from an element descriptor. The element descriptor can contain closed B-spline curve, group hole, shape, complex shape, ellipse and all other surface types of elements. It allocates the memory as necessary.

chainPP points to the address of the resulting surface chain.

edP points to the input element descriptor.

If there is a trim boundary, *tolerance* specifies the accuracy of the approximation process used to create the boundary.

outwardSolidNorms determines if the solid normals will be oriented outward (TRUE) or inward (FALSE).

Returns mdlBspline_convertToSurfaceChain returns SUCCESS to indicate successful completion.

See Also mdlBspline_convertToCurve.

mdlBspline_intersectSurfaceChains

```
int mdlBspline_intersectSurfaceChains
(
  PointList    **pointLists,      /* <= address of point list */
  int          *numPointLists,    /* => length of point list */
  SurfaceChain *surf0,           /* => first surface chain */
  SurfaceChain *surf1,           /* => second surface chain */
  double        tolerance,        /* => accuracy */
  int          displayFlag,       /* => TRUE=display pnts as they are calculated */
  int          wantSurf0Bounds,    /* => TRUE=use 1st surf bounds */
  int          wantSurf1Bounds    /* => TRUE=use 2nd surf bounds */
);
```

Description mdlBspline_intersectSurfaceChains creates a list of points which represent the intersection points between each surface in the first surface chain with every surface in the second surface chain. *pointLists* points to the address of the intersection point list array. Each element in this array contains array of intersection points. This function will allocate memory as necessary.

pointLists points to the address of the returned point list.

numPointLists is length of the point list.

surf0 points to the first surface chain, and *surf1* points to the second surface chain.

tolerance specifies the accuracy of the approximation process used to calculate the intersection curve(s). Smaller values yield more accurate results but at the cost of greater number of points in the output arrays.

If *displayFlag* is TRUE, the function will display each intersection point as it calculates them.

If *wantSurf0Bounds* is TRUE, the intersection will take the boundaries of the first surface chain into account.

If *wantSurf1Bounds* is TRUE, the intersection will take the boundaries of the second surface chain into account.

Returns mdlBspline_intersectSurfaceChains returns SUCCESS to indicate the success completion of the function.

See Also mdlBspline_intersectSurfaces.

mdlBspline_arcLengthFromParameters

```
#include <mdlbspln.h>
#include <msbsplin.fdf>

int mdlBspline_arcLengthFromParameters
(
    double *arcLengthP,          /* <= arc length returned */
    double *errorAchievedP,      /* <= absolute error achieved */
    double startParam,           /* => starting parameter */
    double endParam,             /* => ending parameter */
    MSBsplineCurve *curveP,      /* => given curve structure */
    double relativeTol           /* => req'd rel. tolerance for */
);
```

Description The `mdlBspline_arcLengthFromParameters` function computes the distance between two points along a given curve. The starting point and ending point are defined by the two given parameters. The required accuracy should be given in relative tolerance.

The *arcLengthP* parameter points to the returned arc length.

The *errorAchievedP* parameter is the absolute error achieved, i.e.
`abs(arcLength-trueArcLength)`.

The *startParam* parameter is the parameter value for the starting point.

The *endParam* parameter is the parameter for the ending point.

The *curveP* parameter points to the given B-spline curve.

The *relativeTol* parameter is the relative tolerance used when compute the arc length by integration using Simpson's Rule,
i.e., `abs(errorAcheived)/arcLengthP<=relativeTol`.



This function was implemented in MicroStation 95.

Returns `mdlBspline_arcLengthFromParameters` returns `SUCCESS` to indicate successful completion.

See Also `mdlBspline_parameterFromArcLength`.

mdlBspline_computeEqualChordByLength

```
#include <mdlbspln.h>
#include <msbsplin.fdf>

int mdlBspline_computeEqualChordByLength
(
    Dpoint3d    **pointsPP,      /* <= points on curve, or NULL */
    double      **paramsPP,      /* <= parameters for points */
    int          *numChords,      /* <= number of equal-length chords */
    double       chordLength,     /* => chord length */
    MSBsplineCurve *curveP       /* => B-spline curve */
);
```

Description The `mdlBspline_computeEqualChordByLength` function will return an array of points on a curve together with their parameter values so that the distance between each successive points is the given *chordLength* with the exception of the last two points that the distance may be smaller than *chordLength*.

pointsPP points to the array of points on curve.

paramsPP points to the parameter values of the point array.

numChords is the number of chords created.

chordLength is the given chord length value.

curveP points to the input curve structure.



This function was implemented in MicroStation 95.

Returns `mdlBspline_computeEqualChordByLength` returns `SUCCESS` to indicate successful completion.

See Also `mdlBspline_parameterFromArcLength`.

mdlBspline_cubicInterpolationExt

```
#include <mdlbspln.h>
#include <msbsplin.fdf>

int mdlBspline_cubicInterpolationExt
(
    MSBsplineCurve *curve,        /* <= open cubic spline curve */
    Dpoint3d       *inPts,        /* => points to be interpolated */
    double          *inParams,     /* => u parameters or NULL */
    int             numPts,        /* => number of points */
    boolean         remvData,      /* => TRUE remove coincide points */
    double          remvTol,       /* => only if remvData is TRUE */
    Dpoint3d       *endTangents,   /* => end tangents or NULL */
    int             closedCurve    /* => if TRUE, closed B-spline is created */
);
```


Description The `mdlBspline_cubicInterpolationExt` function creates a B-spline curve which interpolates the given set of data points. The resulting curve is always of order four and non-rational. If *inParams* is not `NULL`, it will be used as B-spline knot vector. Otherwise, the knot vector is calculated based on the accumulations of distances between the data points. The number of poles is two more than the number of input points. The fitting has the property that it is the smoothest interpolation among similar fitting curves. If *remvData* is set to `TRUE`, the coincident data points will be removed before interpolation takes place. The curve tangent vectors at the two end points can be specified. If *endTangents* is set to `NULL`, the routine will use the so called Bessel end condition to determine the tangents at both ends. If the *closedCurve* is set to `TRUE`, a closed B-spline curve will be created. This routine allocates memory for the returned curve as necessary.

curve points to the curve structure returned.

inPts points to the array of input points.

inParams, if not `NULL`, points to the input knot vector.

numPts points to the size of the points array.

remvData is `TRUE` if the coincident points are to be removed before interpolation.

remvTol is the tolerance when determining the coincident points.

endTangents, if not `NULL`, points the given tangents at the two end points.

closedCurve is `TRUE` is a closed B-spline curve is expected.



This function was implemented in MicroStation 95.

Returns `mdlBspline_cubicInterpolationExt` returns `SUCCESS` to indicate successful completion.

See Also `mdlBspline_cubicInterpolation`.

mdlBspline_parameterFromArcLength

```
#include <mdlbsp1n.h>
#include <msbsplin.fdf>

int mdlBspline_parameterFromArcLength
(
    Dpoint3d    *pointP,          /* <= point of paramP */
    double      *paramP,          /* <= parameter to match the arc length */
    double      *errorAchievedP, /* <= the absolute error achieved */
    double      arcLength,        /* => from start parameter to paramP */
    double      startParam,       /* => start parameter */
    MSBsplineCurve *curveP,       /* => input curve */
    double      relativeTol       /* => relative tolerance for integration */
);
```

Description The `mdlBspline_parameterFromArcLength` function computes the parameter value from the starting parameter with the given arc length.

pointP points to the point on the curve which has the returned parameter.

paramP points to the parameter value returned.

errorAchievedP is the absolute error achieved.

arcLength is the given arc length.

startParam the parameter value for the starting point.

curveP points to the given curve structure.

relativeTol is the required relative tolerance.



This function was implemented in MicroStation 95.

Returns `mdlBspline_parameterFromArcLength` returns `SUCCESS` to indicate successful completion.

See Also `mdlBspline_arcLengthFromParameters`.

B-spline Modification Functions

The following table lists all B-spline functions:

Function	Used to
<code>mdlBspline_extractCurve</code>	return information from a B-spline curve element stored in an element descriptor.
<code>mdlBspline_extractSurface</code>	return information about a B-spline surface element stored in an element descriptor.
<code>mdlBspline_allocateCurve</code>	allocate memory for a B-spline curve's poles, weights and knot vector.
<code>mdlBspline_allocateSurface</code>	allocate memory for a B-spline surface's poles, weights and knot vectors.
<code>mdlBspline_freeCurve</code>	free memory allocated for a B-spline curve's poles, weights and knot vector.
<code>mdlBspline_freeSurface</code>	free memory allocated for a B-spline surface's poles, weights and knot vector.
<code>mdlBspline_freeCurveChain</code>	free memory allocated for each curve in the curve chain.
<code>mdlBspline_freeSurfaceChain</code>	free the memory allocated for each surface in the surface chain.

Function	Used to
<code>mdlBspline_addCurveLink</code>	add a curve element to the existing curve chain.
<code>mdlBspline_addSurfaceLink</code>	add a surface element to the existing surface chain.
<code>mdlBspline_weightPoles</code>	calculate the homogeneous coordinates of a B-spline's poles from their respective weights.
<code>mdlBspline_unWeightPoles</code>	calculate the real-world coordinates of a B-spline's poles from their respective weights.
<code>mdlBspline_openCurve</code>	return an equivalent open B-spline curve from a closed B-spline curve.
<code>mdlBspline_closeCurve</code>	return a closed curve that is identical to a given open curve that starts and stops at the same point.
<code>mdlBspline_elevateDegree</code>	elevate the degree (increases the order) of a B-spline curve.
<code>mdlBspline_openSurface</code>	do to surfaces what <code>mdlBspline_openCurve</code> does to curves.
<code>mdlBspline_closeSurface</code>	do to surfaces what <code>mdlBspline_closeCurve</code> does to curves.
<code>mdlBspline_elevateDegreeSurface</code>	do to surfaces what <code>mdlBspline_elevateDegree</code> does to curves.
<code>mdlBspline_makeBezier</code>	return an equivalent Bézier curve from a B-spline curve.
<code>mdlBspline_makeBezierSurface</code>	do to surfaces what <code>mdlBspline_makeBezier</code> does to curves.
<code>mdlBspline_makeRational</code>	return an equivalent rational B-spline curve from a non-rational B-spline curve.
<code>mdlBspline_makeRationalSurface</code>	do to surfaces what <code>mdlBspline_makeRational</code> does to curves.
<code>mdlBspline_reverseCurve</code>	change the direction that a B-spline curve traces as its parameter ranges from zero to one.
<code>mdlBspline_reverseSurface</code>	do to surfaces what <code>mdlBspline_reverseCurve</code> does to curves.
<code>mdlBspline_make2CurvesCompatible</code>	return two B-spline curves with the same order, number of poles and knot vectors.
<code>mdlBspline_makeCurvesCompatible</code>	return equivalent compatible B-spline curves given a list of B-spline curves.

Function	Used to
<code>mdlBspline_make2SurfacesCompatible</code>	return two B-spline surfaces with the same order, number of poles, and knot vectors in U or V.
<code>mdlBspline_swapUV</code>	exchange the parametric direction of a surface.
<code>mdlBspline_segmentDisjointCurve</code>	break a discontinuous B-spline curve into separate continuous pieces.
<code>mdlBspline_segmentDisjointSurface</code>	break a discontinuous B-spline surface into separate continuous pieces.
<code>mdlBspline_curveDataReduction</code>	create a B-spline curve element with fewer poles than that of the original curve.
<code>mdlBspline_copyBoundaries</code>	copy the the boundary elements from an input surface onto an output surface.

Example

See `bspline.mc`.

`mdlBspline_extractCurve`

```
#include <mdlbspln.h>
#include <mselems.h>

int mdlBspline_extractCurve
(
  MSElementUnion *header,      /* <= curve header, or NULL */
  int             *type,        /* <= curve type flag, or NULL */
  int             *rational,    /* <= rational flag, or NULL */
  BsplineDisplay *display,      /* <= display flags, or NULL */
  BsplineParam   *params,       /* <= curve parameters, or NULL */
  Dpoint3d       **poles,       /* <= poles of curve (unweighted) */
  double         **knots,       /* <= full knot vector of curve */
  double         **weights,     /* <= curve weights (if rational) */
  MSElementDescr *elmDescr     /* => elements defining curve */
);
```

Description The `mdlBspline_extractCurve` function returns information from a B-spline curve element stored in an element descriptor. It allocates memory for the poles, knots, and weights as necessary. The poles are returned unweighted.

The `mdlBspline_extractCurve` function returns `SUCCESS` (zero) to indicate successful completion. Otherwise, it returns one of the following error codes:

`MDLERR_BADBSPELM` -508

MDLERR_NOBSPHEADER -505
MDLERR_INSMEMORY -116

See Also mdlBspline_extractSurface.

mdlBspline_extractSurface

```
#include <mdlbspIn.h>
#include <mselems.h>

int mdlBspline_extractSurface
(
MSElementUnion *header,      /* <= surface header, or NULL */
int *type,                   /* <= surface type flag, or NULL */
int *rational,                /* <= rational flag, or NULL */
BsplineDisplay *display,     /* <= display flags, or NULL */
BsplineParam *uParams,       /* <= surface data in U, or NULL */
BsplineParam *vParams,       /* <= surface data in V, or NULL */
Dpoint3d **poles,            /* <= poles (unweighted) or NULL */
double **uKnots,             /* <= full knot vector in U, or NULL */
double **vKnots,             /* <= full knot vector in V, or NULL */
double **weights,            /* <= weights (if rational) or NULL */
int *holeOrigin,             /* <= type of boundary, or NULL */
int *numBounds,              /* <= number of boundaries, or NULL */
BsurfBoundary **bounds,      /* <= surface boundaries, or NULL */
MSElementDescr *elmDescr    /* => elements defining surface */
);
```

Description mdlBspline_extractSurface returns information about a B-spline surface element stored in an element descriptor. It allocates memory for the *poles*, *uKnots*, *vKnots*, *weights* and *bounds* as necessary. The poles are returned unweighted.

Returns mdlBspline_extractSurface returns SUCCESS (zero) to indicate successful completion. Otherwise, it returns one of the following error codes:

MDLERR_BADBSPERM -508
MDLERR_NOBSPHEADER -505
MDLERR_INSMEMORY 116

See Also mdlBspline_extractCurve.

mdlBspline_allocateCurve

```
#include <mdlbspIn.h>

int mdlBspline_allocateCurve
(
MSBsplineCurve *curve        /* <=> curve structure */
);
```

Description `mdlBspline_allocateCurve` allocates memory for the poles, weights and knot vector of a B-spline curve. The values stored in the `MSBsplineCurve` structure (defined in `mdlbspln.h`) determine the amount of memory the function allocates. *curve->params* and *curve->rational* must be valid when this function is called.

Returns `mdlBspline_allocateCurve` returns `SUCCESS` (zero) to indicate successful completion. It returns `MDLERR_INSMEMORY` if there is not enough memory for the allocations.

See Also `mdlBspline_allocateSurface`.

mdlBspline_allocateSurface

```
#include <mdlbspln.h>

int mdlBspline_allocateSurface
(
    MSBsplineSurface *surface /* <=> surface structure */
);
```

Description `mdlBspline_allocateSurface` allocates memory for the poles, weights, and knot vectors of a B-spline surface. The values stored in the `MSBsplineSurface` structure (defined in `mdlbspln.h`) determine the amount of memory the function allocates. *surface->uParams*, *surface->vParams* and *curve->rational* must be valid when this function is called.

Returns `mdlBspline_allocateSurface` returns `SUCCESS` (zero) if successful. It returns `MDLERR_INSMEMORY` if there is not enough memory for the allocations.

See Also `mdlBspline_allocateCurve`.

mdlBspline_freeCurve

```
#include <mdlbspln.h>

void mdlBspline_freeCurve
(
    MSBsplineCurve *curve /* <=> curve structure */
);
```

Description The `mdlBspline_freeCurve` function frees memory allocated for the poles, weights and knot vector of a B-spline curve.

Returns The `mdlBspline_freeCurve` function is of type `void`. It returns no value.

See Also `mdlBspline_freeSurface`.

mdlBspline_freeSurface

```
#include <mdlbspln.h>

void mdlBspline_freeSurface
(
    MSBsplineSurface *surface /* <=> surface structure */
);
```

Description The mdlBspline_freeSurface function frees memory allocated for the poles, weights and knot vectors of a B-spline surface.

Returns The mdlBspline_freeSurface function is of type void. It returns no value.

See Also mdlBspline_freeCurve.

mdlBspline_freeCurveChain

```
void mdlBspline_freeCurveChain
(
    CurveChain **chainPP /* <=> chain to free */
);
```

Description The mdlBspline_freeCurveChain function frees memory allocated for each curve in the curve chain.

chainPP points to the address of the curve chains.

Returns mdlBspline_freeCurveChain returns no value.

See Also mdlBspline_freeCurve.

mdlBspline_freeSurfaceChain

```
void mdlBspline_freeSurfaceChain
(
    SurfaceChain **chainPP /* <=> chain to free */
);
```

Description The mdlBspline_freeSurfaceChain function frees the memory allocated for each surface in the surface chain.

chainPP points to the address of the surface chain.

Returns mdlBspline_freeSurfaceChain returns no value.

See Also mdlBspline_freeSurface.

mdlBspline_addCurveLink

```
CurveChain *mdlBspline_addCurveLink
(
CurveChain  **chainPP,      /* <=> chain to add link to */
MSBsplineCurve *curveP,    /* => curve for link */
void        *userDataP     /* => userdataP for link */
);
```

Description The mdlBspline_addCurveLink function adds a curve element to the existing curve chain.

chainPP points to the address of the curve chain.

curveP points to the curve element to be added to the chain.

userDataP points to the user data to be linked.

Returns mdlBspline_addCurveLink returns pointer to a CurveChain which contains only *curveP* and *userDataP*. It returns NULL if there is not enough memory for allocation.

See Also mdlBspline_addSurfaceLink.

mdlBspline_addSurfaceLink

```
SurfaceChain *mdlBspline_addSurfaceLink
(
SurfaceChain  **chainPP,      /* <=> chain to add link to */
MSBsplineSurface *surfaceP,   /* => surface for link */
void          *userDataP     /* => userdataP for link */
);
```

Description The mdlBspline_addSurfaceLink function adds a surface element to the existing surface chain.

chainPP points to the address of the surface chain.

surfaceP points to the B-spline surface element.

userDataP points to the user data to be linked.

Returns mdlBspline_addSurfaceLink returns pointer to a SurfaceChain which contains only *surfaceP* and *userDataP*. It returns NULL if there is not enough memory for allocation.

See Also mdlBspline_addCurveLink.

mdlBspline_weightPoles

```
#include <mdlbsp1n.h>

void mdlBspline_weightPoles
(
    Dpoint3d    *weightedPoles, /* <= output poles */
    Dpoint3d    *poles,         /* => input poles */
    double      *weights,       /* => input weights */
    int         numPoles        /* => number of poles */
);
```

Description The mdlBspline_weightPoles function calculates the homogeneous coordinates of a B-spline's poles from their respective weights.

weightedPole is the returned array of weighted poles. Enough memory should be allocated to contain *numPoles* Dpoint3d structures.

poles is the array of unweighted poles.

weights is the array of weights for the poles.

numPoles is an integer representing the number of entries in the above arrays.

Returns The mdlBspline_weightPoles function is of type void. It returns no value.

See Also mdlBspline_unWeightPoles.

mdlBspline_unWeightPoles

```
#include <mdlbsp1n.h>

void mdlBspline_unWeightPoles
(
    Dpoint3d    *poles,          /* <= output poles */
    Dpoint3d    *weightedPoles, /* => input poles */
    double      *weights,       /* => input weights */
    int         numPoles        /* => number of poles */
);
```

Description The mdlBspline_weightPoles function calculates the real-world coordinates of a B-spline's poles from their respective weights.

poles is the returned array of unweighted poles. Enough memory should be allocated to contain *numPoles* Dpoint3d structures.

weightedPoles is the array of weighted poles.

weights is the array of weights for the poles.

numPoles is an integer representing the number of entries in the above arrays.

Returns The mdlBspline_unWeightPoles function is of type void. It returns no value.

See Also mdlBspline_weightPoles.

mdlBspline_openCurve

```
#include <mdlbspIn.h>

int mdlBspline_openCurve
(
    MSBsplineCurve  *output, /* <= resulting curve */
    MSBsplineCurve  *input,  /* => original curve */
    double          u        /* => parameter to open at */
);
```

Description The mdlBspline_openCurve function returns an equivalent open B-spline curve when given a closed B-spline curve. This open curve starts and ends at the point where the parameter value equals *u*. The resulting curve appears identical to the initial curve, but is non-periodic. This routine allocates memory for the output curve as necessary.

output points to the returned curve. It can point to the same structure that *input* points to.

input points to the B-spline curve that will be opened.

u contains the parameter where the closed B-spline curve will be opened.

Returns The mdlBspline_openCurve function returns SUCCESS (zero) to indicate successful completion. It returns MDLERR_INSMEMORY if there is not enough memory for the allocations.

See Also mdlBspline_makeBezier, mdlBspline_makeRational, mdlBspline_reverseCurve.

mdlBspline_closeCurve

```
int mdlBspline_closeCurve
(
    MSBsplineCurve  *output, /* <= resulting curve */
    MSBsplineCurve  *input   /* => original curve */
);
```

Description This functions returns a closed curve that is identical to a given open curve that starts and stops at the same point. Output may point to the same curve as input.

Returns mdlBspline_imposeBoundary returns SUCCESS (zero) to indicate successful completion or MDLERR_INSMEMORY if memory is insufficient for the allocations.

See Also mdlBspline_closeSurface.

mdlBspline_elevateDegree

```
int mdlBspline_elevateDegree
(
    MSBsplineCurve  *output,      /* <= resulting curve */
    MSBsplineCurve  *input,       /* => original curve */
    int              newDegree     /* => desired degree */
);
```

Description This function elevates the degree (increases the order) of a B-spline curve. The resulting curve appears the same as the original but its parameters are changed. *output* may point to the same curve as input.

Returns mdlBspline_elevateDegree returns SUCCESS (zero) to indicate successful completion or MDLERR_INSFMEMORY if memory is insufficient for the allocations. It returns MDLERR_BADORDER if *newDegree* is less than the current degree of the curve.

See Also mdlBspline_elevateDegreeSurface.

mdlBspline_openSurface, mdlBspline_closeSurface, mdlBspline_elevateDegreeSurface

```
int mdlBspline_openSurface
(
    MSBsplineSurface *output,      /* <= resulting surface */
    MSBsplineSurface *input,       /* => original surface */
    double            uvValue,     /* => parameter to open at */
    int               edge         /* => direction flag */
);

int mdlBspline_closeSurface
(
    MSBsplineSurface *output,      /* <= resulting surface */
    MSBsplineSurface *input,       /* => original surface */
    int               edge         /* => direction flag */
);

int mdlBspline_elevateDegreeSurface
(
    MSBsplineSurface *output,      /* <= resulting surface */
    MSBsplineSurface *input,       /* => original surface */
    int               newDegree,    /* => desired degree */
    int               edge         /* => direction flag */
);
```

Description These functions do for surfaces what mdlBspline_openCurve, mdlBspline_closeCurve and mdlBspline_elevateDegree functions do for curves.

The value of *edge*, BSSURF_U or BSSURF_V, determines which direction of the surface to act upon.

Returns These functions return SUCCESS (zero) to indicate successful completion or MDLERR_INSMEMORY if memory is insufficient for the allocations. They return MDLERR_BADPARAMETER if they are passed illegal arguments.

See Also mdlBspline_openCurve, mdlBspline_closeCurve, mdlBspline_elevateDegree.

mdlBspline_makeBezier

```
#include <mdlbspIn.h>

int mdlBspline_makeBezier
(
  MSBsplineCurve  *output, /* <= resulting curve */
  MSBsplineCurve  *input  /* => original curve */
);
```

Description The mdlBspline_makeBezier function returns an equivalent Bézier curve when given a B-spline curve. It allocates memory for the output curve as necessary.

output points to the returned curve. It can point to the same structure that *input* points to.

input points to the B-spline curve that will become a Bézier curve.

Returns The mdlBspline_makeBezier function returns SUCCESS (zero) to indicate successful completion. It returns MDLERR_INSMEMORY if there is not enough memory for the allocations.

See Also mdlBspline_makeRational, mdlBspline_openCurve, mdlBspline_reverseCurve.

mdlBspline_makeBezierSurface

```
int mdlBspline_makeBezierSurface
(
  MSBsplineSurface *output, /* <= resulting surface */
  MSBsplineSurface *input  /* => original surface */
);
```

Description This function does for surfaces what mdlBspline_makeBezier does for curves.

Returns The mdlBspline_makeBezierSurface function returns SUCCESS (zero) to indicate successful completion or MDLERR_INSMEMORY if memory is insufficient for the allocations.

mdlBspline_makeRational

```
#include <mdlbsp1n.h>

int mdlBspline_makeRational
(
    MSBsplineCurve  *output, /* <= resulting curve */
    MSBsplineCurve  *input  /* => original curve */
);
```

Description mdlBspline_makeRational accepts a non-rational B-spline curve and returns an equivalent rational B-spline. It allocates memory for the weights as necessary and assigns them the value 1.0.

output points to the returned curve. It can point to the same structure that *input* points to.

input points to the B-spline curve to make rational.

Returns mdlBspline_makeRational returns SUCCESS to indicate successful completion. It returns MDLERR_INSFMEMORY if there is not enough memory for the allocations.

See Also mdlBspline_makeBezier, mdlBspline_openCurve, mdlBspline_reverseCurve.

mdlBspline_makeRationalSurface

```
#include <mdlbsp1n.h>

int mdlBspline_makeRationalSurface
(
    MSBsplineSurface *output, /* <= resulting surface */
    MSBsplineSurface *input  /* => original surface */
);
```

Description mdlBspline_makeRationalSurface accepts a non-rational B-spline surface and returns an equivalent rational B-spline. It allocates memory for the weights as necessary and assigns them the value 1.0.

output points to the returned surface. It can point to the same structure that *input* points to.

input points to the B-spline surface to make rational.

Returns mdlBspline_makeRationalSurface returns SUCCESS to indicate successful completion. It returns MDLERR_INSFMEMORY if there is not enough memory for the allocations.

See Also mdlBspline_makeRational, mdlBspline_makeBezierSurface, mdlBspline_openSurface, mdlBspline_reverseSurface, mdlBspline_swapUV.

mdlBspline_reverseCurve

```
#include <mdlbspn.h>

int mdlBspline_reverseCurve
(
    MSBsplineCurve  *output, /* <= resulting curve */
    MSBsplineCurve  *input  /* => original curve */
);
```

Description The `mdlBspline_reverseCurve` function changes the direction that a B-spline curve traces as its parameter ranges from zero to one. The resulting curve is identical to the initial curve but is oriented in the opposite direction. This routine allocates memory for the output curve as needed.

output points to the returned curve. It can point to the same structure that *input* points to.

input points to the original curve before it is reversed.

Returns The `mdlBspline_reverseCurve` function returns `SUCCESS` (zero) to indicate successful completion. It returns `MDLERR_INSFMEMORY` if there is not enough memory for the allocations.

See Also `mdlBspline_makeBezier`, `mdlBspline_makeRational`, `mdlBspline_openCurve`.

mdlBspline_reverseSurface

```
#include <mdlbspn.h>

int mdlBspline_reverseSurface
(
    MSBsplineSurface *output,          /* <= resulting surface */
    MSBsplineSurface *input,          /* => original surface */
    int               direction        /* => param. direction to reverse */
);
```

Description The `mdlBspline_reverseSurface` function changes the direction that a B-spline surface traces as its parameter ranges from zero to one. The resulting surface is identical to the initial surface but is oriented in the opposite direction. This routine allocates memory for the output surface as needed.

output points to the returned surface. It can point to the same structure to which *input* points.

input points to the original surface before it is reversed.

direction contains a flag indicating which parameter direction of the input surfaces should be reversed. The values `BSSURF_U` and `BSSURF_V` are defined in `mdlbspn.h`.

Returns `mdlBspline_reverseSurface` returns `SUCCESS` to indicate successful completion. It returns `MDLERR_INSFMEMORY` if there is not enough memory for the allocations.

See Also mdlBspline_reverseCurve, mdlBspline_makeBezierSurface, mdlBspline_openSurface, mdlBspline_reverseSurface, mdlBspline_swapUV.

mdlBspline_make2CurvesCompatible

```
#include <mdlbspln.h>

int mdlBspline_make2CurvesCompatible
(
    MSBsplineCurve    *curve1,    /* <=> first curve */
    MSBsplineCurve    *curve2    /* <=> second curve */
);
```

Description mdlBspline_make2CurvesCompatible returns two B-spline curves with the same order, number of poles, and knot vectors. Further, they are both rational or both non-rational. This routine allocates memory for additional poles, knots and weights as necessary.

curve1 points to the first B-spline curve to make compatible.

curve2 points to the second B-spline curve to make compatible.

Returns mdlBspline_make2CurvesCompatible returns SUCCESS (zero) to indicate successful completion, and MDLERR_INSFMEMORY if there is not enough memory for the allocations.

See Also mdlBspline_makeCurvesCompatible, mdlBspline_make2SurfacesCompatible.

mdlBspline_makeCurvesCompatible

```
#include <mdlbspln.h>

int mdlBspline_makeCurvesCompatible
(
    MSBsplineCurve    *outputCurves[],    /* <= resulting curves */
    MSBsplineCurve    *inputCurves[],    /* => original curve */
    int                numCurves          /* => number of curves */
);
```

Description The mdlBspline_makeCurvesCompatible function returns an array of pointers to equivalent compatible B-spline curves when given an array of pointers to B-spline curves. The resulting curves have the same order, number of poles, and knot vector. Further, they are either all rational or non-rational. This routine allocates memory for the output curves as necessary.

outputCurves is an array of pointers to compatible B-spline curves.

inputCurves is an array of pointers to the original B-spline curves.

numCurves is the number of curves to make compatible.

Returns The `mdlBspline_makeCurvesCompatible` function returns `SUCCESS` (zero) to indicate successful completion. It returns `MDLERR_INSMEMORY` if there is not enough memory for the allocations.

See Also `mdlBspline_make2CurvesCompatible`, `mdlBspline_make2SurfacesCompatible`.

mdlBspline_make2SurfacesCompatible

```
#include <mdlbsp1n.h>

int mdlBspline_make2SurfacesCompatible
(
    MSBsplineSurface *surface1,    /* <=> first surface */
    MSBsplineSurface *surface2,    /* <=> second surface */
    int               direction     /* => param. direction to modify */
);
```

Description `mdlBspline_make2SurfacesCompatible` returns two B-spline surface with the same order, number of poles, and knot vectors in one of the two parametric directions, U or V. Further, they are both rational or both non-rational. This routine allocates memory for additional poles, knots and weights as necessary.

surface1 points to the first B-spline surface to make compatible.

surface2 points to the second B-spline surface to make compatible.

direction contains a flag indicating which parameter direction of the input surfaces should be made compatible. The values `BSSURF_U` and `BSSURF_V` are defined in the file `mdlbsp1n.h`.

Returns `mdlBspline_make2SurfacesCompatible` returns `SUCCESS` (zero) to indicate successful completion, and `MDLERR_INSMEMORY` if there is not enough memory for the allocations.

See Also `mdlBspline_make2CurvesCompatible`, `mdlBspline_makeCurvesCompatible`.

mdlBspline_swapUV

```
#include <mdlbsp1n.h>

int mdlBspline_swapUV
(
    MSBsplineSurface *output, /* <= resulting surface */
    MSBsplineSurface *input,  /* => original surface */
);
```

Description The `mdlBspline_swapUV` switches the parameter directions of a B-spline surface. The resulting surface is identical to the initial surface but has it U and V directions exchanged. This routine allocates memory for the output surface as needed.

output points to the returned surface. It can point to the same structure to which *input* points.

input points to the original surface before it is reversed.

Returns mdlBspline_swapUV returns SUCCESS to indicate successful completion. It returns MDLERR_INSMEMORY if there is not enough memory for the allocations.

See Also mdlBspline_reverseSurface.

mdlBspline_segmentDisjointCurve [mdllib.ml], mdlBspline_segmentDisjointSurface [mdllib.ml]

```
#include <mdlbsp1n.h>

int mdlBspline_segmentDisjointCurve
(
    MSBsplineCurve    **output,      /* <=> returned array of curves */
    int               *numCurves,   /* <=> # of curves in array */
    MSBsplineCurve    *input        /* => input curve */
);

int mdlBspline_segmentDisjointSurface
(
    MSBsplineSurface  **output,      /* <=> returned array of surfaces */
    int               *numSurfaces,  /* <=> # of surfaces in array */
    MSBsplineSurface  *input        /* => input curve */
);
```

Description mdlBspline_segmentDisjointCurve and mdlBspline_segmentDisjointSurface return arrays of MSBsplineCurve and MSBsplineSurface structures representing the disjoint components of B-spline curves and surfaces that contain jump discontinuities. These routines allocate memory as necessary.

output points to the array of MSBsplineCurve or MSBsplineSurface structures allocated by this function.

numCurves contains the number of curves in the *output* array.

numSurfaces contains the number of surfaces in the *output* array.

input points to the discontinuous B-spline curve or surface to segment.

Returns These functions return SUCCESS (zero) to indicate successful completion. They return MDLERR_INSMEMORY if there is not enough memory for the allocations.

See Also mdlBspline_segmentCurve.

mdlBspline_curveDataReduction

```
#include <mdlbspln.h>

int mdlBspline_curveDataReduction
(
  MSBsplineCurve  *output,      /* <= resulting curve */
  MSBsplineCurve  *input,       /* => original curve */
  double          tolerance     /* => tolerance value */
);
```

Description `mdlBspline_curveDataReduction` creates a B-spline curve with the number of poles being reduced to a minimum number, defined by *tolerance*. In general, the resulting poles are different from the original poles, but the shape of the curve will remain the same. The order of the curve does not change. If the original curve is rational, the weights of the resulting curve are set to ones. If there are no poles are reduced, the output is the same as the input curve. The output can point to the same data structure that input points to. This routine allocates memory for the resulting curve as necessary.

output points to the curve being returned.

input points to the original curve.

tolerance represents the maximum allowed deviation from the original curve.

Returns `mdlBspline_curveDataReduction` returns `SUCCESS` to indicate successful completion. It returns `MDLERR_INSFMEMORY`, if there is not enough memory for allocation.

See Also `mdlBspline_elevateDegree`.

mdlBspline_copyBoundaries

```
int mdlBspline_copyBoundaries
(
  MSBsplineSurface *out,      /* <= resulting surface */
  MSBsplineSurface *in       /* => input surface */
);
```

Description `mdlBspline_copyBoundaries` copies the boundary elements from the input surface onto the output surface. It allocates memory for the output surface as necessary.

out points to the returned B-spline surface.

in points to the input B-spline surface.

Returns `mdlBspline_copyBoundaries` returns `SUCCESS` to indicate the successful completion. It returns `MDLERR_INSFMEMORY` is returned if there is not enough memory for allocation.

See Also `mdlBspline_copySurface`.

B-spline Knot Functions

The following table lists B-spline knot functions:

Function	Used to
mdlBspline_computeKnotVector	calculate the full knot vector for a B-spline curve.
mdlBspline_normalizeKnotVector	normalize a full knot vector to run from zero to one.
mdlBspline_knotTolerance	return the knot tolerance for a B-spline curve.
mdlBspline_getKnotMultiplicity	return the distinct knots of a knot vector.
mdlBspline_computeGrevilleAbscissa	calculate the Greville abscissa of a B-spline knot vector
mdlBspline_findSpan	calculate the interval of a knot vector that contains the specific knot value.
mdlBspline_addKnot	add a given knot to a B-spline curve.
mdlBspline_addKnotSurface	add a knot to a surface's U or V knot vector.
mdlBspline_numberKnots [mdl1lib.m1]	calculate the size of a full knot vector.

Example

See bspline.mc.

mdlBspline_computeKnotVector

```
#include <mdlbsp1n.h>

int mdlBspline_computeKnotVector
(
    double      *knotVector,    /* <= full knot vector */
    BsplineParam *params,       /* => B-spline parameters */
    double      *interiorKnots /* => interior knots */
);
```

Description The mdlBspline_computeKnotVector function calculates the full knot vector for a B-spline. Sufficient memory should be allocated in *knotVector* to contain the complete calculated knot vector as an array of doubles (*params->numPoles* + *params->order* doubles for an open B-spline, *params->numPoles* + 2**params->order* - 1 doubles for a closed B-spline).

For an open B-spline, the initial and final knots, zero and one respectively, are repeated *params->order* number of times. For a closed B-spline, the knot vector is cyclical, with the first *params->order* - 1 knots less than zero and the last *params->order* - 1 knots greater than one.

params points to the `BsplineParam` structure used to define the B-spline knot vector.

If the curve is non-uniform (when *params->numKnots* is non-zero), *interiorKnots* must contain the interior knots running between zero and one (*params->numPoles* - *params->order* knots if *params->closed* is `FALSE` and *params->numPoles*-1 knots if *params->closed* is `TRUE`).

Returns `mdlBspline_computeKnotVector` returns `SUCCESS` (zero) to indicate successful completion.

See Also `mdlBspline_normalizeKnotVector`, `mdlBspline_knotTolerance`, `mdlBspline_getKnotMultiplicity`, `mdlBspline_computeGrevilleAbscissa`, `mdlBspline_findSpan`, `mdlBspline_addKnot`.

mdlBspline_normalizeKnotVector

```
#include <mdlbspln.h>

int mdlBspline_normalizeKnotVector
(
    double *knotVector,    /* <=> full knot vector */
    int     numPoles,      /* => number of poles in B-spline */
    int     order,         /* => order of B-spline */
    int     closed,        /* => periodicity of B-spline */
);
```

Description The `mdlBspline_normalizeKnotVector` function normalizes a full knot vector to run from zero to one.

knotVector points to a complete knot vector to be normalized.

numPoles contains an integer indicating the number of poles for the B-spline.

order contains an integer indicating the order of the B-spline.

closed indicates whether the B-spline is periodic.

Returns The `mdlBspline_normalizeKnotVector` function returns `SUCCESS` (zero) to indicate successful completion.

See Also `mdlBspline_normalizeKnotVector`, `mdlBspline_knotTolerance`, `mdlBspline_getKnotMultiplicity`, `mdlBspline_computeGrevilleAbscissa`, `mdlBspline_findSpan`, `mdlBspline_addKnot`.

mdlBspline_knotTolerance

```
#include <mdlbspln.h>

double mdlBspline_knotTolerance
(
    MSBsplineCurve *curve    /* => curve structure */
);
```

Description The mdlBspline_knotTolerance function returns the knot tolerance for a B-spline curve. This value is set to less than the minimum difference between successive knots in the knot vector but is greater than zero.

curve points to a valid B-spline curve structure.

Returns The mdlBspline_knotTolerance function returns the knot tolerance as a double.

See Also mdlBspline_computeKnotVector, mdlBspline_normalizeKnotVector, mdlBspline_getKnotMultiplicity, mdlBspline_computeGrevilleAbscissa, mdlBspline_findSpan, mdlBspline_addKnot.

mdlBspline_getKnotMultiplicity

```
#include <mdlbspln.h>

int mdlBspline_getKnotMultiplicity
(
    double *distinctKnots,    /* <= distinct knots of knot vector */
    int *knotMultiplicity,    /* <= multiplicity of distinct knots */
    int *numDistinct,         /* <= number of distinct knots */
    double *knotVector,       /* => full knot vector */
    int numPoles,             /* => number of poles of B-spline */
    int order,                /* => order of B-spline */
    int closed,               /* => periodicity of B-spline */
    double knotTolerance      /* => knot tolerance of B-spline */
);
```

Description The mdlBspline_getKnotMultiplicity function returns the distinct knots of a knot vector. It also returns integers specifying how often these knots occur.

distinctKnots returns an array of doubles containing the distinct knots. It should contain enough memory for *numPoles* + 1 number of doubles.

knotMultiplicity returns an array of integers containing the multiplicities of the individual knots. It should contain enough memory for *numPoles*+1 integers.

numDistinct returns the number of distinct knots in *knotVector*.

knotVector is the full knot vector stored as an array of doubles.

numPoles contains an integer indicating the number of poles for the B-spline.

order contains an integer indicating the order of the B-spline.

closed indicates whether the B-spline is periodic.

knotTolerance contains a double specifying the knot tolerance of the B-spline.

Returns The mdlBspline_getKnotMultiplicity function returns SUCCESS (zero) to indicate successful completion.

See Also mdlBspline_computeKnotVector, mdlBspline_normalizeKnotVector, mdlBspline_knotTolerance, mdlBspline_computeGrevilleAbscissa, mdlBspline_findSpan, mdlBspline_addKnot.

mdlBspline_computeGrevilleAbscissa

```
#include <mdlbspln.h>

int mdlBspline_computeGrevilleAbscissa
(
    double *nodeValue,    /* <= Greville abscissa of B-spline */
    double *knotVector,  /* => full knot vector of B-spline */
    int    numPoles,     /* => number of poles of B-spline */
    int    order,        /* => order of B-spline */
    int    closed,       /* => periodicity of B-spline */
    double knotTolerance /* => knot tolerance of B-spline */
);
```

Description The mdlBspline_computeGrevilleAbscissa function calculates the Greville abscissa of a B-spline's knot vector. These are averages of *order-1* knots at a time taken from the full knot vector.

nodeValue returns the calculated Greville abscissa in an array of doubles. It should contain enough memory for *numPoles* doubles.

knotVector is the full knot vector stored as an array of doubles.

numPoles contains an integer indicating the number of poles for the B-spline.

order contains an integer indicating the order of the B-spline.

closed indicates whether the B-spline is periodic.

knotTolerance contains a double specifying the knot tolerance of the B-spline.

Returns The mdlBspline_computeGrevilleAbscissa function returns SUCCESS (zero) to indicate successful completion.

See Also mdlBspline_computeKnotVector, mdlBspline_normalizeKnotVector, mdlBspline_knotTolerance, mdlBspline_getKnotMultiplicity, mdlBspline_findSpan, mdlBspline_addKnot.

mdlBspline_findSpan

```
#include <mdlbspln.h>

int mdlBspline_findSpan
(
    int      *spanIndex,      /* <= span of u in knot vector */
    double   *knotVector,    /* => full knot vector */
    int      numPoles,        /* => number of poles of B-spline */
    int      order,           /* => order of B-spline */
    int      closed,          /* => periodicity of B-spline */
    double   u                /* => parameter of curve to check */
);
```

Description The mdlBspline_findSpan function calculates the interval of a knot vector that contains the specific knot value. Upon return, *spanIndex* is set as follows:

```
knotVector[spanIndex]< u <= knotVector[spanIndex+1]
```

knotVector is the full knot vector stored in an array of doubles.

numPoles contains an integer indicating the number of poles for the B-spline.

order contains an integer indicating the order of the B-spline.

closed indicates whether the B-spline is periodic.

u contains the parameter value between zero and one to be located in the knot vector.

Returns mdlBspline_findSpan returns SUCCESS (zero) to indicate successful completion.

See Also mdlBspline_computeKnotVector, mdlBspline_normalizeKnotVector, mdlBspline_knotTolerance, mdlBspline_getKnotMultiplicity, mdlBspline_computeGrevilleAbscissa, mdlBspline_addKnot.

mdlBspline_addKnot

```
#include <mdlbspln.h>

int mdlBspline_addKnot
(
    MSBsplineCurve *curve,      /* <=> curve structure */
    double          u,          /* => knot value to add */
    double          knotTolerance, /* => knot tolerance of B-spline */
    int             newMult,     /* => new knot multiplicity */
    int             addToCurrent /* => multiplicity flag */
);
```

Description The `mdlBspline_addKnot` function adds a given knot value to a B-spline curve. It calculates the additional poles, knots and weights for the curve and reallocates memory for them as needed.

curve points to the B-spline curve structure to which a knot should be added.

u contains the parameter value to be added to the knot vector.

knotTolerance contains a `double` specifying the knot tolerance of the B-spline.

newMult contains an integer representing the number of times to add the knot value *u* (*addToCurrent* is `TRUE`), or the desired final multiplicity of the knot value *u* in the knot vector (*addToCurrent* is `FALSE`).

Returns The `mdlBspline_addKnot` function returns `SUCCESS` (zero) to indicate successful completion. It returns `MDLERR_INSFMEMORY` if there is not enough memory for the allocations. If the *u* parameter is not in the range of the full knot vector or the resulting multiplicity of the knot would be less than the current multiplicity, the function returns `MDLERR_BADPARAMETER`.

See Also `mdlBspline_computeKnotVector`, `mdlBspline_normalizeKnotVector`, `mdlBspline_knotTolerance`, `mdlBspline_getKnotMultiplicity`, `mdlBspline_computeGrevilleAbcissa`, `mdlBspline_findSpan`.

mdlBspline_addKnotSurface

```
#include <mdlbspln.h>

int mdlBspline_addKnotSurface
(
    MSBsplineSurface *surface,      /* <=> curve structure */
    double            uv,           /* => knot value to add */
    double            knotTolerance, /* => knot tolerance of spline */
    int               newMult,       /* => new knot multiplicity */
    int               addToCurrent,  /* => multiplicity flag */
    int               direction      /* => direction flag */
);
```

Description `mdlBspline_addKnotSurface` adds a given knot value to one of the knot vectors of a B-spline surface. It calculates the additional poles, knots and weights for the surface and reallocates memory for them as needed.

surface points to the B-spline surface structure to which a knot should be added.

uv contains the parameter value to be added to the knot vector.

knotTolerance contains a `double` specifying the knot tolerance of the B-spline.

newMult contains an integer representing the number of times to add the knot value *uv* (*addToCurrent* is TRUE), or the desired final multiplicity of the knot value *uv* in the knot vector (*addToCurrent* is FALSE).

direction contains a flag indicating to which knot vector the value should be added. The values BSSURF_U and BSSURF_V are defined in the file mdlbspln.h.

Returns mdlBspline_addKnotSurface returns SUCCESS (zero) to indicate successful completion. It returns MDLERR_INSFMEMORY if there is not enough memory for the allocations. If the *u v* parameter is not in the range of the full knot vector or the resulting multiplicity of the knot would be less than the current multiplicity, the function returns MDLERR_BADPARAMETER.

See Also mdlBspline_addKnot, mdlBspline_openSurface.

mdlBspline_numberKnots [mdlLib.mll]

```
#include <mdlbspln.h>

int mdlBspline_numberKnots
(
    int    numPoles,      /* number of poles of B-spline */
    int    order,         /* order of B-spline */
    int    closed,        /* periodicity of B-spline */
);
```

Description The mdlBspline_numberKnots function returns the number of knots in the complete knot vector given the number of poles, order and periodicity of the curve or surface.

numPoles indicates the number of poles in the curve or surface in a particular direction, U or V.

order indicates the order of the curve or surface in a particular direction, U or V.

closed indicates the periodicity (TRUE for closed) of the curve or surface in a particular direction, U or V.

Returns mdlBspline_numberKnots returns
(*closed*? *numPoles* + 2 * *order* - 1 : *numPoles* + *order*)

See Also mdlBspline_isValidKnotVector [mdlLib.mll].

B-spline Query Functions

The following table lists B-spline query functions:

Function	Used to
<code>mdlBspline_computeDerivatives</code>	calculate partial derivatives of a B-spline curve at a parameter value.
<code>mdlBspline_frenetFrame</code>	calculate the Frenet frame, radius of curvature, and torsion of a B-spline curve at a parameter value.
<code>mdlBspline_minimumDistanceToCurve</code>	find the closest point on a B-spline curve to a given point.
<code>mdlBspline_minimumDistanceToSurface</code>	find a B-spline surface's closest point to a given point.
<code>mdlBspline_evaluateCurve</code>	calculate points that lie on a B-spline curve.
<code>mdlBspline_evaluateSurface</code>	calculate points that lie on a B-spline surface.
<code>mdlBspline_evaluateCurvePoint</code>	calculate a point that lies on a B-spline curve.
<code>mdlBspline_evaluateSurfacePoint</code>	calculate a point that lies on a B-spline surface.
<code>mdlBspline_boresiteToSurface</code>	returns the parameter and location of the first point on the surface hit by a ray.
<code>mdlBspline_allBoresiteToSurface</code>	returns the parameter and location of the all points on the surface hit by a ray.
<code>mdlBspline_computeDerivativesSurface</code>	evaluate the position and partial derivatives of a B-spline surface.
<code>mdlBspline_extractCurveNormal</code>	find the plane containing a B-spline curve.
<code>mdlBspline_closestEdge</code>	find the point on the UV parametric square closest to an interior point.
<code>mdlBspline_inBounds</code>	determine whether a parameter point is on a trimmed surface.
<code>mdlBspline_isValidBoundary [mdl1lib.m1]</code>	determine if a trim boundary is OK.
<code>mdlBspline_isValidKnotVector [mdl1lib.m1]</code>	determine if a knot vector is OK.
<code>mdlBspline_isValidSurface [mdl1lib.m1]</code>	determine if a surface is OK.
<code>mdlBspline_isPhysicallyClosed</code>	determine if a curve is physically closed.
<code>mdlBspline_isPhysicallyClosedCurve [mdl1lib.m1]</code>	determine if elements in an element descriptor define a physically closed curve.
<code>mdlBspline_isPhysicallyClosedSurface</code>	determine if a surface is physically closed in either parameter direction.

Function	Used to
mdlBspline_intersectCurves	calculate the intersection points of two B-spline curves.
mdlBspline_intersectSegment	calculate the intersection of a B-spline curve with a line segment.
mdlBspline_intersectSurfaces	calculate the intersection curves of two B-spline surfaces.
mdlBspline_intersectOffsetSurfaces	calculate the intersection curves of the offset surfaces of two B-spline surfaces.
mdlBspline_cuspPoints	calculate the location and parameters of the cusp points of a B-spline curve.
mdlBspline_inflectionPoints	calculate the location and parameters of the inflection points of a B-spline curve.
mdlBspline_extractCapAsCurve	extract either the first or the last rule line in the parameter space.
mdlBspline_extractCapAsCurves	extract either the first or the last rule line from the surface in the parameter space
mdlBspline_extractCapAsSurface [mdlLib.mll]	extract the endcaps of a solid.
mdlBspline_meshSurface	compute quadrilateral/triangle mesh for surface.
mdlBspline_edgeCode	tell if a given point is on one of the four edges in parameter space.
mdlBspline_isDegenerateEdge	check whether an edge of the surface degenerates to a single point.
mdlBspline_isSolid	check whether a B-spline surface encloses a valid space.

Example

See bspline.mc.

mdlBspline_computeDerivatives

```
#include <mdlbspln.h>

int mdlBspline_computeDerivatives
(
    Dpoint3d      *dervPoles,      /* <= derivative of poles */
    double        *dervWeights,    /* <= derivative of weights or NULL */
    MSBsplineCurve *curve,         /* => curve structure */
    int           numDervs,         /* => number of derivatives */
    double        u,               /* => parameter to evaluate at */
);
```

Description The `mdlBspline_computeDerivatives` function calculates the number of derivatives specified by *numDervs* of a B-spline curve at a particular parameter value. If the curve is rational, it also calculates the derivatives of the homogeneous coordinate (weight) function. It does not, however, perform the chain rule computations for the complete derivative if the curve is rational.

dervPoles contains the calculated derivatives of the x, y and z coordinates of the B-spline curve. *dervPoles* should contain enough memory to hold *numDervs* + 1 `Dpoint3ds`. (*dervPoles*[0] is the curve evaluated at *u*).

dervWeights contains the calculated derivative of the homogeneous coordinate (weight) of the B-spline curve. If *curve->rational* is `FALSE`, *dervWeights* can be set to `NULL`. If *curve->rational* is `TRUE`, *dervWeights* should contain enough memory to contain *numDervs* + 1 doubles.

curve points to the B-spline curve that will be evaluated.

numDervs contains the number of derivatives to calculate. Derivatives beyond *curve->params.order* will be zero.

u is the parameter value where the derivatives will be calculated.

Returns The `mdlBspline_computeDerivatives` function returns `SUCCESS` (zero) to indicate successful completion. It returns `MDLERR_NOWEIGHTS` if the curve is rational and *dervWeights* is `NULL`.

See Also `mdlBspline_computeDerivativesSurface`, `mdlBspline_frenetFrame`.

mdlBspline_frenetFrame

```
#include <mdlbspln.h>

int mdlBspline_frenetFrame
(
    Dpoint3d      *frame,          /* <= Frenet frame, or NULL */
    Dpoint3d      *point,         /* <= point on curve , or NULL */
    double        *curvature,     /* <= curvature, or NULL */
    double        *torsion,       /* <= torsion, or NULL */
    MSBsplineCurve *curve,        /* => curve structure */
    double        u,              /* => parameter to evaluate at */
);
```

Description The mdlBspline_frenetFrame function calculates the Frenet frame, radius of curvature, and torsion of a B-spline curve at a particular parameter value.

If *frame* != NULL, when the function returns, *frame* points to the calculated Frenet frame of the B-spline curve consisting of the normalized tangent, main normal, and binormal vectors. It must contain enough memory to hold three Dpoint3d structures.

If *point* != NULL, when the function returns, *point* contains the position of the B-spline curve evaluated at the parameter. It serves as the origin of the coordinate system.

If *curvature* and *torsion* are different from NULL, when the function returns, they will contain values for the B-spline curve evaluated at *u*.

curve points to the B-spline curve that will be evaluated.

u is the parameter value where the above values will be calculated.

Returns The mdlBspline_frenetFrame function returns SUCCESS (zero) to indicate successful completion.

See Also mdlBspline_computeDerivatives.

mdlBspline_minimumDistanceToCurve

```
#include <mdlbspIn.h>

int mdlBspline_minimumDistanceToCurve
(
    double          *distance,      /* <= minimum distance to curve */
    Dpoint3d        *closestPoint,  /* <= closest point on curve */
    double          *u,             /* <= parameter of closest point */
    Dpoint3d        *testPoint,     /* => test point */
    MSBsplineCurve  *curve          /* => curve structure */
);
```

Description mdlBspline_minimumDistanceToCurve finds the closest point on a B-spline curve to a given point. It also returns the distance to that point and its parameter value.

distance is the minimum distance to the B-spline curve from *testPoint*.

closestPoint is the point on the B-spline curve that is nearest *testPoint*.

u is the parameter value of *closestPoint* on the B-spline curve.

testPoint is the point where the nearest point on the B-spline curve will be located.

curve points to the B-spline curve that will be evaluated.

Returns The mdlBspline_minimumDistanceToCurve function returns SUCCESS (zero) to indicate successful completion.

See Also mdlBspline_minimumDistanceToSurface, mdlMinDist_betweenElms.

mdlBspline_minimumDistanceToSurface

```
#include <mdlbsp1n.h>

int mdlBspline_minimumDistanceSurface
(
    double          *distance,      /* <= minimum dist. to surface */
    Dpoint3d        *closestPoint, /* <= closest point on surface */
    Dpoint2d        *uvValue,      /* <= parameter of closest pnt */
    Dpoint3d        *testPoint,    /* => test point */
    MSBsplineSurface *surface      /* => surface structure */
);
```

Description `mdlBspline_minimumDistanceToSurface` finds a B-spline surface's closest point to a given point. It also returns the distance to that point and the point's parameter value.

distance is the minimum distance to the B-spline surface from *testPoint*.

closestPoint is the B-spline surface's point that is nearest *testPoint*.

uvValue is the parameter value of *closestPoint* on the B-spline surface.

This function uses *testPoint* to find the nearest point on the B-spline surface. The minimum distance that this function returns is between *testPoint* and the closest point.

surface points to the B-spline surface that will be evaluated.

Returns The `mdlBspline_minimumDistanceToSurface` function returns `SUCCESS` (zero) if the function completes successfully.

See Also `mdlBspline_minimumDistanceToCurve`, `mdlMinDist_betweenElms`.

mdlBspline_evaluateCurve

```
#include <mdlbsp1n.h>

int mdlBspline_evaluateCurve
(
    Dpoint3d    **points,      /* <= evaluated points */
    double      *data,        /* => parameters, or NULL */
    int         *numPoints,    /* <=> # of evaluated points */
    MSBsplineCurve *curve     /* => curve structure */
);
```

Description The `mdlBspline_evaluateCurve` function calculates points that lie on a B-spline curve. If *numPoints* != 0 and *data* != NULL, when the function returns, *points* contains the B-spline curve values evaluated at each parameter specified in *data*.

If *numPoints* != 0 and *data* = NULL, when the function returns, *points* contains the values of the B-spline curve evaluated at *numPoints* intervals. These intervals are spaced evenly throughout the parameter range (*curve*-

>display.rulesByLength is FALSE) or along the curve's arc length (*curve->display.rulesByLength* is TRUE).

If *numPoints* = 0, *data*[0] holds a chord tolerance that strokes the curve. When the function returns, *points* contains the points generated by MicroStation's B-spline stroking algorithm and *numPoints* is set with the number of Dpoint3d structures in *points*. The true curve will deviate from these returned points by the specified tolerance (at most).

This routine allocates memory for the *points* array. *points* points to the array of calculated points on the B-spline curve.

data contains parameter values that will be evaluated, a chord tolerance to be used by the stroking algorithm, or NULL.

numPoints points to an integer containing the number of points in the output array.

curve points to the B-spline curve that will be evaluated.

Returns mdlBspline_evaluateCurve returns SUCCESS (zero) to indicate successful completion or MDLERR_INSFMEMORY if memory is insufficient for the allocations. It returns MDLERR_INSFINFO if it was not called with the correct information.

See Also mdlBspline_evaluateSurface, mdlBspline_evaluateCurvePoint, mdlBspline_evaluateSurfacePoint.

mdlBspline_evaluateSurface

```
#include <mdlbspIn.h>

int mdlBspline_evaluateSurface
(
    Dpoint3d      **points,      /* <=  evaluated points */
    Dpoint2d      *data,         /* =>  parameters, or NULL */
    int           numPoints[2],  /* <=> # of evaluated points */
    MSBsplineSurface *surface    /* =>  surface structure */
);
```

Description mdlBspline_evaluateSurface calculates points that lie on a B-spline surface. If *numPoints*[0] != 0 and *data* != NULL, when the function returns, *points* contains the values of the B-spline surface evaluated at each parameter specified in *data*.

If *numPoints*[0] != 0 and *data* = NULL, when the function returns, *points* contains the values of the B-spline curve evaluated at intervals of *numPoints*[0] in the u direction and *numPoints*[1] in the v direction. These intervals are spaced evenly throughout the parameter range (*surface->display.rulesByLength* is FALSE) or along the curve's arc length (*surface->display.rulesByLength* is TRUE).

If *numPoints*[0] = 0, *data*[0] holds a chord tolerance to stroke the surface. When the function returns, *points* contains the points generated by MicroStation's B-spline stroking algorithm and *numPoints*[0] is set with the

number of `Dpoint3d` structures in *points*. The true surface will deviate from these returned points by the specified tolerance (at most). This routine allocates memory for the *points* array.

points points to the array of calculated points on the B-spline surface.

data contains parameter values that will be evaluated the B-spline, a chord tolerance to be used by the stroking algorithm, or `NULL`. (The X coordinate corresponds to the U parameter and Y coordinate corresponds to the V parameter).

numPoints points to an array of two ints containing the number of pts in each direction. The total number of points is *numPoints*[0] * *numPoints*[1].

surface points to the B-spline surface that will be evaluated.

Returns The `mdlBspline_evaluateSurface` function returns `SUCCESS` (zero) to indicate successful completion or `MDLERR_INSFMEMORY` if memory is insufficient for the allocations. It returns `MDLERR_INSFINFO` if it was not called with correct information.

See Also `mdlBspline_evaluateCurve`, `mdlBspline_evaluateCurvePoint`, `mdlBspline_evaluateSurfacePoint`.

mdlBspline_evaluateCurvePoint

```
#include <mdlbspln.h>

void mdlBspline_evaluateCurvePoint
(
    Dpoint3d    *pointP,          /* <= point on curve */
    Dpoint3d    *tangentP,       /* <= tangent vector at the point */
    MSBsplineCurve *curveP,      /* => B-spline curves */
    double      parameter        /* => parameter value */
);
```

Description The `mdlBspline_evaluateCurvePoint` function calculates the point and its tangent on a B-spline curve at a input parameter value.

pointP points to the point on curve being returned.

tangentP points to the tangent vector of the point.

curve points to the B-spline curve.

parameter is the parameter value.

See Also `mdlBspline_evaluateSurfacePoint`, `mdlBspline_evaluateCurve`, `mdlBspline_evaluateSurface`.

mdlBspline_evaluateSurfacePoint

```
#include <mdlbsp1n.h>

void mdlBspline_evaluateSurfacePoint
(
    Dpoint3d      *pointP,      /* <= point on the surface */
    double        *weightP,     /* <= weight value, or NULL */
    Dpoint3d      *dPdu,        /* <= u partial derivative or NULL */
    Dpoint3d      *dpdv,        /* <= v partial derivative or NULL */
    double        u,            /* => u parameter value */
    double        v,            /* => v parameter value */
    MSBsplineSurface *surfaceP  /* => B-spline surface */
);
```

Description The mdlBspline_evaluateSurfacePoint function calculates the point and its partial derivatives on a B-spline surface at the input *u* and *v* parameter values.

pointP points to the point on surface being returned.

If *weightP* != NULL, the weight value of the point will be returned.

If *dpdu* != NULL, the u partial derivative will be returned.

If *dpdv* != NULL, the v partial derivative will be returned.

u is the u parameter value, *v* is the v parameter value.

surfaceP points to the B-spline surface.

See Also mdlBspline_evaluateCurvePoint, mdlBspline_evaluateCurve,
mdlBspline_evaluateSurface.

mdlBspline_boresiteToSurface

```
int mdlBspline_boresiteToSurface
(
    Dpoint3d      *borePoint,    /* <= point on surface */
    Dpoint2d      *uvParam,      /* <= parameter of point */
    Dpoint3d      *testPt,       /* => point to bore from */
    Dpoint3d      *direction,     /* => direction to bore */
    MSBsplineSurface *surface    /* => surface structure */
);
```

Description This function returns the parameter and location of the first point on the surface hit by a ray starting at *testPt* and extending in the direction given by *direction*.

Returns The mdlBspline_boresiteToSurface function returns SUCCESS (zero) to indicate that the indicated ray actually hits the surface. It returns ERROR (one) if the ray misses the surface.

See Also mdlBspline_allBoresiteToSurface.

mdlBspline_allBoresiteToSurface

```
int mdlBspline_allBoresiteToSurface
(
    Dpoint3d      **borePoints, /* <= points on surface */
    Dpoint2d      **uvParams,   /* <= parameters of points */
    int           *numPoints,   /* <= number of points returned */
    Dpoint3d      *testPt,      /* => point to bore from */
    Dpoint3d      *direction,   /* => direction to bore */
    MSBsplineSurface *surface   /* => surface structure */
);
```

Description This function returns the parameter and location of the all points on the surface hit by a ray starting at *testPt* and extending in the direction given by *direction*. This routine allocates the memory for the arrays *borePoints* and *uvParam* and returns the number of entries in them in *numPoints*.

Returns The `mdlBspline_allBoresiteToSurface` function returns `SUCCESS` (zero) to indicate that the indicated ray actually hits the surface. It returns `ERROR` (one) if the ray misses the surface.

See Also `mdlBspline_boresiteToSurface`.

mdlBspline_computeDerivativesSurface

```
#include <mdlbspln.h>

int mdlBspline_computeDerivativesSurface
(
    Dpoint3d      *position, /* <= point on surface */
    double        *weight,  /* <= homogeneous coord. of point, or NULL */
    Dpoint3d      *dPdU,    /* <= 1st partial in U direction, or NULL */
    Dpoint3d      *dPdV,    /* <= 1st partial in V direction, or NULL */
    Dpoint3d      *dPdUU,   /* <= 2nd partial in U direction, or NULL */
    Dpoint3d      *dPdVV,   /* <= 2nd partial in V direction, or NULL */
    Dpoint3d      *dPdUV,   /* <= mixed 2nd partial, or NULL */
    Dpoint3d      *normP,   /* <= surface normal direction */
    Dpoint2d      *param,    /* => parameter of point to evaluate */
    MSBsplineSurface *surface /* => surface definition */
);
```

Description `mdlBspline_computeDerivativesSurface` calculates the position and partial derivatives of a B-spline surface at a particular uv parameter value pair. If the surface is rational, it also calculates the derivatives of the homogeneous coordinate (*weight*) function. It does perform the chain rule computations for the partial derivatives if the surface is rational.

position points to a `Dpoint3d` structure that contains the position of the desired surface at the u-v parameter value pair specified by *param*.

weight points to a `double` that contains the value of the homogenous coordinate of the desired surface at the u-v parameter value pair specified by *param*.

dPdU, if not `NULL`, points to a `Dpoint3d` structure that contains the first partial U derivative of the desired surface at the u-v parameter value pair specified by *param*.

dPdV, if not `NULL`, points to a `Dpoint3d` structure that contains the first partial V derivative of the desired surface at the u-v parameter value pair specified by *param*.

dPdUU, if not `NULL`, points to a `Dpoint3d` structure that contains the second partial U derivative of the desired surface at the u-v parameter value pair specified by *param*.

dPdVV, if not `NULL`, points to a `Dpoint3d` structure that contains the second partial V derivative of the desired surface at the u-v parameter value pair specified by *param*.

dPdUV, if not `NULL`, points to a `Dpoint3d` structure that contains the mixed second partial derivative of the desired surface at the u-v parameter value pair specified by *param*.

normP is the surface normal direction, i.e., cross product of the U-partial derivative and V-partial derivative.



`mdlBspline_computeDerivativesSurface` will not compute the surface normal direction unless both *dPdU* and *dPdV* were passed non-`NULL` values.

param is the uv parameter value pair where the surface will be evaluated.

surface points to the B-spline surface that will be evaluated.

Returns `mdlBspline_computeDerivativesSurface` is of type `void`. It returns no value.

See Also `mdlBspline_computeDerivatives`, `mdlBspline_frenetFrame`.

mdlBspline_extractCurveNormal

```
#include <mdlbspln.h>

int mdlBspline_extractCurveNormal
(
    Dpoint3d    *normal,           /* <= normal of planar curve */
    Dpoint3d    *position,        /* <= point in plane of curve */
    double      *planarDeviationP, /* <= deviation from planar */
    MSBsplineCurve *curve,        /* => input curve */
);
```

Description `mdlBspline_extractCurveNormal` calculates the normal and center of a plane containing a planar B-spline curve. The curve does not need to be exactly planar;

the function will compute an averaged normal. This function performs the same calculation as `mdlElmdscr_extractNormal`.

normal points to a structure which contains the normal of the plane of the curve.

position points to a structure which contains the a point in the plane of the curve centered relative the poles of the curve.

planarDeviationP points to a value indicating the deviation from planar. For a non-planar curve, this is the maximum deviation the curve is from the plane defined by *position* and *normal*. *planarDeviationP* can be used to check to see how close a curve is to being planar; if *planarDeviationP* is too large a value, it means the curve is too distant from being planar for the values of *normal* and *position* to be useful.

curve points to the `MSBsplineCurve` structure defining the curve.

Returns `mdlBspline_extractCurveNormal` returns `SUCCESS`.

See Also `mdlElmdscr_extractNormal`.

mdlBspline_closestEdge

```
#include <mdlbspln.h>

void mdlBspline_closestEdge
(
    Dpoint2d    *output,      /* <= output point */
    Dpoint2d    *input       /* => input point */
);
```

Description The `mdlBspline_closestEdge` function returns the point on the boundary of the parameter square ($0.0 \leq u \leq 1.0$), ($0.0 \leq v \leq 1.0$) that is closest to a given point in that square.

output points to a `Dpoint2d` that holds the returned value. It can point to the same structure to which *input* points.

input points to a `Dpoint2d` structure representing a point that lies in the parameter square ($0.0 \leq u \leq 1.0$), ($0.0 \leq v \leq 1.0$).

Returns `mdlBspline_closestEdge` is of type `void`. It returns no value.

See Also `mdlBspline_inBounds`.

mdlBspline_inBounds

```
#include <mdlbspln.h>

boolean mdlBspline_inBounds
(
    Dpoint2d      *point,          /* => uv point to test */
    MSBsplineSurface *surface      /* => surface to test against */
);
```

Description The mdlBspline_inBounds function determines whether a specified u-v parameter value pair lies within the trim boundaries of a surface.

point is the point that is to be checked.

surface is the surface against which to check the uv parameter point in *point*.

Returns mdlBspline_inBounds returns TRUE if the u-v parameter value pair specified by *point* lies on a part of the surface that is present. It returns FALSE if that value pair falls in a region that is trimmed.

See Also mdlBspline_closestEdge, mdlBspline_isValidBoundary [mdl1lib.m1].

mdlBspline_isValidBoundary [mdl1lib.m1]

```
#include <mdlbspln.h>
#include <mdl1lib.fdf>

boolean mdlBspline_isValidBoundary
(
    BsurfBoundary *bound          /* => boundary to check */
);
```

Description mdlBspline_isValidBoundary checks whether a BsurfBoundary structure contains a valid trim boundary.

bound is the boundary to be checked.

Returns mdlBspline_isValidBoundary returns TRUE if the first and last points of the trim boundary are identical. It returns FALSE if the boundary contains fewer than 4 points, if the boundary's array is not allocated, or if the first and last points are not identical.

See Also mdlBspline_isValidSurface [mdl1lib.m1], mdlBspline_isValidKnotVector [mdl1lib.m1].

mdlBspline_isValidKnotVector [mdl.lib.ml]

```
#include <mdlbspln.h>
#include <mdl.lib.fdf>

boolean mdlBspline_isValidKnotVector
(
  double *knots          /* => knot vector to check */
  int    numPoles,        /* => number of poles of B-spline */
  int    order,           /* => order of B-spline */
  int    closed           /* => periodicity of B-spline */
);
```

Description mdlBspline_isValidKnotVector checks whether a full knot vector agrees with the parameters defining it.

knots is the knot vector to be checked.

numPoles indicates the number of poles in the curve or surface in a particular direction, U or V.

order indicates the order of the curve or surface in a particular direction, U or V.

closed indicates the periodicity (TRUE for closed) of the curve or surface in a particular direction, U or V.

Returns mdlBspline_isValidKnotVector returns TRUE if the full knot vector agrees with the supplied parameters. If the number of knots is wrong, or the sequence is not monotone increasing, then FALSE is returned.

See Also mdlBspline_isValidSurface [mdl.lib.ml], mdlBspline_isValidBoundary [mdl.lib.ml].

mdlBspline_isValidSurface [mdl.lib.ml]

```
#include <mselems.h>

boolean mdlBspline_isValidSurface
(
  MSElementDescr *edP    /* => element descriptor to check */
);
```

Description mdlBspline_isValidSurface checks whether the elements in an element descriptor define a valid B-spline surface. The format of a valid B-spline surface is defined in Appendix A of the MDL Programmers Guide.

edP points to the element descriptor to examine.

Returns mdlBspline_isValidSurface returns TRUE if the element descriptor contains a valid B-spline surface. Otherwise it returns FALSE.

See Also mdlBspline_isValidBoundary [mdl.lib.ml], mdlBspline_isValidKnotVector [mdl.lib.ml].

mdlBspline_isPhysicallyClosed

```
#include <mselems.h>
#include <mdlLib.fdf>

boolean mdlBspline_isPhysicallyClosed
(
  MSBsplineCurve   *curve   /* => curve to check */
);
```

Description mdlBspline_isPhysicallyClosed checks whether a B-spline curve is physically closed. A B-spline curve may be non-periodic, but still return TRUE if its first and last poles coincide.

curve points to the B-spline curve structure to examine.

Returns mdlBspline_isPhysicallyClosed returns TRUE if the curve is a physically closed B-spline curve. Otherwise it returns FALSE.

See Also mdlBspline_isValidSurface [mdlLib.m1],
mdlBspline_isPhysicallyClosedCurve [mdlLib.m1],
mdlBspline_isPhysicallyClosedSurface, mdlBspline_isValidBoundary
[mdlLib.m1], mdlBspline_isValidKnotVector [mdlLib.m1].

mdlBspline_isPhysicallyClosedCurve [mdlLib.m1]

```
#include <mselems.h>

boolean mdlBspline_isPhysicallyClosedCurve
(
  MSElementDescr   *edP     /* => element descriptor to check */
);
```

Description mdlBspline_isPhysicallyClosedCurve checks whether the elements in an element descriptor define a physically closed B-spline curve. A B-spline curve may be non-periodic, but still return TRUE if its first and last poles coincide.

edP points to the element descriptor to examine.

Returns mdlBspline_isPhysicallyClosedCurve returns TRUE if the element descriptor contains a physically closed B-spline curve. Otherwise it returns FALSE.

See Also mdlBspline_isValidSurface [mdlLib.m1], mdlBspline_isPhysicallyClosed,
mdlBspline_isPhysicallyClosedSurface, mdlBspline_isValidBoundary
[mdlLib.m1], mdlBspline_isValidKnotVector [mdlLib.m1].

mdlBspline_isPhysicallyClosedSurface

```
#include <mselems.h>

void mdlBspline_isPhysicallyClosedSurface
(
    boolean          *uClosed, /* <=> u direction status */
    boolean          *vClosed, /* <=> v direction status */
    MSBsplineSurface *surface /* => surface to check */
);
```

Description `mdlBspline_isPhysicallyClosedSurface` checks whether a B-spline surface is physically closed in either parameter direction. A B-spline surface may be non-periodic, but still be closed in the v/u direction if its first and last rows/columns of poles coincide.

uClosed will be set to TRUE if the first and last columns of poles of *surface* are the same.

vClosed will be set to TRUE if the first and last rows of poles of *surface* the same.

surface points to the B-spline surface structure to examine.

Returns `mdlBspline_isPhysicallyClosedSurface` is of type void.

See Also `mdlBspline_isValidSurface` [mdl1lib.m], `mdlBspline_isPhysicallyClosed`, `mdlBspline_isPhysicallyClosedSurface`, `mdlBspline_isValidBoundary` [mdl1lib.m], `mdlBspline_isValidKnotVector` [mdl1lib.m].

mdlBspline_intersectCurves

```
int mdlBspline_intersectCurves
(
    Dpoint3d      **intPoints, /* <=> intersectn point(s), or NULL */
    double        **paramo,    /* <=> param(s) on curve0, or NULL */
    double        **param1,    /* <=> param(s) on curve1, or NULL */
    int           *numPoints,   /* <= number of points returned */
    MSBsplineCurve *curve0,    /* => curve to intersect */
    MSBsplineCurve *curve1,    /* => curve to intersect */
    RotMatrix     *rotMatrix   /* => view to calc. in, or NULL */
);
```

Description This function returns the parameters and location of the all intersection points that are common to two B-spline curves. This routine allocates memory for the arrays *intPoints*, *param0* and *param1*; it returns the number of entries in these arrays in *numPoints*.

intPoints is a pointer to the array of intersection points to be allocated, or NULL if this information is not desired.

param0 is a pointer to the array of parameters of the intersection points on *curve0* to be allocated, or NULL if this information is not desired.

param1 is a pointer to the array of parameters of the intersection points on *curve1* to be allocated, or NULL if this information is not desired.

numPoints is the number of intersection points found.

curve0 and *curve1* are the input curves to test.

rotMatrix points to a view rotation matrix used to rotate and flatten the input curves if intersection points as seen from a particular view are desired. If *rotMatrix* == NULL, the input curves will be intersected in 3D space.

Returns The mdlBspline_intersectCurves function returns SUCCESS (zero) to indicate successful completion. It returns ERROR (one) if the curves do not intersect, or MDLERR_INSFMEMORY if memory is insufficient for the allocations.

See Also mdlBspline_intersectSegment, mdlBspline_intersectSurfaces.

mdlBspline_intersectSegment

```
int mdlBspline_intersectSegment
(
    Dpoint3d    *intPoint,      /* <=> intersection point, or NULL */
    double      *param,        /* <=> parameter on curve, or NULL */
    Dvector3d    *segment,      /* => segment to test */
    MSBsplineCurve *curve,      /* => curve to intersect */
    RotMatrix    *rotMatrix     /* => view to calc. in, or NULL */
);
```

Description This function returns the parameter and location of the intersection point between a segment and a B-spline curve.

intPoint is a pointer to the intersection point calculated, or NULL if this information is not desired.

param is a pointer to the parameter of the intersection points on *curve*, or NULL if this information is not desired.

segment is a pointer to the Dvector3d structure holding the segment to test.

curve is the input curve to test.

rotMatrix points to a view rotation matrix used to rotate and flatten the input curves if the intersection point as seen from a particular view is desired. If *rotMatrix* == NULL, the curve and segment will be intersected in 3D space.

Returns The mdlBspline_intersectSegment function returns TRUE (one) to indicate successful completion. It returns ERROR (one) if the curves do not intersect, or MDLERR_INSFMEMORY if memory is insufficient for the allocations.

See Also mdlBspline_intersectCurves, mdlBspline_intersectSurfaces.

mdlBspline_intersectSurfaces

```
#include <mdlbsp1n.h>

int mdlBspline_intersectSurfaces
(
    PointList          **pointLists,          /* <=> intersctn lines, or NULL */
    int                *numPointLists,        /* <=> number of lines */
    BsurfBoundary      **surf0Bounds,         /* <=> surface0 bounds or NULL */
    int                *numSurf0Bounds,        /* <=> # of bounds on surface0 */
    BsurfBoundary      **surf1Bounds,         /* <=> surface1 bounds, or NULL */
    int                *numSurf1Bounds,        /* <=> # of bounds on surface1 */
    MSBsplineSurface   *surface0,             /* => surface to intersect */
    MSBsplineSurface   *surface1,             /* => surface to intersect */
    double              tolerance,             /* => used to calc intersectn */
    int                 displayFlag            /* => disp marching; boolean */
);
```

Description The `mdlBspline_intersectSurfaces` function calculates the intersection curve(s) of two B-spline surfaces. It calculates the 3D points along the intersection curve, as well as the 2D parametric trim boundaries for the B-spline surfaces. Since exact intersection curves are not representable as low order B-splines, this routine uses a marching algorithm to approximate the intersection curves. The supplied tolerance defines the accuracy of this process. Arrays are returned because two surface may intersect in multiple intersection curves. This routine allocates memory for the returned arrays as necessary.

pointLists points to the array of `PointList` data structures returned. `PointLists`, defined in `mdlbsp1n.h`, are similar to `BsurfBoundary`s but contain `Dpoint3ds` instead of `Dpoint2ds`. These 3D points are the approximation to the intersection curve(s) calculated by the routine. This may be set to `NULL` if this information is not desired.

numPointLists returns the number of entries in the *pointLists* array. If *pointLists* == `NULL`, this may also be set to `NULL`.

surf0Bounds points to the array of `BsurfBoundary` data structures returned. These are the trim boundaries for *surface0* that are calculated by the function. This may be set to `NULL` if this information is not desired.

numSurf0Bounds returns the number of entries in the *surf0Bounds* array. If *surf0Bounds* == `NULL`, this may also be set to `NULL`.

surf1Bounds points to the array of `BsurfBoundary` data structures returned. These are the trim boundaries for *surface1* that are calculated by the function. This may be set to `NULL` if this information is not desired.

numSurf1Bounds returns the number of entries in the *surf1Bounds* array. If *surf1Bounds* == `NULL`, this may also be set to `NULL`.

surface0 points to one of the surfaces to intersect.

surface1 points to the other surface to intersect.

tolerance specifies the accuracy of the approximation process used to calculate the intersection curve(s). Smaller values yield more accurate results but at the cost of greater number of points in the output arrays.

displayFlag indicates whether or not to display the marching process as it describes the intersection curve(s). If *displayFlag* == TRUE then the points along the intersection will be displayed in HILITE mode as they are determined.

Returns The mdlBspline_intersectSurfaces function returns SUCCESS (zero) to indicate successful completion. It returns MDLERR_INSFMEMORY if there is not enough memory for the allocations. It returns MDLERR_BADPARAMETER if *tolerance* == 0.0.

See Also mdlBspline_intersectOffsetSurfaces, mdlBspline_intersectCurves.

mdlBspline_intersectOffsetSurfaces

```
#include <mdlbspIn.h>

int mdlBspline_intersectOffsetSurfaces
(
    PointList      **pointLists, /* <=> intersection lines or NULL */
    PointList      **norm0Lists, /* <=> normals to surf0 or NULL */
    PointList      **norm1Lists, /* <=> normals to surf1 or NULL */
    int            *numPointLists, /* <= number of lines */
    MSBsplineSurface *surface0, /* => root surface to intersect */
    int            *numsurf0, /* => number of bounds for surf0 */
    MSBsplineSurface *surface1, /* => root surface to intersect */
    int            *numsurf1, /* => number of bounds for surf1 */
    double          distance0, /* => to offset surface0 */
    double          distance1, /* => to offset surface1 */
    double          tolerance /* => used in calc'ing intrsectn */
);
```

Description The mdlBspline_intersectOffsetSurfaces function calculates the intersection curve(s) of the offset surfaces of two B-spline surfaces. It is more efficient and accurate to use this routine than to offset the surfaces individually and then intersect them. This routine fundamentally uses the same algorithm as mdlBspline_intersectSurfaces. This routine allocates memory for the returned arrays as necessary.

pointLists points to the array of PointList data structures returned. PointLists, defined in mdlbspIn.h, are similar to BsurfBoundaryrs but contain Dpoint3ds instead of Dpoint2ds. These 3D points are the approximation to the intersection curve(s) calculated by the routine. This may be set to NULL if this information is not desired.

norm0Lists points to the array of PointList data structures returned. These contain the normal vectors of the offset of *surface0* at the intersection points returned in *pointLists* array. This may be set to NULL if this information is not desired.

norm1Lists points to the array of `PointList` data structures returned. These contain the normal vectors of the offset of *surface1* at the intersection points returned in *pointLists* array. This may be set to `NULL` if this information is not desired.

numPointLists returns the number of entries in the above array.

surface0 and *surface1* point to the surfaces to intersect. *numsurf0* and *numsurf1* are the number of boundaries for *surface0* and *surface1*, respectively.

distance0 and *distance1* specify the distance by which to offset *surface0* and *surface1* in their respective normal directions before intersecting.

tolerance specifies the accuracy of the approximation process used to calculate the intersection curve(s). Smaller values yield more accurate results but at the cost of greater number of points in the output arrays.

Returns `mdlBspline_intersectOffsetSurfaces` returns `SUCCESS` (zero) to indicate successful completion. It returns `MDLERR_INSFMEMORY` if there is not enough memory for the allocations. It returns `MDLERR_BADPARAMETER` if *tolerance* == 0.0.

See Also `mdlBspline_intersectSurfaces`, `mdlBspline_intersectCurves`.

mdlBspline_cuspPoints

```
int mdlBspline_cuspPoints
(
    Dpoint3d      **cuspPoints, /* <=> cusp pnt(s) on curve or NULL */
    double        **params,     /* <=> param(s) on curve, or NULL */
    int           *numPoints,    /* <=> number of points returned */
    MSBsplineCurve *curve,      /* => curve to check */
    RotMatrix     *rotMatrix     /* => view to calc. in, or NULL */
);
```

Description This function returns the parameters and location of the all cusp points of a B-spline curve. This routine allocates memory for the arrays *cuspPoints* and *param*; it returns the number of entries in these arrays in *numPoints*.

cuspPoints is a pointer to the array of cusp points to be allocated, or `NULL` if this information is not desired.

param is a pointer to the array of parameters of the cusp points on *curve* to be allocated, or `NULL` if this information is not desired.

numPoints is the number of cusp points found.

curve is the input curve to test.

rotMatrix points to a view rotation matrix used to rotate and flatten the input curve if cusp points as seen from a particular view are desired. If *rotMatrix* == `NULL`, the input curve will be tested in 3D space.

Returns The mdlBspine_cuspPoints function returns SUCCESS (zero) to indicate successful completion. It returns ERROR (one) if the curve does not have any cusp points, or MDLERR_INSFMEMORY if memory is insufficient for the allocations.

See Also mdlBspine_inflectionPoints.

mdlBspine_inflectionPoints

```
int mdlBspine_inflectionPoints
(
    Dpoint3d    **inflPoints, /* <=> inflection point(s), or NULL */
    double      **params,    /* <=> param(s) on curve, or NULL */
    int          *numPoints,  /* <=> number of points returned */
    MSBspineCurve *curve,    /* => curve to check */
    RotMatrix    *rotMatrix   /* => of view to calculate in, or NULL */
);
```

Description This function returns the parameters and location of the all inflection points of a B-spline curve. This routine allocates memory for the arrays *inflPoints* and *param*; it returns the number of entries in these arrays in *numPoints*.

inflPoints is a pointer to the array of inflection points to be allocated, or NULL if this information is not desired.

param is a pointer to the array of parameters of the inflection points on *curve* to be allocated, or NULL if this information is not desired.

numPoints is the number of inflection points found.

curve is the input curve to test.

rotMatrix points to a view rotation matrix used to rotate and flatten the input curve if inflection points as seen from a particular view are desired. If *rotMatrix* == NULL, the input curve will be tested in 3D space.

Returns The mdlBspine_inflectionPoints function returns SUCCESS (zero) to indicate successful completion. It returns ERROR (one) if the curve does not have any inflection points, or MDLERR_INSFMEMORY if memory is insufficient for the allocations.

See Also mdlBspine_cuspPoints.

mdlBspline_extractCapAsCurve, mdlBspline_extractCapAsCurves

```
int mdlBspline_extractCapAsCurve
(
MSBsplineCurve  *curve,          /* <= cap curve (possibly disjoint) */
MSBsplineSurface *surface,       /* => input surface */
int              lastRow         /* => FALSE = start of surface */
);
int mdlBspline_extractCapAsCurves
(
MSBsplineCurve  **curves,        /* <= output curves */
int              nCurves,        /* <= number of output curves */
MSBsplineSurface *surface,       /* => input surface */
int              lastRow         /* => FALSE = start of surface */
);
```

Description `mdlBspline_extractCapAsCurve` extracts either the first or the last rule line in U direction in the parameter space. The resulting curve could have several disjoint pieces.

`mdlBspline_extractCapAsCurves` extracts either the first or the last rule line from the surface in U direction in the parameter space. It creates a curve chain in which there are no disjoint pieces for each curve.

The resulting curve or curves are returned in *curve* or *curves*.

nCurves the number of curves in the array.

surface is the input from which to extract the rule line.

If *lastRow* is `TRUE`, the last rule line is extracted. Otherwise, the first rule line is extracted.

Returns `mdlBspline_extractCapAsCurve` and `mdlBspline_extractCapAsCurves` return `TRUE` to indicate successful completion.

See Also `mdlBspline_extractIsoCurve`.

mdlBspline_extractCapAsSurface [mdlplib.m]l

```
#include <mdlbspln.h>

int mdlBspline_extractCapAsSurface
(
MSBsplineSurface *outputCapSurf, /* <= resulting surface */
MSBsplineSurface *inputSurf,     /* => surface to calculate caps */
double           tolerance        /* => for boundary calculations */
int              lastRow,         /* => TRUE=last row of surface */
);
```

Description The `mdlBspline_extractCapAsSurface` function creates a trimmed, planar B-spline surface from a B-spline surface that is closed in the U parameter direction. It can be

used to create the caps of a solid of projection or revolution. It allocates memory as necessary for the surface.

outputCapSurf points to the planar MSBsplineSurface calculated by the function.

inputSurf points to the planar MSBsplineSurface closed in U to be capped by the function.

tolerance contains the value that is used in calculating trim boundaries. Smaller values yield more accurate boundaries, but at the cost of more points in them.

If *lastRow* == TRUE then the *v* = 1.0 end of *output* will be capped. Otherwise, the *v* = 0.0 end will be capped.

Returns mdlBspline_trimmedPlaneFromCurves returns SUCCESS (zero) to indicate successful completion or MDLERR_INSFMEMORY if memory is insufficient for the allocations.

See Also mdlBspline_convertToPlanarSurface [mdlilib.mll],
mdlBspline_trimmedPlaneFromCurves [mdlilib.mll].

mdlBspline_meshSurface

```
include <mdlbspln.h>

int mdlBspline_meshSurface
(
    int          (*meshFunction)(),          /* => mesh function */
    int          (*triangleFunction)(),      /* => triangle strip function */
    double       tolerance,                  /* => tolerance */
    int          toleranceMode,              /* => tolerance mode */
    Transform    *toleranceTransformP,      /* => tolerance transform */
    Dpoint3d     *toleranceCameraP,         /* => tolerance camera position */
    double       toleranceFocalLength,      /* => tolerance focal length */
    Dpoint2d     *parameterScale,           /* => parameter scale */
    MSBsplineSurface *surfaceP,             /* => surface to mesh */
    boolean      normalsRequired,           /* => TRUE to return normals */
    boolean      parametersRequired,        /* => TRUE to return parameters */
    void         *userDataP                 /* => user data */
);

int meshFunction
(
    Dpoint3d *pointP,                        /* => mesh vertices */
    Dpoint3d *normalP,                      /* => mesh normals */
    Dpoint2d *paramP,                       /* => mesh parameters (uv) */
    int      nColumns,                      /* => number of columns */
    int      nRows,                        /* => number of rows */
    void     *userDataP                    /* => user data pointer */
);
```

```
int triangleFunction
(
    Dpoint3d *pointP,           /* => triangle strip vertices */
    Dpoint3d *normalP,         /* => triangle strip normals */
    Dpoint2d *paramP,          /* => triangle strip vertices */
    int      nPoints,          /* => # of points (triangles + 2) */
    void      *userDataP       /* => user data pointer */
);
```

Description mdlBspline_meshSurface computes a mesh of quadrilaterals and triangles that approximates the B-spline surface *surfaceP*. *MicroStation's* meshing algorithm uses a rectangular quadrilateral mesh to represent the interior portion of surfaces and “covers” the surface edges to ensure that no cracking occurs between adjacent surface patches. The quadrilateral meshes are passed to the user function, *meshFunction* and the coving triangles are passed to *triangleFunction*. The vertex normals and parametric values are computed and passed to the user functions if *normalsRequired* and *parametersRequired* are non-zero.

The coarseness of the approximating mesh is determined by the tolerance parameters. The *toleranceMode* controls the manner in which the approximation is computed. The STROKETOL_ definitions in msdefs.h may be combined to produce the desired tolerance mode (i.e., STROKETOL_ChoordHeight | STROKETOL_XYProjection produce chord height tolerance on the xy projection of the mesh). For most applications, *toleranceMode* should be set to STROKETOL_ChoordHeight in this case, only the value of tolerance is used and NULL may be passed for *toleranceTransformP* and *toleranceCameraP*.

If STROKETOL_ChoordHeight is specified, the mesh approximation will deviate from the true surface by a maximum of *tolerance*.

If STROKETOL_StrokeSize is specified, the maximum size of a quadrilateral or triangle is specified by *tolerance*. This mode is not currently supported.

If STROKETOL_XYProjection is specified, only the X-Y projection of the surface (after transformation by *toleranceTransformP*) is used to compute the tolerance parameters. The Z value is ignored. This mode is most useful for minimizing the number of facets for rendering.

If STROKETOL_Perspective is specified, the tolerance is determined by transforming the surface by *toleranceTransformP* and then projecting to the x-y image plane specified by *toleranceCameraP* and *toleranceFocalLength*.

If *parameterScaleP* is NULL, the parameter values for a surface will vary between 0.0 and 1.0. If *parameterScaleP* is specified, the parameter values are scaled by the surface size and divided by the parameter scale. This allows for specification of texture map size in real world coordinates and is used to implement the absolute texture mapping in *MicroStation's* rendering.

userDataP is a pointer to user data that is passed through to *meshFunction* and *triangleFunction* and is not otherwise used.

meshFunction receives a rectangular array of vertices (*pointP*) and optionally, normal (*normalP*) and parameter values (*parameterP*). The values for a row are stored consecutively. If *meshFunction* returns a non-zero value, the mesh generation is aborted and this value is returned to the calling routine.

triangleFunction receives an array of vertices (*pointP*) and optionally, normal (*normalP*) and parameter values (*parameterP*). The *nPoints* vertices represent *nPoints-2* triangles with the first triangle represented by the first three vertices and each additional triangle represented by an additional point and the two previous vertices. If *triangleFunction* returns a non-zero value, the mesh generation is aborted and this value is returned to the calling routine.

Returns All functions return SUCCESS, an error code (see mdlerrs.h), or a non-zero value if one is returned by either *meshFunction* or *triangleFunction*.

See Also mdlBspline_evaluateCurve, meshsurf sample application.

mdlBspline_edgeCode

```
int mdlBspline_edgeCode
(
    Dpoint2d    *uv,           /* => parameter to test */
    double tolerance          /* => tolerance value */
);
```

Description mdlBspline_edgeCode determines if the point specified by *uv* is on one of the four edges in parameter space within the given tolerance.

uv is the point to test.

tolerance is the tolerance value, defined by how close *uv* is to zero. See below for a description of how it is used.

mdlBspline_edgeCode returns:

Return value	When
U0_EDGE	<i>uv->x</i> is closer than <i>tolerance</i> to zero.
U1_EDGE	<i>uv->x</i> is closer than <i>tolerance</i> to one.
V0_EDGE	<i>uv->y</i> is closer than <i>tolerance</i> to zero.
V1_EDGE	<i>uv->y</i> is closer than <i>tolerance</i> to one.
NO_EDGE	<i>uv</i> is not on any edge within <i>tolerance</i> .

See Also mdlBspline_isDegenerateEdge.

mdlBspline_isDegenerateEdge

```
boolean mdlBspline_isDegenerateEdge
(
  int          edgeCode,      /* => edge to check */
  MSBsplineSurface *surf,     /* => surface to check */
  double       tolerance      /* => tolerance value */
);
```

Description The `mdlBspline_isDegenerateEdge` function checks whether an edge of the surface degenerates to a single point.

edgeCode can be `U0_EDGE`, `U1_EDGE`, `V0_EDGE` or `V1_EDGE` to represent different edges of the surface.

surf points to the input B-spline surface.

tolerance, specified in working units, is used to determine the closeness between two points when checking whether an edge degenerates to a point.

Returns `mdlBspline_isDegenerateEdge` returns `TRUE` if the edge degenerates to a single point. Otherwise, it returns `FALSE`.

See Also `mdlBspline_getEdgeFromSurface`.

mdlBspline_isSolid

```
boolean mdlBspline_isSolid
(
  MSBsplineSurface *surf,     /* => surface to check */
  double           tolerance   /* => tolerance value */
);
```

Description The `mdlBspline_isSolid` function checks whether a B-spline surface encloses a valid space. For example, a surface which is closed in both the U and V directions is considered to be a solid. This function also tries to determine that if one direction is closed, whether the two edges in the other direction degenerate to points.

surf points to the input surface.

tolerance, specified in working units, is used to determine the closeness between two points when checking whether an edge encloses a space.

Returns `mdlBspline_isSolid` returns `TRUE` if it encloses valid space. Otherwise, it returns `FALSE`.

See Also `mdlBspline_isDegenerateEdge`.

10

Dimension Functions

The dimension element is used for all types of dimensioning - linear, angular, radial, rectangular coordinate (ordinate) and labels.

The dimension element is divided into sections on parameters, options, points and text:

- The parameters section of the dimension element contains the dimension settings common to all or most dimensions. In general, this is the information from the four dimension settings boxes labeled Dimension Attributes, Geometry, Text Format and Tool Settings.
- The options section of the dimension element contains settings that are needed only in special cases and would result in wasted space if they were included in all dimension elements. Most of these settings are defined in the dimension settings boxes labeled Custom Symbols, Custom Terminators and Tolerance.
- The points section of the dimension element contains an array of points required to define the dimension. Unlike most other elements, dimension elements use `DPoint3d` or `AssocPoint` points. A `DPoint3d` point contains fixed coordinates, while an `AssocPoint` point references a location on another element. The point array can contain any mix of real and associative points. The meaning of each point depends on the dimension type specified in the `mdlDim_create [mdlLib.mli]` function. See the requirements section below for more information.
- The text portion of the dimension element contains literal strings used to override the default dimension text. This portion is an optional part of the dimension element. Only the `mdlDim_setStrings [mdlLib.mli]` function can add it.

Dimension types and requirements

The following table lists the available dimension types and the requirements for each. The type names on the left are constants defined in `mdlDim.h`. The `mdlDim_create [mdlLib.m]` function uses these constants. The type names closely relate to the MicroStation dimension commands. Refer to the user documentation on dimension commands for a description of each dimension type.

Dimension type	Requirements
DIMTYPE_SIZE_ARROW DIMTYPE_SIZE_STROKE DIMTYPE_CUSTOM_LINEAR DIMTYPE_LOCATE_SINGLE DIMTYPE_LOCATE_STACKED	Point 0 - base of first witness line. Point 1 - base of second witness line. Base points of additional witness lines are optional. Use the <code>mdlDim_defineRotMatrix</code> function to set the rotation matrix. Set the dimension height using the <code>mdlDim_setHeight [mdlLib.m]</code> function.
DIMTYPE_ANGLE_SIZE DIMTYPE_ANGLE_LOCATION	Point 0 - center point of arc or angle. Point 1 - base of first witness line. Point 2 - base of second witness line. Base points of additional witness lines are optional. The rotation matrix is set automatically. Set the dimension height using the <code>mdlDim_setHeight [mdlLib.m]</code> function.
DIMTYPE_ANGLE_LINES DIMTYPE_ANGLE_AXIS_X DIMTYPE_ANGLE_AXIS_Y DIMTYPE_ARC_SIZE DIMTYPE_ARC_LOCATION	Point 0 - angle vertex. Point 1 - base of first witness line. Point 2 - base of second witness line. The rotation matrix is set automatically. Set the dimension height using the <code>mdlDim_setHeight [mdlLib.m]</code> function.
DIMTYPE_RADIUS DIMTYPE_RADIUS_EXTENDED DIMTYPE_DIAMETER DIMTYPE_DIAMETER_EXTENDED	Point 0 - center point of circular element. Point 1 - intersection of dimension and element. Point 2 - end of leader line. Optional leader vertices can be added between points one and two. The rotation matrix must be extracted from the arc or circular ellipse and inserted in the dimension with <code>mdlDim_setRotMatrix</code> . Only point 0 can be an associate point.
DIMTYPE_ORDINATE	Point 0 - base of first (zero) ordinate line. Point 1 - base of second ordinate line. Base points of additional ordinate lines are optional. Use the <code>mdlDim_defineRotMatrix</code> function to set the rotation matrix. The length of each ordinate line is determined by the text offset that must be set with the <code>mdlDim_setTextOffset [mdlLib.m]</code> function.
DIMTYPE_CENTER	Point 0 - origin of center mark. The rotation matrix is set automatically.

Dimension type	Requirements
DIMTYPE_LABEL_LINE	Point 0 - origin of labeled line segment. Point 1 - end of labeled line segment.
DIMTYPE_NOTE	Point 0 - origin of leader line. Point 1 - end of leader line. Optional leader vertices can be added between points zero and one. The rotation matrix is set automatically.

The following table lists dimension functions:

Function	Used to
mdlDim_create [mdl1lib.m1]	create a MicroStation dimension element.
mdlDim_getParam	return parameter values from dimension elements.
mdlDim_setParam	set dimension element parameters.
mdlDim_defineRotMatrix [mdl1lib.m1]	define and set dimension element rotation matrix for linear and ordinate dimensions.
mdlDim_getRotMatrix	extract rotation matrix from dimension element.
mdlDim_setRotMatrix	set dimension element rotation matrix.
mdlDim_getHeight [mdl1lib.m1]	get dimension height (length of first witness line) for linear and angular dimensions.
mdlDim_setHeight [mdl1lib.m1]	set dimension height (length of first witness line) for linear and angular dimensions.
mdlDim_setTextOffset [mdl1lib.m1]	set dimension text justification or offset.
mdlDim_insertPoint [mdl1lib.m1]	insert a dimension vertex.
mdlDim_deletePoint	remove a vertex from a dimension element.
mdlDim_extractPoints	retrieve points from a dimension element.
mdlDim_getStrings [mdl1lib.m1]	retrieve dimension strings.
mdlDim_setStrings [mdl1lib.m1]	add or replace dimension strings.
mdlDim_getElementDescr	create an element descriptor of the IGDS primitive representation a dimension element.
mdlDim_validate [mdl1lib.m1]	validate a dimension element.
mdlDim_setExtensionLine [mdl1lib.m1]	change the on/off state of any extension line in the dimension element.
mdlDim_getExtensionLine [mdl1lib.m1]	return the on/off state of an extension line in the dimension element.

Example

See `dimline.mc`.

mdlDim_create [mdlLib.m1]

```
#include <mdlDim.h>

int mdlDim_create
(
  MSElementUnion    *newDim,          /* <= new dimension element */
  MSElementUnion    *seedDim,        /* => seed dimension element or NULL */
  RotMatrix          *rMatrix,         /* => element rotation matrix */
  int                 dimType,         /* => type of dimension to create */
  int                 dimView          /* => view for text orientation */
);
```

Description `mdlDim_create` creates an empty MicroStation dimension element in *newDim*. If *seedDim* is not NULL, all dimension settings are taken from *seedDim*. Otherwise, the active dimension settings are used.

dimType determines the type of dimension created. *dimType* must be set to one of the constants listed in the “Dimension types and requirements” section above.

dimView specifies the view (0-7) used for text orientation.

The dimension element rotation matrix will be taken from *rMatrix*. If the rotation matrix is not required or will be set later, pass NULL for *rMatrix*.



After `mdlDim_create` is called, the required points must be added to the element with `mdlDim_insertPoint [mdlLib.m1]`. The modified dimension element must be validated with `mdlDim_validate [mdlLib.m1]` before it is written to the file.

Returns The `mdlDim_create` function returns `SUCCESS` if a valid dimension element is created. Otherwise, an error status is returned.

See Also `mdlDim_insertPoint [mdlLib.m1]`, `mdlDim_validate [mdlLib.m1]`.

mdlDim_getParam, mdlDim_setParam

```
#include <msdim.fdf>

int mdlDim_getParam
(
  void            *param,          /* <= dimension parameter values */
  MSElementUnion *dim,            /* => dimension element */
  int             paramName        /* => dimension parameter name */
);

int mdlDim_setParam
```

```
(
MSElementUnion    *dim,          /* <=> dimension element */
void               *param,        /* => dimension parameter values */
int                paramName      /* => dimension parameter name */
);
```

Description mdlDim_getParam retrieves parameter values from a MicroStation dimension element.

The parameters returned in *param* are determined by the value of *paramName* as defined in the following table. The mdlDim_setParam function changes parameter values in a MicroStation dimension element. The new parameter values are given in *param*. The parameters to be changed are determined by the value of *paramName*.

paramName	Type for <i>param</i>
DIMPARAM_TEXT	DimParamText *
DIMPARAM_FORMAT	DimParamFormat *
DIMPARAM_GEOMETRY	DimParamGeometry *
DIMPARAM_FLAGS	DimParamFlags *
DIMPARAM_TOLERANCE	DimParamTolerance *
DIMPARAM_TEMPLATE	DimParamTemplate *
DIMPARAM_INFO	DimParamInfo *
DIMPARAM_TERMINATOR	DimParamTerm *

See the “Dimension Parameters” section of the MDL header file mdlDim.h for a complete list of the values allowed for *param*.

Returns mdlDim_getParam and mdlDim_setParam return SUCCESS if successful and an error value if they could not perform their task.

mdlDim_defineRotMatrix [mdlLib.ml]

```
#include <mdlDim.h>

void mdlDim_defineRotMatrix
(
RotMatrix    *rMatrix,          /* <= dimension rotation matrix */
MSElementUnion *dim,          /* <=> dimension element */
DPoint3d     *point1,          /* => first point */
DPoint3d     *point2,          /* => second point */
DPoint3d     *point3           /* => third point */
);
```

Description mdlDim_defineRotMatrix defines and sets a dimension rotation matrix to use in a dimension element of one of the following types: DIMTYPE_SIZE_ARROW,

DIMTYPE_SIZE_STROKE, DIMTYPE_CUSTOM_LINEAR, DIMTYPE_LOCATE_SINGLE, DIMTYPE_LOCATE_STACKED or DIMTYPE_ORDINATE.

The rotation matrix produced is based on the three input points and the current status of the DIMENSION AXIS setting. *point1*, *point2* and *point3* must be defined as the first three data points of the DIMENSION SIZE, DIMENSION LOCATION and DIMENSION ORDINATE commands.

This function automatically adds the rotation matrix to *dim*.

`mdlDim_setRotMatrix` does not need to be called after this function is called. If *rMatrix* is not NULL, a copy of the rotation matrix is returned there.



The modified dimension element must be validated with `mdlDim_validate [mdl1ib.m1]` before it is written to the file.

Returns `mdlDim_defineRotMatrix` is of type `void`. It returns no value.

See Also `mdlDim_getRotMatrix`, `mdlDim_setRotMatrix`, `mdlDim_validate [mdl1ib.m1]`.

mdlDim_getRotMatrix, mdlDim_setRotMatrix [mdl1ib.m1]

```
#include <mdlDim.h>

void mdlDim_getRotMatrix
(
  MSElementUnion  *dim,           /* => dimension element */
  RotMatrix        *rMatrix       /* <= dimension rotation matrix */
);

void mdlDim_setRotMatrix
(
  MSElementUnion  *dim,           /* <= dimension element */
  RotMatrix        *rMatrix       /* => dimension rotation matrix */
);
```

Description `mdlDim_getRotMatrix` extracts the rotation matrix from *dim* and returns it in *rMatrix*. `mdlDim_setRotMatrix` adds the rotation matrix *rMatrix* to the dimension element *dim*.

The modified dimension element must be validated with `mdlDim_validate [mdl1ib.m1]` before it is written to the file.

Returns The `mdlDim_getRotMatrix` and `mdlDim_setRotMatrix` functions are of type `void`; they return no values.

See Also `mdlDim_defineRotMatrix [mdl1ib.m1]`, `mdlDim_validate [mdl1ib.m1]`.

mdlDim_getHeight [mdlLib.mli]

```
#include <mdlDim.h>
#include <msdim.fdf>

int mdlDim_getHeight
(
    double          *height,      /* <= dimension height */
    MSElementUnion *dim,         /* => dimension element */
    int             option        /* => option (currently unused) */
);
```

Description mdlDim_setHeight extracts the height value from a linear or angular dimension. This function works only for the following dimension types: DIMTYPE_SIZE_ARROW, DIMTYPE_SIZE_STROKE, DIMTYPE_CUSTOM_LINEAR, DIMTYPE_LOCATE_SINGLE, DIMTYPE_LOCATE_STACKED, DIMTYPE_ANGLE_SIZE, DIMTYPE_ANGLE_LOCATION, DIMTYPE_ANGLE_LINES, DIMTYPE_ANGLE_AXIS_X and DIMTYPE_ANGLE_AXIS_Y.

The height of the dimension element will be returned in the location pointed to by *height*.

The *option* argument is currently unused. Pass zero to guarantee compatibility with future versions.

Returns The mdlDim_getHeight function returns SUCCESS if the requested operation completes successfully. Otherwise, an error status is returned.

See Also mdlDim_setTextOffset [mdlLib.mli], mdlDim_setHeight [mdlLib.mli].

mdlDim_setHeight [mdlLib.mli]

```
#include <mdlDim.h>

int mdlDim_setHeight
(
    MSElementUnion *dim,          /* <=> dimension to modify */
    double          height,        /* => dimension height */
    int             relative       /* => modification mode */
);
```

Description mdlDim_setHeight sets the height of a linear or angular dimension. This function works only for the following dimension types: DIMTYPE_SIZE_ARROW, DIMTYPE_SIZE_STROKE, DIMTYPE_CUSTOM_LINEAR, DIMTYPE_LOCATE_SINGLE, DIMTYPE_LOCATE_STACKED, DIMTYPE_ANGLE_SIZE, DIMTYPE_ANGLE_LOCATION, DIMTYPE_ANGLE_LINES, DIMTYPE_ANGLE_AXIS_X and DIMTYPE_ANGLE_AXIS_Y.

height is the length of the first witness line in the dimension.

relative determines how the dimension will react to modifications of the first witness line's base. If *relative* is TRUE, the witness line length remains constant. If *relative* is FALSE, the dimension line location remains constant, while the witness length changes. If *relative* is -1, the active setting is used.



The modified dimension element must be validated with `mdlDim_validate [mdl1lib.m1]` before it is written to the file.

Returns The `mdlDim_setHeight` function returns `SUCCESS` if the dimension is modified. Otherwise, an error status is returned.

See Also `mdlDim_getHeight [mdl1lib.m1]`, `mdlDim_setTextOffset [mdl1lib.m1]`, `mdlDim_validate [mdl1lib.m1]`.

mdlDim_setTextOffset [mdl1lib.m1]

```
#include <mdlDim.h>

int mdlDim_setTextOffset
(
  MSElementUnion *dim,      /* <=> dimension to modify */
  int segNo,                /* => dimension segment number */
  int justify,              /* => justification type */
  double offset             /* => offset from segment start */
);
```

Description `mdlDim_setTextOffset` alters the text's location in the direction of the dimension line.

segNo is the zero-based segment containing the dimension text. The first text string encountered in a dimension element is in segment zero.

offset is the distance from the dimension line origin to the dimension text origin measured along the dimension line.

justify must be set to one of the following constants:

Value	Description
DIMTEXT_LEFT	Dimension text is left-justified in the dimension line. The value of <i>offset</i> is ignored.
DIMTEXT_CENTER	Dimension text is centered in the dimension line. The value of <i>offset</i> is ignored.
DIMTEXT_RIGHT	Dimension text is right-justified in the dimension line. The value of <i>offset</i> is ignored.
DIMTEXT_OFFSET	Dimension text origin is determined by the value of <i>offset</i> .



The modified dimension element must be validated with `mdlDim_validate [mdl1lib.m1]` before it is written to the file.

Returns The `mdlDim_setTextOffset` function returns `SUCCESS` if the dimension is modified. Otherwise, an error status is returned.

See Also `mdlDim_setHeight [mdl1lib.m1]`, `mdlDim_validate [mdl1lib.m1]`.

mdlDim_insertPoint [mdlLib.mll]

```
#include <mdlDim.h>

int mdlDim_insertPoint
(
  MSElementUnion *dimP,          /* <=> dimension to modify */
  void *newPoint,                /* => DPoint3d or AssocPoint to add */
  int pointNo,                   /* => vertex number (zero-based) */
  int pointType                  /* => content of newPoint */
);
```

Description mdlDim_insertPoint inserts the point *newPoint* at the position *pointNo* in the element *dimP*.

If *pointNo* is -1, the new point is inserted as the last vertex in the dimension.

The value of *pointType* determines the content of *newPoint* and must be set to one of the following constants:

Value	Description
POINT_STD	<i>newPoint</i> is the address of a DPoint3d structure.
POINT_ASSOC	<i>newPoint</i> is the address of an AssocPoint created by one of the mdlAssoc_create... functions.



The modified dimension element must be validated with mdlDim_validate [mdlLib.mll] before it is written to the file.

Returns The mdlDim_insertPoint function returns SUCCESS if the point is inserted. Otherwise, an error status is returned.

See Also mdlDim_deletePoint.

mdlDim_deletePoint

```
#include <mdlDim.h>

int mdlDim_deletePoint
(
  MSElementUnion *dim,          /* <=> dimension to modify */
  int pointNo                    /* => vertex to delete (zero-based) */
);
```

Description mdlDim_deletePoint deletes the vertex at *pointNo* from the dimension element *dim*.



The modified dimension element must be validated with mdlDim_validate [mdlLib.mll] before it is written to the file.

Returns `mdlDim_deletePoint` returns `SUCCESS` if the point is deleted. Otherwise, an error status is returned.

See Also `mdlDim_insertPoint [mdlLib.m1]`, `mdlDim_validate [mdlLib.m1]`.

mdlDim_extractPoints

```
#include <mdlDim.h>

int mdlDim_extractPoints
(
    DPoint3d      *outPoints,      /* <= extracted points */
    MSElementUnion *dim,          /* => dimension element */
    int           fileName,        /* => design file (master/reference)*/
    int           pointNo,         /* => first point to extract */
    int           nPoints          /* => number of points to extract */
);
```

Description `mdlDim_extractPoints` extracts *nPoints* vertices from *dim* starting at vertex *pointNo*.

outPoints must be at least *nPoints** `sizeof(DPoint3d)` bytes. If a dimension vertex is represented by an associative point, the association is resolved before the point is added to *outPoints*.

The file that the dimension element came from (`MASTERFILE` for the master file and 1-256 for reference files) is passed in *fileName*.

Returns The `mdlDim_extractPoints` function returns the number of points that were extracted.

See Also `mdlDim_insertPoint [mdlLib.m1]`, `mdlDim_deletePoint`.

mdlDim_getStrings [mdlLib.m1]

```
#include <mdlDim.h>

int mdlDim_getStrings
(
    DimStrings *dimStrings,      /* <= dimension text strings */
    DimStringConfig *config,      /* <= string configuration */
    MSElementUnion *dim,        /* => dimension element */
    int           segNo          /* => segment containing strings */
);
```

Description `mdlDim_getStrings` returns the dimension text strings from segment *segNo* of *dim* in *dimStrings*. *segNo* is the zero-based segment of the dimension. For dimensions that have only one segment (one set of strings), *segNo* is 0.

To determine the configuration of strings in *dimStrings*, check the flags in *config*. If *config->dual* is `TRUE`, primary and secondary strings are present.

If *config->tolerance* is TRUE, there are tolerance strings. If *config->limit* is TRUE, the tolerance strings are upper and lower limits (*dimStrings->primary.limit.upper* and *dimStrings->primary.limit.lower*). If *config->limit* is FALSE, the tolerances are plus/minus values (*dimStrings->primary.pmtol.main*, *dimStrings->primary.pmtol.plus* and *dimStrings->primary.pmtol.minus*).

If *config->tolerance* is FALSE, only a single dimension string is used (*dimStrings->primary.single*).

Unless strings were added by `mdlDim_setStrings`, all strings will be blank. (In this case, the string is generated automatically).

An asterisk in a string is used as a place holder for the dimension value.

For more information, refer to the `DimStrings` and `DimStringConfig` type definitions in `mdldim.h`.

Returns The `mdlDim_getStrings` function returns `SUCCESS` if segment *segNo* exists and the strings were returned. Otherwise, an error status is returned.

See Also `mdlDim_setStrings [mdl1ib.m1]`.

mdlDim_setStrings [mdl1ib.m1]

```
#include <mdldim.h>

int mdlDim_setStrings
(
    MSElementUnion    *dim,           /* <=> dimension element */
    int                segNo,          /* => segment containing strings */
    DimStrings         *dimStrings    /* <= dimension text strings */
);
```

Description `mdlDim_setStrings` replaces the strings in *dim* at segment *segNo* with those in *dimStrings*. *segNo* is the zero-based segment of the dimension. For dimensions that have only one segment (one set of strings), *segNo* is 0. It is important to first call `mdlDim_getStrings [mdl1ib.m1]` to determine which strings are used in the dimension.

Any asterisk in the dimension string will be replaced by the dimension value during display.

The use of the `DimStrings` type is discussed with `mdlDim_getStrings [mdl1ib.m1]`.



The modified dimension element must be validated with `mdlDim_validate [mdl1ib.m1]` before it is written to the file.

Returns `mdlDim_setStrings` returns `SUCCESS` if segment *segNo* exists and the strings were added to the dimension. Otherwise, an error status is returned.

See Also `mdlDim_getStrings [mdl1lib.m1]`.

mdlDim_getElementDescr

```
#include <mdldim.h>

int mdlDim_getElementDescr
(
  MSElementDescr  **descr,      /* <= element descriptor */
  MSElementUnion  *dim,        /* => dimension element */
  int      fileName,    /* => design file (master or ref.) */
  int      graphGroup   /* => if TRUE, put element in graphic group */
);
```

Description `mdlDim_getElementDescr` converts a MicroStation dimension element to IGDS-8.8-compatible primitives that are then stored in an element descriptor. The address of the element descriptor is returned in *descr*.

The file that the dimension element came from (MASTERFILE for the master file and 1-256 for reference files) is passed in *fileName*.



It is the responsibility of the calling application to free the memory allocated to *descr* by calling `mdlElmdscr_freeAll`.

Returns `mdlDim_getElementDescr` returns `SUCCESS` if the element descriptor is created. Otherwise, an error status is returned.

See Also Element Descriptor Functions.

mdlDim_validate [mdl1lib.m1]

```
#include <mdldim.h>

int mdlDim_validate
(
  MSElementUnion  *dim      /* <=> dimension to validate */
);
```

Description `mdlDim_validate` sets the dimension range block and checks for invalid points and/or settings. If `mdlDim_insertPoint [mdl1lib.m1]`, `mdlDim_deletePoint` or `mdlDim_setHeight [mdl1lib.m1]` modified *dim*, `mdlDim_validate` must be called before the element is written to the file.

Returns `mdlDim_validate` returns `SUCCESS` if the element is validated. Otherwise, an error status is returned.

See Also `mdlDim_insertPoint [mdl1lib.m1]`, `mdlDim_deletePoint`, `mdlDim_setHeight [mdl1lib.m1]`, `mdlDim_setTextOffset [mdl1lib.m1]`.

mdlDim_setExtensionLine [mdlLib.mli]

```
#include <msdim.fdf>

Public int      mdlDim_setExtensionLine
(
MSElement      *pDim,          /* => dimension element */
int              index,          /* => extension line index */
int              extOn           /* => New state, 0=off, 1=on */
);
```

Description mdlDim_setExtensionLine can change the on/off state of any extension line in the dimension element, *pDim*. The parameter, *index* specifies the (zero based) index of the dimension line to be changed. Pass -1 to change the last extension line in the dimension.

The *extOn* parameter should be set to 1 to turn extension line display on or 0 to turn extension line display off.

Returns mdlDim_setExtensionLine returns SUCCESS if the operation was completed successfully or ERROR if the index is out of range.

mdlDim_getExtensionLine [mdlLib.mli]

```
#include <msdim.fdf>

Public int mdlDim_getExtensionLine
(
DimensionElm     *pDim,
int              index
);
```

Description mdlDim_getExtensionLine returns the on/off state of an extension line in the dimension element, *pDim*. The parameter, *index* specifies the index (zero based) of the requested extension line.

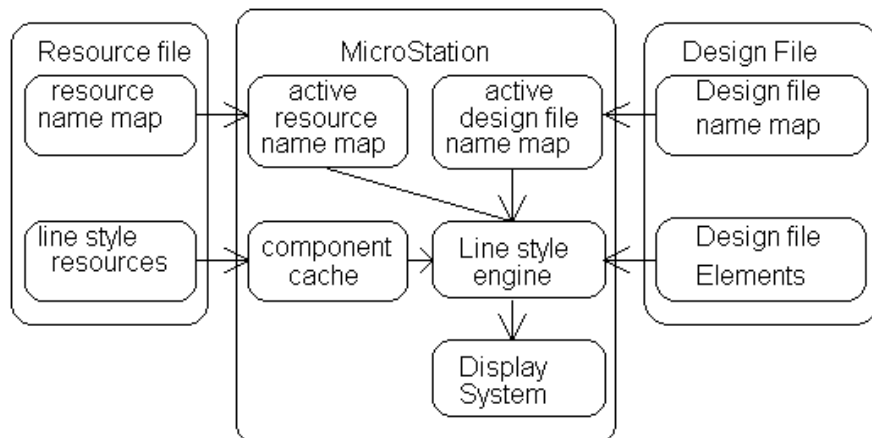
Returns mdlDim_getExtensionLine returns 0 if the extension line is off, 1 if the extension line is on or ERROR if the value of index is invalid.

Line Style Functions

A line style definition is a collection of one or more line style resources stored in a symbology resource file. The line style resources are used actively for display.

A change in a line style resource will be reflected in the display of an element that uses that resource as a part of a line style definition. Line style operation is analogous to the way a font library is used. In the same manner that changing a font library changes the appearance of text in your drawings, changing a line style library changes the appearance of other elements in your drawings.

Most simple line styles can be represented using a single resource, but many of the more complex line styles will require multiple resources in the definition. Whenever multiple resources are involved, there is always one “top level” resource that includes references to all other necessary resources. The resources in a line style definition are often referred to as the “components” of the line style.



Line Style Names

Because it is necessary to refer to line styles by name, each resource file that contains line style definitions must also contain a name map resource. The name map resource contains a list of records that are used to link names to specific resources. Each record in the name map resource contains a name (NULL terminated string) followed by the resource type and resource ID of the top level line style resource.

A similar system is used in each design file. Each element that is linked to a line style has user attribute data containing a line style ID number. Each line style ID number is represented in the design file name map. Each record in the design file name map contains a line style name (NULL terminated string) followed by a line style ID number that is unique only within the design file.

When MicroStation opens a line style resource file, it loads the contents of the name map resource(s) into the active resource name map. The active resource name map contains a complete list of available line style definitions and is the link between line style names and the line style definitions in the resource file.

When a design file is opened, MicroStation loads the contents of the design file name map element(s) into the active design file name map. This map is consulted when MicroStation needs to translate between line style names and the element line style ID numbers.

When a user selects a new line style using the LC= key-in, MicroStation searches the active resource name map for the requested style name. If the name is located, MicroStation then searches for a corresponding record in the target design file name map. If the name does not exist in the design file name map, a new record is generated containing a new line style ID number. Internal to MicroStation, the style ID number from the design file name map record is now the “active” line style.

When subsequent elements are created, MicroStation appends to each element a line style attribute linkage containing the style ID number and any active line style modifiers.

To display an element, MicroStation first extracts the style ID number (and modifiers) from the element attribute data. The line style ID number is then used to search the design file name map and locate the line style name. Once the name is known, it is used to search the active resource name map and locate the top level line style resource. After these operations are complete MicroStation has a link from the line style ID number (on an element) to a line style resource (file, type and ID) which can then be used for display.

Though it may seem otherwise, the complexity of linking a line style ID number to a line style resource does not cause any appreciable delay. Advanced caching and searching mechanisms make the initial operation very fast. Also, once a line style ID number has been linked to a resource, the link (a pointer into the component cache) is

saved. Subsequent display operations using the same line style ID number require no searching.

The line style engine

When MicroStation encounters an element containing line style attribute data, the attribute data and the element vectors are passed to the line style engine for display. The line style engine extracts the style ID number and the style modifiers from the attribute data. It then uses the line style definition combined with the line style modifiers to display the element vectors. The line style definition is located using the name map system (discussed above) and then loaded using the component cache (discussed below).

The line style component cache

The line style engine displays line styles using a processed form of the line style definition that it reads from the component cache. Line style component resources are converted into internal format and loaded into the cache as needed. Once in the cache, a component is available for all future display operations. Components in the cache are identified in exactly the same manner as the component resources themselves—that is, by resource file, resource type and resource ID.

If an application needs to access or modify the component cache, there are two important characteristics of the cache that need to be considered. When a top level component of a line style definition is loaded it automatically triggers the loading of all necessary sub-components. Also, components in the cache can be shared and are never duplicated. If more than one line style definition contains a reference to the same sub-component, the sub-component is only loaded once. Any modification of a component in the cache will effect all line styles that reference it.

The application interface

Line style definitions are stored in MicroStation resource files and can be created and manipulated like any other resources. You can use a text editor to create a resource text (.r) file and then compile it using the MicroStation resource compiler (rcomp). Alternatively, an MDL application can create and manipulate line style resources and resource files using the `mdlResource_...` MDL built-in functions.

The line style resource name maps are stored using the same format as a string list but with a resource type of `RTYPE_LineStyleNames` rather than `RTYPE_STRINGLIST`. The type definition for `LineStyleNames` is in `mslstyle.h`. They can be created and

manipulated using the `mdlStringList_...` MDL built-in functions. Each member of the string list contains a line style name and four or more info fields that define the resource to which the name is linked. The first info field (index 0) contains the line style component resource type. The second info field contains the line style component resource ID. The third and fourth fields are reserved for line style attributes and line style system ID numbers. The third and fourth fields should be set to zero to insure compatibility with future versions of MicroStation.

The design file name maps are created and maintained by MicroStation and it should not be necessary for an MDL application to operate on these name maps directly. If required, the design file name maps can be replaced using the `mdlLineStyle_nameSaveMap` MDL built-in function.

An application program can manipulate the line style component cache directly, using the `mdlLineStyle_cache_...` functions. Since the line style engine loads its components only from the cache, the application is able to bypass the line style resource file when necessary.

The `mdlLineStyle_name_...` functions provide direct access to the active resource name map. An application can use these functions to override or modify values in the name map or to bypass the resource file name map.

Access to the active line style settings (name and modifiers) is provided through new options to the `mdlParams_getActive` and `mdlParams_setActive` functions. Refer to the “Params” section of the “Settings” chapter for information on these functions and the line-style-specific parameters.

Setting and retrieving element line style information is provided by two new MDL built-in functions, `mdlElement_setLineStyle` and `mdlElement_getLineStyle`. Refer to the “Elements” section of the “CAD Engine” chapter for information on these functions.

Resource Definitions

The resource definitions for line styles can be found in the file `mslstyle.h`.

Line Style Functions

The following table lists the Line Style functions:

Function	Used to
<code>mdlLineStyle_nameInsert</code>	insert a name into the active name map.
<code>mdlLineStyle_nameDelete</code>	remove a name from the active name map.
<code>mdlLineStyle_nameModify</code>	modify an entry in the active name map.
<code>mdlLineStyle_nameQuery</code>	retrieve information from the active name map.
<code>mdlLineStyle_nameLoadMap</code>	load one or more map(s) from file.
<code>mdlLineStyle_nameSaveMap</code>	one or more of the active maps to file.
<code>mdlLineStyle_nameGetStringList</code>	get a <code>StringList</code> of all line style names.
<code>mdlLineStyle_getElementDescr</code>	convert an element with a user defined line style to the individual primitive elements that represent the line style components.
<code>mdlLineStyle_cacheInsert</code>	insert a line style resource into the line style component cache.
<code>mdlLineStyle_cacheDelete</code>	remove a line style resource from the line style component cache.
<code>mdlLineStyle_cacheFree</code>	remove all components from the line style component cache and frees all associated memory.
<code>mdlLineStyle_expandSymbolDescr</code> <code>[mdlLib.m1]</code>	duplicate the elements in <i>elmDscr</i> ; expand any elements that cannot be used in a point symbol and append them to the specified element descriptor chain.
<code>mdlLineStyle_createSymbolResource</code> <code>[mdlLib.m1]</code>	allocate and initialize a point symbol resource and append the elements from the element descriptor chain.

mdlLineStyle_nameInsert

```
#include <mdl.h>
#include <mslstyle.h>
#include <mslstyle.fdf>

int mdlLineStyle_nameInsert
(
    char          *pStyleName,    /* => Style name to insert */
    RscFileHandle rscFile,        /* => Resource file handle */
    ULONG         rscType,        /* => Resource type */
    ULONG         rscID,          /* => Resource ID */
    ULONG         nameAttributes, /* => Style name attributes */
    ULONG         option          /* => Function options */
);
```

Description The `mdlLineStyle_nameInsert` function inserts an entry into the active line style name map. Once inserted, the name is available for use by any line style operations.

The *pStyleName* parameter is the address of the style name to be copied into the active line style name map.

The *rscFile*, *rscType* and *rscID* parameters define the resource file, type and ID respectively of the top level line style resource that will be referenced by *pStyleName*.

The *nameAttributes* parameter is currently not used. Pass zero to guarantee compatibility with future versions.

The *option* parameter is currently not used. Pass zero to guarantee compatibility with future versions.



This function works directly on the active name maps. The changes are not written to file and will be lost unless the name maps are saved.

Returns `mdlLineStyle_nameInsert` function returns `SUCCESS` if the insert operation is completed successfully. Otherwise a non-zero error status is returned.

See Also `mdlLineStyle_nameDelete`, `mdlLineStyle_nameModify`, `mdlLineStyle_nameQuery`, `mdlLineStyle_nameLoadMap`, `mdlLineStyle_nameSaveMap`, `mdlLineStyle_nameGetStringList`.

mdlLineStyle_nameDelete

```
#include <mdl.h>
#include <mslstyle.h>
#include <mslstyle.fdf>

int mdlLineStyle_nameDelete
(
    char          *pStyleName,    /* => Name to remove */
    ...
);
```

```
int    option                /* => Function options */
);
```

Description The mdlLineStyle_nameDelete function removes an entry from the active line style name map. Once removed the line style name is no longer available to MicroStation or any line style functions.

The *pStyleName* parameter is the address of the line style name that is to be removed from the active line style name map.

The *option* parameter is currently unused. Pass zero to guarantee compatibility with future versions.



This function works directly on the active name maps. The changes are not written to file and will be lost unless the name maps are saved.

Returns mdlLineStyle_nameDelete returns SUCCESS if the name map entry was successfully removed. Otherwise a non-zero error status is returned.

See Also mdlLineStyle_nameInsert, mdlLineStyle_nameModify, mdlLineStyle_nameQuery, mdlLineStyle_nameLoadMap, mdlLineStyle_nameSaveMap, mdlLineStyle_nameGetStringList.

mdlLineStyle_nameModify

```
#include <mdl.h>
#include <mslstyle.h>
#include <mslstyle.fdf>

int mdlLineStyle_nameModify
(
char          *pNewName,      /* => New line style name */
RscFileHandle, rscFile,      /* => Component file handle */
ULong         rscType,       /* => Component resource type */
ULong         rscID,         /* => Component resource ID */
ULong         attributes,
char          *pOldName,     /* => Existing line style name */
ULong         option         /* => Function options */
);
```

Description mdlLineStyle_nameModify is used to modify the information in the active resource file and design file name maps. This function first searches the active resource name map for the record indicated by *pOldName*. If located, the record will contain the file, type, ID and attributes of the line style component referenced by *pOldName*. The remaining function parameters are used to alter the active name map record as follows.

If the *pNewName* parameter is not NULL, the string it points to will be copied into the active resource name map in place of *pOldName*. If an entry for this name also exists in the active design file name map, it too

will be modified and all elements using the associated style ID number will now reference the new style name.

If the *rscFile* parameter is not zero, it will replace the resource file handle stored in the located name map record.

If the *rscType* parameter is not zero, it will replace the resource type stored in the located name map record.

If the *rscID* parameter is not zero, it will replace the resource ID stored in the located name map record.

The *attributes* and *option* parameters are currently unused. Pass zero for both to guarantee compatibility with future versions.

It is important to note that this function cannot be replaced by similar calls to `mdlLineStyle_nameInsert` and `mdlLineStyle_nameDelete`. Deleting the old resource name map entry and then inserting the new one does not update the design file name map and therefore does not guarantee the use of the same style ID number. This could in turn orphan any elements that use the effected line style.



This function works directly on the active name maps. The changes are not written to file and will be lost unless the name maps are saved.

Returns `mdlLineStyle_nameModify` returns `SUCCESS` (zero) if the modify operation is completed successfully. Otherwise, a non zero error status is returned.

See Also `mdlLineStyle_nameInsert`, `mdlLineStyle_nameDelete`, `mdlLineStyle_nameQuery`, `mdlLineStyle_nameLoadMap`, `mdlLineStyle_nameSaveMap`, `mdlLineStyle_nameGetStringList`.

mdlLineStyle_nameQuery

```
#include <mdl.h>
#include <mslstyle.h>
#include <mslstyle.fdf>

int mdlLineStyle_nameQuery
(
    LineStyleNameInfo *pNameInfo,    /* <= Rsrc file, type, id etc. */
    char               *pStyleName,  /* => Query style name */
    ULONG              *option       /* => Function options */
);
```

Description The `mdlLineStyle_nameQuery` function returns the resource information from the active line style name map entry for *pStyleName*.

The values for the line style component resource file handle, resource type and resource ID are returned in *pNameInfo*. The `LineStyleNameInfo` structure is defined in the include file `mslstyle.h`.

pStyleName represents the line style name map entry from which the resource information is to be extracted.

Currently the *option* parameter is not used. Pass zero to guarantee compatibility with future versions.

Returns mdlLineStyle_nameQuery returns SUCCESS if the query operation was completed successfully. If the name does not exist in the line style name map, a non-zero error status is returned.

See Also mdlLineStyle_nameInsert, mdlLineStyle_nameDelete, mdlLineStyle_nameModify, mdlLineStyle_nameLoadMap, mdlLineStyle_nameSaveMap, mdlLineStyle_nameGetStringList.

mdlLineStyle_nameLoadMap

```
#include <mdl.h>
#include <mslstyle.h>
#include <mslstyle.fdf>

int mdlLineStyle_nameLoadMap
(
    ULong   fileNo,           /* => File to load map from */
    ULong   option           /* => Function options */
);
```

Description The mdlLineStyle_nameLoadMap function loads the line style name map(s) from the specified file(s) into the active line style name maps. Once completed the names from the map are available for use by MicroStation and any line style functions.

The first parameter *fileNo* is the design file number or resource file handle from which the name map will be loaded. Currently the only supported value is zero which indicates that all available line style name maps will be loaded.

The second parameter *option* defines whether the *fileNo* parameter is specifying a resource file handle or a design file number. Currently the only supported option is zero which causes all line style name maps to be loaded.

This function can be used to reload all name maps if an application has modified the resource file or design file name maps directly.

Returns mdlLineStyle_nameLoadMap returns SUCCESS (zero) if the name map(s) were loaded successfully. Otherwise a non-zero error status is returned.

See Also mdlLineStyle_nameInsert, mdlLineStyle_nameDelete, mdlLineStyle_nameModify, mdlLineStyle_nameQuery, mdlLineStyle_nameSaveMap, mdlLineStyle_nameGetStringList.

mdlLineStyle_nameSaveMap

```
#include <mdl.h>
```

```
#include <mslstyle.h>
#include <mslstyle.fdf>

int mdlLineStyle_nameSaveMap
(
    ULong   fileNo,          /* => Name map file number */
    ULong   option           /* => Function options */
);
```

Description The `mdlLineStyle_nameSaveMap` function stores one or more of the active line style name maps in the line style resource file or in a design file.

The first parameter *fileNo* identifies the file in which the name map is to be stored. Currently the only supported value is zero. This indicates that the design file portion of the active name map will be stored in the master design file.

The second parameter *option*, determines whether *fileNo* indicates a design file or a resource file. Currently the only supported option is zero indicating that only the design file portion of the line style name map will be stored.

Returns `mdlLineStyle_nameSaveMap` returns `SUCCESS` (zero) if the name map was stored successfully. Otherwise a non-zero error status is returned.

See Also `mdlLineStyle_nameInsert`, `mdlLineStyle_nameDelete`, `mdlLineStyle_nameModify`, `mdlLineStyle_nameQuery`, `mdlLineStyle_nameLoadMap`, `mdlLineStyle_nameGetStringList`.

mdlLineStyle_nameGetStringList

```
#include <mdl.h>
#include <mslstyle.h>
#include <mslstyle.fdf>

StringList *mdlLineStyle_nameGetStringList
(
    void    *pAuxInfo,      /* => Optional input information */
    ULong   option           /* => Function options */
);
```

Description The `mdlLineStyle_nameGetStringList` function creates a string list containing all line style names from the active resource name map. By default, the string list will contain four info fields (per cell). The first info field will be left empty and should be reserved for use by the list box manager. The second, third and fourth info fields will contain the resource file handle, resource type and resource ID. respectively, from the name map record.

The *pAuxInfo* parameter may be used to specify more details about the format of the string list to be created. Currently the parameter is unused. Pass `NULL` to guarantee compatibility with future versions.



It is the responsibility of the caller to free the string list memory by calling `mdlStringList_destroy`.

Returns `mdlLineStyle_nameGetStringList` returns the address of the new string list if successful. Otherwise the function returns `NULL`.

See Also `mdlStringList_destroy`, String List Manager.

mdlLineStyle_getElementDescr

```
int mdlLineStyle_getElementDescr
(
  MSElementDescr  **outEdPP,      /* <= Primitives */
  MSElementDescr  *inEdP,        /* => Base element */
  int              fileNo,         /* => File number */
  int              option          /* => Options (zero=default) */
);
```

Description `mdlLineStyle_getElementDescr` converts an element with a user defined line style, *inEdP*, to the individual primitive elements that represent the line style components. *inEdP* is an element with a user defined line style (UDLS).

outedPP points to the address of an element descriptor containing the component elements representing the line style.

fileNo is the file number for *inEdPP*. 0 is the master file and 1 is the first reference file.

If *option* is set to one, the output elements are placed in a graphic group. If it is zero, they are not.

Returns `mdlLineStyle_getElementDescr` returns `SUCCESS` or an appropriate error code. See `mdlerrs.h` for a list of codes.

mdlLineStyle_cacheInsert

```
#include <mdl.h>
#include <mslstyle.h>
#include <mslstyle.fdf>

int mdlLineStyle_cacheInsert
(
  RscFileHandle rscFile,      /* => Line style resource file */
  ULONG         rscType,      /* => Line style resource type */
  ULONG         rscID,        /* => Line style resource ID */
  void          *pRsc,        /* => Line style resource (or NULL) */
  ULONG         option        /* => Function options */
);
```

Description The `mdlLineStyle_cacheInsert` function inserts a line style component resource into the line style component cache. MicroStation always attempts to load a line

style resource from the cache before attempting to read it from the resource file. By using this function an application can override a line style resource from a style file or cause MicroStation to use a line style resource that does not exist in any resource file. When a line style resource is inserted into the cache it automatically loads any dependent resources. So if you are trying to override all resources in a line style, make sure the dependent resources are inserted into the cache first.

The first three parameters *rscFile*, *rscType* and *rscID* identify the file, type and ID respectively, of the resource that will be loaded into the component cache.

If the fourth parameter *pRsc* is not `NULL` it must be the address of a valid line style resource that will be copied into the cache directly. If *pRsc* is `NULL`, the resource will be read from the line style resource file indicated by *rscFile*.

Returns `mdlLineStyle_cacheInsert` returns `SUCCESS` if the resource was inserted into the component cache successfully. Otherwise, a non-zero error status is returned.

See Also `mdlLineStyle_cacheDelete`, `mdlLineStyle_cacheFree`.

mdlLineStyle_cacheDelete

```
#include <mdl.h>
#include <mslstyle.h>
#include <mslstyle.fdf>

int mdlLineStyle_cacheDelete
(
  RscFileHandle    rscFile,      /* => Component resource file */
  ULONG           rscType,      /* => Component resource type */
  ULONG           rscID,        /* => Component resource ID */
  ULONG           option        /* => Function options */
);
```

Description The `mdlLineStyle_cacheDelete` function removes a line style component resource from the line style component cache. If any other resources in the cache reference the resource being removed, they too will be removed.

The first three parameters, *rscFile*, *rscType* and *rscID* identify the line style component that is to be removed.

The *option* parameter is currently unused. Pass zero to guarantee compatibility with future versions.

Returns `mdlLineStyle_cacheDelete` returns `SUCCESS` if the component was successfully removed from the line style component cache. Otherwise an error status is returned.

See Also `mdlLineStyle_cacheInsert`, `mdlLineStyle_cacheFree`.

mdlLineStyle_cacheFree

```
#include <mdl.h>
#include <mslstyle.h>
#include <mslstyle.fdf>

void mdlLineStyle_cacheFree
(
void
);
```

Description The mdlLineStyle_cacheFree function removes all components from the line style component cache and frees all associated memory. The cache is automatically reallocated and reloaded with MicroStation's next request for a line style component.

It is rarely necessary for an application to use this function but it can be helpful when the application has made numerous changes to the line style library. Calling this function will force MicroStation to reread the line style components from the library rather than relying on any copies that might be in the component cache.

Returns mdlLineStyle_cacheFree returns no value.

See Also mdlLineStyle_cacheInsert, mdlLineStyle_cacheDelete.

Creating Point Symbol Resources

Creating a point symbol resource is a two step process. First the element descriptor must be processed (expanded) using mdlLineStyle_expandSymbolDescr [mdl1lib.m1] to remove all compound elements.

Dimensions, multi-lines and shared cells are dropped to their component primitives. Second, the expanded descriptor is processed by mdlLineStyle_createSymbolResource [mdl1lib.m1] which allocates memory for the symbol resource and inserts the symbol elements into the resource.

Prototypes for the following functions are located in mslstyle.fdf.

To use these functions, an application must link with mdl1lib.m1.

mdlLineStyle_expandSymbolDescr [mdl1lib.m1]

```
#include <mslstyle.fdf>
int mdlLineStyle_expandSymbolDescr /* <= SUCCESS or ERROR */
(
MSElementDescr **symDescr, /* <= element descrip. (append!=NULL)*/
MSElementDescr *elmDscr, /* => existing element descriptor */
int fileNo /* => file number for descriptor */
);
```

Description `mdlLineStyle_expandSymbolDescr` duplicates the elements in *elmDescr*, expands any elements that cannot be used in a point symbol and appends them to the element descriptor chain *symDescr*. If **symDescr* is NULL, a new element descriptor is allocated.⁴

The *fileNo* argument is required (0 for masterfile) to resolve associative points and shared cell definitions.



It is the responsibility of the caller to free the memory used by *symDescr* by calling `mdlElmdscr_freeAll`.

Returns `mdlLineStyle_expandSymbolDescr` returns SUCCESS if the operation completed successfully. Otherwise an error status is returned.

mdlLineStyle_createSymbolResource [mdl.lib.ml]

```
#include <mslstyle.fdf>

int mdlLineStyle_createSymbolResource /* <= SUCCESS or ERROR */
(
    PointSymRsc      **pSymRsc,      /* <= new sym. resource (allocated) */
    MSElementDescr  *symDescr,      /* => input element descriptor */
    Dpoint3d         *pOrigin        /* => origin point */
);
```

Description `mdlLineStyle_createSymbolResource` allocates and initializes a point symbol resource and appends the elements from the element descriptor chain *symDescr*. The element descriptor, *symDescr* should be created using `mdlLineStyle_expandSymbolDescr` so that it does not contain elements that cannot be used in a line style definition.

The coordinates of the symbol origin must be specified in *pOrigin*.



It is the responsibility of the caller to free the memory allocated for *pSymRsc*.

A symbol description string can be added by filling in the point symbol field, *sym.header.descr* (see `mslstyle.h`), with a null terminated ASCII string. Use `mdlResource_add` to add the point symbol resource to a style library. It's a good idea to pass the point symbol description to `mdlResource_add` as the resource alias so that the symbol resource can be located by name for use in the PLACE SYMBOL command.

Returns `mdlLineStyle_createSymbolResource` returns SUCCESS if the operation completed successfully. Otherwise an error status is returned.

12

Multi-Line Functions

The multi-line functions create and modify MicroStation multi-line elements. MicroStation multi-line elements are divided into four major sections: profile, caps, work line and breaks.

A multi-line can contain up to 16 parallel lines. Each line can have its own color, style, weight and offset from the work line collectively referred to as a **line definition**. The group of up to 16 line definitions is referred to as the **multi-line profile**.

The ends of multi-lines can have **caps**. The two end caps display as lines connecting the two outermost lines, circular arcs, or a combination of lines and arcs. Joints in multi-lines are also loosely referred to as caps. The joints display only as lines. Each cap can also have its own color, style and weight.

The work line is defined by the multi-line vertices. The work line itself is never displayed. It is the reference line for the line definitions mentioned above. If the multi-line profile contains a line definition with an offset value of zero, this line will be displayed exactly on the work line.

Each segment of a multi-line can contain breaks, each of which appears as a blank area in the line. A break is defined by its offset from the previous vertex, its length, and a line mask that indicates which lines in the profile are affected. The values used for a break are always measured along the work line and then projected along a perpendicular vector to the affected line.

The following table lists multi-line functions:

Function	Used to
<code>mdlMline_create [mdl.lib.m]</code>	create a multi-line element.
<code>mdlMline_getInfo [mdl.lib.m]</code>	get general information about a multi-line.
<code>mdlMline_insertPoint [mdl.lib.m]</code>	insert a new vertex into a multi-line.
<code>mdlMline_deletePoint</code>	remove a vertex from a multi-line.
<code>mdlMline_extractPoints [mdl.lib.m]</code>	extract work line vertex points from a multi-line.

Function	Used to
mdlMline_modifyPoint [mdl1ib.ml]	move a vertex of a multi-line.
mdlMline_getBreak [mdl1ib.ml]	retrieve break information.
mdlMline_insertBreak [mdl1ib.ml]	insert a break into a multi-line segment.
mdlMline_deleteBreak	remove a break from a multi-line segment.
mdlMline_getLineDef [mdl1ib.ml]	retrieve line definition information.
mdlMline_setLineDef [mdl1ib.ml]	set line symbology and offset.
mdlMline_getCapDef [mdl1ib.ml]	retrieve cap definition.
mdlMline_setCapDef [mdl1ib.ml]	set cap definition.
mdlMline_setClosure [mdl1ib.ml]	set closure status of multi-line.
mdlMline_getElementDescr	convert a MicroStation multi-line element to IGDS-8.8-compatible primitives.
mdlMline_getBoundaryDescr	create an element descriptor representing the boundary of a multi-line element.
mdlMline_validate [mdl1ib.ml]	validate a multi-line element.

The mdlMline_join... functions provide a program interface to all Multi-line joint cleanup operations previously available only through the Multi-line joint application (cutter.ma).

Function	Used to
mdlMline_joinTee [mdl1ib.ml]	create Multi-line Tee Joint.
mdlMline_joinCross [mdl1ib.ml]	create Multi-line Cross Joint.
mdlMline_joinCorner [mdl1ib.ml]	create Multi-line Corner Joint.

The following definitions in mdl.h are for use with the mdlMline_join... functions:

```
#define MLJOIN_CLOSE 0    /* Request open type joint */
#define MLJOIN_OPEN  1    /* Request closed type joint */
#define MLJOIN_MERGE 2    /* Request merged type joint */
```

To facilitate application user interface design, the arguments required by the mdlMline_join... functions are similar to the user inputs required by the Multi-line joints (cutter.ma) application supplied with MicroStation. For a detailed description of each of the available joint types, see the *Multi-line Joints palette* section of the *Advanced Element Manipulations* chapter in the *MicroStation Reference Guide*.

Example

See mline.mc.

mdlMline_create [mdl.lib.ml]

```
#include <mdl.h>

int mdlMline_create
(
    MSElementUnion    *mline,    /* <= new multi-line element */
    MSElementUnion    *seed,     /* => seed multi-line element */
    DPoint3d           *normal,   /* => multi-line normal vector */
    DPoint3d           *points,   /* => multi-line vertices */
    int                nPoints    /* => number of vertices */
);
```

Description The mdlMline_create function creates a multi-line element in *mline*. If *seed* is not NULL, the multi-line definition is extracted from *seed*. Otherwise, the active multi-line definition is used.

The plane of the multi-line is defined by the unit vector in *normal*. If *normal* is NULL or the current file is 2D, the vector [0,0,1] is used.

The vertices for the multi-line are taken from *points*. *nPoints* specifies the number of vertices in *points*.

This function does not allow associative points. To add associative points to a multi-line, you must use the mdlMline_insertPoint [mdl.lib.ml] function.

Returns The mdlMline_create function returns SUCCESS if a valid multi-line was created. Otherwise, it returns an error status.

See Also mdlMline_insertPoint [mdl.lib.ml].

mdlMline_getInfo [mdl.lib.ml]

```
#include <mdl.h>

void mdlMline_getInfo
(
    int                *nPoints,   /* <= number of vertices */
    int                *nLines,    /* <= number of lines (definitions) */
    double             *minDist,   /* <= minimum line definition offset */
    double             *maxDist,   /* <= maximum line definition offset */
    DPoint3d           *normal,    /* <= multi-line normal vector */
    MSElementUnion    *mline      /* => multi-line element */
);
```

Description The mdlMline_getInfo function retrieves general information from a multi-line. If NULL is passed for any of the output arguments, it is ignored.

The value returned in *nPoints* is the number of vertices in the multi-line work line. *nLines* is the number of line definitions in the multi-line profile. *minDist* and *maxDist* contain the line definition offsets of the two outermost lines. The multi-line normal vector is returned in *normal*.

Returns The mdlMline_getInfo function returns no value.

mdlMline_insertPoint [mdl.lib.mll]

```
#include <mdl.h>

int mdlMline_insertPoint
(
  MSElementUnion *mline,      /* <=> multi-line to modify */
  void *newPoint,             /* => DPoint3d or AssocPoint to add */
  int pointNo,                /* => vertex number (zero-based) */
  int pointType               /* => contents of newPoint */
);
```

Description The mdlMline_insertPoint function inserts the point *newPoint* at the position *pointNo* in the element *mline*. If *pointNo* is -1, the new point is inserted as the last vertex in the multi-line. The value of *pointType* determines the content of *newPoint* and must be set to one of the following constants:

Value	Description
POINT_STD	<i>newPoint</i> is the address of a DPoint3d.
POINT_ASSOC	<i>newPoint</i> is the address of an AssocPoint created by an mdlAssoc_create... function.



Before the modified multi-line element is written to the file, it must be validated with mdlMline_validate [mdl.lib.mll].

Returns The mdlMline_insertPoint function returns SUCCESS if the point was inserted. Otherwise, it returns an error status.

See Also mdlMline_deletePoint, mdlMline_validate [mdl.lib.mll].

mdlMline_deletePoint

```
#include <mdl.h>

int mdlMline_deletePoint
(
  MSElementUnion *mline,      /* <=> multi-line to modify */
  int pointNo                 /* => vertex to delete (zero-based) */
);
```

Description The mdlMline_deletePoint function deletes the vertex at *pointNo* from the element multi-line *mline*.



Before the modified multi-line element is written to the file, it must be validated with mdlMline_validate [mdl.lib.mll].

Returns The mdlMline_deletePoint function returns SUCCESS if the point was deleted. Otherwise, it returns an error status.

See Also mdlMline_insertPoint [mdl.lib.mdl], mdlMline_validate [mdl.lib.mdl].

mdlMline_extractPoints [mdl.lib.mdl]

```
#include <mdl.h>

int mdlMline_extractPoints
(
    DPoint3d      *outPoints,    /* <= extracted points */
    MSElementUnion *mline,      /* => multi-line element */
    int           fileName,      /* => design file (master/reference file) */
    int           pointNo,       /* => first point to extract */
    int           nPoints        /* => # of points to extract */
);
```

Description The mdlMline_extractPoints function extracts *nPoints* vertices of the work line from *mline* starting at vertex *pointNo*. The file that the multi-line element came from (MASTERFILE for master file, 1-256 for reference files) is passed in *fileName*.

outPoints must be at least *nPoints* * sizeof(DPoint3d) bytes. If a multi-line vertex is represented by an associative point, the association is resolved before the point is added to *outPoints*.

Returns The mdlMline_extractPoints function returns the number of points. Otherwise, it returns an error status.

See Also mdlMline_insertPoint [mdl.lib.mdl], mdlMline_deletePoint.

mdlMline_modifyPoint [mdl.lib.mdl]

```
#include <mdl.h>

int mdlMline_modifyPoint
(
    MSElementUnion *mline,      /* <=> multi-line to modify */
    int             pointNo,     /* => vertex to modify (zero-based) */
    DPoint3d        *newPoint,   /* => new point */
    int             option       /* => modification option */
);
```

Description The mdlMline_modifyPoint function replaces the multi-line vertex at *pointNo* in *mline* with the point data in *newPoint*. If *option* is set to SHIFT_BREAKS, any breaks following *pointNo* will be adjusted so that they appear to remain as close as

mdlMline_insertBreak [mdlLib.m]l

```
#include <mdl.h>

int mdlMline_insertBreak
(
  MSElementUnion    *mline,          /* <=> multi-line to modify */
  int                segNo,           /* => segment to contain break */
  double             offset,          /* => break offset */
  double             length,          /* => break length */
  int                lineMask,        /* => mask of lines cut by break */
  int                breakFlags       /* => length and offset options */
);
```

Description The mdlMline_insertBreak function inserts a break in *mline* starting at a distance of *offset* from the origin of the segment indicated by *segNo*.

The length of the break is specified by *length*. The low order word of *lineMask* determines which of the 16 possible lines are cut by the break.

If any bit 0 through 15 is set, the break will cut the corresponding line. For example, setting *lineMask* to 0xff cuts all lines. *breakFlags* overrides the values in *length* and *offset* when either of the following constants, combined with the bitwise OR operator (|), is used:

Value	Description
MLBREAK_FROM_JOINT	The break starts at the joint line at the specified segment's origin, and the value of <i>offset</i> is ignored.
MLBREAK_TO_JOINT	The break ends at the joint line at the specified segment's end, and the value of <i>length</i> is ignored.
MLBREAK_STD	Use offset and length.

If the inserted break overlaps an existing break, the conflicting break is removed.

Returns mdlMline_insertBreak returns SUCCESS if the break was inserted. Otherwise, it returns an error status.

See Also mdlMline_getBreak [mdlLib.m]l, mdlMline_deleteBreak.

mdlMline_deleteBreak

```
#include <mdl.h>

int mdlMline_deleteBreak
(
  MSElementUnion    *mline,          /* <=> multi-line to modify */
  int                segNo,           /* => segment # (zero-based) */
  int                breakNo          /* => break to delete */
);
```

Description The mdlMline_deleteBreak function deletes a break from a multi-line segment. *segNo* is the multi-line segment number (zero-based) containing the break.

breakNo is the index of the break within the segment *segNo*.

Returns The mdlMline_deleteBreak function returns SUCCESS if the break was located and deleted. Otherwise, it returns an error status.

See Also mdlMline_insertBreak [mdl.lib.m]l, mdlMline_getBreak [mdl.lib.m]l.

mdlMline_getLineDef [mdl.lib.m]l

```
#include <mdl.h>

int mdlMline_getLineDef
(
    int          *color,          /* <= line color */
    int          *style,          /* <= line style */
    int          *weight,         /* <= line weight */
    int          *level,          /* <= line level */
    int          *conClass,       /* <= TRUE for construction class */
    double       *offset,         /* <= line offset from work line */
    MSElementUnion *mline,       /* => multi-line element */
    int          lineNo           /* => line number - profile index */
);
```

Description mdlMline_getLineDef retrieves the definition of 1 of the 16 possible lines in a multi-line. If NULL is passed for an output argument, that argument is ignored.

The values returned in *color*, *style* and *weight* are the line color, line style and line weight, respectively. The value of each can be -1. In this case, the line uses the value from the multi-line display header, which can be extracted using the mdlElement_getSymbology function.

Valid values for *level* are 0-63. If *level* is 0 (zero), the default value from the element header is used.

If *conClass* is TRUE, the line will be treated as construction class. Otherwise it will be treated as primary class.

The value returned in *offset* is the distance of *lineNo* from the multi-line work line.

Returns mdlMline_getLineDef returns SUCCESS if the requested line definition was found. Otherwise, it returns an error status.

See Also mdlMline_setLineDef [mdl.lib.m]l.

mdlMline_setLineDef [mdl.lib.m]l

```
#include <mdl.h>

int mdlMline_setLineDef
(
```

```
MSElementUnion *mline,          /* <=> multi-line element */
int             lineNo,          /* => line number/profile index */
int             color,           /* => line color */
int             style,           /* => line style */
int             weight,          /* => line weight */
int             level,           /* <= line level */
int             conClass,        /* <= TRUE for construction class */
double          offset           /* => line offset from work line */
);
```

Description The mdlMline_setLineDef function changes the definition of the line *lineNo* in *mline* to the values of *color*, *style*, *weight* and *offset*.

Valid values for *color* are -1 to 253; valid values for *style* are -1 to 7; and valid values for *weight* are -1 to 15.

If the *color*, *style* or *weight* is -1, the default value from the multi-line element display header is used. This value can be set with the mdlElement_setSymbology function.

Valid values for *level* are 0-63. If *level* is 0 (zero), the default value from the element header is used.

If *conClass* is TRUE, the line will be treated as construction class. Otherwise it will be treated as primary class.

The value of *offset* is the new distance of *lineNo* from the multi-line work line.

Returns mdlMline_setLineDef returns SUCCESS if the requested line definition was changed. Otherwise, it returns an error status.

See Also mdlMline_getLineDef [mdl.lib.ml].

mdlMline_getCapDef [mdl.lib.ml]

```
#include <mdl.h>

int mdlMline_getCapDef
(
int             *color,          /* <= cap color */
int             *style,          /* <= cap style */
int             *weight,         /* <= cap weight */
int             *level,          /* <= line level */
int             *conClass,        /* <= TRUE = construction class */
int             *capOptions,      /* <= cap options/type */
double          *capAngle,        /* <= cap angle from work line */
MSElementUnion *mline,          /* => multi-line element */
int             capNo            /* => cap def. to extract */
);
```

Description The mdlMline_getCapDef function retrieves the definition of one of three possible caps in a multi-line. A NULL pointer can be passed for any output arguments. In this case, the argument is ignored.

The value of *capNo* must be 0 (origin cap), 1 (joint cap), or 2 (end cap).

The values returned in *color*, *style*, and *weight* are the cap color, cap style and cap weight, respectively. If the value is -1, the cap uses the value from the display header, which can be extracted with the mdlElement_getSymbology function.

Valid values for *level* are 0-63. If *level* is 0 (zero) the default value from the element header is used.

If *conClass* is TRUE the line will be treated as construction class. Otherwise, it will be treated as primary class.

The value in *capOptions* is used as a bit mask and it is set when any of the following constants are set with the bitwise OR (|) operator:

Value	Description
MLCAP_NONE	Use no end caps.
MLCAP_LINE	Display a line connecting the two outermost lines.
MLCAP_INNER_ARCS	Display circular arcs connecting all but the two outermost lines. (Not used for joint cap).
MLCAP_OUTER_ARC	Display a circular arc connecting the two outermost lines (Not used for joint cap).

If the requested cap is an end cap, the counter-clockwise angle in radians from the work line to the cap line is returned in *capAngle*.

Returns The mdlMline_getCapDef function returns SUCCESS if the requested cap definition was found. Otherwise, it returns an error status.

See Also mdlMline_setCapDef [mdl.lib.ml].

mdlMline_setCapDef [mdl.lib.ml]

```
#include <mdl.h>

int mdlMline_setCapDef
(
  MSElementUnion *mline,      /* <=> multi-line element */
  int capNo,                  /* => cap number */
  int color,                  /* => cap color */
  int style,                  /* => cap style */
  int weight,                 /* => cap weight */
  int *level,                 /* <= line level */
  int *conClass,              /* <= TRUE=construction class */
  int capOptions,             /* => type of cap */

```



```
double      capAngle      /* => cap angle from work line */
);
```

Description The mdlMline_setCapDef function changes the definition of *capNo* in *mline* to the values of *color*, *style*, *weight*, *capOptions* and *capAngle*.

The value of *capNo* must be 0 (origin cap), 1 (joint cap) or 2 (end cap).

Valid values for *color* are -1 to 253; valid values for *style* are -1 to 7; and valid values for *weight* are -1 to 15.

If *color style* or *weight* is -1, the default value from the element display header is used. This value can be set with the mdlElement_setSymbology function.

Valid values for *level* are 0-63. If *level* is 0 (zero), the default value from the element header is used.

If *conClass* is TRUE, the line will be treated as construction class. Otherwise it will be treated as primary class.

The value of *capOptions* must be set to one or more of the following constants combined with the bitwise OR (|) operator:

Value	Description
MLCAP_NONE	Use no end caps or arcs.
MLCAP_LINE	Display a line connecting the two outermost lines.
MLCAP_INNER_ARCS	Display circular arcs connecting all but the two outermost lines. (Not used for joint cap).
MLCAP_OUTER_ARC	Display a circular arc connecting the two outermost lines. (Not used for joint cap).

capAngle is the counter-clockwise angle in radians from the work line to the cap line. This value is used only for end caps (0 and 2).

Returns mdlMline_setCapDef returns SUCCESS if the requested cap definition was changed. Otherwise, it returns an error status.

See Also mdlMline_getCapDef [mdlLib.m]l.

mdlMline_setClosure [mdlLib.m]l

```
#include <mdl.h>

int mdlMline_setClosure
(
MSElementUnion *mline, /* <=> multi-line element */
int close /* => close multi-line TRUE/FALSE */
);
```

Description The `mdlMline_setClosure` function sets the closure status of a multi-line element. If *close* is `TRUE`, the multi-line is closed. If *close* is `FALSE`, the multi-line is not closed.

When a multi-line is closed, the start and end cap definitions are no longer used. Instead, the start and end caps are automatically adjusted to appear as a single joint using the joint options and symbology.

A multi-line cannot be closed unless it has a minimum of four points and the first and last points are identical.

Returns The `mdlMline_setClosure` function returns `SUCCESS` if the operation is successful. Otherwise, it returns an error status.

mdlMline_getElementDescr

```
#include <mdl.h>

int mdlMline_getElementDescr
(
  MSElementDescr  **descr,          /* <= element descriptor */
  MSElementUnion  *mline,          /* => multi-line element */
  int              fileName,        /* => design file
                                     (master or reference file) */
  int              graphGroup       /* => TRUE=put elms in graphic grp */
);
```

Description The `mdlMline_getElementDescr` function converts a MicroStation multi-line element to IGDS 8.8 compatible primitives, which are then stored in an element descriptor. The address of the element descriptor is returned in *descr*.

The file that the multi-line element came from (`MASTERFILE` for master file, 1-256 for reference files) is passed in *fileName*.

Upon successful completion of this function, the low order byte of the `userData1` member of each element descriptor in *descr* contains the multi-line profile index from which the element was derived. An application can use this value in coordination with the `mdlMline_getLineDef` function to determine the multi-line profile information for each element in *descr*.



It is the responsibility of the calling application to free the memory allocated to *descr* by calling `mdlElmdscr_freeAll`.

Returns The `mdlMline_getElementDescr` function returns `SUCCESS` if the element descriptor was created. Otherwise, it returns an error status.

See Also Element Descriptor Functions.

mdlMline_getBoundaryDescr

```
#include <mdl.h>
#include <msmline.fdf>

int mdlMline_getBoundaryDescr
(
    MSElementDescr    **edPP,          /* <= shape representing boundary */
    MSElement         *mlineP,         /* => multi-line element */
    int                minLine,         /* => min. profile line (-1 for limit) */
    int                maxLine,         /* => max. profile line (-1 for limit) */
    int                fileNum          /* => file number */
);
```

Description The mdlMline_getBoundaryDescr function creates an element descriptor in *edPP* containing one or more elements that represent the boundary of the multi-line element *mlineP*. If the boundary requires fewer than 101 the boundary is represented by a primitive shape element. Otherwise a complex shape is created.

The *minLine* and *maxLine* parameters indicate the minimum and maximum profile index numbers (line numbers) used to define the boundary. Pass -1 for both to use the outermost lines of the multi-line element.

fileNum is the file number of the multi-line *mlineP*. Pass MASTERFILE (0) for the master file of 1-255 for reference files.



It is the responsibility of the calling application to free the memory allocated to *edPP* by calling mdlElmdscr_freeAll.

Returns mdlMline_getBoundaryDescr returns SUCCESS if the operation was successful. Otherwise an error status is returned.

See Also mdlMline_getElementDescr, mdlElmdscr_freeAll.

mdlMline_validate [mdl.lib.mdl]

```
#include <mdl.h>

int mdlMline_validate
(
    MSElementUnion    *mline          /* <=> multi-line to validate */
);
```

Description The mdlMline_validate function sets the multi-line range block and checks for invalid points and/or settings. If mdlMline_insertPoint [mdl.lib.mdl], mdlMline_deletePoint, or mdlMline_modifyPoint [mdl.lib.mdl] has modified *mline*, mdlMline_validate must be called before the element is written to the file.

Returns The mdlMline_validate function returns SUCCESS if the element was validated. Otherwise, it returns an error status.

See Also mdlMline_insertPoint [mdlLib.mli], mdlMline_deletePoint, mdlMline_modifyPoint [mdlLib.mli].

mdlMline_joinTee [mdlLib.mli]

```
int mdlMline_joinTee
(
  MSElement *line1,          /* <=> First line. (will be extended) */
  int segNo1,                /* => Segment number in first line */
  DPoint3d *locatePoint,     /* => Locate point on first line */
  MSElement *line2,          /* <=> Second line (base line) */
  int segNo2,                /* => Segment number in second line */
  int type                    /* => Joint type (MLJOIN_...) */
);
```

Description This function creates open, closed or merged “tee joints” in precisely the same manner as the Tee Joint tools described in the Multi-line joints section of the *MicroStation Reference Guide*.

line1 is a Multi-line element that will be extended or shortened to create a tee joint with Multi-line element *line2*.

segNo1 and *segNo2* are the zero based segment numbers of *line1* and *line2* that will be modified by the join operation.

type must be one of the following three constants:

Value	Description
MLJOIN_CLOSE	Create a closed tee joint.
MLJOIN_OPEN	Create an open tee joint.
MLJOIN_MERGE	Create a merged tee joint.

locatePoint is an arbitrary point that identifies the portion of the Multi-line *line1* that will be kept if *line1* is shortened by the join operation.



If a Multi-line element is located by means of the mdlState_start...Modify... or mdlLocate_... functions, the locate point and segment number can be found in the TCB structure *tcb->locateInfo* defined in the header file tcb.h.

Returns This function returns SUCCESS (zero) if successful. Otherwise an error status is returned.

See Also mdlMline_joinCross [mdlLib.mli], mdlMline_joinCorner [mdlLib.mli].

mdlMline_joinCross [mdlLib.mli]

```
int mdlMline_joinCross
(
  MSElement *line1, /* <=> First line */
  int segNo1, /* => Segment number in first line */
  MSElement *line2, /* <=> Second line (base line) */
  int segNo2, /* => Segment number in second line */
  int type /* => Joint type (MLJOIN_...) */
);
```

Description mdlMline_joinCross creates open, closed or merged “cross joints” in precisely the same manner as the Cross Joint tools described in the Multi-line joints section of the *MicroStation Reference Guide*.

line1 and *line2* are the two multi-line elements that will be joined.

segNo1 and *segNo2* are the zero based segment numbers of *line1* and *line2* that will be modified by the join operation.

type must be one of the following three constants:

Value	Description
MLJOIN_CLOSE	Create a closed cross joint.
MLJOIN_OPEN	Create an open cross joint.
MLJOIN_MERGE	Create a merged cross joint.



If a Multi-line element is located by means of the mdlState_start...Modify... or mdlLocate... functions, the locate point and segment number can be found in the TCB structure *tcb->locateInfo* defined in the header file tcb.h.

Returns mdlMline_joinCross returns SUCCESS (zero) if successful. Otherwise an error status is returned.

See Also mdlMline_joinTee [mdlLib.mli], mdlMline_joinCorner [mdlLib.mli].

mdlMline_joinCorner [mdlLib.mli]

```
int mdlMline_joinCorner
(
  MSElement *line1, /* <=> First line. (will be extended) */
  int segNo1, /* => Segment number in first line */
  DPoint3d *point1, /* => Locate point on first line */
  MSElement *line2, /* <=> Second line. (will be extended) */
  int segNo2, /* => Segment number in second line */
  DPoint3d *point2 /* => Locate point on second line */
);
```

Description This function creates “corner joints” in precisely the same manner as the Corner Joint tool described in the Multi-line joints section of the *MicroStation Reference Guide*.

line1 and *line2* are Multi-line elements that will be shortened or extended to create the corner joint.

segNo1 and *segNo2* are the zero based segment numbers of *line1* and *line2* that will be modified.

point1 is an arbitrary point that identifies the portion of the Multi-line *line1* that will be kept if *line1* is shortened by the join operation.

point2 is an arbitrary point that identifies the portion of the Multi-line *line2* that will be kept if *line2* is shortened by the join operation.



If a Multi-line element is located by means of the `mdlState_start...Modify...` or `mdlLocate_...` functions, the locate point and segment number can be found in the TCB structure *tcb->locateInfo* defined in the header file *tcb.h*.

Returns This function returns `SUCCESS` (zero) if successful. Otherwise an error status is returned.

See Also `mdlMline_joinTee [mdlLib.mll]`, `mdlMline_joinCross [mdlLib.mll]`.

13

Element Association Functions

The element association functions (`mdlAssoc_...`) provide access to the element association capabilities used to create association points in dimensions, multi-lines and shared cells.

An **association point** (`AssocPoint`) is information that represents a point on a tagged element. The actual coordinates of the association point can be extracted at any time.

To create an association, you must first add a tag (unique ID number) to the element that is the association's target or object. The tag is added as attribute data appended to the element. After the element is tagged, the `mdlAssoc_create...` functions use the tag value to create an association point.

Once an association point is created, it can be passed to any of the dimension, multi-line, or shared cell functions that use association points to define geometry.

Outside the use of association points, the `mdlAssoc_tagElement` and `mdlAssoc_getElement` functions provide a powerful general-purpose method for tagging and retrieving elements. This method is a critical part of many applications.

Function	Used to
<code>mdlAssoc_createArc</code>	create an association representing a point on an arc or ellipse.
<code>mdlAssoc_createIntersection</code>	create an association representing the intersection of two elements.
<code>mdlAssoc_createKeypoint</code>	create an association representing a keypoint on a line, line string or shape.
<code>mdlAssoc_createLinear</code>	create an association representing a point on a linear element.
<code>mdlAssoc_createMline</code>	create an association representing a point on a multi-line.
<code>mdlAssoc_createOrigin</code>	create an association representing the origin of a base element.

Function	Used to
mdlAssoc_createBCurve	create an association representing a point along a B-spline curve.
mdlAssoc_getElement	retrieve an element given its tag value.
mdlAssoc_getPoint	get a geometric point from an association point.
mdlAssoc_isTagged	test an element to see if it is tagged.
mdlAssoc_tagElement	add a tag attribute (unique ID number) to an element.
mdlAssoc_tagElementComplex	add a tag to each component of a complex element.
mdlAssoc_getCurrent	retrieve the “current” associative (or “active”) associative point.

mdlAssoc_createArc

```
#include <mdl.h>

int mdlAssoc_createArc
(
    AssocPoint  *assoc,          /* <= arc association */
    ULong       tagValue,        /* => tag value of arc element */
    double      angle,           /* => angle from arc start */
    int         keyPoint         /* => keypoint on arc element */
);
```

Description The mdlAssoc_createArc function creates an association that represents a point on an arc or ellipse element.

If a valid association is created, the association information is returned in *assoc*.

tagValue is the tag (unique ID number) of the arc or ellipse element that is the association’s object (or root).

angle is the angle (in radians) from the primary axis of the arc or ellipse to the association point. This argument is used only when the value of *keyPoint* is ASSOC_ARC_ANGLE.

keyPoint determines the type of association that is created. This argument must be set to one of the following defined constants:

Association type	Association point location
ASSOC_ARC_ANGLE	<i>angle</i> radians from the primary axis
ASSOC_ARC_CENTER	center of the arc or ellipse

Association type	Association point location
ASSOC_ARC_START	arc start point
ASSOC_ARC_END	arc end point

Returns The mdlAssoc_createArc function returns SUCCESS (zero) if it is able to create a valid association. Otherwise, it returns a non-zero value.

See Also mdlAssoc_getPoint.

mdlAssoc_createIntersection

```
#include <mdl.h>

int mdlAssoc_createIntersection
(
  AssocPoint  *assoc,          /* <= arc association */
  ULong       tagValue1,       /* => tag value of first element */
  ULong       tagValue2,       /* => tag value of second element */
  int         index            /* => intersection index */
);
```

Description The mdlAssoc_createIntersection function creates an association that represents a point at the intersection of two elements.

If a valid association is created, the association information is returned in *assoc*. Valid element types are lines, line strings, shapes, arcs and ellipses.

tagValue1 is the tag (unique ID number) of the first intersecting root element; *tagValue2* is the tag value of the second intersecting root element.

index is the intersection index as returned by mdlIntersect_... routines.

Returns The mdlAssoc_createIntersection function returns SUCCESS (zero) if it is able to create a valid association. Otherwise, it returns a non-zero value.

See Also mdlAssoc_getPoint.

mdlAssoc_createKeypoint

```
#include <mdl.h>

int mdlAssoc_createKeypoint
(
  AssocPoint  *assoc,          /* <= arc association */
  ULong       tagValue,        /* => tag value of arc element */
  int         vertex,          /* => vertex before assoc point */
  int         numerator,       /* => dist from vertex * divisor */
  int         divisor          /* => # of keypoints in segment */
);
```

Description The mdlAssoc_createKeypoint function creates an association that represents a point on a linear element.

If a valid association is created, the association information is returned in *assoc*. Linear elements are lines, line strings and shapes.

tagValue is the tag (unique ID number) of the linear element that is the association's object (or root).

vertex is the index of the vertex directly preceding the association point.

numerator is the distance from vertex number *vertex* in units of *divisor* as described below. Its range must be between 0 and 32767.

divisor is the number of units (segments) to be considered between the points at *vertex* and *vertex+1*. The values of *numerator* and *divisor* are used together as the fraction of the distance between the points at *vertex* and *vertex+1*, where the association point will be located. The denominator must be between 1 and 32767.

Returns The mdlAssoc_createKeypoint function returns SUCCESS (zero) if it is able to create a valid association. Otherwise, it returns a non-zero value.

See Also mdlAssoc_createLinear, mdlAssoc_getPoint.

mdlAssoc_createLinear

```
#include <mdl.h>
#include <msassoc.fdf>

int mdlAssoc_createLinear
(
  AssocPoint  *assoc,    /* <= keypoint association */
  ULong       tagValue,  /* => element to associate to */
  DPoint3d    *inPoint,  /* => locate point */
  int         vertex,    /* => vertex before selected segment */
  int         lineNo     /* => line number (multilines only) */
);
```

Description The mdlAssoc_createLinear function creates an association that represents a point on a linear element. This function is similar to mdlAssoc_createKeypoint but has the advantage that it creates associations multi-line elements as well as other linear elements. Also, this function does require the *numerator* and *divisor* parameters as does mdlAssoc_createKeypoint. Instead, this function automatically calculates the keypoint numerator and divisor based on *inPoint* and the specified segment of the linear element.

If a valid association is created, the association information is returned in *assoc*.

tagValue is the tag (unique ID number) of the linear element that is the association's object (or root).

inPoint is the location on the specified linear element that is converted to an associative point.

vertex is the index of the vertex directly preceding the associative point.

fileNo is the file number of the element containing the unique ID *tagValue*.

Pass MASTERFILE (0) for the master file or 1-255 for reference files.

Returns mdlAssoc_createLinear returns SUCCESS if a valid linear association is created. Otherwise an non-zero error status is returned.

See Also mdlAssoc_createKeypoint.

mdlAssoc_createMline

```
#include <mdl.h>

int mdlAssoc_createMline
(
    AssocPoint *assoc,          /* <= arc association */
    ULong tagValue,             /* => tag value of multi-line elem */
    int vertex,                 /* => vertex before assoc point */
    int lineNo,                 /* => line num in mline profile */
    double offset,              /* => distance from vertex to point */
    int joint                    /* => assoc to joint; ignore offset */
);
```

Description The mdlAssoc_createMline function creates an association that represents a point on a multi-line element.

If a valid association is created, the association information is returned in *assoc*.

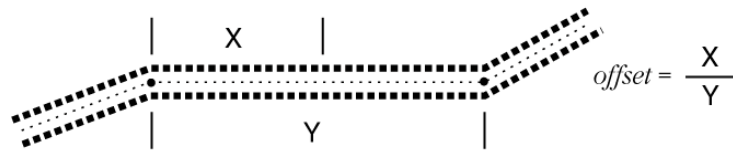
tagValue is the tag (unique ID number) of the multi-line element that is the association's object (or root).

vertex is the index of the vertex directly preceding or at the association point.

lineNo indicates which line in the multi-line is the association's object. This value is an index into the multi-line profile information that was copied into the multi-line from the tcb when the multi-line was created.

offset is the distance from the specified vertex to the association point, measured along the work line and divided by the work line length. The

information needed to calculate this parameter can be obtained using the mdlVec_... functions. *offset* is not used if *joint* is set to TRUE.



If *joint* is TRUE, the association point is at the intersection of the line specified by lineNo and the joint vector is at vertex. In other words, the association point will always be on the joint.

Returns The mdlAssoc_createMline function returns SUCCESS (zero) if it is able to create a valid association. Otherwise, it returns a non-zero value.

See Also mdlAssoc_getPoint, mdlVec_distance.

mdlAssoc_createOrigin

```
#include <mdl.h>

int mdlAssoc_createOrigin
(
  AssocPoint  *assoc,          /* <= origin association */
  ULONG       tagValue,        /* => tag value of base element */
  int         futureUse        /* => for future use, always pass 0 */
);
```

Description mdlAssoc_createOrigin creates an association that represents the origin of the base element. Origin associations can currently be created for cell headers, shared cells, text nodes and text elements. For elements other than these, mdlAssoc_createOrigin will create an associative point representing the lower left-hand corner of the range block for the element.

If a valid association is created, the association information is returned in *assoc*.

tagValue is the tag (unique ID number) of the element that is the association's object (or root).

Returns mdlAssoc_createOrigin returns SUCCESS (zero) if it is able to create a valid association. Otherwise, it returns a non-zero value.

See Also mdlAssoc_getPoint.

mdlAssoc_createBCurve

```
#include <mdl.h>

int mdlAssoc_createBCurve
(
    AssocPoint    *assoc,          /* <= origin association */
    ULong         tagValue,        /* => tag value of B-spline curve */
    double        *uParam,        /* => param. along B-spline curve */
);
```

Description mdlAssoc_createBCurve creates an association that represents a point along a B-spline curve.

If a valid association is created, the association information is returned in *assoc*.

tagValue is the tag (unique ID number) of the element that is the association's object (or root).

uParam is the parameter along the B-spline curve representing the associative point.

Returns mdlAssoc_createBCurve returns SUCCESS (zero) if it is able to create a valid association. Otherwise, it returns a non-zero value.

See Also mdlAssoc_getPoint.

mdlAssoc_getElement

```
#include <mdl.h>

int mdlAssoc_getElement
(
    MSElementUnion *element,      /* <= elem located by ID number */
    ULong           *filePos,      /* <= file position of element */
    ULong           elemID,        /* => element ID number */
    int             fileNum        /* => file number to search in */
);
```

Description mdlAssoc_getElement attempts to locate the element with the unique ID *elemID* in the design file specified by *fileNum*.

If found, the element is returned in *element* and the element's file position is returned in *filePos*. If NULL is passed for *element* or *filePos*, the corresponding data is not returned.

Returns mdlAssoc_getElement returns SUCCESS (zero) if the requested element is located. If the function is unable to locate the element, it returns a non-zero value.

See Also mdlAssoc_tagElement.

mdlAssoc_getPoint

```
#include <mdl.h>

int mdlAssoc_getPoint
(
    DPoint3d      *dPoint,          /* <= point extracted from assoc */
    AssocPoint    *assocPoint,      /* => association information */
    int           fileNumber        /* => file number */
);
```

Description mdlAssoc_getPoint uses the information in *assocPoint* to create a 3D data point that is returned in *dPoint*. An mdlAssoc_create... function must create the information in *assocPoint*.

fileNumber should be set to the file that the association is created in (generally MASTERFILE).

Returns mdlAssoc_getPoint returns SUCCESS (zero) if a valid point is created in *dPoint*. If the information in *assocPoint* is invalid, the function returns a non-zero value.

See Also mdlAssoc_createArc, mdlAssoc_createIntersection, mdlAssoc_createKeypoint, mdlAssoc_createMline.

mdlAssoc_isTagged

```
#include <mdl.h>

int mdlAssoc_isTagged
(
    ULong          *tagValue,        /* <= element ID number */
    MSElementUnion *element        /* => element to be tested */
);
```

Description mdlAssoc_isTagged checks an element for a tag (unique ID) added by MicroStation or the mdlAssoc_tagElement function.

If a tag is found, its value is returned in *tagValue*. If *tagValue* is NULL, the tag value will not be returned.

Returns mdlAssoc_isTagged returns TRUE if the element is tagged. Otherwise, it returns FALSE.

See Also mdlAssoc_tagElement.

mdlAssoc_tagElement

```
#include <mdl.h>

int mdlAssoc_tagElement
(
    ULONG    *tagValue,      /* <= element ID number */
    ULONG    filePos,       /* => file position of element to tag */
    int      futureArg      /* => for future use. Always pass 0 */
);
```

Description mdlAssoc_tagElement adds a tag (unique ID number) to the element at *filePos* in the current design file.

If a non-NULL address is passed, the tag value is returned in *tagValue*. To add tags to component elements of a complex element, you must tag the outermost complex header.

Returns The mdlAssoc_tagElement function returns SUCCESS if the element is successfully tagged. If the element at *filepos* is a member of a complex element or is too large to accommodate the tag data, an error status is returned.

See Also mdlAssoc_tagElementComplex, mdlAssoc_isTagged, mdlAssoc_getElement.

mdlAssoc_tagElementComplex

```
#include <mdl.h>
#include <msassoc.fdf>

int mdlAssoc_tagElementComplex
(
    ULONG    *newTagP,      /* <= tag for component of interest */
    ULONG    headerPos,     /* => header file position */
    ULONG    compPos,       /* => component file position */
    int      fileNo,        /* => file number */
    int      option         /* => for future use - pass zero */
);
```

Description The mdlAssoc_tagElementComplex function adds a tag (unique ID number) to each component element in the complex element located at file position *headerPos* in the design file indicated by *fileNo*. Each element in the complex element receives a different tag and the value of the tag assigned to the complex component located at the file position *compPos* is returned in *newTagP*.

The *option* parameter is currently unused. Pass zero to guarantee compatibility with future versions.

Returns mdlAssoc_tagElementComplex returns SUCCESS if the operation is successful. Otherwise a non zero error status is returned.

See Also mdlAssoc_tagElement.

mdlAssoc_getCurrent

```
#include <msassoc.fdf>

int mdlAssoc_getCurrent
(
  AssocPoint  *assoc,    /* <= current associative point */
  int         option     /* => option (pass zero) */
);
```

Description The `mdlAssoc_getCurrent` function copies the “current” associative point to the buffer pointed to by *assoc*. The “current” or “active” associative point is created each time a tentative point is entered while the association lock is on.

This function is usually called from the *dataPoint* function of a primitive placement command to retrieve the current associative point for use in element creation.

Returns `mdlAssoc_getCurrent` returns `TRUE` if a valid associative point is retrieved. A return value of `FALSE` indicates there is no valid “current” associative point.

See Also `mdlAssoc_getElement`, `mdlAssoc_getPoint`.

14

Reference Files

Reference files provide a convenient way to attach read-only files as references to the master design file.

Versions 4.0 and later support up to 255 reference files which can be configured through user preferences. The reference files can coincide with the master design file or can be translated, rotated and scaled.



Any reference file attachments not compatible with version 3.x or Vax (such as 2DREF/3DMASTER or those containing voids) are stored as Type 66, Level 5. These are identical in format to Type 5, Level 9.

The following functions are provided to manipulate reference files:

Function	Used to
mdlRefFile_attach mdlRefFile_extendedAttach [mdlLib.mli]	attach a reference file.
mdlRefFile_attachCoincident	attach a reference file coinciding with master file.
mdlRefFile_attachByView	attach a reference file by saved view.
mdlRefFile_createAttachment	create attachment elements for writing into other design files using the Work File functions.
mdlRefFile_writeAttachment	write reference file information to attachment element.
mdlRefFile_setClip	set reference file clipping points.
mdlRefFile_detach	detach a single reference file, or all attached reference files.
mdlRefFile_rotate	rotate an attached reference file.
mdlRefFile_scale [mdlLib.mli]	scale an attached reference file about a point.
mdlRefFile_reload	reload a reference file to reflect changes.
mdlRefFile_updateReference [mdlLib.mli]	update a reference file.
mdlRefFile_getParameters	retrieve reference file attachment settings.
mdlRefFile_setParameters	change reference file attachment settings.

Reference file attachment parameters

The reference file attachment information is stored in the `ReferenceFile` structure defined in the include file `reference.h`. The base of an array containing all attachment information is stored in the global variable `refFileP`. Reference file information can be accessed directly through `refFileP`. For example, `locate_lock` for reference file 3 could be enabled as follows:

```
#include <mdl.h>
#include <reference.h>
extern ReferenceFile *refFileP;
/* Turn on locate lock for reference file 3 */
refFileP[2].display.fb_opts.locate_lock = TRUE;
/* Write change back to the reference file attachment elem */

mdlRefFile_writeAttachment (2);
```

Changing the reference file attachment data by manipulating the `refFileP` structure does not change the reference file attachment element stored in the design file. Changing the file's attachment data is similar to changing a MicroStation setting without executing the FILE DESIGN command: the changes take effect immediately but are not remembered after the file is exited. To update the reference file attachment element and make the change permanent, call `mdlRefFile_writeAttachment`.

Clipping boundaries

MicroStation supports both inside (boundary) and outside (masking) clip polygons. Each reference file attachment must have exactly one boundary polygon and can optionally have one or more masking polygons. The boundary polygon is always the first polygon stored in the clip points array. Any masking polygons follow the boundary polygon. The clipping polygons are separated by disconnect points. These separator points have X and Y values of -2147483648 (defined as `DISCONNECT` in `msdefs.h`). Each clipping polygon must be closed, with identical first and last points. A reference file with a rectangular clipping boundary and a single rectangular clipping mask would have 11 points: 5 points for the boundary, followed by a disconnect point, and 5 points for the mask rectangle.

Transforming between reference and master file coordinates

```
typedef struct ref_display
{
    short    file_num;           /* file number */
    Fb_opts  fb_opts;           /* file builder options mask */
}
```

```

    Fd_opts    fd_opts;           /* file displayer options mask */
    short      trm_opts;          /* terminal options mask */
    short      disp_flags[8];     /* display flags */
    short      lev_flags[8][4];   /* level on/of flags */
    Point3d     mast_org;
    Point3d     ref_org;
    double      scale;
    byte        tds_levs[8];      /* levels currently in TDS */
    double      trns_mtrx[9];
} Ref_display;

```

The parameters for transforming reference file coordinates to master file coordinates are stored in the `Ref_display` structure displayed above. The *ref_org* and *mast_org* points represent the coordinate values in the reference and master files that coincide. It may be convenient to think of this as the attachment “pivot point.” The reference file coordinates are scaled (by *scale*) and rotated (by *trns_mtrx*) about this point.

A point in reference file coordinates is transformed to master file coordinates in the following manner:

1. Subtract the reference file origin
2. Scale by *scale*
3. Rotate by *trns_mtrx*
4. Add the master file origin

The following fragment of MDL code convert the point *refPoint* (reference file coordinates) to *masterPoint* (master file coordinates) for reference file 1:

```

Dpoint3d masterPoint, refPoint, referenceOrigin, masterOrigin;
mdlCnv_IPointToPoint(&masterOrigin, &refFileP[0].display.mast_org);
mdlCnv_IPointToPoint(&referenceOrigin, &refFileP[0].display.ref_org);
mdlVec_subtractPoint(&masterPoint, &refPoint, &referenceOrigin);
mdlVec_scale(&masterPoint, &masterPoint, refFileP[0].display.scale);
mdlRMMatrix_rotatePoint(&masterPoint, &refFileP[0].display.trns_mtrx);
mdlVec_addPoint(&masterPoint, &masterPoint, &masterOrigin);

```

Alternatively, `mdlTMatrix_referenceToMaster` returns the Transform from reference file to master coordinates (`mdlTMatrix_masterToReference` returns the Transform from master file to reference file coordinates):

```

mdlTMatrix_referenceToMaster(&transform, 1);
mdlTMatrix_transformPoint(&referencePoint, &transform);

```

mdlRefFile_attach

```
#include <mdl.h>

void mdlRefFile_attach
(
    char          *filename,          /* => name of file to attach */
    char          *logical,          /* => logical name */
    char          *description,      /* => description */
    Dpoint3d      *masterOrigin,     /* => master file origin */
    Dpoint3d      *referenceOrigin,  /* => reference file origin */
    double        scale,            /* => scale factor */
    RotMatrix     *rotMatrix,        /* => rotation from ref to master */
    int           nClipPoints,       /* => number of clipping points */
    Point2d       *clipPoints,       /* => clipping points */
    short         levels[8][4],      /* => level bit maps */
    boolean       snapLock,          /* => initial state of snap lock */
    boolean       locateLock        /* => initial state of locate lock */
);
```

Description mdlRefFile_attach provides the most flexible method for attaching a reference file. All reference file attachment parameters are passed as arguments to this function.

filename points to the name of the reference file to be attached. If a full file specification is not provided, the master file path and the environment variables MS_RFDIR and MS_DEF are searched sequentially for the file.

logical points to the reference file logical name, with a maximum of 20 characters. Unique logical names are required for duplicate reference file attachments.

description points to the reference file description, with a maximum of 40 characters. The description is optional.

masterOrigin points to the attachment origin in the current coordinate system.

referenceOrigin points to the attachment origin in the reference file. The reference file origin must be specified in reference file UORs, not the current coordinate system.

scale is the ratio of master file UORs to reference file UORs.

If the working unit values for each file are known, *scale* should be set to the ratio of master file UORs to reference file UORs. If the working units of the reference files are not known, the reference files should be attached coincident so that their subunits and positional units can be extracted. The scale factor can then be computed and submitted as the *scale* parameter.

rotMatrix points to a rotation matrix specifying the rotation from the reference file to the master file. NULL can be passed if no rotation is required.

nClipPoints and *clipPoints* are the number of clipping points and a pointer to the clipping point array. If the number of clipping points is zero or if NULL is passed for *clipPoints*, the clipping boundary is set to the entire design plane. The format of the clipping points array is discussed in detail at the beginning of this chapter.

levels points to an array of level bit maps for each of the eight views. If NULL is passed, the level configuration is extracted from the reference file.

snapLock and *locateLock* specify whether elements in the reference file can be snapped to or located.

The mdlRefFile_attach function returns SUCCESS if the reference file is successfully attached and the following values if an error occurs:

Value	Description
MDLERR_CANNOTOPENFILE	Reference file cannot be opened.
MDLERR_NOTDESIGNFILE	The specified file is not a valid design file.
MDLERR_3DREF2DMASTER	3D reference to 2D master file (not supported).
MDLERR_DUPLICATELOGICAL	Duplicate attachment with non-unique logical name.
MDLERR_INVALIDREFORG	Reference file origin off from design plane.
MDLERR_INVALIDMASTERORG	Master file origin is located beyond design plane.

See Also mdlRefFile_extendedAttach [mdl.lib.mli], mdlRefFile_attachCoincident, mdlRefFile_attachByView, mdlRefFile_getParameters.

mdlRefFile_extendedAttach [mdl.lib.mdl]

```

#include <mdl.lib.fdf>
#include <mdlerrs.h>

int mdlRefFile_extendedAttach
(
    char          *fileName,          /* => file name */
    char          *logicalName,       /* => logical name */
    char          *description,       /* => description */
    Dpoint3d      *masterOrigin,      /* => master origin */
    Dpoint3d      *refOrigin,         /* => reference origin */
    double        scale,              /* => scale */
    RotMatrix     *rotMatrix,         /* => rotation matrix */
    int           nClipPoints,        /* => number of clipping points */
    Point2d       *clipPoints,        /* => clipping points */
    double        zDelta,             /* => z Clipping limit */
    short         levels[MAX_VIEWS][4], /* => level map */
    boolean       snapLock,           /* => snap lock */
    boolean       locateLock,         /* => locate lock */
    int           nestDepth,          /* => nest depth limit */
    boolean       coincident          /* => attach coincident */
);

```

Description The `mdlRefFile_extendedAttach` function attaches a reference file with several extensions to the `mdlRefFile_attach` routine. The most significant extension is the ability to attach nested reference files.

fileName points to the name of the reference file to attach. If a full file specification is not provided, the master file path and the environment variables `MS_RFDIR` and `MS_DEF` are searched sequentially for the file.

logicalName points to the reference file logical name, with a maximum of 20 characters. Unique logical names are required for duplicate reference file attachments.

description points to the reference file description with a maximum of 40 characters. The description is optional.

masterOrigin points to the attachment origin in the current coordinate system.

refOrigin points to the attachment origin in the reference file. The reference file origin must be specified in reference file UORs, not the current coordinate system.

scale is the ratio between master file UORs and reference file UORs.

rotMatrix points to a rotation matrix specifying the rotation from the reference file to the master file. `NULL` can be passed if no rotation is required.

nClipPoints and *clipPoints* are the number of clipping points and a pointer to the clipping point array. If the number of clipping points is zero or if NULL is passed for clipPoints, the clipping boundary is set to the entire design plane.

zDelta is the distance from the master file origin to both the front and back clipping plane. As only one value is stored for the clipping distance, the master file origin must be centered between the front and back clipping planes.

levels points to an array of level bit maps for each of the eight views. If NULL is passed, the level configuration is extracted from the reference file views.

snapLock and *locateLock* controls whether snap lock and locate lock are enabled in the reference file.

nestDepth controls the attachment of nested reference files. A value of zero specifies that no nested attachments are to be made. A value of one indicates that an reference file attachment from the reference file should also be attached to the master file. Higher *nestDepth* values indicate that the attachments for the nested files themselves should be attached.

coincident should be non-zero if the attachment is coincident (no scaling, rotation, translation or clipping).

Returns mdlRefFile_extendedAttach returns the reference file slot number if the reference file is attached successfully. This slot number is the index into the *refFileP* global array where the reference file attachment information was inserted. One of the following values are returned if an error occurs:

Value	Description
MDLERR_CANNOTOPENFILE	Reference file cannot be opened.
MDLERR_NOTDESIGNFILE	The specified file is not a valid design file.
MDLERR_3DREF2DMASTER	3D reference to 2D master file (not supported).
MDLERR_DUPLICATELOGICAL	Duplicate attachment with non-unique logical name.
MDLERR_INVALIDREFORG	Reference file origin off from design plane.
MDLERR_INVALIDMASTERORG	Master file origin is located beyond design plane.

See Also mdlRefFile_attach, mdlRefFile_attachCoincident, mdlRefFile_attachByView, mdlRefFile_getParameters.

mdlRefFile_attachCoincident

```
#include <mdl.h>

void mdlRefFile_attachCoincident
(
    char    *filename,      /* => name of file to attach */
    char    *logical,      /* => logical name */
    char    *description,  /* => description (optional) */
    short   levels[8][4],  /* => level bit maps */
    boolean snapLock,      /* => initial state of snap lock */
    boolean locateLock     /* => initial state of locate lock */
);
```

Description The `mdlRefFile_attachCoincident` function attaches a reference file that coincides exactly with the master file.

filename points to the name of the reference file to be attached. If a full file specification is not provided, the master file path and the environment variables `MS_RFDIR` and `MS_DEF` are searched sequentially for the file.

logical points to the reference file logical name with a maximum of 20 characters. Unique logical names are required for duplicate reference file attachments.

description points to the reference file description with a maximum of 40 characters. The description is optional.

levels points to an array of level bit maps for each of the eight views. If `NULL` is passed, the level configuration is extracted from the reference file.

snapLock and *locateLock* specify whether elements in the reference file can be snapped to or located.

The `mdlRefFile_attachCoincident` function returns `SUCCESS` if the reference file is successfully attached and the following values if an error occurs:

Value	Description
<code>MDLERR_CANNOTOPENFILE</code>	Reference file cannot be opened.
<code>MDLERR_NOTDESIGNFILE</code>	The specified file is not a valid design file.
<code>MDLERR_3DREF2DMASTER</code>	3D reference to 2D master file (not supported).
<code>MDLERR_DUPLICATELOGICAL</code>	Duplicate attachment with non-unique logical name.

See Also `mdlRefFile_attach`, `mdlRefFile_attachByView`, `mdlRefFile_getParameters`.

mdlRefFile_attachByView

```
#include <mdl.h>

void mdlRefFile_attachByView
(
    char          *filename,      /* => name of file to attach */
    char          *logical,      /* => logical name */
    char          *description,  /* => description (optional) */
    char          *viewName,     /* => name of saved view */
    double        scale,         /* => scale factor */
    Dpoint3d      *centerPoint,  /* => center of view in master */
    short         levels[8][4],  /* => level bit maps */
    boolean       snapLock,      /* => initial state of snap lock */
    boolean       locateLock     /* => initial state of locate lock */
);
```

Description The mdlRefFile_attachByView function attaches a reference file with a saved view from the reference file, a center point and a scale factor.

filename points to the name of the reference file to be attached. If a full file specification is not provided, the master file path and the environment variables MS_RFDIR and MS_DEF are searched sequentially for the file.

logical points to the reference file logical name, with a maximum of 20 characters. Unique logical names are required for duplicate reference file attachments.

description points to the reference file description, with a maximum of 40 characters. The description is optional.

viewName is the name of the saved view to be used in attaching the reference file. This view must have been saved in the reference file.

scale is the ratio of master file UORs to reference file UORs.

centerPoint points to the center of the view attachment in the current coordinate system.

levels points to an array of level bit maps for each of the eight views. If NULL is passed, the level configuration is extracted from the saved view.

snapLock and *locateLock* specify whether elements in the reference file can be snapped to or located.

Returns The mdlRefFile_attachByView function returns SUCCESS if the reference file is successfully attached and the following values if an error occurs:

Value	Description
MDLERR_CANNOTOPENFILE	Reference file cannot be opened.
MDLERR_NOTDESIGNFILE	The specified file is not a valid design file.
MDLERR_3DREF2DMASTER	3D reference to 2D master file (not supported).

Value	Description
MDLERR_DUPLICATELOGICAL	Duplicate attachment with non-unique logical name.
MDLERR_VIEWNOTFOUND	<i>viewName</i> does not exist.
MDLERR_INVALIDMASTERORG	Master file origin is located beyond design plane.

See Also mdlRefFile_attach, mdlRefFile_attachCoincident, mdlRefFile_getParameters.

mdlRefFile_createAttachment

```
#include <mdl.h>

void mdlRefFile_createAttachment
(
    short      *type5ElmP      /* <= type 5 element */
    char       *filename,      /* => name of file to attach */
    char       *logical,       /* => logical name */
    char       *description,    /* => description */
    Dpoint3d   *masterOrigin,   /* => master file origin */
    Dpoint3d   *referenceOrigin, /* => reference file origin */
    double     scale,          /* => scale factor */
    RotMatrix  *rotMatrix,      /* => rotation from ref to master */
    int        nClipPoints,     /* => number of clipping points */
    Point2d    *clipPoints,     /* => clipping points */
    double     zlimit,          /* => z-limits */
    short      levels[8][4],    /* => level bit maps */
    boolean    snapLock,        /* => initial state of snap lock */
    boolean    locateLock,      /* => initial state of locate lock */
    int        slot,           /* => slot number */
    short      *displayFlagsP   /* => display flags (or NULL) */
);
```

Description mdlRefFile_createAttachment provides a flexible method for creating reference file attachment elements without storing the attachment information into the current master design file or actually performing the attachment. This function should only be used to create attachment elements for writing into other design files using the mdlWorkDgn_... functions. If any application needs to create attachments to the master design file, the mdlRefFile_attach... functions should be used. All reference file attachment parameters are passed as arguments to this function.

type5ElmP points to the MicroStation Type 5 element structure into which the newly created attachment is to be written. The structure should be defined as an MSElementUnion in the calling function. The type and level information returned in the element header will be either Type 5, Level 9 or Type 66, Level 5 depending upon which slot number was specified by the *slot* parameter (*slot* > 32 causes the creation of type 66, level 5 elements).

filename points to the name of the reference file to be attached. If a full file specification is not provided, the master file path and the environment variables `MS_RFDIR` and `MS_DEF` are searched sequentially for the file.

logical points to the reference file logical name, with a maximum of 20 characters. Unique logical names are required for duplicate reference file attachments.

description points to the reference file description, with a maximum of 40 characters. The description is optional.

masterOrigin points to the attachment origin in the current coordinate system.

referenceOrigin points to the attachment origin in the reference file. The reference file origin must be specified in reference file UORs, not the current coordinate system.

scale is the ratio of master file UORs to reference file UORs.

rotMatrix points to a rotation matrix specifying the rotation from the reference file to the master file. `NULL` can be passed if no rotation is required.

nClipPoints and *clipPoints* are the number of clipping points and a pointer to the clipping point array. If the number of clipping points is zero or if `NULL` is passed for *clipPoints*, the clipping boundary is set to the entire design plane. The format of the clipping points array is discussed in detail at the beginning of this chapter.

zlimit is the distance between the hither (near) clipping plane and projection screen and the distance between the yon (far) clipping plane and the projection screen. There is a distance of $2 * zlimit$ between the hither and yon projection planes.

levels points to an array of level bit maps for each of the eight views. If `NULL` is passed, the level configuration is extracted from the reference file.

snapLock and *locateLock* specify whether elements in the reference file can be snapped to or located.

slot is the slot number to be used in the `refFileP` array of reference file attachments. If this field is less than 0, MicroStation will find an open slot to use.

displayFlagsP points to the display (view) flags to be used. If none are to be used, pass `NULL`. The view flags structure, `Viewflags`, is published in `mstypes.h`.

Returns The `mdlRefFile_createAttachment` function always returns `SUCCESS`.

See Also `mdlRefFile_attachCoincident`, `mdlRefFile_attachByView`, Work File Functions.

mdlRefFile_writeAttachment

```
#include <mdl.h>

int mdlRefFile_writeAttachment
(
    int      refNumber      /* => reference file number */
);
```

Description Reference file attachment information is modified when the `refFileP` array is changed as described at the beginning of this section. Such changes will take effect immediately. However, they do not become permanent until they are written back to the reference file attachment elements. This process is similar to changing a MicroStation setting without executing the FILE (file design) command: the changes take effect immediately but are not remembered after the design file is exited. The `mdlRefFile_writeAttachment` function writes the attachment information for reference file *refNumber* to the reference file attachment element.

refNumber specifies the reference file number (beginning with zero).

Returns The `mdlRefFile_writeAttachment` function returns `SUCCESS` if the attachment information is successfully written and `MDLERR_INVALIDREF` if an invalid reference file number is passed.

See Also `mdlRefFile_createAttachment`.

mdlRefFile_setClip

```
int mdlRefFile_setClip
(
    int      refNumber,      /* => reference file number */
    Point2d  *points,        /* => clipping points */
    int      numPoints       /* => number of clipping points */
);
```

Description The `mdlRefFile_setClip` function resets the clipping polygon for a reference file. The clipping points are specified in UORs relative to the master file origin. The clipping points must include a boundary polygon. The boundary polygon can be optionally followed by one or more masking polygons. The format of the clipping polygons is discussed in detail at the beginning of this section. A maximum of 101 clipping points can be specified in each reference file attachment.



A pointer to the reference file clipping points is stored in the `ReferenceFile` structure. Due to memory allocation restrictions, an MDL application cannot change this pointer. The `mdlRefFile_setClip` function is provided for changing the reference file clipping without changing the clipping pointer directly.

Returns `mdlRefFile_setClip` returns `SUCCESS` if the clipping boundary is successfully updated and `MDLERR_INVALIDREF` if an invalid reference file number is passed.

mdlRefFile_detach

```
#include <mdl.h>

int mdlRefFile_detach
(
    int      refNumber      /* => reference file number */
);
```

Description The `mdlRefFile_detach` function detaches a single reference file specified by *refNumber*. It also detaches all reference files if *refNumber* is set to -1.

refNumber specifies the reference file number (beginning with zero). A value of -1 detaches all reference files.

Returns `mdlRefFile_detach` returns `SUCCESS` if the reference file is successfully detached. It returns `MDLERR_INVALIDREF` if an invalid reference file number is passed.

mdlRefFile_rotate

```
int mdlRefFile_rotate
(
    int      slot,          /* => reference slot (0-tcb->maxRefs-1) */
    DPoint3d *pivotP,       /* => clipping points */
    double xRotation,       /* => x rotation (radians) */
    double yRotation,       /* => y rotation (radians) */
    double zRotation,       /* => z rotation (radians) */
    int      view           /* => view number */
);
```

Description The `mdlRefFile_rotate` function rotates an attached reference file. The rotation is performed using the specified axis rotation values and a view number defining the view in which the rotation is to be based upon.

slot is the slot number (array index) in the `refFileP` attachment variable defining the reference file attachment the rotation is to be performed upon.

pivotP defines the pivot point in the current master file coordinate system upon which the rotation is based.

xRotation, *yRotation*, and *zRotation* are the rotation angles in radians by which the attachment is to be rotated. Only *zRotation* is valid for 2D design files – *xRotation* and *yRotation* are ignored in 2D design files.

view defines the master file view number in which the rotation is to be applied in 3D design files. This parameter is ignored in 2D design files.

Returns The `mdlRefFile_rotate` function returns `SUCCESS` if the reference file is successfully rotated and the following values if an error occurs:

Value	Description
<code>MDLERR_BADSLOT</code>	Reference file slot number was invalid.
<code>MDLERR_INVALIDMASTERORG</code>	Master file pivot point is located beyond the design plane.

See Also `mdlRefFile_scale` [`mdl1lib.m1`].

mdlRefFile_scale [mdl1lib.m1]

```
int mdlRefFile_scale
(
    int          refNumber,      /* => ref file # (0 - tcb->maxRefs-1) */
    Dpoint3d     *scalePointP,  /* => point to scale about */
    double       scale          /* => scale factor (applied to existing scale) */
);
```

Description The `mdlRefFile_scale` function scales the reference file specified by *refNumber* about the point *scalePointP* by the scale factor, *scale*.

Returns `mdlRefFile_scale` returns `SUCCESS` if the reference file is successfully scaled.

See Also `mdlRefFile_rotate`.

mdlRefFile_reload

```
int mdlRefFile_reload
(
    int          refNumber,      /* => reference file number */
    int          updateDisplay /* => TRUE=update display once reloaded */
);
```

Description The `mdlRefFile_reload` function reloads the reference file specified by *refNumber*. The reference file is closed and reopened, and the element cache for the reference file is regenerated to reflect the current file contents. This is useful for synchronizing a reference file whose contents have been altered while MicroStation has it attached. If *updateDisplay* is nonzero, the reference file display is updated to reflect the reloaded geometry.

Returns `mdlRefFile_reload` returns `SUCCESS` if the reference file is successfully reloaded and `MDLERR_INVALIDREF` if an invalid reference file number is passed.

See Also `mdlRefFile_updateReference` [`mdl1lib.m1`].

mdlRefFile_updateReference [mdl.lib.mli]

```
Public int mdlRefFile_updateReference
(
    int    refNumber,      /* => reference file number (0 - maxRefs-1) */
    int    displayMode     /* => display mode */
);
```

Description The mdlRefFile_updateReference function updates the reference file specified by *refNumber* in the display mode specified by *displayMode*. The display mode definitions are included in msdefs.h (NORMALDRAW, ERASE, TEMPDRAW, TEMPERASE, etc.)

Returns mdlRefFile_updateReference returns SUCCESS if the reference file is successfully updated and MDLERR_INVALIDREF if an invalid reference file number is passed.

See Also mdlRefFile_reload.

mdlRefFile_getParameters, mdlRefFile_setParameters

```
#include <mdl.h>

int mdlRefFile_getParameters
(
    void    *param,
    int     paramName,
    int     refSlot
);

int mdlRefFile_setParameters
(
    void    *param,
    int     paramName,
    int     refSlot
);
```

Description The mdlRefFile_getParameters function is used to retrieve reference file attachment settings for the reference file specified by the *refSlot* parameter. The parameter returned in *param* is determined by the value of *paramName*.

The mdlRefFile_setParameters function is used to change reference file attachment settings. The setting to be changed is determined by the value of *paramName* as tabulated below:



In releases of MicroStation prior to version 5, it was possible to simply look in the refFileP built-in variable and get or set these values. In version 5.0, this is no longer advisable, particularly for the REFERENCE_DISPLAY value. In general, using the functions rather than

accessing the structure directly results in better insulation from MicroStation details.

paramName	Type for <i>param</i>
REFERENCE_DISPLAY	int
REFERENCE_SNAP	int
REFERENCE_LOCATE	int
REFERENCE_SLOTACTIVE	int
REFERENCE_SCALELINESTYLES	int
REFERENCE_FILENOTFOUND	int
REFERENCE_COMPLETE_PATH_STORED	int
REFERENCE_HIDDEN_LINE	int
REFERENCE_DESCRIPTION	char *
REFERENCE_LOGICAL	char *
REFERENCE_SCALE	double
REFERENCE_ROTATION	rotmatrix <reference.h>
REFERENCE_LEVELFLAGS	short[8][4] <reference.h>
REFERENCE_UNITDESC	Unit_desc <reference.h>
REFERENCE_SCALE_MASTERUNITS	double



REFERENCE_SLOTACTIVE is valid only for mdlRefFile_getParameters.



When passing values listed below to mdlParams_setActive, pay special attention to whether you should be passing a pointer or a value in *param*. mdlParams_getActive will always require a pointer for *param*. Integer parameters are passed by value in mdlRefFile_setParameters.



In calls to mdlRefFile_setParameters, if *param* is an int, pass value rather than pointer.

Example

```
/* To set display on: */
mdlRrfFile_setParamters((void *) 1, REFERENCE_DISPLAY, SLOT)
```

Returns mdlRefFile_getParameters returns SUCCESS if the reference file slot is in range and the specified slot is in use. If *refSlot* is less than 0 or greater than *tcb->maxRefs*, MDLERR_BADSLOT is returned.

See Also mdlRefFile_attach.

15

Cells

This chapter contains the functions that allow for manipulation of MicroStation cells.

This chapter discusses:

- Cell functions
- Shared cell functions

Cell Functions

The following table lists cell functions:

Function	Used to
<code>mdlCell_attachLibrary</code>	attach a new cell library.
<code>mdlCell_getFilePosition</code> <code>mdlCell_getFilePosInLibrary</code>	get a cell's position in a cell library.
<code>mdlCell_existsInLibrary</code>	indicate whether the specified cell exists in the current cell library.
<code>mdlCell_createFromFence</code>	create a cell in a cell library from elements in the fence or selection set.
<code>mdlCell_addLibDescr</code>	add a cell to a cell library from an element descriptor.
<code>mdlCell_addLibElement</code>	add an element to a cell library.
<code>mdlCell_rewriteLibElement</code>	rewrite an element in a cell library.
<code>mdlCell_deleteInLibrary</code>	delete an existing cell from a cell library.
<code>mdlCell_rename</code>	rename a cell in a cell library.
<code>mdlCell_placeCell</code>	place a cell from a cell library in the design file.
<code>mdlCell_getElmDescr</code>	read a cell definition from a cell library and create an element descriptor.
<code>mdlCell_fixLevels</code>	adjust element levels in a cell element descriptor.
<code>mdlCell_isPointCell</code>	determine if a cell is a point or normal cell.
<code>mdlCell_createLibraryHeader</code>	create a new cell library header element.
<code>mdlCell_setRange</code>	set a range for a cell element.

Function	Used to
<code>mdlCell_generateLibIndex</code>	re-index the current cell library.
<code>mdlCell_begin</code>	add a cell header to the design file.
<code>mdlCell_end</code>	finish a cell definition in the design file.

Example

See `cell.mc`.

mdlCell_attachLibrary

```
int mdlCell_attachLibrary
(
    char    *fullName,      /* <= full name of attached library */
    char    *inputName,     /* => name of library to attach */
    char    *defaultDir,    /* => default dir to search (or NULL) */
    boolean fromKeyin       /* => if TRUE, save from inputName */
);
```

Description `mdlCell_attachLibrary` attaches a new cell library to the current design file. If a library is successfully attached, *fullName* is set to the cell library's full file specification.

inputName is the name of the cell library. *inputName* can contain a path specification. However, if it does not then *defaultDir* is used for the path.

If *fromKeyin* is TRUE, the path information from *inputName* is used. If it is FALSE, `mdlCell_attachLibrary` assumes that *inputName* came from the information stored in the design file header (where the path information is not always correct), and it only uses the path information from *inputName* if the library cannot be found anywhere else. MDL applications should normally set *fromKeyin* to TRUE.



To detach a cell library, pass a NULL string for *inputName*.

Returns `mdlCell_attachLibrary` returns SUCCESS if a cell library is attached. If it cannot find the cell library, it returns a non-zero value.

mdlCell_getFilePosition, mdlCell_getFilePosInLibrary

```
int mdlCell_getFilePosition
(
    ULong   *filePos,       /* <= position in cell library of cell */
    char    *cellName,     /* => name of cell */
    long    *indexPos,     /* <= OPTIONAL, position in index file */
    int     searchAll       /* => search libs in MS_CELLLIST? */
);
```

```
ULong mdlCell_getFilePosInLibrary
(
char    *cellName      /* => name of cell */
);
```

Description mdlCell_getFilePosition returns the file position cell *cellName* in a cell library. This function is used by MicroStation whenever it searches for a cell to be placed. It first looks for the cell in the currently attached cell library. If the cell cannot be found in that library, MicroStation begins searching the cell libraries in the MS_CELLLIST list. If the cell is located in a library defined by MS_CELLLIST, that library becomes the active “alternate” library and that library is accessed by all MDL functions that read from cell libraries until another call is made to mdlCell_getFilePosition.

The file position of the cell is returned in *filePos*.

If *indexPos* is non-NULL, the position of the cell entry in the cell index file is returned. This is rarely useful.

If *searchAll* is TRUE, mdlCell_getFilePosition searches all libraries in MS_CELLLIST, otherwise it only searches the currently attached cell library.

The mdlCell_getFilePosInLibrary function returns *cellName*’s file position in the current cell library. mdlCell_getFilePosInLibrary only searches the current cell library. This routine exists for compatibility with version 4.0 MDL.

Returns mdlCell_getFilePosition returns SUCCESS if the cell is found and *filePos* is valid. Otherwise it returns MDLERR_CELLNOTFOUND.

The mdlCell_getFilePosInLibrary function returns *cellName*’s file position in the cell library. If the cell cannot be located, it returns 0 and sets mdlErrno to a value that indicates the error cause. Possible values for mdlErrno are MDLERR_NOCELLLIBRARY and MDLERR_NOMATCH.

See Also mdlCell_existsInLibrary, mdlCell_placeCell, mdlCell_getElmDscr.

mdlCell_existsInLibrary

```
#include <mselems.h>

boolean mdlCell_existsInLibrary
(
char    *cellName      /* => name of cell */
);
```

Description The mdlCell_existsInLibrary function returns an indication of whether *cellName* exists in the current cell library.

Returns The mdlCell_existsInLibrary function returns TRUE if the cell name exists in the library; otherwise it returns FALSE.

See Also mdlCell_getFilePosInLibrary, mdlCell_placeCell, mdlCell_getElmDscr.

mdlCell_createFromFence, mdlCell_addLibDescr

```
#include <mselems.h>

ULong mdlCell_createFromFence
(
    char    *cellName,      /* => name of cell */
    char    *description,   /* => description of cell */
    Dpoint3d *cellOrigin,   /* => origin of cell */
    int     cellType        /* => cell type */
);

ULong mdlCell_addLibDescr
(
    MSElementDescr *cellDescrP /* => element descr for cell */
);
```

Description `mdlCell_createFromFence` and `mdlCell_addLibDescr` create a new cell in the current cell library.

`mdlCell_createFromFence` creates a new cell that contains the elements in the fence. It creates the new cell from the current selection set if no fence is defined.

cellName is the name of the cell. *cellName* should have six or fewer ASCII characters.

description is an ASCII string of 27 or fewer characters.

cellOrigin is the origin of the cell. If `NULL` is passed for *cellOrigin*, the (0, 0) point for the current coordinate system is used.

cellType is an integer defining the type of cell to be created. Possible values for *cellType* are 1 for a normal graphic cell, 2 for a point cell, 3 for a menu cell, and 5 for a tutorial cell.

The `mdlCell_addLibDescr` function adds a new cell from the element descriptor *cellDescrP* to the currently attached cell library.

`mdlCell_addLibDescr` is a more advanced routine than `mdlCell_createFromFence` and requires a valid cell header (type 1) element descriptor. You must transform all elements in the element descriptor about the cell origin before calling this routine.

Returns The `mdlCell_createFromFence` and `mdlCell_addLibDescr` functions return the file position in the cell library. If an error occurs during cell processing, these functions return 0 for the file position. They also set `mdlErrno` to the value that indicates the error cause. Possible values are `MDLERR_CELLTOOLARGE`, `MDLERR_INVALIDLIBRARY`, `MDLERR_NOCELLLIBRARY` and `MDLERR_FILEONLY`.

See Also `mdlCell_addLibElement`, `mdlCell_rewriteLibElement`, `mdlCell_createLibraryHeader`.

mdlCell_addLibElement, mdlCell_rewriteLibElement

```
#include <mselems.h>

ULong mdlCell_addLibElement
(
    MSElementUnion    *cellElement    /* => elm to add to library */
);

void mdlCell_rewriteLibElement
(
    MSElementUnion    *cellElement,    /* => elm to update in library */
    ULong              filePos          /* => file pos. in cell library */
);
```

Description mdlCell_addLibElement and mdlCell_rewriteLibElement directly write elements to the current cell library. These functions should be used with care, since they can corrupt the cell library. mdlCell_addLibDescr or mdlCell_createFromFence should almost always be used for adding cells to a cell library.

mdlCell_addLibElement adds a single element, *cellElement*, to a cell library.

The mdlCell_rewriteLibElement function overwrites the element in the cell library at file position, *filePos*, with the new element, *cellElement*.



If cells are added to a cell library with these calls, the cell library must be re-indexed manually with mdlCell_generateLibIndex.

Returns The mdlCell_addLibElement function returns the added element's file position in the cell library. If an error occurs, it returns 0 for the file position and sets mdlErrno to the specific error cause. Possible values for mdlErrno are MDLERR_INVALIDLIBRARY, MDLERR_NOCELLLIBRARY and MDLERR_FILEREADONLY.

The mdlCell_rewriteLibElement function is of type void. It returns no value.

See Also mdlCell_addLibDescr, mdlCell_createFromFence, mdlCell_createLibraryHeader, mdlCell_generateLibIndex.

mdlCell_deleteInLibrary, mdlCell_rename

```
#include <mselems.h>

int mdlCell_deleteInLibrary
(
```

```

char    *cellName      /* => name of cell to be deleted */
);

int mdlCell_rename
(
char    *newName,      /* => new name for cell in library */
char    *oldName       /* => old name of cell in library */
);

```

Description The function `mdlCell_deleteInLibrary` removes the cell, *cellName*, from the current cell library.

The `mdlCell_rename` function changes the name of cell, *oldName*, in the current cell library to the new name, *newName*.

Returns The `mdlCell_deleteInLibrary` and `mdlCell_rename` functions return `SUCCESS` if the specified cell was deleted or renamed. If an error occurs, they return one of the following values: `MDLERR_CELLNOTFOUND`, `MDLERR_CELLEXISTS`, `MDLERR_INVALIDLIBRARY`, `MDLERR_NOCELLLIBRARY` or `MDLERR_FILEONLY`.

mdlCell_placeCell

```

#include <mselems.h>

ULong mdlCell_placeCell
(
ULong    cellFilePos,  /* => cell's file pos. in library */
Dpoint3d *origin,      /* => origin for placement */
Dpoint3d *scale,       /* => scale factors */
RotMatrix *rMatrix,    /* => rotation matrix for placement */
short    *attributes,  /* => attr data to append to header */
int       ggroup,      /* => graphic group number */
int       relativeMode, /* => relative to baseLevel */
int       baseLevel,    /* => used for point cells/relative mode */
int       sharedFlag,   /* => 0=not shared, 1=shared, 2=current */
char     *cellName     /* => name of cell (optional) */
);

```

Description The `mdlCell_placeCell` function places a cell in the design file.

cellFilePos is the cell's file position in the cell library. This value can be obtained with the `mdlCell_getFilePosInLibrary` function.

If *cellFilePos* is 0, `mdlCell_placeCell` scans the library for the cell using *cellName*. If *sharedFlag* is 1 and a shared cell definition for *cellName* exists in the design file, a new shared cell instance is placed without the cell library being read, and *cellFilePos* is ignored.

origin is the location to place the cell origin. If *origin* is `NULL`, the cell is placed at the (0, 0, 0) point in the current coordinate system.

scale points to a `Dpoint3d` structure holding the X, Y, and (in 3D) Z scale factors to be applied to the cell elements before they are placed. If *scale* is `NULL`, the cell is placed at a scale factor of 1.0.

rMatrix is the rotation matrix that defines the orientation for cell placement. If *rMatrix* is `NULL`, the identity matrix is used. *rMatrix* does not necessarily need to be orthogonal or normalized. (Cells can be placed skewed).

attributes is an array of attribute information to append to the cell header before it is placed. The first `short` of this array is the length, in `shorts`, of the attribute data. Note that attribute linkage lengths must be a multiple of four words. If *attributes* is `NULL`, the cell has no attributes attached to the header.

ggroup is the graphic group number for the cell's elements. A value of 0 means that the elements will not be part of a graphic group.

relativeMode and *baseLevel* determine how the levels for the cell's elements are assigned. All elements for a point cell are assigned to *baseLevel*. If *relativeMode* is `TRUE`, the lowest level used in the cell definition is assigned to *baseLevel* and all other element levels are adjusted accordingly. If *relativeMode* is `FALSE`, the levels are taken from the cell library unchanged.

sharedFlag indicates whether the cell should be placed as shared or unshared. If *sharedFlag* is 0, an unshared cell is placed. If *sharedFlag* is 1, a shared cell is created. If *sharedFlag* is 2, the state of the user-specified shared flag is used. `mdlCell_placeCell` also creates and places the shared cell definition.

cellName is the name of the cell to be placed. It is used only when *cellFilePos* is 0. Or it is used for placing a shared cell. Otherwise, it is ignored.



The rotation matrix, *rMatrix*, is applied before the scale factors.

Returns The `mdlCell_placeCell` function returns the file position of a newly placed cell (or the file position of the shared cell definition if the function needed to create one). If an error occurs, it returns 0 and sets `mdlErrno` to the specific error cause.

See Also `mdlCell_getFilePosInLibrary`, `mdlCell_getElmDscr`.

mdlCell_getElmDscr

```
#include <mselems.h>

int mdlCell_getElmDscr
(
  MSElementDescr **cellDscrPP, /* <= cell element descriptor */
  MSElementDescr **txtNodeDscrPP, /* <= elem descr for text node */

```

```

ULong      cellFilePos,      /* => cell's file pos in lib */
Dpoint3d   *origin,         /* => origin for placement */
Dpoint3d   *scale,          /* => scale factors */
RotMatrix  *rMatrix,        /* => rot matrix for placement */
short      *attributes,     /* => append attr data to header*/
int        ggroup,          /* => graphic group number */
int        sharedFlag,      /* => 0=no, 1=shared, 2=current */
char       *cellName        /* => name of cell (optional) */
);

```

Description The `mdlCell_getElmDscr` function reads the specified cell from the cell library and returns the cell's element descriptor. The address of the element descriptor for the cell's graphic elements is returned in *cellDscrPP*.

If empty text nodes are in a cell, MicroStation removes them before it places the cell. These empty text nodes are placed immediately after the cell. For this reason, `mdlCell_getElmDscr` also extracts the empty text node elements and returns them in a separate element descriptor, *txtNodeDscrPP*. Applications should treat these empty text nodes similarly.

cellFilePos is the cell's file position in the cell library. This value can be obtained through `mdlCell_getFilePosInLibrary`. If *cellFilePos* is 0, `mdlCell_getElmDscr` scans the library for the cell using *cellName*. If *sharedFlag* is 1 or 2 and the user has turned shared cells on, the function places a new shared cell instance without reading the cell library and *cellFilePos* is ignored.

origin is the location of the cell's origin. If *origin* is `NULL`, the cell is transformed to the (0, 0, 0) point in the current coordinate system.

scale points to a `Dpoint3d` structure holding the X, Y and (in 3D) Z scale factors to be applied to the cell's elements. If *scale* is `NULL`, the cell is not scaled.

rMatrix is the rotation matrix that defines the cell's orientation. If *rMatrix* is `NULL`, the identity matrix is used.

attributes is an array of attribute information to append to the cell header. If *attributes* is `NULL`, the cell has no attributes attached to the header.

ggroup is the graphic group number for the cell's elements. A value of 0 means that the elements will not be part of a graphic group.

sharedFlag indicates whether the element descriptor is for a shared or unshared cell. If *sharedFlag* is 0, an unshared cell is returned. If *sharedFlag* is 1, a shared cell is returned. If *sharedFlag* is 2, the state of the user-specified shared flag is used.

cellName is the name of the cell to be returned. The function uses it only when *cellFilePos* is 0 or when the function reads a shared cell. Otherwise, *cellName* is ignored.

Returns The mdlCell_getElmDscr function returns SUCCESS if the cell is read and *cellDscrPP* is valid. Otherwise, it returns ERROR.

See Also mdlCell_getFilePosInLibrary, mdlCell_placeCell.

mdlCell_fixLevels

```
#include <mselems.h>

void mdlCell_fixLevels
(
  MSElementDscr    *cellDscrP,    /* => cell element descriptor */
  int               relativeMode, /* => relative or absolute */
  int               baseLevel      /* => used for point cells and
                                   relative mode */
);
```

Description The mdlCell_fixLevels function adjusts element levels in a cell element descriptor, *cellDscrP*. This adjustment is based on the cell type and the values of *relativeMode* and *baseLevel*. It uses the following logic:

```
if (cell is a point cell)
{
    set all levels to baseLevel
}
else if (relativeMode is TRUE)
{
    adjust so the lowest level used in cell is baseLevel
}
```

This function is needed because mdlCell_getElmDscr does not apply the above logic before it returns the cell's element descriptor. This functionality was left out of mdlCell_getElmDscr so that you could get the original level values.

Returns The mdlCell_fixLevels function is of type void. It returns no value.

See Also mdlCell_getElmDscr.

mdlCell_isPointCell

```
#include <mselems.h>

boolean mdlCell_isPointCell
(
  MSElementUnion    *cellHeader    /* => cell header element */
);
```

Description The `mdlCell_isPointCell` function determines whether the cell *cellHeader* is a point cell.

Returns The `mdlCell_isPointCell` function returns `TRUE` if *cellHeader* is a point cell. Otherwise, it returns `FALSE`.

mdlCell_createLibraryHeader, mdlCell_setRange

```
#include <mselems.h>

void mdlCell_createLibraryHeader
(
  Cell_Lib_Hdr      *cellHeader,    /* <= cell header element */
  char              *cellName,       /* => name of cell to be created*/
  char              *cellDescription, /* => cell description */
  int               cellType         /* => cell type */
);

void mdlCell_setRange
(
  MSElementDescr   *cellDescrP     /* <=> cell's element descriptor */
);
```

Description The `mdlCell_createLibraryHeader` and `mdlCell_setRange` functions are generally used to create library cells in memory for subsequent addition to a cell library using `mdlCell_addLibDescr`. The programmer typically creates a library header, creates a new element descriptor, adds the component elements to that element descriptor, calls `mdlCell_setRange`, and then adds the cell to the library using `mdlCell_addLibDescr`.

The `mdlCell_createLibraryHeader` function creates a new library header (type 1 element) in *cellHeader*.

cellName is the name of the cell (in ASCII).

cellDescription is the description of the cell (in ASCII).

cellType is the type of cell to be created. See `mdlCell_createFromFence` for values for *cellType*.

The `mdlCell_setRange` function must be called to update the range block diagonal in the header of a cell library element descriptor. The range block diagonal is the range of the un-rotated cell, so this function must be called after the cell definition is complete (immediately before `mdlCell_addLibDescr` is called).

Returns The `mdlCell_createLibraryHeader` and `mdlCell_setRange` functions are of type `void`. They return no values.

See Also `mdlCell_addLibDescr`.

mdlCell_generateLibIndex

```
#include <mselems.h>

void mdlCell_generateLibIndex
(
    boolean      messages /* => put out indexed messages */
);
```

Description MicroStation maintains an index file for cell libraries. This file contains the names and file positions of all cells in the library. If a cell library's contents are changed with `mdlCell_addLibElement` or `mdlCell_rewriteLibElement`, `mdlCell_generateLibIndex` must be called to regenerate the index file.

messages is a flag that indicates whether to display the "Cells indexed" status message.

Returns The `mdlCell_generateLibIndex` function is of type `void`. It returns no value.

mdlCell_begin

```
#include<mselems.h>

ULong mdlCell_begin
(
    char          *cellName,      /* => name of cell */
    Dpoint3d      *origin,        /* => origin of cell */
    short         *attributes,     /* => attributes to append to hdr */
    int           pointCell       /* => create point cell? */
);
```

Description `mdlCell_begin` creates a cell header in the design file and returns its file position.

cellName points to a character string that contains the cell name. Cell names should have six characters or less and use only valid Radix50 characters. (See `mdlCnv_fromAsciiToR50`.)

origin points to a `Dpoint3d` structure that contains the cell's origin in the current coordinate system.

attributes points to an array of shorts containing any attribute information to be attached to the cell header. The first member of the array is the length of the attribute data. If you do not want to append attributes to the cell header, set *attributes*=NULL.

pointCell indicates whether the cell will be a point cell. If *pointCell* is TRUE, a point cell header is created. Otherwise, a normal cell header is created.



Calls to `mdlCell_begin` should be matched with subsequent calls to `mdlCell_end`.



The `mdlCell_begin` function is provided mainly for compatibility with MicroCSL. Its use is discouraged. Element descriptors should usually be used for creating cells.

Returns The `mdlCell_begin` function returns the file position of the cell header created. If an error occurs, it returns zero and `mdlErrno` is set to the specific error cause. See `mdlElement_add` for the possible values of `mdlErrno`.

See Also `mdlCell_end`.

mdlCell_end

```
#include <mselems.h>

int mdlCell_end
(
    ULong    headerFilePos    /* => returned by mdlCell_begin */
);
```

Description The `mdlCell_end` function finishes the cell definition started with a call to `mdlCell_begin`. *headerFilePos* is the file position of the cell header that `mdlCell_begin` returned. All elements between *headerFilePos* and the end of file should be part of the cell definition.

The `mdlCell_end` function sets the cell header range to the union of the ranges of cell elements. It also sets the level bit mask, the words in the description, and the class bit mask in the cell header. And it ensures that complex bits are set for all elements in the cell definition.

Returns `mdlCell_end` returns `SUCCESS` if the operations described above are accomplished; it returns `ERROR` if the element that *headerFilePos* points to is not a cell header.

See Also `mdlCell_begin`.

Shared Cell Functions

The shared cell functions provide techniques for manipulating shared cells and shared cell definitions.

The following table lists shared cell functions:

Function	Used to
mdlSharedCell_read	given a shared cell instance, read the shared cell definition.
mdlSharedCell_create	create a shared cell instance element.
mdlSharedCell_fromCellHeader	given a cell header, create a shared cell instance element.
mdlSharedCell_addToFile	add a shared cell instance to the file and display it.
mdlSharedCell_createDefinitionElement [mdl1lib.m1]	create a shared cell definition element.
mdlSharedCell_addDefinitionElements	add a shared cell definition in the file (if a definition does not already exist).
mdlSharedCell_makeSureDefExists	ensure that the definition for a shared cell exists in the current design file.
mdlSharedCell_find	find shared cell definition if one exists.
mdlSharedCell_dropOneLevel	drop sharing one level.
mdlSharedCell_dropToNormalCell	drop a shared cell to a normal (unshared) cell.
mdlSharedCell_redefine	redefine shared cell definition.
mdlSharedCell_deleteDefinition	delete a shared cell definition from the design file.
mdlSharedCell_drawAllShares	draw all instances of a shared cell name.
mdlSharedCell_setRange	set range of a shared cell instance based on current shared cell definition.
mdlSharedCell_toNormalCell	get an element descriptor that is the normal (unshared) cell equivalent of a shared cell instance.

mdlSharedCell_read

```
#include <mselems.h>

int mdlSharedCell_read
(
  MSElementDescr    **scDefPP,          /* <= shared cell elm descr */
  MSElementUnion *sCell,                /* => shared cell instance element */
  boolean            transformToWorld,    /* => trans. def by instance */
  int                fileName,           /* => origin of sCell */
  boolean            expandNested        /* => expand nested shared cells*/
);
```

Description `mdlSharedCell_read` reads the shared cell definition for a shared cell instance, *sCell*. It allocates a new element descriptor and returns its address in *scDefPP*.

If *transformToWorld* is `TRUE`, the function transforms the elements of the shared cell definition to the origin, scale, and rotation of the shared cell instance. Otherwise, they remain at the origin of the world coordinate system with a scale factor of 1 and no rotation. Applications will normally pass `TRUE` for *transformToWorld*.

fileName is the file number from which *sCell* originated. This number determines the design file in which to search for the shared cell definition.

If *expandNested* is `TRUE`, `mdlSharedCell_read` expands any nested shared cell instances. Otherwise, it leaves them as shared cell instances in the element descriptor. Applications normally pass `TRUE` for *expandNested*.

Returns The `mdlSharedCell_read` function returns `SUCCESS` if the shared cell definition is read and *scDefPP* is valid. Otherwise, it returns `ERROR`.

mdlSharedCell_create, mdlSharedCell_fromCellHeader

```
#include <mselems.h>

void mdlSharedCell_create
(
  MSElementUnion *scElement,    /* <= shared cell instance element */
  MSElementUnion *in,           /* => existing shared cell instance */
  Dpoint3d        *origin,       /* => origin of instance */
  RotMatrix        *rMatrix,     /* => rotation matrix of instance */
  Dpoint3d        *scale,        /* => scale factors for instance */
  char             *cellName,    /* => cell name */
  SCOverride       *scOverride,  /* => override bit mask */
  boolean          relativeMode, /* => levels assigned relative */
  int              baseLevel,    /* => level for instance */
  AssocPoint       *assocOrigin /* => origin is assoc point */
);

void mdlSharedCell_fromCellHeader
(
```

```

MSElementUnion *scElement,    /* <= shared cell instance element */
MSElementUnion *cellHeader,   /* => cell header element */
RotMatrix *rotScale,          /* => rotation and scaling */
boolean relativeMode,         /* => levels assigned relative */
int baseLevel,                /* => level for instance */
AssocPoint *assocOrigin      /* => origin is assoc point */
);

```

Description The `mdlSharedCell_create` function creates a shared cell instance element from the parameters passed to it.

in is an existing shared cell instance element.

origin is the origin of the shared cell instance. If *origin* is `NULL`, the (0, 0, 0) point of the current coordinate system is used.

rMatrix is the rotation matrix for the shared cell instance. If *rMatrix* is `NULL`, the identity matrix is used.

scale points to a `Dpoint3d` structure that contains the X, Y and Z scaling factors to be applied to the shared cell instance. If *scale* is `NULL`, a scale of 1.0 is used.

cellName is the name of the shared cell instance. If *cellName* is `NULL`, the name is taken from *in*.

scOverride is the override mask for the shared cell instance.

relativeMode indicates whether the shared cell levels will be relative or absolute to the levels in the shared cell definition.

baseLevel is the level assigned to the shared cell instance element. This level is used if *relativeMode* is `TRUE`.

assocOrigin is an associative point to use for the shared cell instance origin. If *assocOrigin* is `NULL`, the origin of *cellHeader* is used.

The `mdlSharedCell_fromCellHeader` function converts the cell header, *cellHeader*, to a shared cell instance element in *scElement*.

rotScale is the non-orthogonal rotation and scaling matrix applied to the shared cell definition.



The `mdlSharedCell_fromCellHeader` function assumes that the range block in the cell header element is valid. If it is not, `mdlSharedCell_setRange` should be called for the new shared cell instance.

Returns The `mdlSharedCell_create` and `mdlSharedCell_fromCellHeader` functions are of type `void`. They return no values.

See Also `mdlSharedCell_setRange`.

mdlSharedCell_addToFile

```

ULong mdlSharedCell_addToFile
(
  MSElementUnion *cellHeader, /* => cell hdr or shared cell def. */
  RotMatrix *rMatrix, /* => rot. matrix for shared cell */
  int relativeMode, /* => 0 = absolute, 1 = relative */
  int baseLevel, /* => base level if relative mode */
  AssocPoint *assocOrigin /* => always pass NULL */
);

```

Description The `mdlSharedCell_addToFile` function adds a shared cell instance to the file and displays the new shared cell.

cellHeader is a pointer to a shared cell definition element or a cell header element.

rMatrix is a pointer to a rotation matrix. If *cellHeader* is a library cell header, then *rMatrix* is used as the rotation matrix for the new shared cell instance. If *cellHeader* is a shared cell definition, the rotation matrix from the shared cell definition is used and *rMatrix* can be `NULL`.

relativeMode is an integer which indicates whether the levels in the new shared cell instance should be determined in relative or absolute mode. In absolute mode the levels of the components of the shared cell definition are used. In relative mode, the difference between the level of the shared cell instance and *baseLevel* is added to the components of the shared cell definition.

baseLevel is an integer which specifies the base level from which component levels are calculated in relative mode.

assocOrigin should always be `NULL`.

Returns `mdlSharedCell_addToFile` returns the position of the new shared cell instance within the file if successfully added to the file.

See Also `mdlSharedCell_find`.

mdlSharedCell_createDefinitionElement [mdl.lib.mdl]

```

int mdlSharedCell_createDefinitionElement
(
  MSElementUnion *hdrP, /* <= location of buffer */
  char *cellName, /* => name of cell */
  boolean pointCell /* => nonzero for Point cell */
);

```

Description The `mdlSharedCell_createDefinitionElement` function creates a definition element for the cell name *cellname* in the buffer *hdrP*.

hdrP is a pointer to an `MSElementUnion` where the shared cell definition should be stored.

cellName is a character array containing the name of the shared cell.

pointCell is a boolean indicating whether the shared cell definition is for a point cell. If this is `TRUE`, the view independent attribute is set for the element and the snappable attribute is unset.

Returns `mdlSharedCell_createDefinitionElement` returns `SUCCESS` if the shared cell definition is successfully created.

mdlSharedCell_addDefinitionElements

```
#include <mselems.h>

void mdlSharedCell_addDefinitionElements
(
    MSElementDescr    *cellDscrP,    /* <=> cell element descriptor */
    RotMatrix          *rotScale,      /* => rotation and scaling */
    Dpoint3d           *origin         /* => origin of cell */
);
```

Description The `mdlSharedCell_addDefinitionElements` function converts the cell definition element descriptor pointed to by *cellDscrP* to a shared cell definition and adds the definition to the file.

rotScale is the rotation and scaling matrix applied to the elements of *cellDscrP* when they were created. *origin* is the origin used to generate the cell element descriptor.

The `mdlSharedCell_addDefinitionElements` function inverts the rotation and scaling matrix. It then subtracts from the origin to shift the shared cell definition to the origin at zero rotation and a scale factor of 1.0.

If nested cell definitions are in *cellDscrP*, `mdlSharedCell_addDefinitionElements` creates a shared cell definition for each definition.

Returns The `mdlSharedCell_addDefinitionElements` function is of type `void`. It returns no value.

See Also `mdlCell_getElmDscr`.

mdlSharedCell_makeSureDefExists

```
#include <mselems.h>

ULong mdlSharedCell_makeSureDefExists
(
    MSElementUnion    *sharedCell     /* => shared cell element */
);
```

Description The `mdlSharedCell_makeSureDefExists` function checks for a shared cell definition corresponding to the shared cell instance, *sharedCell*. If the shared cell definition is not found, it reads and creates one from the current cell library.

Returns The `mdlSharedCell_makeSureDefExists` function returns `SUCCESS` if the shared cell definition already exists in the file or if one was read from the cell library and added to the file. It returns an error status if no cell with the proper cell name could be located or if the cell with the proper name could not be read.

mdlSharedCell_find

```
#include <mselems.h>

int mdlSharedCell_find
(
    ULong    *foundFilePos, /* <= file position for shared cell definition */
    int      *foundFileNum, /* <= file number where definition was found */
    char     *cellName,     /* => cell to search for */
    int      preferredFile, /* => search here first */
    boolean  allowRefs      /* => allow definition to be in reference file */
);
```

Description The `mdlSharedCell_find` function searches the lists of shared cell definitions for the address of the shared cell definition with *cellName*. When the shared cell definition is located, the function sets *foundFilePos* and *foundFileNum* to the file position and file number where the definition is located.

preferredFile is the number of the file where the function will search first for the cell. File number 0 is the master file. If *allowRefs* is `TRUE` and the shared cell instance cannot be found in the preferred file, `mdlSharedCell_find` searches all other files.

Returns The `mdlSharedCell_find` function returns `SUCCESS` if the definition was located and *foundFilePos* and *foundFile* are valid. It returns `ERROR` if no shared cell definition with the given name can be located.

See Also `mdlSharedCell_makeSureDefExists`, `mdlSharedCell_read`.

mdlSharedCell_dropOneLevel, mdlSharedCell_dropToNormalCell

```
#include <mselems.h>

void mdlSharedCell_dropOneLevel
(
  MSElementUnion  *sharedCell,  /* => shared cell instance to drop */
  ULong           filePos,       /* => file position of instance */
  boolean         displayFlag    /* => if TRUE, display new cell */
);

void mdlSharedCell_dropToNormalCell
(
  MSElementUnion  *sharedCell,  /* => shared cell instance to drop */
  ULong           filePos,       /* => file position of instance */
  boolean         displayFlag,    /* => if TRUE, display new cell */
  boolean         freeze         /* => freeze rather than delete */
);
```

Description The `mdlSharedCell_dropOneLevel` function drops *sharedCell* one nesting level. Any nested shared cell instances or nested complex components in the shared cell definition remain unchanged. *filePos* is the address of the original shared cell instance. If *displayFlag* is `TRUE`, the new elements are drawn as they are added to the file.

The `mdlSharedCell_dropToNormalCell` function drops the sharing status of *sharedCell* and creates a new normal (or unshared) cell. Any nested shared cell instances in the shared cell definition are dropped. *filePos* is the address of the original shared cell instance. If *displayFlag* is `TRUE`, the new elements are drawn as they are added to the file.

If *freeze* is `TRUE`, `mdlSharedCell_dropToNormalCell` marks the original shared cell instance as frozen rather than deleted. It also assigns a new graphic group number to the new elements and saves that number in the frozen element. The original shared cell instance can be retrieved later with the `THAW` command.



The `mdlSharedCell_dropOneLevel` and `mdlSharedCell_dropToNormalCell` functions can cause the file size to grow substantially, because they replace one element with many.

Returns The `mdlSharedCell_dropOneLevel` and `mdlSharedCell_dropToNormalCell` functions are of type `void`. They return no values.

mdlSharedCell_redefine

```
#include <mselems.h>

void mdlSharedCell_redefine
(
  ULong   cellFilePos,    /* => cell's position in cell library */
  char    *cellName       /* => name of cell to redefine */
);
```

Description The `mdlSharedCell_redefine` function re-reads the cell definition from the cell library and creates a new shared cell definition. It then redraws all instances of the cell with the new definition.

cellFilePos is the new cell definition's position in the cell library. *cellName* is the name of the shared cell to be redefined. This argument is required. If *cellFilePos* is 0, `mdlSharedCell_redefine` finds the cell in the library using *cellName*.



Since `mdlSharedCell_redefine` erases and redraws all instances of the shared cell, this function can take awhile to complete.

Returns The `mdlSharedCell_redefine` function returns `SUCCESS` if the cell was processed properly. It returns an error if it could not read the new cell definition.

See Also `mdlSharedCell_drawAllShares`.

mdlSharedCell_deleteDefinition

```
int mdlSharedCell_deleteDefinition
(
  char    *cellName       /* => name of shared cell def to delete */
);
```

Description `mdlSharedCell_deleteDefinition` deletes a shared cell definition from the design file.

cellName determines which shared cell definition is deleted.



It is dangerous to delete a shared cell definition while there are still shared cell instances in the file that reference it. Doing so will cause them to be “lost” and never display again.

Returns `mdlSharedCell_deleteDefinition` returns `SUCCESS` if a shared cell definition is deleted. It returns `ERROR` if no shared cell definition can be found.

See Also `mdlSharedCell_redefine`.

mdlSharedCell_drawAllShares

```
#include <mselems.h>

void mdlSharedCell_drawAllShares
(
    char    *cellName,      /* => name of cell */
    int     drawMode       /* => drawing mode */
);
```

Returns mdlSharedCell_drawAllShares finds all shared cell instances of *cellName* in the master file and draws these instances using the drawing mode specified by *drawMode*.

Returns The mdlSharedCell_drawAllShares function is of type void. It returns no value.

mdlSharedCell_setRange

```
#include <mselems.h>

void mdlSharedCell_setRange
(
    MSElementUnion *sharedCell, /* <=> shared cell instance element */
    int             fileNumber   /* => file number for shared cell */
);
```

Description mdlSharedCell_setRange reads the shared cell definition for the shared cell, *sharedCell*, and sets its range block according to that definition. It needs the file number, *fileNumber*, to determine which file to search first for the definition.

Returns The mdlSharedCell_setRange function is of type void. It returns no value.

mdlSharedCell_toNormalCell

```
int mdlSharedCell_toNormalCell
(
    MSElementDescr **cellDPP, /* <= cell element descriptor */
    MSElementUnion *scInstanceP, /* => shared cell instance element */
    int             fileNum      /* => file number of instance */
);
```

Description mdlSharedCell_toNormalCell returns an element descriptor that is the normal (unshared) cell equivalent of a shared cell instance.

The function returns a pointer to an element descriptor holding the shared cell in **cellDPP*.

scInstanceP is a pointer to the shared cell instance element.

fileNum is the file number used to locate the shared cell definition for *scInstanceP*.

Returns `mdlSharedCell_toNormalCell` returns `SUCCESS` if the shared cell instance is converted to a normal cell or `ERROR` if the shared cell definition can not be found.

See Also `mdlSharedCell_dropToNormalCell`.

16

Patterning Functions

Patterning functions create element descriptors containing elements to pattern specified areas or along linear elements.

The following sections comprise this chapter:

- Basic Patterning Functions
- Associative Patterning Functions

Basic Patterning Functions

The following table describes the basic patterning functions. Associative patterning functions are documented in the next section of this chapter.

Function	Used to
mdlPattern_area	create an area pattern with pattern cell.
mdlPattern_hatch	create an area pattern with hatch lines.
mdlPattern_linear	create a linear pattern with pattern cell.

Example

See patrnmdl.mc.

mdlPattern_area

```
#include <mdl.h>
#include <mselems.h>

int mdlPattern_area
(
MSElementDescr  **patternEdPP, /* <= pattern elements */
MSElementDescr  *solid,        /* => solid element */
MSElementDescr  *holes,        /* => hole elements (optional) */
MSElementDescr  *cell,         /* => pattern descriptor (optional) */
char             *cellName,     /* => pattern cell name (optional) */
double           scale,         /* => pattern scale */
double           angle,         /* => pattern angle in radians */
double           rowSpacing,    /* => row spacing */
double           columnSpacing, /* => column spacing */
int              view,          /* => view number (optional) */

```

```
boolean      searchForHoles, /* => TRUE to search for holes */
Dpoint3d     *originPoint   /* => pattern origin (optional) */
);
```

Description The `mdlPattern_area` function creates an element descriptor. This descriptor contains elements to pattern the area enclosed by the closed element *solid* with the cell specified by *cell* and *cellName*.

patternEdPP points to the pattern element descriptor address. The pattern components can be displayed with the `mdlElmdscr_display` function. They can also be added to the design file with the `mdlElmdscr_add` function. The element descriptor should then be freed with `mdlElmdscr_freeAll`.

solid points to an element descriptor containing the closed element to be patterned. Valid closed elements include shapes, ellipses, complex shapes and closed B-spline curves.

holes points to an element descriptor containing one or more closed hole elements. The holes designate areas within the solid that will not be patterned. A `NULL` value can be passed for holes if no hole elements are required. If *searchForHoles* is nonzero, the master design file is searched for all hole elements on the same level as the solid.

The pattern cell can be specified by an element descriptor pointer *cell* or by the cell name in *cellName*. If neither argument is used, `NULL` should be used. If both arguments are `NULL`, the active pattern cell is used. If the cell is passed as an element descriptor, the cell name should still be passed in *cellName*, because the cell name is stored in the type 5 patterning element and is used for the SHOW PATTERN or MATCH PATTERN tool.

scale represents the scale factor between the cell library definition and the pattern component size.

angle and *view* determine the pattern orientation. *angle* specifies the angle in radians between the view's X-axis and the pattern rows. If *view* is less than zero, the angle is measured from the design file's X-axis.

rowSpacing and *columnSpacing* represent the distance between the pattern rows and columns in the current coordinate system. In most cases, these values are zero.

originPoint determines the pattern position. The pattern rows and columns begin at this point and are repeated in either direction to fill the solid. If `NULL` is passed, the current coordinate system origin is used.

Returns The `mdlPattern_area` function returns zero if the pattern is successfully created. The following values are returned if an error occurs:

Return value	Meaning
<code>MDLERR_CELLNOTFOUND</code>	Pattern cell cannot be found.
<code>MDLERR_NONCLOSEDPATELM</code>	Solid element is not closed.

Return value	Meaning
MDLERR_NONSOLIDPATELM	Solid element has a hole bit set.
MDLERR_NOPATTERNS	No pattern components are within solid.
MDLERR_INVALIDPATSPACE	Both pattern spacing and cell size equal zero.

mdlPattern_hatch

```
#include <mdl.h>
#include <mselems.h>

int mdlPattern_hatch
(
  MSElementDescr  **patternEdPP,      /* <= pattern components */
  MSElementDescr  *solid,             /* => solid to pattern */
  MSElementDescr  *holes,             /* => holes elements (optional) */
  MSElement        *template,         /* => template for pattern */
  double           angle,              /* => hatch angle */
  double           spacing,            /* => hatch spacing */
  int              view,               /* => view number */
  boolean          searchForHoles,     /* => TRUE to search for holes */
  Dpoint3d         *originPoint       /* => hatch origin (optional) */
);
```

Description mdlPattern_hatch creates an element descriptor. This descriptor contains the elements to pattern the area enclosed by the closed element *solid* with the hatch lines.

patternEdPP points to the pattern element descriptor pointer. The pattern components can be displayed with the mdlElmdscr_display function. They can also be added to the design file with the mdlElmdscr_add function. The element descriptor should then be freed with mdlElmdscr_freeAll.

solid points to an element descriptor containing the closed element to be patterned. Valid closed elements include shapes, ellipses, complex shapes and closed B-spline curves.

holes points to an element descriptor containing one or more closed hole elements. The holes designate areas within the solid that should not be patterned. A NULL value can be passed for holes if no hole elements are required. If *searchForHoles* is non-zero, the master design file is searched for all hole elements on the same level as the solid.



You should either specify a pointer to an element descriptor for holes or pass TRUE to the *searchForHoles* argument. If you do both, the results will be unpredictable.

angle and *view* determine the pattern orientation. *angle* specifies the angle in radians between the view's *X*-axis and the hatch lines. If a view number is less than zero, the angle is measured from the design file's *X*-axis.

spacing represents the distance between the hatch lines in the current coordinate system. This value must be greater than zero.

originPoint determines the hatch line position. The hatch lines begin at this point and are repeated in either direction to fill the solid. If *NULL* is passed, the current coordinate system origin is used.

Returns The `mdlPattern_hatch` function returns zero if the pattern is successfully created. The following values are returned if an error occurs:

Return value	Meaning
<code>MDLERR_NONCLOSEDPATELM</code>	Solid element is not closed.
<code>MDLERR_NONSOLIDPATELM</code>	Solid element has a hole bit set.
<code>MDLERR_NOPATTERNS</code>	No pattern components are within solid.
<code>MDLERR_INVALIDPATSPACE</code>	Spacing is not greater than zero.

mdlPattern_linear

```
#include <mdl.h>
#include <mselems.h>

int mdlPattern_linear
(
  MSElementDescr  **patternEdPP,      /* <= Pattern elements */
  MSElementDescr  *elementEdP,        /* => element to pattern */
  ULong            elementPosition,     /* => elm position (optional) */
  MSElementDescr *cell,                /* => pattern cell (optional) */
  char             *cellName,           /* => pattern cellname ("") */
  double           scale,                /* => pattern scale */
  int              patternType          /* => pattern type */
);
```

Description The `mdlPattern_linear` function creates an element descriptor. This descriptor contains the elements to linear pattern the element specified by *elementEdP*.

patternEdPP points to the pattern's element descriptor address. The pattern's components can be displayed with the `mdlElmdscr_display` function. They can also be added to the design file with the `mdlElmdscr_add` function. The element descriptor should then be freed with `mdlElmdscr_freeAll`.

elementEdP points to an element descriptor containing the element to be patterned. Elements that can be linear patterned include lines, line strings, shapes, curves, B-spline curves, complex chains, and complex shapes.

When an element is linear patterned, its class changes to 5 because the element has an associated linear pattern. When patterns are enabled for a view, the pattern components display and the linear patterned elements are inhibited. When patterns are not enabled, the linear patterned elements display and the patterns do not. If the element class specified in *elementEdP* will be changed to linear patterned class, *elementPosition* should point to the element's position in the master design file. If *elementPosition* is set to zero, the element is not changed.

The pattern cell can be specified by an element descriptor pointer *cell* or by the cell name in *cellName*. If either argument is not used, *NULL* should be used. If both arguments are *NULL*, the active pattern cell is used. If the cell is passed as an element descriptor, the cell name should still be passed in *cellName*, because the cell name is stored in the type 5 patterning element and is used for the SHOW PATTERN or MATCH PATTERN tool.

scale represents the scale factor between the cell library definition and the pattern component size.

Returns *patternType* specifies the linear patterning mode. These modes are discussed in detail in the *MicroStation Reference Guide*:

patternType	Description
0	Truncated cycle (unscaled)
1	Untruncated cycle (scaled)
2	Single cycle
3	Multiple cycle

Returns The *mdlPattern_linear* function returns zero if the pattern is successfully created. It returns *MDLERR_CELLNOTFOUND* if the pattern cell cannot be located.

Associative Pattern Functions

The associative and non-associative methods of patterning are similar in many ways. Both involve gathering a group of parameters to define a pattern (spacing, angle, scale cell name, etc.) and then using a closed region defined by design file elements to generate the pattern elements. The difference between the two methods is a matter of *when* the pattern elements are generated.

The elements of a non-associative pattern are generated at *placement* time. That is, when a user executes a pattern command (or an application executes an *mdlPattern_...* function), the pattern elements are generated and added to the design file. Subsequent changes to the element(s) that define the pattern boundary, or changes to the pattern parameters will have no effect unless the original pattern elements are deleted and the pattern commands are manually re-executed.

The elements of an associative pattern are generated at *display* time and are not actually added to the design file (unless the DROP PATTERN command is used). When a user executes an associative pattern command, the command doesn't actually generate any pattern elements. Instead, all the information necessary to define the pattern is gathered together and attached to the boundary element(s) in the form of a user data linkage. In all subsequent display operations MicroStation will recognize the associative pattern linkage on the element and will then generate and display the pattern elements each time the boundary element is displayed. Because the pattern elements are generated at *display* time they will automatically adjust to any changes in the boundary element(s) or changes to the pattern parameters that are stored in the associative pattern linkage.

The design file space used by an associative pattern is only the few bytes required to store the associative pattern linkage and possibly a shared cell definition for the pattern cell. This is usually a tiny fraction of the space required to store the same pattern in a non-associative form.

If the pattern definition involves a complicated cell requiring extensive clipping to adapt to the pattern boundary, the associative pattern may be slower to display than a non-associative pattern. This is not usually the case for simpler patterns, especially for hatching. In these cases, pattern elements can be generated and displayed very quickly without any overhead of reading the pattern elements from the design file. The benefits of automatic updates and extremely low design file overhead make associative patterning the method of choice for many applications.

If an associative pattern operation requires a pattern cell, a shared cell definition for the pattern cell must exist in the design file before the pattern can be displayed. The `mdlPattern_addAssociative` function will automatically add any necessary shared cell definition(s) to the file when the function is executed, if the required cells can be located in an attached cell library. If the cell library will not be available when the function is executed then the application will have to create and add the required shared cell definition to the design file before `mdlPattern_addAssociative` is called.

The following table lists the associative pattern functions:

Function	Used to
<code>mdlPattern_addAssociative</code>	add an associative pattern (or hatch) to a closed element.
<code>mdlPattern_deleteAssociative</code>	delete an associative pattern from a closed element.
<code>mdlPattern_extractAssociative</code>	extract associative pattern information from an element.
<code>mdlPattern_getElementDescr</code>	convert an associative pattern to an IGDS 8.8 compatible primitive elements.

mdlPattern_addAssociative

```
#include <mdl.h>
#include <msmisc.fdf>

int mdlPattern_addAssociative
(
    MSElementDescr    **dscrPP,        /* <=> element to modify */
    int                line1,           /* => first boundary line
                                       (multi-lines only) */
    int                line2,           /* => second boundary line
                                       (multi-lines only) */
    PatternParams      *paramsP,        /* => pattern parameters */
    DPoint3d           *originP,        /* => pattern origin */
    RotMatrix          *rotPt,          /* => element rotation matrix */
    int                option           /* => pattern type */
);
```

Description mdlPattern_addAssociative adds the associative pattern linkage defined by *paramsP* to the pattern boundary element(s) in the element descriptor *dscrPP*. The element(s) in *dscrPP* can be either a closed primitive element (multi-line, ellipse or shape) or a complex element defining the pattern boundary. If *dscrPP* contains a complex element, the associative pattern linkage is appended to the complex header.

The *line1* and *line2* parameters are used only if *dscrPP* contains a MicroStation multi-line element (type=MULTILINE_ELM). In this case, *line1* and *line2* indicate the multi-line profile index of each of the two lines in the multi-line element that define the pattern boundary. Pass -1 for both *line1* and *line2* to use the outermost lines of any multi-line element.

The parameters that define the pattern are specified in the PatternParams structure *paramsP* which is defined in the header file mdl.h. For each parameter specified, the value must be assigned to the appropriate member of the PatternParams structure (*paramsP*) and then the corresponding bit of the modifiers member (*paramsP->modifiers*) must be set to indicate the presence of the specified pattern parameter. The type and number of specified may vary depending on the type of pattern being created. See the MDL include file mdl.h. for the complete definition of the PatternParams structure. If *paramsP* is NULL the active pattern parameters are used.

A PatternParams structure for creation of a simple 45 degree hatch pattern can be defined as follows:

```
..
PatternParams params;

memset(&params, 0, sizeof(params));

params.space1 = 100.0;
params.modifiers |= PATMOD_SPACE1;
```

```
params.angle1 = 45.0 * fc_piover180;
params.modifiers |= PATMOD_ANGLE1;
...
```

The point specified in *originP* is used as the origin or intersection point of the pattern or hatch. If *originP* is `NULL` MicroStation will define the pattern origin based on the origin of the element(s) in *dscrPP*.

The rotation matrix in *pRot* defines the orientation of the coordinate system from which the pattern angle(s) in *paramsP* are measured. Usually this is the view rotation matrix from the view in which the pattern is being placed. The view rotation matrix can be determined using the `mdlRMatrix_fromView` function. If *pRot* is `NULL` the identity matrix is used which indicates the orientation of the design plane.

The *option* parameter specifies the type of pattern to create. If the value of *option* is set to `PATTERN_HATCH`, a simple hatch is created. In this case *paramsP* must contain a value for hatch spacing in *space1*. A hatch angle (in radians) may also be specified in *angle1*.

If the value of *option* is `PATTERN_CROSSHATCH` then a cross hatch pattern is created. In addition to the information specified for `PATTERN_HATCH`, *paramsP* must also contain a value for the secondary hatch spacing in *space2*. A secondary hatch angle can also be specified in *angle2*.

If the value of *option* is `PATTERN_AREA` an area pattern is created using the pattern cell specified in *paramsP*->*cell*, the row spacing specified in *paramsP*->*space1* and the column spacing specified in *paramsP*->*space2*. A pattern angle may also be specified in *paramsP*->*angle1*.

For each of the pattern options, values for pattern scale, and pattern element color, style and weight can also be specified in the `PatternParams` structure pointed to by *paramsP*.



All element types except multi-lines can contain only one associative pattern. If the element in *dscrPP* is not a multi-line and already contains an associative pattern linkage the existing pattern linkage is replaced. If the element in *dscrPP* is a multi-line element the pattern linkage is appended without effecting any previous pattern linkages. To replace an associative pattern on a multi-line element, it is necessary to first call `mdlPattern_deleteAssociative` to remove previously applied patterns.

Returns The `mdlPattern_addAssociative` function returns `SUCCESS` if the requested operation was successful. Otherwise an error status is returned.

See Also `mdlPattern_deleteAssociative`, `mdlPattern_extractAssociative`.

mdlPattern_deleteAssociative

```
#include <mdl.h>
#include <msmisc.fdf>

int mdlPattern_deleteAssociative
(
  MSElementDescr  **dscrPP,      /* <=> element to modify */
  int              index          /* => pattern index (multi-lines only) */
);
```

Description The mdlPattern_deleteAssociative function removes an associative pattern linkage from the element(s) contained in *dscrPP*.

If *dscrPP* contains a multi-line element then there may be more than one associative pattern linkage on the element. In this case an application can use the mdlPattern_extractAssociative function to search for the desired pattern linkage and then specify the pattern to be removed using the *index* parameter when calling mdlPattern_deleteAssociative.

To delete all patterns from a multi-line element call mdlPattern_deleteAssociative in a loop continually deleting the first pattern (*index=0*) until the function returns a non SUCCESS status:

```
...
while (mdlPattern_deleteAssociative(mlineDscrPP, 0)) == SUCCESS)
;
...
```

Returns mdlPattern_deleteAssociative returns SUCCESS if the requested operation completed successfully. Otherwise an error status is returned.

See Also mdlPattern_addAssociative, mdlPattern_extractAssociative.

mdlPattern_extractAssociative

```
#include <mdl.h>
#include <msmisc.fdf>

int mdlPattern_extractAssociative
(
  PatternParams  *paramsP,      /* <= pattern parameters */
  DPoint3d      *originP,      /* <= pattern origin point */
  MSElementUnion *elemP,      /* => patterned element */
  int           fileNo,        /* => element file number */
  int           index          /* => requested pattern index */
);
```

Description The mdlPattern_extractAssociative function extracts associative pattern information from the element pointed to by *elemP*.

If *paramsP* is not NULL the members of the PatternParams structure pointed to by *paramsP* are filled in with the pattern parameters extracted

from the element pointed to by *elemP*. See the MDL include file `mdl.h` for the definition of the `PatternParams` structure. Also, see the MDL documentation on `mdlPattern_addAssociative` for additional information on the contents of the `PatternParams` structure.

If *originP* is not `NULL` the pattern origin point will be returned in *originP*.

fileNo indicates the file number of the element in *elemP*. Pass `MASTERFILE` (0) for the master file or 1 - 255 for reference files. Currently *fileNo* is only used if *elemP* is a multi-line element that contains associative points.

If *elemP* is a multi-line element it may contain more than one associative pattern. In this case pass the index (zero based) of the desired pattern in the *index* parameter. If *elemP* is not a multi-line element pass a value of zero for *index*.

Returns `mdlPattern_extractAssociative` returns `SUCCESS` if the requested operation was successful. Otherwise an error status is returned.

See Also `mdlPattern_addAssociative`, `mdlPattern_deleteAssociative`.

mdlPattern_getElementDescr

```
#include <mdl.h>
#include <msmisc.fdf>

int mdlPattern_getElementDescr
(
  MSElementDescr  **outDscrPP,  /* <= output primitive pattern elems */
  MSElementDescr  *inDscrP,     /* => input associative pattern elem */
  int              fileNo,       /* => file number of input element */
  int              index,        /* => pattern index (multi-lines only) */
  int              option        /* => graphic group options */
);
```

Description The `mdlPattern_getElementDescr` function creates an element descriptor containing primitive elements that represent the associative pattern on the element in *inDscrP*.

outDscrPP is a pointer to the address of an element descriptor that, on successful completion of this function will contain the pattern elements.

inDscrP is a pointer to an element descriptor containing an associative patterned element. If the element in *inDscrP* does not contain an associative pattern, an error status will be returned and *outDscrPP* will be unchanged.

fileNo indicates the file number of the element in *inDscrP*. Pass `MASTERFILE` (0) for the master file or 1 - 255 for reference files. Currently *fileNo* is only used if *inDscrP* is a multi-line element that contains associative points.

If *inDscrP* is a multi-line element it may contain more than one associative pattern. In this case pass the index (zero based) of the desired pattern in the *index* parameter. If *inDscrP* is not a multi-line element pass a value of zero for *index*.

If *option* is `TRUE` all output elements will have their graphic group number set to the next available graphic group number and then the next available graphic group number will be updated (incremented). If *option* is `FALSE`, all output elements will have their graphic group number set to zero.

Returns `mdlPattern_getElementDescr` returns `SUCCESS` if the requested operation was successful. Otherwise an error status is returned.

See Also `mdlPattern_addAssociative`, `mdlPattern_deleteAssociative`, `mdlPattern_extractAssociative`.

Color Management Functions

In general, applications use MicroStation's Color Management functions to specify the colors used to display design file elements in View windows and dialog items in dialog boxes.

This chapter contains the following topics:

- Color table functions
- Color descriptor functions
- Color palette functions
- RGB conversion functions
- Color configuration functions
- Color map functions

Color Codes, Color Tables, Color Maps and Display Colors

Each element's color in a design file is defined by a numeric color code stored in the element's header. For raster data elements (Type 88), a scanline is represented by an array of color codes. Valid color codes range from 0 to 254.

The element color code is used as an index into a **color table**. A color table is an array of 256 RGB values that represent the optimum colors to be used to display design file elements. A maximum of one color table may be defined for a given design file. (See the Color Table Functions section of this chapter for functions that manipulate Color Tables).

MicroStation's Color Manager sets up a mapping between the design file color table, and the actual colors available in the video hardware or native windowing system (XWindows, Windows NT, etc.). This mapping of "logical" design colors to "physical" display colors is called a **color map**. (See the Color Map Functions section of this chapter for functions that manipulate Color Maps).

Although the MicroStation Color Manager ultimately decides which colors are to be actually loaded to the video hardware or native windowing system, applications and users can exercise some control over this color selection process. (See the Color Configuration functions section for more information).

Generating Display Colors

The Color Manager generates the display colors as follows:

1. It defines colors to be used for window borders, menus and dialog boxes. These colors are typically white, black, and several shades of gray.
2. It inserts **balanced colors**, a variety of colors that should satisfy the color-matching needs of most color tables. The number of balanced colors depends on the number of total colors supported by the video graphic hardware. The balanced colors are spread evenly throughout RGB space. Their number will always be a perfect cube (8, 27, 64, 125 or 216). The balanced colors are also required for rendering operations. (As the number of balanced colors increases, the realism of the rendered images also increases).
3. If enough remaining slots exist, it allocates up to four historical colors: pink, greenish-yellow, aquamarine and dark purple. These colors are generated for enhanced compatibility with previous MicroStation releases.
4. For Intergraph Interpro workstations only, it adds pre-defined Environ V menu colors to the list of display colors. This step maintains the screen colors used by Environ V for window borders, while MicroStation is the active application.

Display colors are candidates for the color-table-matching process described above.

Exact Colors

In most cases, the display colors generated by the Color Manager will contain enough unique colors to closely approximate all colors specified in the application's color table. In some cases, however, a close approximation will not suffice. For example, an application might set up 32 or more unique shades of gray in its color table to be used to display raster images. Unfortunately, the matching algorithm employed by the Color Manager would match many of the grays to the same display color, compromising the raster image resolution.

You can avoid this problem by telling the Color Manager to copy a specific number of exact color definitions from the active design file's color table into the internal set of display colors. You can do so by setting the Exact Colors field of the Preferences dialog

box (see the “User menu” chapter of the *MicroStation Reference Guide*) or by calling `mdlColor_reallocateColors`. When the design file is opened, the Color Manager will incorporate the 32 shades of gray in its display colors. However, increasing the number of exact colors can reduce the number of balanced colors and thus reduce the quality of rendered images.

Color Descriptors vs. Color Indexes

In MicroStation version 4.x, a “color” argument to any MDL built-in function was generally an integer value. This integer typically ranged in value from 0 to 255 and was used to index the system's native hardware color palette.

In version 5.0, MicroStation stops treating colors as hardware indices and instead treats them as color objects, commonly known as Color Descriptors. Any new V5 built-in functions that return or accept Color Descriptors will be prototyped as such. That is, they will be prototyped as returning or accepting *BSIColorDescr* *.

Existing V4 built-in functions were also changed internally to return or accept Color Descriptors. However, the existing function prototypes could not be changed to reflect this as it would keep existing applications from compiling successfully. Therefore, there are a few functions that appear to accept or return integer color values as they did in V4, but they actually use Color Descriptors.

These functions, as well as the correctly prototyped V5 functions, are listed below:

Functions that return Color Descriptors:

```
mdlWindow_systemColorGet, mdlWindow_backgroundCDGet,  
mdlColorPal_getColorDescr
```

Functions that return Color Descriptors cast as integers:

```
mdlWindow_fixedColorIndexGet, mdlWindow_colorIndexGet,  
mdlColor_convertRGBtoIndex
```

Functions that have Color Descriptor arguments:

```
mdlWindow_backgroundColorSet, mdlWindow_lineStyleSetCD,  
mdlWindow_textDrawCD, mdlDialog_rectDrawCD, mdlDialog_rectFillCD,  
mdlDialog_textDrawCD, mdlDialog_textDrawNCD, mdlDialog_underlineDrawCD,  
  
mdlDialog_diamondDrawCD, mdlDialog_ellipseDrawCD, mdlDialog_ellipseFillCD,  
mdlDialog_rectDrawBeveledCD, mdlDialog_rectDrawDropShadowCD,
```

mdlDialog_rectDrawEdgeCD, mdlDialog_scrollArrowDrawCD,
mdlDialog_2DArrowDrawCD

Functions that have Color Descriptor arguments cast as integers:

mdlWindow_iconDraw, mdlWindow_lineStyleSet, mdlWindow_rectFill,
mdlWindow_textDraw

Arguments:

mdlWindow_backgroundColorSet *bgColorP*

mdlWindow_lineStyleSetCD *colorP*

mdlWindow_textDrawCD *fgColorP, bgColorP*

mdlDialog_diamondDrawCD *topShadowP, bottomShadowP, fillP*

mdlDialog_ellipseDrawCD *colorP*

mdlDialog_ellipseFillCD *colorP*

mdlDialog_rectDrawCD *colorP*

mdlDialog_rectDrawBeveledCD *topShadowP, bottomShadowP*

mdlDialog_rectDrawDropShadowCD *frameColorP, bottomShadowP*

mdlDialog_rectDrawEdgeCD *topShadowP, bottomShadowP*

mdlDialog_rectFillCD *fillP*

mdlDialog_scrollArrowDrawCD *topShadowP, bottomShadowP, fillP*

mdlDialog_2DArrowDrawCD *fillP, dimColorP, topShadowP*

mdlDialog_textDrawCD *foregroundP, backgroundP, dimColorP*

mdlDialog_textDrawNCD *foregroundP, backgroundP, dimColorP*

mdlDialog_underlineDrawCD *foregroundP, dimColorP*

Arguments:

mdlWindow_iconDraw *onColor, offColor*

```
mdlWindow_lineStyleSet color

mdlWindow_rectFill color

mdlWindow_textDraw fgColor, bgColor
```

Color Table Functions

The following table lists color table functions:

Function	Used to
mdlColor_attachColorTable	attach a color table to the active design file.
mdlColor_getName	obtain the name of the current color table.
mdlColor_getColorTable	get a copy of the active design file's color table.
mdlColor_getColorTableByFileNumber	get the color table associated with a particular reference file.
mdlColor_getDefaultColorTable	get a copy of the MicroStation default color table.
mdlColor_searchRGBArray	scan an array of RGB's (color table) for the closest match to a given RGB value.

mdlColor_attachColorTable

```
#include <mdl.h>

int mdlColor_attachColorTable
(
    char    *ctName,  /* => brief string describing ctbl */
    char    *ctbl     /* => Pointer to colortable to attach */
);
```

Description The mdlColor_attachColorTable function lets an application attach a new color table to a design file. Each color definition in *ctbl* is mapped to the closest matching color in the Color Manager's display colors. The color code associated with each design file element will then correspond to a color in the newly attached color table. (For a more detailed discussion of color tables and their relationships to design file elements, see the beginning of this section).

When the Exact Colors User Preferences variable is greater than zero, that number of color definitions is copied from the beginning of the caller's

color table to the Color Manager's display colors. Thus, exact color matches can be obtained.

ctName is the color table's name as it will appear in the MicroStation Command Window after the attachment is complete. This is also the name that will appear when the CT=? command is entered to display the currently attached color table. *ctName* does not necessarily correspond to a file name.

ctbl points to 256 RGB color definitions containing a total of 768 bytes. The first entry in the table defines the background color to be used in the design file. Color definitions 2 through 256 represent color indices 0 through 254 as used by design file element color codes.

Returns The mdlColor_attachColorTable function returns either SUCCESS or a non-zero value if the color table attachment fails.

See Also mdlColor_getColorTable, mdlColor_getName.

mdlColor_getName

```
int mdlColor_getName
(
    char    *ctName    /* <= 64-byte array to contain name */
);
```

Description The mdlColor_getName function returns the name of the active color table in use with the design file.

ctName is the address of a 64-byte array to receive the name of the color table.

Returns mdlColor_getName always returns SUCCESS.

See Also mdlColor_attachColorTable, mdlColor_getColorTable.

mdlColor_getColorTable

```
#include <mdl.h>

void mdlColor_getColorTable
(
    char    *ctbl      /* <= Where to store the colortable */
);
```

Description The mdlColor_getColorTable function obtains a copy of the color table that is currently attached to the active design file. If no color table is attached to the design file, the MicroStation default color table is copied.

ctbl points to a memory area that will receive a copy of the design file's currently attached color table. This memory area must be large enough to hold 256 RGB color definitions containing a total of 768 bytes.

Returns The mdlColor_getColorTable function returns no value.

See Also mdlColor_attachColorTable, mdlColor_getName.

mdlColor_getColorTableByFileNumber

```
#include <mscolor.fdf>

int mdlColor_getColorTableByFileNumber
(
    byte    *colortable,    /* <= Store it here */
    int     fileNum        /* => 0=master dgn file, 1-255 = ref files */
);
```

Description The mdlColor_getColorTableByFileNumber function is used to retrieve a color table from the master design file or one of it's attached reference files.

colortable points to an array of 768 bytes in which the colortable RGB values are stored.

fileNum is set to zero for the master design file, or 1-255 for an attached reference file.

Returns mdlColor_getColorTableByFileNumber returns SUCCESS.

See Also mdlColor_getColorTable.

mdlColor_getDefaultColorTable

```
#include <mdl.h>

void mdlColor_getDefaultColorTable
(
    char    *ctbl          /* <= Where to store the colortable */
);
```

Description The mdlColor_getDefaultColorTable function obtains a copy of the default color table used by MicroStation when no color table is attached to the active design file.

ctbl points to a memory area that will receive a copy of the default color table. This memory area must be large enough to hold 256 RGB color definitions containing a total of 768 bytes.

Returns The mdlColor_getDefaultColorTable function returns no value.

See Also mdlColor_getColorTable.

mdlColor_searchRGBArray

```
#include <mscolor.h>
#include <mscolor.fdf>

int mdlColor_searchRGBArray          /* <= Closest RGB's array index */
```

```
(
  RGBColorDef *rgbColorP,    /* => find cmap index for this rgb */
  void        *rgbTableP,    /* => table of RGB's to search */
  int         numColors      /* => num rgbs in table */
);
```

Description The `mdlColor_searchRGBArray` function is used to find the closest matching RGB to a target RGB from an array of RGB's. All RGB color comparisons are done by first converting the two RGB values into HSV (Hue, Saturation, Value) values, and comparing those converted values. The hue component of the HSV is given the highest priority. Comparing HSV values results in more desirable colors in terms of what the human eye expects to see.

Typically, an application would use this function in conjunction with `mdlColor_getColorTable` to determine the best element color number to use for an arbitrary RGB value. When using `mdlColor_searchRGBArray` in this fashion, remember that RGB offset zero in a color table corresponds to the view background color and that RGB index one corresponds to element color number zero.

rgbColorP points to the target RGB to be matched against the array of RGBs.

rgbTableP points to the RGB values that are to be searched for the closest match to the RGB value pointed to by *rgbColorP*.

numColors indicates the number of RGB triads in the array *rgbTableP*.

Returns `mdlColor_searchRGBArray` returns the zero-based index of the RGB triad that most closely matches the target RGB value.

See Also `mdlColor_getColorTable`.

Color Descriptor Functions

Color descriptors allow MDL applications to specify color for MicroStation drawing routines in a hardware independent way. They also allow an application to specify colors in a variety of forms. An application will typically use the Color Palette and Color Descriptor functions in the following order:

1. `mdlColorPal_create` is used to allocate a block of color descriptors.
2. `mdlColorPal_setEntries` is used to initialize the palette.
3. `mdlColorPal_getColorDescr` is used to obtain a pointer to an individual color descriptor from a palette.
4. (optional) One of the `mdlColorDescr_set...` or `mdlColorDescr_get...` functions are used to manipulate a color descriptor.

5. The color descriptor is passed as an argument to one of the mdlDialog_... or mdlWindow_... drawing routines.
6. mdlColorPal_destroy is used to deallocate a palette that is no longer needed.

The following table lists Color Descriptor functions:

Function	Used to
mdlColorDescr_getColorId	get color ID number of a color descriptor.
mdlColorDescr_setByColorId	set color descriptor using color ID.
mdlColorDescr_getBestBWContrast	get the best black or white contrast color for a given background color.
mdlColorDescr_getColorName	get the color name of a color descriptor.
mdlColorDescr_setByColorName	set color descriptor from a color name.
mdlColorDescr_getDrawIndex	get hardware draw value from color descriptor.
mdlColorDescr_setDrawIndex	override draw value of a color descriptor.
mdlColorDescr_getElemColorNumber	get an element color number from a color descriptor.
mdlColorDescr_setByElemColorNumber	set a color descriptor from an element color number.
mdlColorDescr_getHsv	get a Hue-Saturation-Value triad from a color descriptor.
mdlColorDescr_setByHsv	set a color descriptor from a Hue-Saturation-Value triad.
mdlColorDescr_getMenuColor	get menu color ID from color descriptor.
mdlColorDescr_setByMenuColor	set color descriptor from a menu color ID.
mdlColorDescr_getRgb	get an RGB value from a color descriptor.
mdlColorDescr_setByRgb	set a color descriptor from an RGB value.

mdlColorDescr_getColorId

```
#include <mscolor.h>
#include <mscolor.fdf>

int mdlColorDescr_getColorId
(
    int          *colorIdP,      /* <= store color id here */
    BSIColorDescr *cdP          /* => get from this color descr */
);
```

Description The `mdlColorDescr_getColorId` function is used to query a color descriptor for its X Color ID value. Valid X Color ID values are defined in `colname.h`.

colorIdP points to an integer that is to receive the color ID.

cdP is a color descriptor pointer usually obtained from a MicroStation color palette using the `mdlColorPal_getColorDescr` function.

Returns `mdlColorDescr_getColorId` returns `SUCCESS` or `MDLERR_BADCOLORDESCR` if the color descriptor pointer *cdP* is invalid.

See Also `mdlColorDescr_getColorName`, `mdlColorDescr_setByColorId`, `mdlColorPal_getColorDescr`.

mdlColorDescr_setByColorId

```
#include <colname.h>
#include <mscolor.fdf>
#include <mscolor.h>

int mdlColorDescr_setByColorId/* <= SUCCESS or error. */
(
    BSIColorDescr    *cdP,          /* <= color descr to get new value */
    int               colorID,      /* => Color Id. */
    int               matchToDgnCtbl /* => match to dgn file colortable? */
);
```

Description The `mdlColorDescr_setByColorId` function is used to set the RGB value of the given color descriptor according to the RGB associated with *colorID*.

cdP points to the color descriptor to receive the new setting.

colorID is one of the color ID macros defined in `colname.h` and indicates (indirectly) the RGB value to set in *cdP*.

matchToDgnCtbl controls how the draw value for *cdP* is calculated. If *matchToDgnCtbl* is `TRUE`, the draw value will correspond to the closest RGB in the attached colortable. If *matchToDgnCtbl* is `FALSE`, the draw value will correspond to the closest RGB in the video hardware. This parameter is usually set to `FALSE`.

Returns `mdlColorDescr_setByColorId` returns `SUCCESS` if the color descriptor was set. `MDLERR_BADCOLORDESCR` is returned if *cdP* is an invalid pointer. `MDLERR_RSCSTRINGNOTFOUND` is returned if the stringlist resource containing the colorids could not be loaded or if *colorID* did not match any IDs in MicroStation's list of valid color IDs.

See Also `mdlColorDescr_getColorId`, `mdlColorDescr_setByColorName`.

mdlColorDescr_getBestBWContrast

```
#include <mcolor.fdf>
#include <mcolor.h>

BSIColorDescr *mdlColorDescr_getBestBWContrast /* <= B or W cdP */
(
    int          screen, /* => Screen number being drawn to */
    BSIColorDescr *cdP    /* => Background color descr */
);
```

Description The *mdlColorDescr_getBestBWContrast* function returns a pointer to either a black or white color descriptor, whichever provides the best contrast to *cdP* as a foreground color.

screen indicates the screen being drawn to.

cdP is a color descriptor pointer for the background color.

Returns *mdlColorDescr_getBestBWContrast* returns a pointer to a color descriptor.

mdlColorDescr_getColorName

```
#include <mcolor.h>
#include <mcolor.fdf>

int mdlColorDescr_getColorName
(
    char          *nameP, /* <= store color name here */
    int           sizeofName, /* => max size of nameP */
    BSIColorDescr *cdP      /* => get from this color descr */
);
```

Description The *mdlColorDescr_getColorName* function is used to query a color descriptor for its X color name.

nameP is a buffer to receive the color name.

sizeofName indicates the maximum size of *nameP*.

cdP is a color descriptor pointer usually obtained from a MicroStation color palette using the *mdlColorPal_getColorDescr* function.

Returns *mdlColorDescr_getColorName* returns *SUCCESS* or *MDLERR_BADCOLORDESCR* if the color descriptor pointer *cdP* is invalid.

See Also *mdlColorDescr_getColorId*, *mdlColorDescr_setByColorName*,
mdlColorPal_getColorDescr.

mdlColorDescr_setByColorName

```
#include <mcolor.fdf>
#include <mcolor.h>

int mdlColorDescr_setByColorName /* <= SUCCESS or error */
```

```
(
BSIColorDescr      *cdP,          /* <= color descr to get new value */
char               *colorName,    /* => XColor name */
int                matchToDgnCtbl /* => match to dgn file colortable? */
);
```

Description The `mdlColorDescr_setByColorName` function is used to set the RGB value of the given color descriptor according to the RGB associated with *colorName*.

cdP and *matchToDgnCtbl* are identical to those used in the `mdlColorDescr_setByColorId` function.

colorName is an ASCII color name which is contained in one of the standard MicroStation color name stringlists. The string is located in the stringlists and its associated RGB value is saved. This RGB value will be used to set the color descriptor.

Returns `mdlColorDescr_setByColorName` returns `SUCCESS` if the color name was valid and the color descriptor could be set. `MDLERR_BADCOLORDESCR` is returned if *cdP* is an invalid pointer. `MDLERR_RSCSTRINGNOTFOUND` is returned if the stringlist resource containing the color names could not be loaded or if *colorName* did not match any names in MicroStation's list of valid color names.

See Also `mdlColorDescr_setByColorId`.

mdlColorDescr_getDrawIndex

```
#include <mcolor.fdf>
#include <mcolor.h>

int mdlColorDescr_getDrawIndex /* <= SUCCESS or error. */
(
long               *drawValueP,   /* <= Draw value */
BSIColorDescr      *cdP,          /* => color descr to query */
int                screen         /* => screen to get draw value for */
);
```

Description `mdlColorDescr_getDrawIndex` is used to acquire a draw value from a color descriptor for a given screen. This function rarely needs to be called from MDL applications since drawing routines accept pointers to color descriptors directly. MicroStation uses this function internally to obtain the draw values contained in color descriptors.

drawValueP is the address of a memory location to receive the draw value. Depending on the hardware/operating system platform, the draw value will be a video hardware pixel index or the native operating system's equivalent type.

cdP is the color descriptor from which *drawValueP* will be obtained.

screen indicates the screen for which *drawValueP* will be obtained.

Returns mdlColorDescr_getDrawIndex returns SUCCESS if *drawValueP* can be obtained. MDLERR_BADSCREENNUMBER is returned if the *screen* parameter was invalid. MDLERR_BADCOLORDESCR is returned if *cdP* is an invalid pointer.

See Also mdlColorDescr_setDrawIndex.

mdlColorDescr_setDrawIndex

```
#include <mscolor.fdf>
#include <mscolor.h>

int mdlColorDescr_setDrawIndex /* <= SUCCESS or error. */
(
    BSIColorDescr    *cdP,          /* <= color descr to get new draw index */
    long             drawIndex,      /* => Draw Index corr. to screen */
    int              screen          /* => screen to set drawIndex for */
);
```

Description The mdlColorDescr_setDrawIndex function is used to force a color descriptor to always yield a specific draw value for a given screen. (A color descriptor “yields” a draw value when queried by the function mdlColorDescr_getDrawIndex). It is up to the caller to maintain this draw value as the MicroStation color environment changes. This function is not recommended for general use. Instead, use one of the mdlColorDescr_setBy... routines to set color descriptors since this will allow MicroStation to maintain the draw values automatically.

cdP points to a color descriptor to receive the setting.

drawIndex specifies the draw value to store in the color descriptor.

screen indicates for which screen the color descriptor will yield *drawIndex* when queried by mdlColorDescr_getDrawIndex.

Returns mdlColorDescr_setDrawIndex returns SUCCESS if the color descriptor could be set. MDLERR_BADCOLORDESCR is returned if *cdP* is an invalid pointer.

See Also mdlColorDescr_getDrawIndex.

mdlColorDescr_getElemColorNumber

```
#include <mscolor.h>
#include <mscolor.fdf>

int mdlColorDescr_getElemColorNumber /* <= SUCCESS or error */
(
    int              *elemColorNumberP, /* <= elem color number */
    BSIColorDescr    *cdP               /* => get from this color descr */
);
```

Description The `mdlColorDescr_getElemColorNumber` function is used to query a color descriptor for its associated element color number.

elemColorNumberP is a pointer to an integer to receive the element color number.

cdP is a color descriptor pointer usually obtained from a MicroStation color palette using the `mdlColorPal_getColorDescr` function.

Returns `mdlColorDescr_getElemColorNumber` returns `SUCCESS` or `MDLERR_BADCOLORDESCR` if the color descriptor pointer *cdP* is invalid.

See Also `mdlColorDescr_setByElemColorNumber`, `mdlColorPal_getColorDescr`.

mdlColorDescr_setByElemColorNumber

```
#include <mcolor.fdf>
#include <mcolor.h>

int mdlColorDescr_setByElemColorNumber /* <= SUCCESS or error. */
(
    BSIColorDescr *cdP,          /* <= clr descr to get new value */
    int elemColorNumber /* => elem color number */
);
```

Description The `mdlColorDescr_setByElemColorNumber` function is used to set the RGB value of the given color descriptor with the design file colortable RGB indexed by *elemColorNumber*.

cdP points to the color descriptor to receive the new setting.

elemColorNumber is an index into the currently attached master design file colortable, or the MicroStation default colortable if no colortable is attached.

Returns `mdlColorDescr_setByElemColorNumber` returns `SUCCESS` if the color descriptor could be set. `MDLERR_BADCOLORDESCR` is returned if *cdP* is an invalid pointer.

See Also `mdlColorDescr_getElemColorNumber`.

mdlColorDescr_getHsv

```
#include <mcolor.h>
#include <mcolor.fdf>

int mdlColorDescr_getHsv /* <= SUCCESS or error */
(
    HSVColorDef *hsvP, /* <= HSV value */
    BSIColorDescr *cdP /* => get from this color descr */
);
```

Description `mdlColorDescr_getHsv` is used to query a color descriptor for its HSV value.

hsvP points to an HSV variable to receive the HSV data.

cdP is a color descriptor pointer usually obtained from a MicroStation color palette using the `mdlColorPal_getColorDescr` function.

Returns `mdlColorDescr_getHsv` returns `SUCCESS` or `MDLERR_BADCOLORDESCR` if the color descriptor pointer *cdP* is invalid.

See Also `mdlColorDescr_getRgb`, `mdlColorDescr_setByHsv`, `mdlColorPal_getColorDescr`.

mdlColorDescr_setByHsv

```
#include <mcolor.fdf>
#include <mcolor.h>

int mdlColorDescr_setByHsv /* <= SUCCESS or error. */
(
    BSIColorDescr    *cdP,          /* <= color descr to get new value */
    HSVColorDef      *hsvP,        /* => HSV value */
    int              matchToDgnCtbl /* => match to dgn file colortable? */
);
```

Description `mdlColorDescr_setByHsv` is used to set a color descriptor from an HSV color structure. The HSV data is stored internally to the color descriptor as RGB data.

cdP points to a color descriptor to receive the setting.

hsvP points to the HSV data. This data will be converted to RGB and used to set the color descriptor.

matchToDgnCtbl controls how the draw value for *cdP* is calculated. If *matchToDgnCtbl* is `TRUE`, the draw value will correspond to the closest RGB in the attached colortable. If *matchToDgnCtbl* is `FALSE`, the draw value will correspond to the closest RGB in the video hardware. This parameter is usually set to `FALSE`.

Returns `mdlColorDescr_setByHsv` returns `SUCCESS` if the color descriptor could be set. `MDLERR_BADCOLORDESCR` is returned if *cdP* is an invalid pointer. `MDLERR_BADARG` is returned if *hsvP* is `NULL`.

See Also `mdlColorDescr_getHsv`, `mdlColorDescr_setByRgb`.

mdlColorDescr_getMenuColor

```
#include <mcolor.h>
#include <mcolor.fdf>

int mdlColorDescr_getMenuColor /* <= SUCCESS or error */
(
    int              *menuColorIdP, /* <= store menu color id here */
    BSIColorDescr    *cdP          /* => get from this color descr */
);
```

Description The `mdlColorDescr_getMenuColor` function is used to query a color descriptor for its associated menu color, also known as a “fixed color.”

menuColorIdP points to an integer that is to receive the color ID.

cdP is a color descriptor pointer usually obtained from a MicroStation color palette using the `mdlColorPal_getColorDescr` function.

Returns `mdlColorDescr_getMenuColor` returns `SUCCESS` or `MDLERR_BADCOLORDESCR` if the color descriptor pointer *cdP* is invalid.

See Also `mdlColorDescr_setByMenuColor`, `mdlColorPal_getColorDescr`.

mdlColorDescr_setByMenuColor

```
#include <mcolor.fdf>
#include <mcolor.h>
#include <msdefs.h> /* For definitions of menu colors */

int mdlColorDescr_setByMenuColor /* <= SUCCESS or error */
(
    BSIColorDescr *cdP,          /* <= color descr to get new value */
    int menuColorId             /* => menu color id */
);
```

Description The `mdlColorDescr_setByMenuColor` function is used to set a color descriptor according to the MicroStation defined “menu colors.” The menu colors are: `BLACK_INDEX`, `BLUE_INDEX`, `GREEN_INDEX`, `CYAN_INDEX`, `RED_INDEX`, `MAGENTA_INDEX`, `YELLOW_INDEX`, `WHITE_INDEX`, `LGREY_INDEX`, `DGREY_INDEX`, `MGREY_INDEX` and `PSEUDOWHITE_INDEX`.

cdP points to a color descriptor to receive the setting.

menuColorId is one of the menu color IDs (shown above) defined in `msdefs.h`.

Returns `mdlColorDescr_setByMenuColor` returns `SUCCESS` if the color descriptor could be set. `MDLERR_BADCOLORDESCR` is returned if *cdP* is an invalid pointer.

See Also `mdlColorDescr_getMenuColor`.

mdlColorDescr_getRgb

```
#include <mcolor.h>
#include <mcolor.fdf>

int mdlColorDescr_getRgb /* <= SUCCESS or error */
(
    RGBColorDef *rgbP,          /* <= RGB value */
    BSIColorDescr *cdP         /* => get from this color descr */
);
```

Description `mdlColorDescr_getRgb` is used to query a color descriptor for its RGB value.

rgbP points to an RGB variable to receive the RGB data.

cdP is a color descriptor pointer usually obtained from a MicroStation color palette using the `mdlColorPal_getColorDescr` function.

Returns `mdlColorDescr_getRgb` returns `SUCCESS` or `MDLERR_BADCOLORDESCR` if the color descriptor pointer *cdP* is invalid.

See Also `mdlColorDescr_getHsv`, `mdlColorDescr_setByRgb`, `mdlColorPal_getColorDescr`.

mdlColorDescr_setByRgb

```
#include <mcolor.fdf>
#include <mcolor.h>

int mdlColorDescr_setByRgb /* <= SUCCESS or error */
(
    BSIColorDescr    *cdP,          /* <= color descr to get new value */
    RGBColorDef    *rgbP,          /* => RGB value */
    int             matchToDgnCtbl /* => match to dgn file colortable? */
);
```

Description `mdlColorDescr_setByRgb` is used to set a color descriptor from the given RGB triad.

cdP points to a color descriptor to receive the setting.

rgbP points to the RGB data that is used to set the color descriptor.

matchToDgnCtbl controls how the draw value for *cdP* is calculated. If *matchToDgnCtbl* is `TRUE`, the draw value will correspond to the closest RGB in the attached colortable. If *matchToDgnCtbl* is `FALSE`, the draw value will correspond to the closest RGB in the video hardware. This parameter is usually set to `FALSE`.

Returns `mdlColorDescr_setByRgb` returns `SUCCESS` if the color descriptor could be set. `MDLERR_BADCOLORDESCR` is returned if *cdP* is an invalid pointer. `MDLERR_BADARG` is returned if *rgbP* is `NULL`.

See Also `mdlColorDescr_getRgb`, `mdlColorDescr_setByHsv`.

Color Palette Functions

Color palettes are essentially arrays of color descriptors. They allow an application to allocate and organize color descriptors as a group.

An application will typically use the Color Palette and Color Descriptor functions in the following order:

1. `mdlColorPal_create` is used to allocate a block of color descriptors.
2. `mdlColorPal_setEntries` is used to initialize the palette.
3. `mdlColorPal_getColorDescr` is used to obtain a pointer to an individual color descriptor from a palette.
4. (optional) One of the `mdlColorDescr_set...` or `mdlColorDescr_get...` functions are used to manipulate a color descriptor.
5. The color descriptor is passed as an argument to one of the `mdlDialog...` or `mdlWindow...` drawing routines.
6. `mdlColorPal_destroy` is used to deallocate a palette that is no longer needed.

The following table lists Color Palette functions:

Function	Used to
<code>mdlColorPal_create</code>	create a color palette.
<code>mdlColorPal_destroy</code>	destroy a color palette.
<code>mdlColorPal_getColorDescr</code>	get a pointer to a color descriptor from a color palette and index.
<code>mdlColorPal_modifyStatus</code>	modify the status of a color palette.
<code>mdlColorPal_getDrawValue</code>	get draw value of a color palette entry.
<code>mdlColorPal_setEntries</code>	set the entries in a color palette.
<code>mdlColorPal_setPaletteFromColortable</code>	set a color palette from a colortable.
<code>mdlColorPal_getNumColors</code>	get the number of entries in color palette.

mdlColorPal_create

```
#include <mcolor.fdf>
#include <mcolor.h>

int mdlColorPal_create
(
    BSIColorPalette  **palettePP,    /* <= address of palette ptr */
    int              numEntries      /* => desired palette size */
);
```

Description The `mdlColorPal_create` function is used to create a MicroStation color palette. A MicroStation color palette is a collection of color descriptors. Each color descriptor, in turn, allows a programmer to define a color for drawing to a window without regard to the color capabilities of the video hardware present. For example, a color palette of 300 color descriptors may be set up even if the hardware capabilities of

the current system can only display 16 colors. This helps to provide for hardware independent applications.

palettePP is the address of a color palette pointer. The pointer will be set to the newly allocated palette. When an application terminates, the palette and all associated descriptors are automatically freed. After allocating a color palette, `mdlColorPal_setEntries` must be called to initialize the color descriptors therein.

The *numEntries* parameter specifies the number of color descriptors in the palette.

Returns `mdlColorPal_create` returns `SUCCESS` if the palette could be created. `MDLERR_BADARG` is returned if *numEntries* is zero or *palettePP* is `NULL`. `MDLERR_INSMEMORY` is returned if there was not enough memory to allocate a palette of *numEntries* in size.

See Also `mdlColorPal_destroy`, `mdlColorPal_setEntries`, `mdlColorPal_getColorDescr`.

mdlColorPal_destroy

```
#include <mscolor.fdf>
#include <mscolor.h>

int mdlColorPal_destroy
(
    BSIColorPalette  **palettePP    /* <=> Palette to be destroyed */
);
```

Description The `mdlColorPal_destroy` function is used to free a color palette and its constituent color descriptors.

palettePP is the address of a color palette pointer which will be freed and afterwards set to `NULL`.

Returns `mdlColorPal_destroy` returns `SUCCESS` after the palette is successfully destroyed. `MDLERR_BADCOLORPALETTE` is returned if *palettePP* is not the address of a valid palette pointer.

See Also `mdlColorPal_create`.

mdlColorPal_getColorDescr

```
#include <mscolor.fdf>
#include <mscolor.h>

BSIColorDescr *mdlColorPal_getColorDescr
(
    BSIColorPalette  *paletteP,      /* => palette */
    int              index            /* => get CD at this pal entry */
);
```

Description The `mdlColorPal_getColorDescr` function is used to obtain a color descriptor pointer from the palette entry indicated by *index*.

paletteP is a pointer to a palette obtained from `mdlColorPal_create`. If *paletteP* is `NULL`, the color palette associated with the currently attached colortable is used.

index indicates the entry in the palette for which a color descriptor pointer is desired.

Returns `mdlColorPal_getColorDescr` returns a pointer to a color descriptor or `NULL` if *index* is larger than the size of the color palette.

See Also `mdlColorPal_create`, `mdlColorPal_setEntries`.

mdlColorPal_modifyStatus

```
#include <mcolor.fdf>
#include <mcolor.h>

int mdlColorPal_modifyStatus
(
    BSIColorPalette *paletteP,      /* => palette to be modified */
    ULONG          statusMask,      /* => bit mask */
    int            setFlag          /* => set or reset? */
);
```

Description `mdlColorPal_modifyStatus` is used to change the status of a color palette.

paletteP points to the palette whose status is to be changed.

statusMask indicates the status bits to be affected. Currently the only valid status bit mask is `PALSTATUS_SYNCHABLE`. This is used by the MicroStation Color Manager to determine if a color palette should be synchronized to changes in the MicroStation color environment. By default, this status bit is set to the ON position.

setFlag indicates whether the status bits should be turned on (1) or off (0).

Returns `mdlColorPal_modifyStatus` always returns `SUCCESS`.

mdlColorPal_getDrawValue

```
#include <mcolor.fdf>
#include <mcolor.h>

int mdlColorPal_getDrawValue /* <= SUCCESS or error. */
(
    long          *drawValueP,      /* <= draw value corr. to screen */
    BSIColorPalette *paletteP,      /* => qry this palette */
    int           index,            /* => qry this palette entry */
    int           screen            /* => screen to get pixel index for */
);
```

Description mdlColorPal_getDrawValue is used to acquire a draw value from a color palette entry for a given screen. This function rarely needs to be called from MDL applications since drawing routines accept pointers to color descriptors directly. MicroStation uses this function internally to obtain the draw values contained in color palette entries. This function is equivalent to calling mdlColorPal_getColorDescr followed by mdlColorDescr_getDrawIndex.

drawValueP is the address of a memory location to receive the draw value. Depending on the hardware/operating system platform, the draw value will be a video hardware pixel index or the native operating system's equivalent type.

paletteP is a pointer to a palette obtained from mdlColorPal_create. If *paletteP* is NULL, the color palette associated with the currently attached colortable is used.

index indicates the entry in the palette for which a draw value is desired.

screen indicates for which drawing surface a draw value is desired.

Returns mdlColorPal_getDrawValue returns SUCCESS if *drawValueP* can be obtained. MDLERR_BADSCREENNUMBER is returned if the *screen* parameter was invalid. MDLERR_BADCOLORPALETTE is returned if *paletteP* is an invalid pointer.

See Also mdlColorDescr_getDrawIndex.

mdlColorPal_setEntries

```
#include <mscolor.fdf>
#include <mscolor.h>

int mdlColorPal_setEntries
(
    BSIColorPalette *paletteP,      /* => palette w/color descrs. */
    BSIPaletteEntry *srcData,       /* => source data */
    int startIndex,                 /* => where to begin setting */
    int numEntries                  /* => num entries to set */
);
```

Description mdlColorPal_setEntries is used to initialize the array of color descriptors contained in the color palette pointed to by *paletteP*. This is done in two steps. First, an array of color descriptor setting data is first initialized by the caller in the array pointed to by *srcData*. Then mdlColorPal_setEntries is called with a starting index and a count of entries to be initialized from the setting data. Once a palette has been initialized this way, individual color descriptors in the palette may be reset at a later time by using mdlColorPal_setEntries again, or by calling mdlColorPal_getColorDescr and then setting the color descriptor using one of the mdlColorDescr_setBy... functions.

paletteP points to the color palette to be initialized. A color palette pointer is obtained from a call to mdlColorPal_create.

srcData points to an array of `BSIPaletteEntry` structures used to initialize the specified entries in the color palette. The *setMethod* member of each `BSIPaletteEntry` entry must be set to one of the following values: `COLORD_RGB`, `COLORD_HSV`, `COLORD_XCOLORID`, `COLORD_ELEM_COLOR_NUMBER` or `COLORD_MENUCOLORID`. Depending on the set method used, additional information must also be provided:

Set Method	Additional Data Required
<code>COLORD_RGB</code>	<i>setDataP</i> -> <i>setData.rgb</i>
<code>COLORD_HSV</code>	<i>setDataP</i> -> <i>setData.hsv</i>
<code>COLORD_XCOLORID</code>	<i>setDataP</i> -> <i>setData.xColorId</i>
<code>COLORD_ELEM_COLOR_NUMBER</code>	<i>setDataP</i> -> <i>setData.elemColorNumber</i>
<code>COLORD_MENUCOLORID</code>	<i>setDataP</i> -> <i>setData.menuColorId</i>

startIndex indicates the first entry in the palette to be initialized. The beginning of the palette is signified by a *startIndex* value of zero.

numEntries indicates the number of sequential entries in the palette to initialize.

Returns `mdlColorPal_setEntries` returns `SUCCESS` or one of the following error codes: `MDLERR_BADCOLORPALETTE` if the color palette pointer is invalid; `MDLERR_BADCOLORPALETTEINDEX` if *startIndex* + *numEntries* is beyond the range of the color palette; `MDLERR_BADSETMETHOD` if an invalid set method was used.

See Also `mdlColorPal_create`, `mdlColorPal_setPaletteFromColortable`.

mdlColorPal_setPaletteFromColortable

```
#include <mcolor.fdf>
#include <mcolor.h>

int mdlColorPal_setPaletteFromColortable
(
    BSIColorPalette *paletteP,      /* => to receive RGB definitions */
    byte            *ctb1P          /* => ctbl with RGB definitions */
);
```

Description `mdlColorPal_setPaletteFromColortable` is used to set a color palette of 256 entries from a 256 color colortable. Internally, this function will call `mdlColorPal_setEntries`, automatically specifying a *setMethod* of `COLORD_RGB` for each of the colors in the colortable.

The *paletteP* parameter points to the color palette to be initialized.

The *ctb1P* parameter points to the RGB values to use to set the color palette.

Returns mdlColorPal_setPaletteFromColortable returns SUCCESS or one of the following error codes: MDLERR_BADCOLORPALETTE if the color palette pointer is invalid; MDLERR_BADCOLORTABLE if *ctbIP* is invalid; MDLERR_PALETTEWRONGSIZE if the color palette is not 256 entries in size.

See Also mdlColorPal_setEntries.

mdlColorPal_getNumColors

```
#include <mscolor.fdf>
#include <mscolor.h>

int mdlColorPal_getNumColors
(
    BSIColorPalette *paletteP      /* => palette */
);
```

Description mdlColorPal_getNumColors returns the number of color entries defined in the color palette pointed to by *paletteP*.

The *paletteP* parameter is a pointer to a palette obtained from mdlColorPal_create.

Returns mdlColorPal_getNumColors returns the number of entries in the color palette.

See Also mdlColorPal_create, mdlColorPal_setEntries.

RGB Conversion Functions

The following table lists RGB Conversion Functions:

Function	Used to
mdlColor_convertRGBtoIndex	get the host windowing system draw color corresponding to the given RGB color definition.
mdlColor_elementColorToRGB	obtain the RGB value corresponding to a design file colortable color number.
mdlColor_luminosityOf	calculate the luminosity of an RGB color.
mdlColor_elementColorFromRGB	obtain a design or reference file color number corresponding to a given RGB value.
mdlColor_rgbToHsv	translate an RGB color definition to its equivalent HSV color definition.

Function	Used to
<code>mdlColor_hsvToRgb</code>	translate an HSV color definition to its equivalent RGB color definition.
<code>mdlColor_interpolateColors</code>	generate a range of colors from a start color and an end color.

mdlColor_convertRGBtoIndex

```
#include <mdl.h>

int mdlColor_convertRGBtoIndex
(
    int          screen,          /* => 0 = Right, 1 = Left */
    RGBColorDef *rgb             /* => Pointer to rgb triad */
);
```

Description `mdlColor_convertRGBtoIndex` obtains an index of the Color Manager's display colors representing the closest match to *rgb*.

screen indicates the right (0) or left (1) screen.

rgb is the color definition to match against the Color Manager's display colors.

Returns `mdlColor_convertRGBtoIndex` returns the Color Manager display color index corresponding to *rgb*. The index will be in the 0-255 range, depending on the video hardware in use.

See Also `mdlColor_matchColorMap`.

mdlColor_elementColorToRGB

```
#include <mcolor.h>
#include <mcolor.fdf>

int mdlColor_elementColorToRGB /* <= SUCCESS or ERROR */
(
    byte      *rgbP,              /* <= Where to store the rgb triad */
    int       *mstrElemColorP,    /* <= closest master dgn file color */
    int       fileNum,            /* => 0=Master. 1-255 = ref file */
    int       colorIndex,         /* => Idx to screen's dgn file ctbl */
    HSVColorDef *hsvTableP        /* => hsv master dgn ctbl or NULL */
);
```

Description `mdlColor_elementColorToRGB` is used to obtain the RGB value corresponding to a design file colortable color number. It can also be used to translate reference file color numbers to the closest matching master design file color number when different color tables are attached to each design file.

rgbP points to a buffer to receive the RGB value.

mstrElemColorP is only valid when *colorIndex* is an index to a reference file's color table; otherwise *mstrElemColorP* is ignored. (*colorIndex* will be an index to a reference file's color table when *fileNum* is greater than zero). The integer pointed to by *mstrElemColorP* will be updated to contain the closest matching color to *colorIndex* that exists in the master colortable index.

fileNum indicates to which design file's colortable the *colorIndex* parameter applies. If zero, the master design file's color table is used. 1-255 indicates an attached reference file.

colorIndex is the colortable index for which an RGB value is needed.

hsvTableP is a pointer to the master design file's colortable converted to HSV values. This is usually set to NULL.

Returns *mdlColor_elementColorToRGB* returns SUCCESS or ERROR if a colortable corresponding to *fileNum* could not be obtained.

See Also *mdlColor_elementColorFromRGB*.

mdlColor_luminosityOf

```
#include <mscolor.fdf>

double mdlColor_luminosityOf/* <= luminosity of RGB */
(
    ULong    red,        /* => red component of RGB */
    ULong    green,      /* => green component of RGB */
    ULong    blue        /* => blue component of RGB */
);
```

Description *mdlColor_luminosityOf* returns the luminosity of an RGB triad. The red, green and blue members of the RGB are passed in as individual parameters.

red, *green*, and *blue* are the red, green and blue members of an RGB value.

Returns *mdlColor_luminosityOf* returns the luminosity of the RGB triad.

mdlColor_elementColorFromRGB

```
#include <mscolor.h>
#include <mscolor.fdf>

int mdlColor_elementColorFromRGB
(
    int      fileNum,    /* => filenum (0=master, 1-255 = ref file */
    byte     *rgbP       /* => find elem color number for this rgb */
);
```

Description `mdlColor_elementColorFromRGB` is used to obtain an appropriate element color number corresponding to a given file number and an RGB value.

fileNum indicates the master design file (0) or an attached reference file (1-255).

rgbP points to an RGB value that needs a corresponding colortable color number.

Returns `mdlColor_elementColorFromRGB` returns an element color number.

See Also `mdlColor_elementColorToRGB`.

mdlColor_rgbToHsv

```
#include <mdl.h>

void mdlColor_rgbToHsv
(
    HSVColor      *hsvP,      /* <= HSV style color coordinate */
    RGBColorDef   *rgbP       /* => RGB color to translate */
);
```

Description `mdlColor_rgbToHsv` translates an RGB (Red-Green-Blue) color definition into its corresponding HSV (Hue-Saturation-Value) color definition.

In the HSV color system, Hue ranges from 0-360 degrees with each value representing the spread of colors ranging from Red (0 & 360 degrees), Yellow (60 degrees), Green (120 degrees), Cyan (180 degrees), Blue (240 degrees), Magenta (300 degrees) and all colors in between. Saturation determines how much “whiteness” is mixed in with the color chosen by hue. A fully saturated (S = 100) color has no whiteness mixed in. On the other hand, an HSV color with a saturation value of zero will appear to be all white (or some shade of gray) regardless of the hue. The Value parameter of an HSV color determines the brightness of a color. When Value = 0, the color will appear black.

hsvP is a pointer to where the translated color definition should be placed.

rgbP is a pointer to the Red-Green-Blue definition to be translated into an HSV color.

Returns `mdlColor_rgbToHsv` has no return value.

See Also `mdlColor_hsvToRgb`.

mdlColor_hsvToRgb

```
#include <mdl.h>

void mdlColor_hsvToRgb
(
    RGBColorDef *rgbP,      /* <= RGB style color coordinate */
    HSVColor    *hsvP      /* => HSV color to translate */
);
```

Description mdlColor_hsvToRgb translates an HSV (Hue-Saturation-Value) color definition into its corresponding RGB (Red-Green-Blue) color definition.

rgbP is a pointer to where the translated color definition should be placed.

hsvP is a pointer to the Hue-Saturation-Value definition to be translated.

Returns mdlColor_hsvToRgb has no return value.

See Also mdlColor_rgbToHsv.

mdlColor_interpolateColors

```
#include <mscolor.h>
#include <mscolor.fdf>

int mdlColor_interpolateColors /* <= SUCCESS or error */
(
    byte      *interpRgbsP, /* <= generated rgbs (packed) */
    RGBColorDef *startP,     /* => start rgb */
    RGBColorDef *endP,       /* => end rgb */
    int        rangeCount,   /* => num steps */
    int        hsvRgbFlag    /* => 0=hsv space, 1=rgb space */
);
```

Description mdlColor_interpolateColors is used to generate a graduating scale of RGB values from a beginning and ending set of RGB values. The scale starts with the RGB value pointed to by *startP* and gradually blends over to the value pointed to by *endP*.

interpRgbsP points to an array of RGB values to be filled in. The memory for this array must be allocated by the caller.

startP points to a beginning RGB value specifying one end of the color scale.

endP points to an ending RGB value specifying the opposite end of the color scale from *startP*.

rangeCount indicates the number of RGB values to be generated in the scale. The array pointed to by *interpRgbsP* must be large enough to hold *rangeCount* RGBs.

hsvRgbFlag indicates whether the color interpolation should be generated across RGB or HSV space. Zero indicates HSV color space. Non-zero indicates RGB color space.

Returns `mdlColor_interpolateColors` returns `SUCCESS` or `MDLERR_BADPARAMETER` if *rangeCount* is invalid.

Color Configuration Functions

The following lists Color Configuration Functions:

Function	Used to
<code>mdlColor_getColorConfig</code>	get the Color Manager's public color configuration information.
<code>mdlColor_getExactColorPositions</code>	get the positions of the exact colors in the current design file's colortable.
<code>mdlColor_reallocateColors</code>	change part of the Color Manager's color configuration.
<code>mdlColor_setExactColorPositions</code>	set the positions of the exact colors in the current design file's colortable.

mdlColor_getColorConfig

```
#include <mcolor.h>
#include <mcolor.fdf>

int mdlColor_getColorConfig
(
    BSIColorSettings *configP, /* => store color info here */
    int              screen    /* => get info for this screen */
);
```

Description `mdlColor_getColorConfig` is used to obtain the Color Manager color configuration for a particular screen. The `BSIColorSettings` structure is defined in `mcolor.h`.

configP points to a `BSIColorSettings` buffer to receive the configuration information.

screen indicates the screen for which the color configuration information is to be obtained.

`mdlColor_getColorConfig` returns `SUCCESS` or one of the following error codes:

Return Value	Meaning
<code>MDLERR_BADARG</code>	<i>configP</i> is invalid.
<code>MDLERR_COLORMGRNOTINITIALIZED</code>	The color manager has not been initialized.
<code>MDLERR_BADSCREENNUMBER</code>	The indicated screen number does not exist.

See Also `mdlColor_reallocateColors`.

mdlColor_getExactColorPositions

```
#include <mscolor.h>
#include <mscolor.fdf>

void mdlColor_getExactColorPositions
(
    byte    *exactColorsMap,    /* <= map of exact rgbs in dgn ctbl */
    boolean *inDgnFileP        /* <= TRUE = stored in dgn file */
);
```

Description `mdlColor_getExactColorPositions` is used to obtain the positions of the exact colors in the design file colortable. The maximum number of exact colors loaded in the host windowing system's color palette is governed by the Exact Colors User Preference. The non-zero bytes in the *exactColorsMap* array only indicate which colortable colors are to be loaded if the current color configuration supports them.

exactColorsMap points to a byte map of 256 bytes to receive the color position information. Each non-zero byte indicates the position of a potential exact color in the design file colortable with the first byte representing color 0 in the colortable and the last representing the background color.

inDgnFileP points to a boolean variable that indicates if the exact position data was obtained from the design file itself (`TRUE`) or from the system default (`FALSE`).

Returns `mdlColor_getExactColorPositions` has no return value.

See Also `mdlColor_setExactColorPositions`, `mdlColor_reallocateColors`.

mdlColor_reallocateColors

```
#include <mscolor.h>
#include <mscolor.fdf>

int mdlColor_reallocateColors /* <= SUCCESS or error code */
(
```

```

ULONG   colorConfigComponent,    /* => color component */
int      newSize,                /* => new size for color component */
int      allowUpdate             /* => update screen & color pals? */
);

```

Description `mdlColor_reallocateColors` is used to alter the number of exact colors to be downloaded to the video hardware. This in turn could cause the MicroStation Color Manager to alter other color allocations to satisfy the request. In particular, the number of balanced colors used by MicroStation may need to decrease if the number of exact colors is increased. This could compromise the quality of rendered images.

colorConfigComponent indicates the color component to alter. Currently, only `COLORCOMPONENT_EXACT` is supported.

newSize indicates the new color allocation size for *colorConfigComponent*.

allowUpdate is used to force a screen update after the new allocation takes effect. If `TRUE`, an update will occur.

Returns `mdlColor_reallocateColors` returns `SUCCESS` or `MDLERR_COLORMGRNOTINITIALIZED` if the Color Manager has not yet initialized itself.

See Also `mdlColor_getColorConfig`.

mdlColor_setExactColorPositions

```

#include <mcolor.h>
#include <mcolor.fdf>

void mdlColor_setExactColorPositions
(
byte    *exactColorsMap,    /* => new exact color positions */
boolean inDgnFile          /* => TRUE = store in dgn file */
);

```

Description The `mdlColor_setExactColorPositions` function is used to select the positions of potential exact colors in the design file color table. The selected positions may be applied to a specific design file (the current design) or they may be applied as a system default setting. An individual file's settings, if any, will override the system default settings. For a visual representation of the exact color positions, see the Edit->Exact Colors pull-down menu of the MicroStation Color Table dialog box.

If the number of positions specified exceeds the number of exact colors that can be downloaded to the host windowing system's color palette, the excess positions are ignored. `mdlColor_getColorConfig` may be used to determine the number of exact colors that are currently downloaded. `mdlColor_reallocateColors` may be used to alter the number of exact colors that may be downloaded.

exactColorsMap is a byte map of flags indicating the positions of the exact colors in the design file colortable. The byte map is 256 bytes in size with

the first byte representing color 0 in the colortable and the last representing the background color. A non-zero byte indicates that the corresponding design file color is to be used as an exact color, if possible.

inDgnFile, if TRUE, indicates that the flags apply only to the current design file. If FALSE, it indicates that the positions should be stored as the system default.

Returns mdlColor_setExactColorPositions has no return value.

See Also mdlColor_getExactColorPositions, mdlColor_getColorConfig, mdlColor_reallocateColors.

Color Map Functions

The color map functions exist mainly for backwards compatibility to MDL applications that were written prior to MicroStation version 5. With the advent of Color Descriptors and Color Palettes in MicroStation version 5, the need for generating and manipulating color maps directly is greatly reduced, if not eliminated.

For more information on Color Descriptors and Color Palettes, see the sections of the same names in this chapter.

The following table lists color map functions:

Function	Used to
mdlColor_assignColorMap	assign a new colortable-to-screen-color map to a given screen.
mdlColor_getColorMap	get the current colortable-to-screen-color map in effect for a given screen.
mdlColor_matchColorMap	generate an array of byte-sized draw values which represents a colortable-to-screen-color map.
mdlColor_matchLongColorMap	generate an array of unsigned long draw values which represents a colortable-to-screen-color map.

mdlColor_assignColorMap

```
#include <mscolor.fdf>

int mdlColor_assignColorMap/* <= SUCCESS or error code */
(
    int      screen,          /* => fileNum[screen] gets the colormap */
    CmapCtbl *colormap,/* => colormap to assign to fileNum[screen] */
```

```
int      fileNum          /* => master dgn file (0) or ref file (1-255) */
);
```

Description `mdlColor_assignColorMap` is used to associate a new colormap with either the master design file colortable or an attached reference file colortable. A colormap is an array of draw values corresponding to each RGB triad in the colortable. Every colortable will have a colormap for each screen in the system.

screen indicates for which screen the colormap is to be used.

colormap is a `CmapCtbl` structure (shown below) containing an array of 256 unsigned longs followed by a pointer to the colortable used to create the colormap:

```
typedef struct
{
    ULong cmap [256];
    byte *colortable;
} CmapCtbl;
```



`CmapCtbl` is not defined in any system header file.

filenum is used to identify the colortable to be mapped. 0 indicates the master design file. 1-255 indicates an attached reference file.

Returns `mdlColor_assignColorMap` returns `SUCCESS`.

See Also `mdlColor_matchLongColorMap`, `mdlColor_getColorMap`.

mdlColor_getColorMap

```
#include <mcolor.fdf>

int mdlColor_getColorMap /* <= SUCCESS or error. */
(
    ULong *colormapP,    /* <= Where to store the colormap. */
    int *numMappingsP,  /* <= Num color mappings. */
    int screen,          /* => colormap for this screen. */
    int maxSize          /* => sizeof *colormapP */
);
```

Description `mdlColor_getColorMap` is used to obtain an existing colormap for the master design file's colortable. A colormap is an array of draw values corresponding to each RGB triad in the colortable. A colortable has a colormap for each screen in the system.

colormapP must point to an array of unsigned long values to receive all or part of the system colormap information.

numMappingsP points to an integer where the number of entries in the system colormap are stored. If *numMappings* is `NULL`, it is ignored.

screen indicates for which screen the colormap is to be obtained.

maxSize specifies how many entries from the system colormap are to be copied to *colormapP*.

Returns `mdlColor_getColorMap` returns `SUCCESS` or `MDLERR_BADARG` if *colormapP* is `NULL`.

See Also `mdlColor_assignColorMap`.

mdlColor_matchColorMap, mdlColor_matchLongColorMap

```
#include <mdl.h>

void mdlColor_matchColorMap
(
    char    *cmap,           /* <= Where to store byte indices */
    char    *ctbl,          /* => Pointer to colortable */
    int     screen,          /* => 0 = right. 1 = left */
    boolean enableDisable   /* => TRUE = allow downloading */
);

void mdlColor_matchLongColorMap
(
    ULong   *cmap,           /* <= Where to store ULong indices */
    char    *ctbl,          /* => Pointer to colortable */
    int     screen,          /* => 0 = right. 1 = left */
    boolean enableDisable   /* => TRUE = allow downloading */
);
```

Description `mdlColor_matchColorMap` and `mdlColor_matchLongColorMap` generate a color map in *cmap*. *cmap* contains the Color Manager display color draw values corresponding to each RGB color definition in *ctbl*.

The only difference between these two functions is in the size of the color map elements they generate. `mdlColor_matchColorMap` generates a color map array of bytes. In this case, the byte values are always indices into a hardware color palette. `mdlColor_matchLongColorMap` generates an array of `unsigned long` values that can be either indices into a hardware color palette or other “draw values” that are specific to the host windowing system. For example, under Windows NT, each color map entry would be equivalent to a `COLORREF` specification.



With the advent of color descriptors and color palettes in MicroStation 5.0, the need for generating and manipulating color maps directly is greatly reduced if not eliminated. Furthermore, `mdlColor_matchColorMap` is not available on all platforms. It is only supported for backwards compatibility to the platforms supported by MicroStation 4.x. If color maps must be used, `mdlColor_matchLongColorMap` is the preferred function for generating them.

cmap is an array of 256 elements (bytes or `unsigned longs`) to receive the matched color indices.

ctbl is a 768-byte color table (256 RGB specifications) to be matched against the Color Manager's display colors.

screen indicates the right (0) or left (1) screen.

If *enableDisable* is set to `TRUE` and the Exact Colors User Preferences variable is non-zero, the number of color definitions defined by the Exact Colors preference are copied from the design file's colortable into the color palette of the host windowing system before the colormap is generated. (Loading colors into the windowing system's color palette like this can be considered a side-effect of this function). If *enableDisable* is set to `FALSE`, the colormap is generated immediately without loading any new colors to the windowing system's color palette.

Returns `mdlColor_matchColorMap` and `mdlColor_matchLongColorMap` have no return value.

See Also `mdlColor_convertRGBtoIndex`, `mdlColor_getExactColorPositions`, `mdlColor_setExactColorPositions`.

Rendering and Image Utilities

This section contains functions for rendering design files, manipulating raster data and working with material tables.

Functions for Importing, Exporting and Manipulating Raster Images

The raster image functions allow an MDL application to import, export and manipulate bitmapped (raster) color images. This allows MDL applications to interface with scanning, paint, and image processing programs.

Bitmapped color images can be loosely divided into three categories:

- **Monochrome** images represent each pixel as either a 0 or 1 bit. The number of colors is two, a foreground color and a background color. Typically, white is the foreground color when viewed on a graphics display, and black is the foreground color when printed on a white piece of paper.
- **Mapped** images represent each pixel with an index into a color palette. The color palette consists of an array of RGB triplets. The number of colors in the image varies with the number of bits used to designate each pixel index, a 4 bit image can contain 16 different colors, an 8 bit image can have 256 colors.

Although the quality of mapped images is limited by the small number of simultaneously available colors, mapped images offer several advantages. The format of mapped images mimics the way most color graphics displays work and mapped images can therefore be displayed without significant preprocessing. Support for mapped images is also much more common among desktop publishing applications.

- **RGB, or unmapped**, images represent each pixel with a three byte RGB triplet designating the red, green and blue intensities. They are also commonly referred to as “24 bit” images since 24 bits of data are required to specify each pixel. This allows for the display of over 16 million (2^{24}) colors. RGB images provide the highest quality format for storing color images.

There are certain disadvantages to using RGB images. RGB image files are usually larger than those in other formats and some word processing or paint applications cannot import them. In order to display RGB images on most displays, it is necessary to reduce the image to a limited number of colors. Color reduction from RGB to a fixed number of colors is frequently done using dithering techniques.



The Save Image As... item in the File menu can be used to save only RGB images.

The following table lists the raster image functions:

Function	Used to
<code>mdlImage_readFileInfo</code>	read image file parameters.
<code>mdlImage_getExtension</code>	get the default extension for image file type.
<code>mdlImage_typeFromExtension</code>	get the image file type from the extension.
<code>mdlImage_getImportFormat</code>	get name and image type for import format.
<code>mdlImage_getExportFormat</code>	get name and image type for export format.
<code>mdlImage_getExportSupport</code>	read whether an image file type is supported for export in the various color modes.
<code>mdlImage_getOSFileType</code>	get the operating system file type for image file type.
<code>mdlImage_readFileToMap</code>	read mapped image file.
<code>mdlImage_readFileToRGB</code>	read image file to RGB (unmapped) image.
<code>mdlImage_createFileFromMap</code>	create file from mapped image.
<code>mdlImage_createFileFromRGB</code>	create file from RGB (unmapped) image.
<code>mdlImage_getBalancedPalette</code>	get balanced color palette.
<code>mdlImage_getOptimizedPalette</code>	get optimized color palette for RGB image.
<code>mdlImage_ditherInitialize</code>	initialize for dithering RGB to mapped image.
<code>mdlImage_ditherRow</code>	dither row of RGB data to map indices.
<code>mdlImage_ditherCleanup</code>	clean up dithering buffers.
<code>mdlDither_drawRow</code>	draw a smoothly shaded image, row by row.

Function	Used to
<code>mdlImage_captureScreenMap</code>	capture screen image map.
<code>mdlImage_getScreenPalette</code>	get screen color palette.
<code>mdlImage_applyGammaToPalette</code>	apply a gamma value to an existing palette.
<code>mdlImage_negate</code>	get an RGB image that is the negated version of an RGB image.
<code>mdlImage_negatePalette</code>	get a negated version of supplied color palette.
<code>mdlImage_mapToRGB</code> <code>mdlImage_mapToRGBBuffer</code>	create an RGB image from a mapped image.
<code>mdlImage_RGBToMap</code>	create a mapped image from an RGB image.
<code>mdlImage_RGBToMapWithGamma</code>	create a mapped image from RGB with gamma correction.
<code>mdlImage_RGBToScreenMap</code>	create a mapped image from RGB with palette/gamma correction for screen display.
<code>mdlImage_resize</code>	expand or compress image.
<code>mdlImage_applyGamma</code>	apply gamma correction to RGB image.
<code>mdlImage_createRasterFromRGB</code>	create a raster element (type 87/88) from an RGB image.
<code>mdlImage_renderViewToRGB</code>	save a (rendered) view to RGB.
<code>mdlImage_checkStop [mdlLib.mtl]</code>	returns <code>TRUE</code> if user enters <Ctrl-C> and displays spinning character (can be passed as <i>stopFunc</i> to other <code>mdlImage_...</code> functions).
<code>mdlImage_readMovie</code>	read a sequence of images into a movie structure.
<code>mdlImage_saveMovie</code>	save a movie structure out to a movie file.
<code>mdlImage_insertMovie</code>	insert movie sequence into movie.
<code>mdlImage_insertMovieFrame</code>	insert movie frame into movie.
<code>mdlImage_deleteMovieFrame</code>	delete frame of movie.
<code>mdlImage_freeMovie</code>	free movie structure.
<code>mdlImage_createFli</code>	create FLI file.
<code>mdlImage_appendFliFromMap</code>	append FLI frame with mapped image.
<code>mdlImage_appendFliFromRGB</code>	append FLI frame with RGB image.
<code>mdlImage_completeFli</code>	complete FLI file creation.

Function	Used to
<code>mdlImage_addMovieFrame</code>	add a single frame to the movie sequence initialized with <code>mdlImage_createMovie</code> .
<code>mdlImage_byteMapToBitMap</code>	create bitmap from byte-mapped image.
<code>mdlImage_byteMapToGreyScale</code>	create greyscale from byte-mapped image.
<code>mdlImage_captureScreen</code>	capture truecolor or palettized screen images.
<code>mdlImage_completeMovie</code>	complete process of saving an animation file.
<code>mdlImage_computeScreenMap</code>	find closest palette colors to screen colors.
<code>mdlImage_convertToUpperLeftHorizontal</code>	convert image orientation to top-left-horizontal.
<code>mdlImage_createFileFromBitMap</code>	create image file from bitmap.
<code>mdlImage_createFileFromBuffer</code>	create image file with specific format.
<code>mdlImage_createMovie</code>	begin process of saving a sequence of images to disk.
<code>mdlImage_displayDescrGet</code>	get display descriptor for a screen.
<code>mdlImage_doubleImage</code>	double the size of movie image.
<code>mdlImage_extMapToRGB</code>	convert byte-mapped to RGB image.
<code>mdlImage_extractBitMapSubImage</code>	stretch bitmap image to subimage.
<code>mdlImage_extractByteMapSubImage</code>	stretch byte-mapped image to subimage.
<code>mdlImage_extractIngrAttach</code>	get Intergraph file header information.
<code>mdlImage_extractPackByteSubImage</code>	stretch packbyte image to subimage.
<code>mdlImage_extractRGBSubImage</code>	stretch RGB image to subimage.
<code>mdlImage_extractSubImage</code>	stretch image to subimage in same format.
<code>mdlImage_extReadFileToMap</code>	read byte-mapped file and optionally compress.
<code>mdlImage_extRGBToMap</code>	convert RGB to byte-mapped image with optional compression.
<code>mdlImage_extRGBToScreenMap</code>	convert RGB to byte-mapped image with palette of closest matching screen colors.
<code>mdlImage_freeImage</code>	free memory associated with image.
<code>mdlImage_getMapUsage</code>	determine which palette indices are used.
<code>mdlImage_greyScaleToBitMap</code>	create bitmap from greyscale image.

Function	Used to
<code>mdlImage_isAVIAvailable</code>	used with <code>mdlImage_getExportFormat</code> to determine if the <code>IMAGEFILE_AVI</code> type is supported on the current platform.
<code>mdlImage_memorySize</code>	find initial size of memory required for image.
<code>mdlImage_mirror</code>	mirror image across side.
<code>mdlImage_packByteBuffer</code>	encode a byte-mapped image in packbyte format.
<code>mdlImage_packByteToBitMap</code>	create bitmap from packbyte image.
<code>mdlImage_packByteToGreyscale</code>	create greyscale from packbyte image.
<code>mdlImage_paletteToGreyscale</code>	create greyscale palette from color palette.
<code>mdlImage_readFileToBitMap</code>	read image from file to bitmap.
<code>mdlImage_remapToScreenPalette</code>	convert mapped image to screen colors.
<code>mdlImage_rgbToBitMap</code>	create bitmap from RGB image.
<code>mdlImage_rgbToGreyscale</code>	create greyscale from RGB image.
<code>mdlImage_RGBToPackByte</code>	convert RGB to packbyte image.
<code>mdlImage_RGBToPackByteWithGamma</code>	convert RGB to packbyte image with gamma correction.
<code>mdlImage_rotate</code>	rotate image about origin.
<code>mdlImage_runLengthDecodeBitMapRow</code>	convert from RLE bitmap to bitmap.
<code>mdlImage_runLengthEncodeBitMap</code>	convert from bitmap to RLE bitmap.
<code>mdlImage_runLengthEncodeBitMapRow</code>	convert from bitmap to RLE bitmap.
<code>mdlImage_setMapIfRGBMatch</code>	set colors of mapped image based on RGB counterpart.
<code>mdlImage_setMapPolygon</code>	mask portion of bitmap or byte-mapped image.
<code>mdlImage_setRGBPolygon</code>	mask portion of RGB image.
<code>mdlImage_stretchRLEToBitMap</code>	convert RLE bitmap to bitmap.
<code>mdlImage_subByteMapFromBitMap</code>	stretch bitmap image to byte-mapped subimage.
<code>mdlImage_subByteMapFromPackByte</code>	stretch packbyte to byte mapped image.
<code>mdlImage_subByteMapFromRLEBitMap</code>	stretch RLE bitmap image to byte-mapped subimage.
<code>mdlImage_tintImage</code>	tint byte-mapped image.
<code>mdlImage_tintPalette</code>	tint palette to RGB color.
<code>mdlImage_typeFromFile</code>	determine file type from file itself.

Function	Used to
mdlImage_unpackByteBuffer	decode packbyte to byte-mapped format.
mdlImage_updateIngrAttach	change Intergraph file header information.
mdlImage_warp	warp image to set of points.

Image File Formats

Raster Format	Extension	Common uses	Bits	Compression	Key Points
Ingr Type 2	.cot	monochrome, color trained	8	none	Used for continuous tone (gray scale) images.
Ingr Type 9	.rle	“old” standard engineering drawings	1	run-length	Uses 1-dimensional run-length compression. Prefer .cit over .rle for binary images.
Ingr Type 24	.cit	engineering drawings	1	CCITT-G4	Uses 2-dimensional CCITT Group 4 Fax compression. Provides highly compressed binary format.
Ingr Type 27/28	.rgb	full color imagery	24	variable	Compressed or uncompressed RGB (red, green, blue) 24-bit color image format.
Ingr Type 27/28	.bum	bump maps	24	variable	.rgb file with a different extension.
Ingr Type 29	(.any)	compressed 8-bit	8	run-length	preferred Intergraph format.
Ingr Type 65	.tg4	engineering drawings	1	tilted CCITT-G4 COT	Tiled version of .cit and .cot formats. For use with large drawings to conserve memory requirements.

Raster Format	Extension	Common uses	Bits	Compression	Key Points
TIF	.tif	interchange format	Any	various	Often used as a “wrapper” for other formats such as CCITT Group 4 Fax.
JPEG	.jpg	gray scale or color photographs	8/24	JPEG	Joint Photographic Experts Group (JPEG). Very high compression rates, but “lossy” compression requires careful use.
Sun Raster	.rs	Sun compatible format	8	run-length	Sun compatibility
X- Windows IMG file	.p	X-Windows palette format	8	none	X-Windows compatibility
X- Windows IMG file	.a	X-Windows palette format	24	none	X-Windows compatibility
GIF*	.gif	popular BBS format	8	LZW	GIF is a trademark of CompuServe, Inc. Supports only mapped (256 color) images.
Targa	.tga	popular PC format supporting binary, mapped pseudo and true color	8/ 16/ 24/ 32	variable	TGA and TARGA are trademarks of Truevision, Inc. Provides uncompressed and run-length encoding.
PCX	.pcx	popular PC format supporting binary, mapped and truecolor	1/8/ 24	run-length	Developed by Zsoft for PC Paintbrush
CALS Type 1	.cal	US DoD compatibility	1	CCITT-G4	SGML compatibility
PICT	.pct	popular Apple and Macintosh format	8/24	run-length	Apple compatibility

Raster Format	Extension	Common uses	Bits	Compression	Key Points
BMP	.bmp	Microsoft Windows bitmap	8/24	none	Microsoft Windows compatibility
Encap. Postscript**	.eps	binary			Encapsulated Postscript Specification by Adobe Systems. Verbose ASCII raster format.
WPG**	.wpg	Word Perfect bitmap format	8	run-length	Word Perfect compatibility
FLI**	.fli	Autodesk Animator "movie" file format		run-length delta-frame	Limited to 320x200 resolution. First frame is decoded.
FLC**	.flc	Autodesk Animator "movie" file format		run-length delta-frame	Faster/better version of .fli files. First frame is decoded.
AVI	.avi	Windows AVI format. Used by Microsoft media player.	24	variable	Only supported for export and only under windows/ windows NT.

* MicroStation supports import of GIF format, but not export.

** MicroStation Review supports export of these file formats, but not import.

Sample Applications

These applications, shipped with MDE, demonstrate the use of the raster image functions:

Application	Description
magnify	Display a magnified portion of the screen in a dialog box.
plaimage	Place image into design file as a raster (type 87/88) element.
rastconv	Convert between the various image file formats and color modes.
scrncapt	Capture screen contents to an image file.
rastshow	Displays attached raster reference file.

Internal Image Format

Internal image formats are identified by the `IMAGEFORMAT` types in `image.h`. The following defines the internal structure of each of these formats.

IMAGEFORMAT_BitMap

This is one form of monochrome data. Each bit represents a single pixel. The left-most bit of byte 1 is the first bit, and the right-most bit is the 8th bit. Each bitmap row is padded to a full byte boundary. The macros `BITMAP_ROWBYTES` and `BITMAP_BYTES` can be used to compute the required storage for a row or bitmap image.

IMAGEFORMAT_RLEBitMap

This is one form of monochrome data. An array of pointers to each row is maintained, and each individual row is an array of unsigned short integers (16-bit integers). Each short integer is the count of the number of 0-bits, then the count of the number of 1-bits, until the image width is reached.

IMAGEFORMAT_ByteMap

This is one form of byte-mapped data. Each byte represents a single color index to a palette of up to 256 colors. No word or long-word alignment is done at the end of each row.

IMAGEFORMAT_GreyScale

This is one form of byte-mapped data. Each byte represents a continuous tone (greyscale) image intensity. Greyscale data has an associated greyscale palette, so greyscale is treated as a special subset of byte-mapped data.

IMAGEFORMAT_PackByte

This is one form of byte-mapped data. An array of pointers to each row is maintained, and each individual row is a run-length encoding scheme.

Packbyte format has a 1-byte tag followed by one or more data bytes. If the tag is zero, it is ignored. If the tag has a range of 1 to 127, that number of bytes follows the tag, and each byte represents a single image pixel. If the tag is in the range of 128 to 255, a single byte follows the tag, and a run of pixels will be generated with that value. The length of the run is 256 minus the value of the tag.

IMAGEFORMAT_RLEByteMap

Users should not specify this format.

IMAGEFORMAT_RGBSeperate

This is one form of RGB data. Red, green and blue intensities are interleaved on a row by row basis. For example:

```
Row1 RRRRRRRRRR...
      GGGGGGGGGG...
      BBBBBBBBBB...

Row2 RRRRRRRRRR...
      GGGGGGGGGG...
      BBBBBBBBBB...
```

IMAGEFORMAT_RGB

This is one form of RGB data. Red, green and blue intensities immediately follow each other. For example:

```
Row1 RGBRGBRGBRGB...
```

IMAGEFORMAT_RGBA

This is one form of RGB data. Red, green, blue and a zero byte immediately follow each other. For example:

```
Row1 RGBARBGBARBGB...
```

mdlImage_readFileInfo

```
#include <image.h>
#include <image.fdf>

int mdlImage_readFileInfo
(
    Point2d *imageSize,    /* <= image size */
    int      *colorMode,    /* <= color mode (see image.h) */
    int      *orientation,  /* orientation (see image.h) */
    char     *name,         /* file name */
    int      fileType       /* file type (see image.h) */
);
```

Description mdlImage_readFileInfo returns information about the image file specified by *name* and *fileType*. *imageSize* is set to the size of the image; *orientation* (file orientation) and *colorMode* (color mode) are set as defined in image.h.

Returns mdlImage_readFileInfo returns SUCCESS if the information is successfully extracted from the image file. MDLERR_BADFILETYPE is returned if *filetype* is not supported for import. MDLERR_CANTOPENFILE is returned if the file cannot be opened.

See Also mdlImage_readFileToMap, mdlImage_readFileToRGB.

mdlImage_getExtension, mdlImage_typeFromExtension

```
int mdlImage_getExtension
(
    char     *extension,    /* <= default extension (3 chars) */
    int      type           /* => image type (see image.h) */
);

int mdlImage_typeFromExtension
(
    char     *fileName      /* => image file name */
);
```

Description mdlImage_getExtension returns the default three character extension for the image file format specified by *type*. The definitions for the various image file types are in image.h. Examples include IMAGEFILE_TIFF, IMAGEFILE_PCX and IMAGEFILE_PICT.

Returns mdlImage_typeFromExtension returns the image file type from the extension of the file specified by *fileName*. If the extension does not have a match, MDLERR_BADARG is returned.

See Also mdlImage_getExportSupport, mdlImage_getOSFileType.

mdlImage_getImportFormat, mdlImage_getExportFormat

```

int mdlImage_getImportFormat
(
    char    *formatName,    /* <= image format name */
    int     *importType,    /* <= IMAGEFILE_ type for format */
    int     index           /* => index of import type to return */
);

int mdlImage_getExportFormat
(
    char    *formatName,    /* <= image format name */
    int     *exportType,    /* <= IMAGEFILE_ type for format */
    int     index           /* => index of import type to return */
);

```

Description mdlImage_getImportFormat and mdlImage_getExportFormat are useful for determining the raster file formats that are supported for import and export by MicroStation. By using these functions, an application can support all of the file formats supported by MicroStation without having to explicitly know what the formats are. The example below, illustrates the use of mdlImage_getExportFormat to fill in the labels of the file type option button for the scrncapt example.

Returns mdlImage_getImportFormat and mdlImage_getExportFormat return SUCCESS if *index* is supported and ERROR otherwise. If an *index* return ERROR, all higher index will also support ERROR.

Example

```

int exportType, i=0;

while (!mdlImage_getExportFormat(formatName, &exportType, i++))
{
    mdlDialog_optionButtonInsertItem(formatName, NULL, NULL, NULL,
                                     &importType, NULL, NULL, NULL,
                                     dimP->dialogItemP->rawItemP, -1);
}

```

See Also mdlImage_getExtension, mdlImage_getExportSupport.

mdlImage_getExportSupport, mdlImage_getOSFileType

```

int mdlImage_getExportSupport
(
    int     *defaultMode,    /* <= default color mode */
    int     *rgbSupport,     /* <= support for 24 bit RGB */
    int     *palette256Support, /* <= support for 256 color palette */
    int     *palette16Support, /* <= support for 16 color palette */
    int     *greyScaleSupport, /* <= support for grey scale */
    int     *colorMode,      /* <= support for monochrome */
    int     imageType        /* => image type (see image.h) */
);

int mdlImage_getOSFileType /* <= SUCCESS or error. */

```



```
(
    ULONG    *osFileType,    /* <= 0.S. file type. */
    int      imageType      /* => image type (see image.h) */
);
```

Description `mdlImage_getExportSupport` returns whether the image file type specified by *imageType* is supported for export in the various color modes. It returns `SUCCESS` if the file type is supported, and `MDLERR_BADARG` if it is not.

`mdlImage_getOSFileType` returns the operating system file type for the image type, *imageType*. The only operating system that currently supports file types is the Macintosh Finder.

See Also `mdlImage_getExtension`, `mdlImage_typeFromExtension`, `mdlFile_setFileType` (Macintosh only).

mdlImage_readFileToMap

```
#include <image.h>
#include <msimage.fdf>

int mdlImage_readFileToMap
(
    byte    **imageMap,      /* <= image map */
    Point2d *imageSize,      /* <= image size */
    byte    *redPalette,     /* <= red palette entries */
    byte    *grnPalette,     /* <= green palette entries */
    byte    *bluPalette,     /* <= palette entries */
    int      *paletteSize,   /* <= palette size */
    char     *fileName,      /* => input file name */
    int      fileType,       /* => input file type (see image.h) */
    MdIFunctionP stopFunc /* => stop function (or NULL) */
);
```

Description `mdlImage_readFileToMap` reads the mapped image file specified by *fileName* and *fileType*. The image file must contain a mapped image with no more than 256 color palette entries. The image map is allocated and the address is returned in *imageMap*. This image map memory should be free by the calling routine when it is no longer required. The size of the image in pixels is returned in *imageSize*.

The red, green and blue color palette entries are returned in *redPalette*, *grnPalette* and *bluPalette*. These buffers should be configured to 256 bytes to accept the largest supported color palette. *paletteSize* is set to the number of color palette entries.

The indirect function *stopFunc* is called for every row of the image. If *stopFunc* returns a non-zero value, processing is aborted. A stop function can be used to allow the user to abort what could be a fairly long process. `mdlImage_checkStop` is provided as a generic stop function. If no stop function is desired, `NULL` can be specified.

Returns `mdlImage_readFileToMap` returns `SUCCESS` if the file is read successfully. If processing is aborted by a non-zero return value from *stopfunc*, this value is returned. Other error values include `MDLERR_INSMEMORY`, `MDLERR_CANTOPENFILE` and `MDLERR_BADFILETYPE`.

See Also `mdlImage_readFileInfo`, `mdlImage_readFileToRGB`, `mdlImage_extReadFileToMap`.

mdlImage_readFileToRGB

```
#include <image.h>
#include <msimage.fdf>

int mdlImage_readFileToRGB
(
    byte          **imageBuffer,      /* <= image buffer */
    Point2d       *imageSize,        /* <= image size */
    char          *filename,          /* => input file name */
    int           fileType,           /* => input file type (see image.h) */
    Point2d       *requestedSize,     /* => req. image size (or NULL) */
    MdlFunctionP stopFunc             /* => stop function (or NULL) */
);
```

Description `mdlImage_readFileToRGB` reads the image file specified by *filename* and *fileType* and returns an RGB image (RGB for each pixel). The image file can be either mapped or unmapped. The image buffer is allocated and the address is returned in *imageBuffer*. This image memory should be freed by the calling routine when it is no longer required. The size of the image in pixels is returned in *imageSize*. If `NULL` is specified for *requestedSize*, *imageSize* is set to the size of the file image, otherwise *imageSize* is set to *requestedSize*.

requestedSize can be used to request an image size other than the one at which the image is stored. The process of resizing the image uses pixel averaging and can substantially increase the time required to read the image. The resizing algorithm produces excellent results, producing expanded or compressed images of significantly higher quality than those produced using simple pixel replication or decimation.

The indirect function *stopFunc* is called for every image row. If the function returns a non-zero status processing is aborted and the that status is returned. This can be used to allow the user to abort what may be a fairly long process. Refer to the section on `mdlImage_checkStop` for additional information on this generic *stopFunc*. If no stop function is required, `NULL` may be specified.

Returns `mdlImage_readFileToRGB` returns `SUCCESS` if the file is read successfully. If processing is aborted by a non-zero return value from *stopfunc*, this value is returned. Other error values include `MDLERR_INSMEMORY`, `MDLERR_CANTOPENFILE` and `MDLERR_BADFILETYPE`.

See Also `mdlImage_readFileInfo`, `mdlImage_readFileToMap`, `mdlImage_extReadFileToRGB`.

mdlImage_createFileFromMap

```
#include <image.h>
#include <msimage.fdf>

int mdlImage_createFileFromMap
(
char          *fileName,      /* => name of file */
int           fileType,      /* => type of file (see image.h) */
int           colorMode,     /* => color mode (see image.h) */
Point2d      *size,         /* => size of image */
byte         *imageMap,      /* => palette index for each pixel */
byte         *redPalette,    /* => red palette values */
byte         *grnPalette,    /* => green palette values */
byte         *bluPalette,    /* => blue palette values */
int           paletteSize,   /* => number of palette entries */
MdlFunctionP stopFunc       /* => stop function (or NULL) */
);
```

Description mdlImage_createFileFromMap creates the image file specified by *fileName*, *fileType* and *colorMode* from a mapped image.

imageMap is the base address of the image map with a single byte color palette index for each of the (*size*->*x* * *size*->*y*) pixels.

The red green and blue color palette entries are passed in *redPalette*, *grnPalette* and *bluPalette*. *paletteSize* specifies number of color palette entries.

The indirect function *stopFunc* is called for every row of the image. If *stopFunc* returns a non-zero value, processing is aborted. A stop function can be used to allow the user to abort what could be a fairly long process. mdlImage_checkStop is provided as a generic stop function. If no stop function is desired, NULL can be specified.

Returns mdlImage_createFileFromMap returns SUCCESS if the file is created successfully. If processing is aborted by a non-zero return value from *stopfunc*, this value is returned. Other error values include MDLERR_INSFMEMORY, MDLERR_CANNOTOPENFILE and MDLERR_BADFILETYPE.

See Also mdlImage_createFileFromRGB, mdlImage_createFileFromBitMap, mdlImage_createFileFromBuffer.

mdlImage_createFileFromRGB

```
#include <image.h>
#include <msimage.fdf>

int mdlImage_createFileFromRGB
(
    char          *fileName,      /* => name of file */
    int           fileType,       /* => type of file (see image.h) */
    int           colorMode,      /* => color mode (see image.h) */
    Point2d       *size,         /* => size of image */
    byte          *imageBuffer,   /* => RGB values for each pixel */
    MdlFunctionP  stopFunc       /* => stop function (or NULL) */
);
```

Description The `mdlImage_createFileFromRGB` creates the image file specified by *fileName*, *fileType* and *colorMode* from an RGB image.

imageBuffer is the base address of the image buffer with red, green and blue rows stored separately for each scan line. The size of *imageBuffer* should be $(3 * size \rightarrow x * size \rightarrow y)$.

The indirect function *stopFunc* is called for every row of the image. If *stopFunc* returns a non-zero value, processing is aborted. A stop function can be used to allow the user to abort what could be a fairly long process. `mdlImage_checkStop` is provided as a generic stop function. If no stop function is desired, NULL can be specified.

Returns `mdlImage_createFileFromRGB` returns `SUCCESS` if the file is created successfully. If processing is aborted by a non-zero return value from *stopfunc*, this value is returned. Other error values include `MDLERR_INSFMEMORY`, `MDLERR_CANNOTOPENFILE` and `MDLERR_BADFILETYPE`.

See Also `mdlImage_createFileFromMap`, `mdlImage_createFileFromBuffer`, `mdlImage_createFileFromBitMap`.

mdlImage_getBalancedPalette

```
#include <image.h>
#include <msimage.fdf>

void mdlImage_getBalancedPalette
(
    byte  *redMap,      /* <= red map */
    byte  *grnMap,      /* <= green map */
    byte  *bluMap,      /* <= blue map */
    int   *numColors    /* <= number of colors */
);
```

Description `mdlImage_getBalancedPalette` calculates a balanced color palette. A balanced color palette will contain a balanced mixture of all possible colors. This palette

could be used for dithering an RGB image, although computing an optimized palette with `mdlImage_getOptimizedPalette` is preferable.

See Also `mdlImage_readFileToRGB`, `mdlImage_ditherInitialize`, `mdlImage_ditherRow`, `mdlImage_ditherCleanup`, `mdlImage_getOptimizedPalette`.

mdlImage_getOptimizedPalette

```
#include <image.h>
#include <msimage.fdf>

int mdlImage_getOptimizedPalette
(
    byte      *redPalette,    /* <= red values for palette */
    byte      *grnPalette,    /* <= green values for palette */
    byte      *bluPalette,    /* <= blue values for palette */
    int        *paletteSize,   /* <= # of entries (<= maxColors */
    int        maxColors,      /* => maximum colors in palette */
    byte      *imageBuffer,    /* => RGB values for each pixel */
    Point2d    *size,          /* => size of imageBuffer */
    MdlFunctionP stopFunc      /* => function to stop (or NULL) */
);
```

Description `mdlImage_getOptimizedPalette` calculates an optimized color palette for an RGB image. The RGB values for each pixel are examined to determine the color palette that will most accurately display the image. The RGB image can then be converted to a mapped image using the `mdlImage_dither...` functions or `mdlImage_RGBToMap`.

The red, green and blue color palette values are returned in *redPalette*, *grnPalette* and *bluPalette*. The number of color palette entries used is returned in *paletteSize*, the maximum number of colors is specified by *maxColors*.

imageBuffer is the base address of the image buffer with red, green, and blue rows stored separately for each scan line. The size of *imageBuffer* should equal $(3 * size->x * size->y)$.

The indirect function *stopFunc* is called for every row of the image. If *stopFunc* returns a non-zero value, processing is aborted. A stop function can be used to allow the user to abort what could be a fairly long process. `mdlImage_checkStop` is provided as a generic stop function. If no stop function is desired, NULL can be specified.

Returns `mdlImage_getOptimizedPalette` returns `SUCCESS` if the palette is calculated successfully. If processing is aborted by a non-zero return value from *stopfunc*, this value is returned. `MDLERR_INSMEMORY` is returned if insufficient memory is available to complete the process.

See Also `mdlImage_readFileToRGB`, `mdlImage_ditherInitialize`, `mdlImage_ditherRow`, `mdlImage_ditherCleanup`, `mdlImage_getBalancedPalette`.

mdlImage_ditherInitialize, mdlImage_ditherRow, mdlImage_ditherCleanup

```

#include <image.h>
#include <msimage.fdf>

int mdlImage_ditherInitialize
(
    int      nColors,          /* => number of palette entries */
    byte     *redPaletteP,     /* => red palette entries */
    byte     *grnPaletteP,     /* => green palette entries */
    byte     *bluPaletteP,     /* => blue palette entries */
    int      width             /* => width of scan lines */
);

int mdlImage_ditherRow
(
    byte     *outputLine,      /* <= Palette indices for each pixel */
    byte     *redP,           /* => red values for each pixel */
    byte     *grnP,           /* => green values for each pixel */
    byte     *bluP,           /* => blue values for each pixel */
    int      width            /* => number of pixels in scan line */
);

void mdlImage_cleanUp(void);

```

Description The `mdlImage_dither...` functions convert unmapped RGB pixel data into color palette indices for a user-specified color palette. A Floyd-Steinberg filter is used to diffuse pixel error in both directions. When used along with the `mdlImage_getOptimizedPalette` function, the conversion from RGB to mapped images can be accomplished with minimal image degradation. This process is used internally when creating mapped files from RGB data using `mdlImage_createFileFromRGB` or converting from RGB to mapped images with `mdlImage_RGBToMap`.

The `mdlImage_ditherInitialize` function must be called first to specify the color palette and scan line width. The `mdlImage_ditherRow` function should then be called for each row. The rows should be processed in order for the error diffusion to occur correctly. `mdlImage_ditherCleanup` should then be called to free internal buffers allocated during the dithering process.

Returns The dithering functions return `SUCCESS` if the process is completed successfully. `MDLERR_INSFMEMORY` is returned if there is insufficient memory to complete the process.

See Also `mdlDither_drawRow`, `mdlImage_getOptimizedPalette`, `mdlImage_RGBToMap`.

mdlDither_drawRow

```
#include <image.h>
#include <msimage.fdf>

void mdlDither_drawRow
(
    MSWindow      *windowP,      /* <= Window to dither in */
    long   x0,      /* => x coord of first pixel in row */
    long   x1,      /* => x coord of last pixel in row */
    long   y,      /* => y coord of first pixel in row */
    byte   *reds,   /* => red values for each pixel */
    byte   *grns,   /* => green values for each pixel */
    byte   *blus    /* => blue values for each pixel */
);
```

Description mdlDither_drawRow lets an application draw a smoothly shaded image, row by row, using the same dithering algorithm that MicroStation employs for rendering. The caller specifies the number of pixels in the row to draw ($x1 - x0 + 1$), the row number (y), and the RGB values associated with each pixel in the row (*reds*, *grns*, *blus*).

windowP points to the window where the dithered row will be drawn. A pointer to a dialog box can be used in place of a pointer to an MSWindow. For more information on the MSWindow structure, see the “Window Manipulation” section of the “User Interface” chapter.

x0 is the X-coordinate of the first pixel (or column) in the dithered row. *x1* is the X-coordinate of the last pixel in the dithered row.

y is the Y-coordinate of all pixels in the dithered row.

reds, *grns*, and *blus* are arrays of RGB color components, corresponding to each pixel in the row. There are $(x1 - x0 + 1)$ elements in each array.

Returns mdlDither_drawRow is of type void; it returns no value.

See Also mdlImage_ditherInitialize, mdlImage_ditherRow, mdlImage_ditherCleanup.

mdlImage_captureScreenMap, mdlImage_getScreenPalette

```

int mdlImage_captureScreenMap
(
    byte    **imageMap,      /* <= image map */
    BSIRect *rectangleP,     /* => rectangle to capture */
    int     screenNum        /* => 0 = main, 1 = secondary */
);

int mdlImage_getScreenPalette
(
    byte    *redPalette,     /* <= red palette entries */
    byte    *grnPalette,     /* <= green palette entries */
    byte    *bluPalette,     /* <= blue palette entries */
    int     *paletteSize,    /* <= number of palette entries */
    int     screenNum        /* => screen number */
);

```

Description `mdlImage_captureScreenMap` and `mdlImage_getScreenPalette` are used to capture part of a MicroStation graphics screen into a mapped image. These functions are used by the SCRNCAPT screen capture utility to capture and store the screen contents as image files.

`mdlImage_captureScreenMap` captures the rectangular portion of the MicroStation display specified by *screenNum* and *rectangleP*. The image map is allocated and the base address returned in *imageMap*. The image map memory should be freed by the calling program when it is no longer required.

`mdlImage_getScreenPalette` returns the color palette for the MicroStation display screen specified by *screenNum*. The red, green and blue palette entries are returned in *redPalette*, *grnPalette* and *bluPalette*. These buffers must be able to contain a maximum of 256 single byte values.

Returns The color palette functions return `SUCCESS` if the process is completed successfully. `MDLERR_INSMEMORY` is returned if insufficient memory is available to complete the process.

See Also `mdlImage_computeScreenMap`.

mdlImage_applyGammaToPalette

```

void mdlImage_applyGammaToPalette
(
    byte    *redMap,         /* <=> red palette values */
    byte    *greenMap,       /* <=> green palette values */
    byte    *blueMap,        /* <=> blue palette values */
    int     paletteSize,     /* => number of palette entries */
    double   gamma           /* => gamma value */
);

```


Description The `mdlImage_applyGammaToPalette` function applies a gamma value to an existing palette. The red, green and blue color palettes values are passed to the function by *redMap*, *greenMap* and *blueMap*. The number of color palette entries is supplied by *paletteSize*. The gamma value to be applied to the palette is supplied by *gamma*.

See Also `mdlImage_getOptimizedPalette`, `mdlImage_getBalancedPalette`.

mdlImage_negate

```
void mdlImage_negate
(
    byte    *rgbOutP, /* <= output buffer */
    byte    *rgbInP,  /* => input buffer */
    Point2d *sizeP    /* => buffer size */
);
```

Description `mdlImage_negate` returns an RGB image that is the negated version of the RGB image supplied to the function. *rgbInP* is the address of the RGB image to use as the seed. The resulting image buffer is allocated and the address is returned in *rgbOutP*. The size of the seed buffer is supplied by *sizeP*.

See Also `mdlImage_negatePalette`.

mdlImage_negatePalette

```
void mdlImage_negatePalette
(
    byte    *redMap,      /* <=> red palette values */
    byte    *greenMap,    /* <=> green palette values */
    byte    *blueMap,     /* <=> blue palette values */
    int     paletteSize   /* => number of palette entries */
);
```

Description The `mdlImage_negatePalette` function returns a negated version of supplied color palette. The red, green and blue color palette values are returned in *redMap*, *greenMap* and *blueMap*. The size of the color palette is supplied by *paletteSize*.

See Also `mdlImage_negate`, `mdlImage_tintPalette`.

mdlImage_mapToRGB, mdlImage_mapToRGBBuffer

```

int mdlImage_mapToRGB
(
    byte          **rgbBufferPP, /* <= image buffer (RGB) */
    byte          *mapP,         /* => image Palette */
    Point2d       *size,         /* => image size */
    byte          *redPalette,    /* => red palette entries */
    byte          *grnPalette,    /* => green palette entries */
    byte          *bluPalette,    /* => blue palette entries */
    MdlFunctionP   stopFunc      /* => stop function (or NULL) */
);

int mdlImage_mapToRGBBuffer
(
    byte          *rgbBufferP,    /* <= image buffer (RGB) */
    byte          *mapP,         /* => image Palette */
    Point2d       *size,         /* => image size */
    byte          *redPalette,    /* => red palette entries */
    byte          *grnPalette,    /* => green palette entries */
    byte          *bluPalette,    /* => blue palette entries */
    MdlFunctionP stopFunc        /* => stop function (or NULL) */
);

```

Description `mdlImage_mapToRGB` and `mdlImage_mapToRGBBuffer` convert a mapped image to an RGB image. The output image is allocated by `mdlImage_mapToRGB` and the base address is returned in *rgbBufferPP*. This buffer should be freed by the calling application when it is no longer required. `mdlImage_mapToRGBBuffer` performs the same task as `mdlImage_mapToRGB`, but does not allocate memory for the output image.

The indirect function *stopFunc* is called for every row of the image. If *stopFunc* returns a non-zero value, processing is aborted. A stop function can be used to allow the user to abort what could be a fairly long process. `mdlImage_checkStop` is provided as a generic stop function. If no stop function is desired, `NULL` can be specified.

Returns The image conversion functions return `SUCCESS` if the palette is calculated successfully. If processing is aborted by a non-zero return value from *stopfunc*, this value is returned. `MDLERR_INSFMEMORY` is returned if insufficient memory is available to allocate the output image.

See Also `mdlImage_RGBToMap`, `mdlImage_RGBToMapWithGamma`, `mdlImage_RGBToScreenMap`, `mdlImage_applyGamma`, `mdlImage_extMapToRGB`.

mdlImage_RGBToMap, mdlImage_RGBToMapWithGamma, mdlImage_RGBToScreenMap

```
int mdlImage_RGBToMap
(
    byte          **imageMapPP, /* <= image map */
    byte          *rgbBuffer,   /* => image buffer (RGB) */
    Point2d       *size,        /* => image size */
    byte          *redPalette,  /* => red palette entries */
    byte          *grnPalette,  /* => green palette entries */
    byte          *bluPalette,  /* => blue palette entries */
    int           paletteSize,   /* => number of palette entries */
    MdlFunctionP  stopFunc      /* => stop function (or NULL) */
);

int mdlImage_RGBToMapWithGamma
(
    byte          **imageMapPP, /* <= image map */
    byte          *rgbBuffer,   /* => image buffer (RGB) */
    Point2d       *size,        /* => image size */
    byte          *redPalette,  /* => red palette entries */
    byte          *grnPalette,  /* => green palette entries */
    byte          *bluPalette,  /* => blue palette entries */
    int           paletteSize,   /* => number of palette entries */
    double        gamma,        /* => gamma correction value */
    MdlFunctionP  stopFunc      /* => stop function (or NULL) */
);

int mdlImage_RGBToScreenMap
(
    byte          **imageMapPP, /* <= image map */
    byte          *rgbBuffer,   /* => image buffer (RGB) */
    Point2d       *sizeP,       /* => image size */
    int           screen,        /* => screen number */
    MdlFunctionP  stopFunc      /* => stop function (or NULL) */
);
```

Description mdlImage_RGBToMap converts an RGB image to an mapped image.

mdlImage_RGBToMapWithGamma converts an RGB image to an mapped image and performs gamma correction to the value specified by *gamma*.

mdlImage_RGBToScreenMap converts an RGB image to a mapped image that is gamma corrected and dithered to the palette for the graphics device indicated by *screen*. The mapped image is then suitable for display by mdlWindow_rasterDataDraw.

The output image is allocated by the conversion routine and the base address is returned in *imageMapPP* for these functions. This buffer should be freed by the calling application when it is no longer required.

The indirect function *stopFunc* is called for every row of the image. If *stopFunc* returns a non-zero value, processing is aborted. A stop function can be used to allow the user to abort what could be a fairly long process. `mdlImage_checkStop` is provided as a generic stop function. If no stop function is desired, `NULL` can be specified.

Returns The image conversion functions return `SUCCESS` if the palette is calculated successfully. If processing is aborted by a non-zero return value from *stopfunc*, this value is returned. `MDLERR_INSFMEMORY` is returned if insufficient memory is available to allocate the output image.

See Also `mdlImage_mapToRGB`, `mdlImage_mapToRGBBuffer`, `mdlImage_applyGamma`, `mdlImage_RGBToPackByteWithGamma`.

mdlImage_resize

```
int mdlImage_resize
(
    byte          **outputImage, /* <= resized output image (RGB) */
    Point2d       *outputSize,   /* => output size */
    byte          *inputImage,   /* => input map (RGB) */
    Point2d       *inputSize,    /* => input size */
    MdlFunctionP  stopFunc      /* => stop function (or NULL) */
);
```

Description `mdlImage_resize` compresses or expands the RGB buffer specified by *inputImage* and *inputSize* to the size specified by *outputSize*. The process of resizing the image uses pixel averaging and can require substantial processing time. The resizing algorithm produces expanded or compressed images of significantly higher quality than those produced using simple pixel replication or decimation techniques.

The indirect function *stopFunc* is called for every row of the image. If *stopFunc* returns a non-zero value, processing is aborted. A stop function can be used to allow the user to abort what could be a fairly long process. `mdlImage_checkStop` is provided as a generic stop function. If no stop function is desired, `NULL` can be specified.

Returns `mdlImage_resize` returns `SUCCESS` if the image is resized successfully. If processing is aborted by a non-zero return value from *stopfunc*, this value is returned. `MDLERR_INSFMEMORY` is returned if insufficient memory is available to allocate the output image.

mdlImage_applyGamma

```
void mdlImage_applyGamma
(
    byte    *rgbOutP,      /* <= output buffer */
    byte    *rgbInP,       /* => input buffer */
    Point2d *sizeP,        /* => buffer size */
    double   gamma         /* => gamma value (1.0 == NONE) */
);
```

Description mdlImage_applyGamma applies the gamma correction value *gamma* to the RGB buffer *rgbInP* and places the result in the RGB buffer *rgbOutP*. The size of the buffer is specified by *sizeP*.

See Also mdlImage_RGBToMapWithGamma.

mdlImage_createRasterFromRGB

```
int mdlImage_createRasterFromRGB
(
    MSElementDescr **edPP,      /* <= raster element */
    byte    *imageBuffer,       /* => RGB image buffer */
    Point2d  *imageSize,        /* => image size (in pixels) */
    Dpoint3d *origin,           /* => origin (current system) */
    double   scale,             /* => scale (current/pixels) */
    int      level,             /* => level */
    int      useCurrentPalette,  /* => use current color table */
    int      transparentBackground, /* => trans. background */
    MdIFunctionP stopFunc      /* => stop func. (or NULL) */
);
```

Description The mdlImage_createRasterFromRGB function creates an element descriptor for a raster (type 87/88) element representing the RGB image specified by *imageBuffer* and *imageSize*. *edPP* is set to point to the element descriptor. This function is used by the plaimage example to implement the **Place Image** tool.

origin specifies the upper left corner of the raster element in the design file.

scale specifies the ratio of design file coordinates in the current coordinate system to pixels.

level specifies the level of the raster element (1-63).

If *useCurrentPalette* is non-zero, the current design file color palette is used and the raster element is calculated to match this palette. Otherwise an optimized color palette is calculated for the image and the image colors are placed at the end of the design file palette. The first 15 palette colors are never changed, the remaining 240 colors may be changed if the input image contains a wide spectrum of colors.

If *transparent* is non-zero, the background (black) pixels are transparent.

The indirect function *stopFunc* is called for every row of the image. If *stopFunc* returns a non-zero value, processing is aborted. A stop function can be used to allow the user to abort what could be a fairly long process. `mdlImage_checkStop` is provided as a generic stop function. If no stop function is desired, `NULL` can be specified.

Returns `mdlImage_createRasterFromRGB` returns `SUCCESS` if the raster element is successfully created. If processing is aborted by a non-zero return value from *stopfunc*, this value is returned. `MDLERR_INSMEMORY` is returned if insufficient memory is available to create the raster element.

mdlImage_renderViewToRGB

```
int mdlImage_renderViewToRGB
(
    byte    **imageBufferPP,    /* <= structure of r,g,b colors */
    Point2d *size,              /* => image size */
    int     renderMode,         /* => 5=Constant, 6=Smooth, 7=Phong */
    boolean stereo,             /* => True for stereo image */
    boolean antiAlias,          /* => True to anti-alias */
    int     view                 /* => view number */
);
```

Description `mdlImage_renderViewToRGB` renders the view specified by *view* and stores the image in the RGB buffer pointed to by *imageBufferPP*. If *imageBufferPP* is `NULL`, the memory for the image buffer is allocated and *imageBufferPP* points to the allocated memory. The size of the image is controlled by *size*. The rendering is controlled by *renderMode*, *stereo*, and *antiAlias*.

Returns `mdlImage_renderViewToRGB` returns `SUCCESS` if the view is successfully rendered. `MDLERR_INSMEMORY` is returned if insufficient memory is available to render the view.

mdlImage_checkStop [mdl.lib.mdl]

```
int mdlImage_checkStop(void);
```

Description Several of the raster image functions require a significant amount of processing time. In order to allow for a graceful method of aborting these processes, an indirect call is made to a function (*stopFunc*) at least once for each row of image data (there is no guarantee that it will be called only once per line). If this function returns a non-zero status, the processing is aborted and that status is returned to the calling routine.

`mdlImage_checkStop` is a function that can be specified for *stopFunc* in any function that accepts this argument. It calls the built-in function `mdlSystem_abortRequested` and returns `MDLIMAGE_PROCESSABORTED` if the user presses <Ctrl-C>. A “spinning” character is displayed in the error field to give the user visual confirmation that processing is continuing. If `mdlImage_checkStop` is used, `mdlSystem_userAbortEnable(FALSE)` must

be called first so that the MDL application itself is not aborted completely, `mdlSystem_userAbortEnable(TRUE)` should be called subsequently to re-enable the user abort functionality.



`mdlImage_checkStop` is contained in the `mdlLib.ml` library, installed in MDE's `mdl/library` directory, and must be specified as an input to the MDL linker.

See Also `mdlSystem_abortRequested`, `mdlSystem_userAbortEnable`.

mdlImage_readMovie, mdlImage_saveMovie

```
#include <image.h>

int mdlImage_readMovie
(
    MSMovie      *movieP,          /* <= movie */
    char         *fileName,        /* => file name */
    int          fileType,         /* => file type */
    byte         *redMap,          /* => red map */
    byte         *grnMap,          /* => green map */
    byte         *bluMap,          /* => blue map */
    int          mapSize,          /* => map size */
    double        gammaCorrection, /* => gamma (1.0 == NONE) */
    boolean      firstFrameOnly,   /* => TRUE for only one frame */
    MdlFunctionP stopFunc,         /* => stop function (or NULL) */
    MdlFunctionP frameFunc        /* => frame function */
);

int mdlImage_saveMovie
(
    char         *fileName,        /* => file name */
    int          fileType,         /* => file type */
    int          colorMode,        /* => color mode */
    MSMovie      *movieP,          /* => movie to save */
    MdlFunctionP stopFunc,         /* => stop function (or NULL) */
    MdlFunctionP frameFunc        /* => frame function */
);
```

Description `mdlImage_readMovie` reads a sequence of images from a single animation file or a series of single image files and stores them in the `MSMovie` structure *movieP*. If an animation file is to be read, *fileName* should contain the file name. If a series of single image files are to be used, *fileName* should point to the first image and an embedded numeric string in *fileName* will be incremented to locate the remaining frames (i.e. if *fileName* is `test01.rgb`, then `test02.rgb`, `test03.rgb` etc. are used for the additional frames). Currently, the only animation file format supported is FLI (*fileType* = `IMAGEFILE_FLI`). *redMap*, *grnMap*, *bluMap* and *mapSize* specify the movie palette. Usually these will be the palette for the graphics screen on which

the movie is to be displayed. If *firstFrameOnly* is non-zero, only the first frame of the movie is read.

`mdlImage_saveMovie` saves the movie *movieP* to a single animation file or series of single image files. If a single image format is used (currently only FLI is supported for animation) *filename* should contain an embedded frame number.

The indirect function *stopFunc* is called for every row of the movie. If *stopFunc* returns a non-zero value, processing is aborted. A stop function can be used to allow the user to abort what could be a fairly long process. `mdlImage_checkStop` is provided as a generic stop function. If no stop function is desired, `NULL` can be specified.

The indirect function *frameFunc* is called for every movie frame. It is useful for indicating progress when processing a long movie. The arguments are as indicated below.

```
void frameFunc
(
    int      frameNumber,    /*<= current frame number */
    int      nFrames        /*<= total number of frames */
);
```

See Also `mdlImage_freeMovie`.

`mdlImage_insertMovie`, `mdlImage_insertMovieFrame`, `mdlImage_deleteMovieFrame`

```
#include <image.h>

int mdlImage_insertMovie
(
    MSMovie      *movieP,          /* <=> movie */
    MSMovieFrame *insertFrameP,    /* => frame to insert at */
    MSMovie      *insertMovieP,    /* => movie to insert */
    int          transitionType,    /* => type of transition */
    int          transitionFrames,  /* => transition frames */
    int          stillTransition,   /* => still transition */
    MdlFunctionP stopFunc          /* => stop function (or NULL) */
);

int mdlImage_insertMovieFrame
(
    MSMovie      *movieP,          /* <=> movie */
    MSMovieFrame *insertFrameP,    /* => frame to insert at */
    byte         *rgbFrameP,       /* => rgb buffer for new frame */
    MdlFunctionP stopFunc          /* => stop function (or NULL) */
);

int mdlImage_deleteMovieFrame
(
```



```
MSMovie      *movieP,          /* <=> movie */
MSMovieFrame **framePP        /* => frame to delete */
);
```

Description mdlImage_insertMovie inserts the movie, *insertMovieP* into the movie *movieP* at *insertFrameP*. The transition type (cut, horizontal wipe, vertical wipe, fade) and the number of transition frames are specified by *transitionType* (see #defines in image.h) and *transitionFrames*. If *stillTransition* is nonzero, the transition is made from the last frame of one sequence to the first frame of the next, otherwise the two sequences are overlapped for a rolling transition.

mdlImage_insertMovieFrame inserts a single frame from the RGB buffer *rgbFrameP* into the movie *movieP* at the frame *insertFrameP*.

mdlImage_deleteMovieFrame deletes the frame *framePP* from the movie *movieP*.

See Also mdlImage_readMovie, mdlImage_saveMovie, mdlImage_freeMovie.

mdlImage_freeMovie

```
#include <image.h>

void mdlImage_freeMovie
(
MSMovie *movieP    /* <=> movie to free */
);
```

Description mdlImage_freeMovie frees the memory associated with a movie.

See Also mdlImage_saveMovie, mdlImage_readMovie.

mdlImage_createFli, mdlImage_appendFliFromMap, mdlImage_appendFliFromRGB, mdlImage_completeFli

```
int mdlImage_createFli
(
void      **fliFilePP,    /* <= fli file pointer */
Point2d *size,           /* => size of the frame */
int      speed,          /* => playback speed in jiffies */
char     *fileName       /* => file to create */
);

int mdlImage_appendFliFromMap
(
void      *fliP,          /* => fli file */
byte      *mapP,          /* => image */
byte      *redMap,        /* => red map */
byte      *grnMap,        /* => green map */
byte      *bluMap,        /* => blue map */
int       paletteSize,    /* => palette size */
);
```

```

MdlFunctionP stopFunc      /* => stop function (or NULL) */
);

int mdlImage_appendFliFromRGB
(
void      *fliP,           /* => fli file (returned by create) */
byte      *redMap,         /* => red map */
byte      *grnMap,         /* => grn map */
byte      *bluMap,         /* => blu map */
int        mapSize,        /* => map size */
byte      *imageBuffer,    /* => RGB buffer */
MdlFunctionP stopFunc      /* => stop function (or NULL) */
);

void mdlImage_completeFli
(
void      *fliP            /* => close the fli file */
);

```

Description AutoDesk Animator FLI files are the only true animation files currently supported by MicroStation. These files contain an entire sequence of images, with each frame defined by the screen changes from the previous frame. MicroStation supports only mapped, 256 color FLI files. The palette for each frame can be different, although on most players this causes distracting flicker.

A FLI file is created by `mdlImage_createFli`. The speed is specified in “jiffies” ($1/60$ of a second) by *speed* and the size of the image in pixels is specified by *size*. A pointer to an internal structure is returned in *fliFilePP*. The contents of this structure are not published or required. The pointer is simply passed directly to the other FLI routines.

Frames are added to the file with either `mdlImage_appendFliFromMap` or `mdlImage_appendFliFromRGB`. The size of these images must match the *size* parameter passed to `mdlImage_createFli`.

`mdlImage_completeFli` is called to complete and close the FLI file.

See Also `mdlImage_readMovie`, `mdlImage_saveMovie`, `mdlImage_freeMovie`.

mdlImage_addMovieFrame

```

#include <image.h>
#include <msimgd1m.fdf>

int mdlImage_addMovieFrame
(
void      *movieContextP, /* => context pointer */
MSMovie   *movieP,       /* => movie to save */
byte      *imageP,       /* => image for frame */
char      *fileNameP,    /* <=> file name */
int        fileType,     /* => file type */

```

```
int          colorMode,      /* => color mode */
int          compression,    /* => quality of compression */
MdlFunctionP stopFunc,       /* => stop function (or NULL) */
MdlFunctionP frameFunc       /* => frame function (or NULL) */
);
```

Description The `mdlImage_addMovieFrame` function adds a single frame to the movie sequence initialized with `mdlImage_createMovie`. For the FLI and FLC single image formats, the frame is appended to the movie *movieP*.

movieContextP is a pointer to the buffer allocated by the call to `mdlImage_createMovie`.

imageP is an RGB buffer for the movie frame to add.

fileNameP for formats that save to a series of single images, is the file name for the single image file to be created; the *fileNameP* returned will be that for the next frame in the series. For the single file formats *fileNameP* is the name of the animation file.

fileType specifies the image format for the animation sequence, these begin with `IMAGEFILE_` and are defined in `IMAGE.H`.

colorMode specifies whether a *imageP* is to be saved as a true-color or mapped image. The `COLORMODE_` defines can be found in `image.h`.

compression specifies the quality of compression to be used, currently this option is only supported by the JPEG format.

stopFunc is an indirect function called for every image row. If the function returns a non-zero status processing is aborted. This allows the user to abort what may be a fairly long process. Refer to the section on `mdlImage_checkStop` for additional information on this generic *stopFunc*. If no stop function is required, `NULL` may be specified.

frameFunc is an indirection function called for every movie frame. It is useful for indicating progress when processing a long movie.



This function was implemented in MicroStation 95.

Returns `mdlImage_addMovieFrame` returns `SUCCESS` if the movie frame is successfully added/created.

See Also `mdlImage_createMovie`, `mdlImage_completeMovie`, `mdlImage_checkStop`.

mdlImage_byteMapToBitMap

```
#include <image.h>
#include <msdefs.h>
#include <mdlerrs.h>
#include <msimgdlm.fdf>

int mdlImage_byteMapToBitMap
```

```
(
byte      *bitMapP,      /* <= bitmap */
byte      *byteMapP,     /* => rgb image */
byte      *redMapP,      /* => red palette values */
byte      *grnMapP,      /* => green palette values */
byte      *bluMapP,      /* => blue palette values */
int       paletteSize,   /* => number of palette entries */
Point2d   *sizeP,        /* => size */
int       ditherMode,    /* => dither mode */
MdIFuncP stopFunc        /* => stop function */
);
```

Description The `mdlImage_byteMapToBitMap` function converts an uncompressed byte-mapped color image to an uncompressed bitmap by dithering the color image.

bitMapP is a pointer to the bitmap image buffer in which results are stored. The caller must allocate memory for this buffer. The format of this buffer is `IMAGEFORMAT_BitMap`.

byteMapP is a pointer to the uncompressed byte-mapped buffer to be converted to monochrome. The format of the buffer is `IMAGEFORMAT_ByteMap`.

redMapP, *grnMapP*, *bluMapP* are pointers to the red, green and blue color palette entries.

paletteSize is the number of palette entries.

sizeP is a pointer to the size of the image in pixels.

ditherMode is either `DITHERMODE_Pattern` to apply a Bayer dither or `DITHERMODE_ErrorDiffusion` to apply a Floyd and Steinberg dithering algorithm. Both defines are in `msdefs.h`.

stopFunc is not used.



This function was implemented in MicroStation 95.

Returns `mdlImage_byteMapToBitMap` returns `SUCCESS` if the operation is completed successfully. `MDLERR_INSFMEMORY` is returned if insufficient memory is available. `MDLERR_BADARG` is returned if an invalid `DITHERMODE` is detected.

See Also `mdlImage_packByteToBitMap`, `mdlImage_greyScaleToBitMap`, `mdlImage_rgbToBitMap`.

mdlImage_byteMapToGreyScale

```
#include <image.h>
#include <msimgdlm.fdf>

int mdlImage_byteMapToGreyScale
(
byte      *greyP,        /* <= greyscale image */
```

```
int      *minGreyP,      /* <= minimum grey value (or NULL) */
int      *maxGreyP,      /* <= maximum grey value (or NULL) */
byte     *byteMapP,      /* => source byte-mapped image */
byte     *redMapP,       /* => red palette entries */
byte     *grnMapP,       /* => green palette entries */
byte     *bluMapP,       /* => blue palette entries */
int      paletteSize,    /* => number of palette entries */
Point2d *sizeP           /* => image size in pixels */
);
```

Description The `mdlImage_byteMapToGreyScale` function converts an uncompressed byte-mapped color image with format `IMAGFORMAT_ByteMap` to a greyscale image based on the formula:

$$\text{grey} = 0.30 * \text{red} + 0.59 * \text{green} + 0.11 * \text{blue}$$

greyP is a pointer to the resulting greyscale image buffer. The caller must allocate memory for this buffer.

minGreyP is the minimum greyscale value. If the pointer is `NULL`, this value is not returned.

maxGreyP is the maximum greyscale value. If the pointer is `NULL`, this value is not returned.

byteMapP is a pointer to the byte-mapped color image data to be converted to greyscale.

redMapP, *grnMapP*, *bluMapP* are pointers to the red, green and blue color palette entries.

paletteSize is the number of palette entries.

sizeP is a pointer to the size of the image in pixels.



This function was implemented in MicroStation 95.

Returns `mdlImage_byteMapToGreyScale` returns `SUCCESS`.

See Also `mdlImage_paletteToGreyScale`, `mdlImage_packByteToGreyScale`, `mdlImage_rgbToGreyScale`.

mdlImage_captureScreen

```
#include <msimage.fdf>
#include <mdlerrs.h>

int mdlImage_captureScreen
(
byte     **image,        /* <= returned image */
BSIRect *rectangleP,    /* => rectangle to Capture */
int      screenNum,      /* => physical screen number */
int      imageFormat     /* => returned image format */
);
```

Description The `mdlImage_captureScreen` function is used to capture part of a MicroStation graphics screen into an RGB or byte-mapped image buffer.

image is the address of a pointer that will point to the image data buffer allocated by `mdlImage_captureScreen`. When the user no longer needs this buffer, this memory should be freed.

rectangleP is a pointer to a rectangular portion of the screen in screen coordinates.

screenNum is the screen number to capture.

imageFormat is either `IMAGEFORMAT_RGB` or `IMAGEFORMAT_ByteMap`. Use `mdlImage_getScreenPalette` to determine the RGB colors for a byte-mapped image.



This function was implemented in MicroStation 95.

Returns `mdlImage_captureScreen` returns `SUCCESS` if image is successfully captured. `MDLERR_BADARG` is returned if an invalid image format is specified. `MDLERR_INSMEMORY` is returned if insufficient memory is available to capture the screen image. Otherwise, it returns a non-zero value.

See Also `mdlImage_captureScreenMap`.

mdlImage_completeMovie

```
#include <msimgdlm.fdf>
#include <image.h>

int mdlImage_completeMovie
(
    void            *movieContextP, /* => context pointer */
    MSMovie         *movieP,        /* => movie pointer */
    char            *fileName,       /* => file name */
    int             fileType,        /* => file type */
    int             colorMode,       /* => color mode */
    int             compression,     /* => quality of compressed image (1-90) */
    MdlFunctionP    stopFunc,        /* => stop function (or NULL) */
    MdlFunctionP    frameFunc       /* => frame function (or NULL) */
);
```

Description The `mdlImage_completeMovie` function is called to complete the process of saving an animation file. In the case of FLI and FLC, *movieP* is processed at this time and the animation file is created. For AVI files, the file handle for the open animation file is closed.

movieContextP is a pointer to the buffer allocated by the call to `mdlImage_createMovie`.

fileNameP is only required for FLI and FLC and is the name of the animation file.

fileType specifies the image format for the animation sequence, these begin with `IMAGEFILE_` and are defined in `image.h`.

colorMode specifies whether a movieP is to be saved as a true-color or mapped image. The `COLORMODE_` defines can be found in `image.h`.

compression specifies the quality of compression to be used, currently this option is only supported by the JPEG format.

stopFunc is an indirect function called for every image row. If the function returns a non-zero status processing is aborted. This allows the user to abort what may be a fairly long process. Refer to the section on `mdlImage_checkStop` for additional information on this generic `stopFunc`. If no stop function is required, `NULL` may be specified.

frameFunc is an indirection function called for every movie frame. It is useful for indicating progress when processing a long movie. If no frame function is required, `NULL` may be specified.



This function was implemented in MicroStation 95.

Returns `mdlImage_completeMovie` returns `SUCCESS` if the animation file was successfully created.

See Also `mdlImage_createMovie`, `mdlImage_addMovieFrame`, `mdlImage_checkStop`.

mdlImage_computeScreenMap

```
#include <image.h>
#include <msimgdlm.fdf>

int mdlImage_computeScreenMap
(
    byte    screenMap[256], /* <= mapping from input palette to screen */
    int     screenNumber,   /* => screen number */
    byte    *redMapP,       /* => red map */
    byte    *grnMapP,       /* => green map */
    byte    *bluMapP,       /* => blue map */
    int     paletteSize     /* => palette size */
);
```

Description The `mdlImage_computeScreenMap` function matches each entry of the palette identified by the red, green and blue palette entries to the closest screen palette entry and returns an index that can be used to remap the image to the screen's palette.

screenMap[256] is an array of indices that will map from the image palette to the closest color in the screen palette. Matching is based on closest HSV values.

screenNumber is the screen number on which the image data is to be displayed. The *screenNumber* is 0 for the main screen, 1 for the secondary screen.

redMapP, *grnMapP*, *bluMapP* are the red, green and blue color palettes for an image.

paletteSize is the number of palette entries. Only *paletteSize* entries of *screenMap* are filled.



This function was implemented in MicroStation 95.

Returns `mdlImage_computeScreenMap` returns `SUCCESS`.

See Also `mdlImage_getScreenPalette`, `mdlImage_remapToScreenPalette`.

mdlImage_convertToUpperLeftHorizontal

```
#include <image.h>
#include <mdlerrs.h>
#include <msimgdlm.fdf>

int mdlImage_convertToUpperLeftHorizontal
(
    byte          **imagePP,      /* <=> image buffer to be modified */
    Point2d       *sizeP,        /* => size of image in pixels */
    int           ingrOrientation, /* => INGR_ORIENT from image.h */
    int           imageFormat,    /* => IMAGEFORMAT from image.h */
    MdIFunctionP  stopFunc       /* => user stop function */
);
```

Description The `mdlImage_convertToUpperLeftHorizontal` function converts an image to upper-left-horizontal orientation, so the first image pixel is the upper-left corner, and subsequent pixels proceed from left to right, then from top to bottom.

This function is normally used with Intergraph files which allow 8 possible image orientations, but it can be used with any other type of image as well.

imagePP is the address of the image buffer. This address can change depending on the type of mirroring or rotation required to normalize the orientation to upper-left-horizontal. If the image is already in upper-left-horizontal orientation, no change occurs.

sizeP is a pointer to the size of the image in pixels.

ingrOrientation is a `INGR_ORIENT_` defined constant in `image.h`.

imageFormat is a `IMAGEFORMAT_` defined constant in `image.h`.

stopFunc is a pointer to a MDL function that is called for every row of the image. If *stopFunc* returns a non-zero value, processing is aborted. A stop function can be used to allow the user to abort what could be a fairly

lengthy process. `mdlImage_checkStop` is provided as a generic stop function. If no stop function is desired, `NULL` can be specified.



This function was implemented in MicroStation 95.

Returns `mdlImage_convertToUpperLeftHorizontal` returns `SUCCESS` the image could be successfully reoriented. `MDLERR_INSMEMORY` is returned if insufficient memory is available. `MDLERR_USERABORT` is returned if the user aborts the read by pressing the reset button.

See Also `mdlImage_extractIngrAttach`, `mdlImage_checkStop`.

mdlImage_createFileFromBitMap

```
#include <image.h>
#include <mdlerrs.h>
#include <msimgdlm.fdf>

int mdlImage_createFileFromBitMap
(
    char          *nameP,           /* => file name */
    int           type,             /* => file type (image.h) */
    Point2d       *sizeP,          /* => image size */
    byte          *bitMapP,        /* => bit map */
    boolean       runLengthEncode, /* => bit map is run length encoded */
    RGBColorDef   *foregroundColorP, /* => foreground color */
    RGBColorDef   *backgroundColorP, /* => background color */
    MdlFunctionP  stopFunc         /* => stop function (or NULL) */
);
```

Description The `mdlImage_createFileFromBitMap` function creates an image file from a memory bitmap.

nameP is the full path name of the file to be written.

type is one of the possible `IMAGEFILE_` types contained in `image.h`. If *type* is other than a monochrome type, this function converts the bitmap to a byte mapped format using the specified *foregroundColorP* and *backgroundColorP* colors, then calls `mdlImage_createFileFromMap`. The monochrome values for *type* include: `IMAGEFILE_CIT`, `IMAGEFILE_RLE`, `IMAGEFILE_TIFF` and `IMAGEFILE_RAWTIFF`.

sizeP is the size of the bitmap in pixels.

bitMapP is a pointer to the bitmap image data. Data must be in either `IMAGEFORMAT_BitMap` or `IMAGEFORMAT_RLEBitMap` formats.

runLengthEncode indicates whether the data has been compressed or not. The following data formats are assumed depending on the value of *runLengthEncode*:

<i>runLengthEncode</i> Value	Image Format Type	Description
TRUE	IMAGEFORMAT_RLEBitMap	Run length encoded (generally most compressed)
FALSE	IMAGEFORMAT_BitMap	Bit per pixel (generally least compressed)

foregroundColorP is the RGB value for the foreground color. This value is used only if type is not one of the monochrome types listed above.

backgroundColorP is the RGB value for the background color. This value is used only if type is not one of the monochrome types listed above.

stopFunc is a pointer to a MDL function that is called for every row of the image. If *stopFunc* returns a non-zero value, processing is aborted. A stop function can be used to allow the user to abort what could be a fairly lengthy process. *mdlImage_checkStop* is provided as a generic stop function. If no stop function is desired, NULL can be specified.



This function was implemented in MicroStation 95.

Returns *mdlImage_createFileFromBitMap* returns SUCCESS if the file is created successfully. MDLERR_INSMEMORY is returned if insufficient memory is available. MDLERR_USERABORT is returned if the user aborts the read by pressing the reset button.

See Also *mdlImage_createFileFromMap*, *mdlImage_createFileFromRGB*, *mdlImage_checkStop*.

mdlImage_createFileFromBuffer

```
#include <image.h>
#include <mdlerrs.h>
#include <msimgdlm.fdf>

int mdlImage_createFileFromBuffer
(
    char        *saveAsFileNameP, /* => name of file */
    int         fileType,          /* => type of file (see image.h) */
    int         colorMode,        /* => color mode (see image.h) */
    Point2d     *sizeP,           /* => size of image */
    byte        *imageDataP,      /* => data for each pixel (various formats) */
    int         imageFormat,      /* => input buffer format */
    void        *param1P,         /* => pointer to parameter 1 */
    void        *param2P,         /* => pointer to parameter 2 */

```

```
void      *param3P,      /* => pointer to parameter 3 */
int       paletteSize,   /* => number of palette entries */
int       compression,   /* => compression (1-90) */
MdlFunctionP stopFunc    /* => stop function (or NULL) */
);
```

Description The mdlImage_createFileFromBuffer function creates a file from an image buffer by using the *imageFormat* to determine how to best write the file. The following subfunctions are called:

<i>imageFormat</i> Value	Function Used
IMAGEFORMAT_BitMap IMAGEFORMAT_RLEBitMap	mdlImage_createFileFromBitMap
IMAGEFORMAT_ByteMap IMAGEFORMAT_GreyScale	mdlImage_extCreateFileFromMap
IMAGEFORMAT_PackByte	mdlImage_createFileFromPackByte or if JPEG, then mdlImage_extCreateFileFromRGB
IMAGEFORMAT_RGBSeperate IMAGEFORMAT_RGB IMAGEFORMAT_RGBA	mdlImage_extCreateFileFromRGB

saveAsFileNameP is a pointer to the full path name of the file created by this function.

fileType is one of the IMAGEFILE_ constants in image.h identifying the disk format or -1 to use the file name extension to determine the disk format.

colorMode is one of the COLORMODE_ constants in image.h identifying the color mode of the source buffer.

sizeP is a pointer to a Point2d structure that identifies the size in pixels of the source image.

imageDataP is a pointer to the source data in one of the indicated formats.

imageFormat is one of the IMAGEFORMAT_ constants in image.h identifying the memory storage format.

param1P is a pointer that has several uses depending on the *imageFormat*.

<i>imageFormat</i> Value	<i>param1P</i> usage
IMAGEFORMAT_BitMap IMAGEFORMAT_RLEBitMap	RGBColorDef <i>*foregroundColorP</i>
IMAGEFORMAT_ByteMap IMAGEFORMAT_GreyScale IMAGEFORMAT_PackByte	byte <i>*redMapP</i>
Other	unused

param2P is a pointer that has several uses depending on the *imageFormat*.

<i>imageFormat</i> Value	<i>param2P</i> usage
IMAGEFORMAT_BitMap IMAGEFORMAT_RLEBitMap	RGBColorDef * <i>backgroundColorP</i>
IMAGEFORMAT_ByteMap IMAGEFORMAT_GreyScale IMAGEFORMAT_PackByte	byte * <i>grnMapP</i>
Other	unused

param3P is a pointer that has several uses depending on the *imageFormat*.

<i>imageFormat</i> Value	<i>param3P</i> usage
IMAGEFORMAT_BitMap IMAGEFORMAT_RLEBitMap	unused
IMAGEFORMAT_ByteMap IMAGEFORMAT_GreyScale IMAGEFORMAT_PackByte	byte * <i>bluMapP</i>
Other	unused

paletteSize identifies the number of palette entries if the image format is IMAGEFORMAT_ByteMap, IMAGEFORMAT_GreyScale or IMAGEFORMAT_PackByte.

compression identifies the compression ratio for JPEG images. Otherwise, it is not used.

stopFunc is a pointer to a MDL function that is called for every row of the image. If *stopFunc* returns a non-zero value, processing is aborted. A stop function can be used to allow the user to abort what could be a fairly lengthy process. `mdlImage_checkStop [mdl1ib.m1]` is provided as a generic stop function. If no stop function is desired, `NULL` can be specified.



This function was implemented in MicroStation 95.

Returns `mdlImage_createFileFromBuffer` returns `SUCCESS` if the file is created successfully. `MDLERR_INSMEMORY` is returned if insufficient memory is available. `MDLERR_USERABORT` is returned if the user aborts the read by pressing the reset button.

See Also `mdlImage_createFileFromBitMap`, `mdlImage_extCreateFileFromMap`, `mdlImage_extCreateFileFromRGB`, `mdlImage_checkStop [mdl1ib.m1]`.

mdlImage_createMovie

```
#include <msimgdlm.fdf>
#include <image.h>

int mdlImage_createMovie
(
void    **movieContextPP,    /* <=> context pointer */
MSMovie *movieP,            /* => movie pointer */
char    *fileNameP,         /* => file name */
int     fileType,           /* => file type */
);
```

Description The `mdlImage_createMovie` function is called to begin the process of saving a sequence of images to disk. In the case of FLI, FLC and AVI, the images are saved to a single animation file; whereas the other supported image formats save to a series of single image files.

movieContextPP is a pointer to a buffer which is allocated to store information that will be needed by `mdlImage_addMovieFrame` and `mdlImage_completeMovie`.

movieP is a pointer to a `MSMovie` structure. There are members of this structure to specify the palette and size of a movie frame. For FLI, FLC and AVI formats, you must also specify a playback speed.

fileNameP specifies the name of the animation file for the single image formats FLI, FLC and AVI. For the other formats which create a series of single image files *fileNameP* must contain an imbedded frame number (e.g. FRAME001.TIF).

fileType specifies the image format for the animation sequence, these begin with `IMAGEFILE_` and are defined in `image.h`.



This function was implemented in MicroStation 95.

Returns `mdlImage_createMovie` returns `SUCCESS` if *fileType* is valid and *movieContextPP* is successfully initialized.

See Also `mdlImage_addMovieFrame`, `mdlImage_completeMovie`.

mdlImage_displayDescrGet

```
#include <image.h>
#include <global.h>
#include <msimgdlm.fdf>

MSDisplayDescr *mdlImage_displayDescrGet
(
int     screen    /* => screen number */
);
```

Description The `mdlImage_displayDescrGet` function returns a pointer to the display descriptor associated with the indicated screen. From the display descriptor, defined in `global.h`, information can be obtained such as the display resolution and the number of colors that the screen supports.

This function is similar to `mdlWindow_displayDescrGet`, except that during image processing, it is sometimes useful to obtain information about the display attributes without necessarily having an associated window.

screen is the screen number on which the image data is to be displayed. The screen is 0 for the main screen, 1 for the secondary screen.



This function was implemented in MicroStation 95.

Returns `mdlImage_displayDescrGet` returns a pointer to the display descriptor for the appropriate screen.

See Also `mdlWindow_displayDescrGet`.

mdlImage_doubleImage

```
#include <image.h>
#include <msimgdlm.fdf>

void mdlImage_doubleImage
(
    byte    *outMapP,      /* <= output image (caller malloc) */
    byte    *inMapP,       /* => source image */
    Point2d *sizeP,        /* => image size in pixels */
    int     bytesPerPixel /* => bytes per pixel (1=Map, 3=RGB) */
);
```

Description The `mdlImage_doubleImage` function doubles the size of the source image to produce the output image. This function is useful when producing movies to double the size of the image so that it can be viewed easier.

outMapP is a pointer to a buffer to hold the destination image. Since the size of the image is doubled in each dimension, the buffer is actually 4 times as large as the source image. Its size is computed as follows:

$4 * sizeP->x * sizeP->y * bytesPerPixel$

inMapP is a pointer to the source image. The image format is either `IMAGEFORMAT_ByteMap` or `IMAGEFORMAT_RGB`.

sizeP is a pointer to the size of the image in pixels.

bytesPerPixel is the number of bytes required per pixel. If the image has format `IMAGEFORMAT_ByteMap`, *bytesPerPixel* is 1. If the image has format `IMAGEFORMAT_RGB`, *bytesPerPixel* is 3.



This function was implemented in MicroStation 95.

Returns mdlImage_doubleImage is of type void; it returns no value.

See Also mdlImage_readMovie.

mdlImage_extMapToRGB

```
#include <image.h>
#include <mdlerrs.h>
#include <msimgd1m.fdf>

int mdlImage_extMapToRGB
(
    byte          **rgbBufferPP, /* <= image buffer (RGB) */
    byte          *mapP,         /* => image map */
    Point2d       *sizeP,        /* => image size */
    byte          *redMapP,       /* => red palette entries */
    byte          *grnMapP,       /* => green palette entries */
    byte          *bluMapP,       /* => blue palette entries */
    boolean        packBytes,     /* => pack bytes (run length encode) */
    MdIFunctionP   stopFunc      /* => stop function (or NULL) */
);
```

Description The mdlImage_extMapToRGB function converts a byte-mapped image to a RGB image. The byte-mapped image can be either compressed or uncompressed. This function supplements the mdlImage_mapToRGB function by handling compressed byte-mapped data.

The function converts from IMAGEFORMAT_ByteMap or IMAGEFORMAT_PackByte to IMAGEFORMAT_RGBSeperate.

rgbBufferPP is a pointer to a buffer allocated by mdlImage_extMapToRGB. When this memory is no longer required, it should be freed by the calling routine by using mdlImage_freeMovie.

mapP is the address of the byte-mapped image data that is converted to RGB format.

sizeP is a pointer to the size of the image in pixels.

redMapP, *grnMapP*, *bluMapP* are the red, green and blue color palette for an image.

paletteSize is the number of palette entries.

stopFunc is a pointer to a MDL function that is called for every row of the image. If *stopFunc* returns a non-zero value, processing is aborted. A stop function can be used to allow the user to abort what could be a fairly lengthy process. mdlImage_checkStop [mdl11b.m1] is provided as a generic stop function. If no stop function is desired, NULL can be specified.

mdlImage_extMapToRGB returns SUCCESS if the image could be successfully converted. MDLERR_INSFMEMORY is returned if insufficient memory is

available. `MDLERR_USERABORT` is returned if the user aborts the operation by pressing the reset button.



This function was implemented in MicroStation 95.

See Also `mdlImage_mapToRGB`, `mdlImage_extRGBToMap`, `mdlImage_freeImage`, `mdlImage_checkStop` [`mdlLib.mli`].

mdlImage_extractBitMapSubImage

```
#include <image.h>
#include <mdlerrs.h>
#include <msimgd1m.fdf>

int mdlImage_extractBitMapSubImage
(
    byte          *outBitMapP,    /* <= output bitmap (allocated by caller) */
    Point2d       *outSizeP,      /* => output image size */
    byte          *inBitMapP,     /* => input bitmap */
    Point2d       *inSizeP,       /* => input image size */
    Drectangle    *rectP          /* => sub rectangle to extract (or NULL) */
);
```

Description The `mdlImage_extractBitMapSubImage` function extracts a bitmapped subimage from a larger image and stretches or decimates it to fit a user-defined output image size.

outMapP is a pointer to a buffer allocated by the caller. The format of this buffer is `IMAGEFORMAT_BitMap` and the size of this buffer is:

```
mdlImage_memorySize(outSizeP, IMAGEFORMAT_BitMap)
```

outSizeP defines the output image size in pixels.

inBitMapP is a pointer to the source image buffer in `IMAGEFORMAT_BitMap` format.

inSizeP defines the size of the source image in pixels.

rectP defines a rectangular subimage in pixels. The rectangle defined by *rectP* should be a subset of the rectangle with origin at (0,0) and corner at (*inSizeP*->x - 1, *inSizeP*->y - 1). If *rectP* is NULL, the subimage rectangle is the entire image.



This function was implemented in MicroStation 95.

Returns `mdlImage_extractBitMapSubImage` returns `SUCCESS` if the operation is completed successfully. `MDLERR_INSFMEMORY` is returned if insufficient memory is available.

See Also `mdlImage_extractSubImage`, `mdlImage_extractByteMapSubImage`, `mdlImage_extractPackByteSubImage`, `mdlImage_extractRGBSubImage`, `mdlImage_extractRLEBitMapSubImage`, `mdlImage_memorySize`.

mdlImage_extractByteMapSubImage

```
#include <image.h>
#include <mdlerrs.h>
#include <msimgdlm.fdf>

int mdlImage_extractByteMapSubImage
(
    byte          *outMapP, /* <= output byte map (allocated by caller) */
    Point2d       *outSizeP, /* => output image size */
    byte          *inMapP,  /* => input byte map */
    Point2d       *inSizeP, /* => input image size */
    Drectangle    *rectP,   /* => sub rectangle to extract (or NULL) */
    byte          *mapP     /* => redirection map (or NULL) */
);
```

Description The `mdlImage_extractByteMapSubImage` function extracts a byte mapped subimage from a larger image and stretches or decimates it to fit a user-defined output image size.

outMapP is a pointer to a buffer allocated by the caller. The format of this buffer is `IMAGEFORMAT_ByteMap` and the size of this buffer is:

```
mdlImage_memorySize(outSizeP, IMAGEFORMAT_ByteMap)
```

outSizeP defines the output image size in pixels.

inMapP is a pointer to the source image buffer in `IMAGEFORMAT_ByteMap` format.

inSizeP defines the size of the source image in pixels.

rectP defines a rectangular subimage in pixels. The rectangle defined by *rectP* should be a subset of the rectangle with origin at (0,0) and corner at (*inSizeP*->x - 1, *inSizeP*->y - 1). If *rectP* is `NULL`, the subimage rectangle is the entire image.

mapP defines an array of color indices used to remap the colors. *mapP* is a pointer to an array of 256 elements, or it is `NULL` if no color mapping is to occur.

`mdlImage_extractByteMapSubImage` returns `SUCCESS` if the operation is completed successfully. `MDLERR_INSFMEMORY` is returned if insufficient memory is available.



This function was implemented in MicroStation 95.

See Also `mdlImage_extractSubImage`, `mdlImage_extractPackByteSubImage`, `mdlImage_extractBitMapSubImage`, `mdlImage_extractRGBSubImage`, `mdlImage_extractRLEBitMapSubImage`, `mdlImage_memorySize`.

mdlImage_extractIngrAttach

```
#include <image.h>
#include <mdlerrs.h>
#include <msimgd1m.fdf>

int mdlImage_extractIngrAttach
(
    Dpoint3d    *originP,      /* <=  origin of Ingr. file (or NULL) */
    Dpoint2d    *extentP,     /* <=  extent of Ingr. file (or NULL) */
    int         *orientationP, /* <=> Ingr. orientation (or NULL) */
    char        *fileNameP    /* <=  Ingr. filename to read (or NULL) */
);
```

Description The `mdlImage_extractIngrAttach` function reads the origin, extent and orientation from the header of an Intergraph image file, or if it is not an Intergraph file, will modify the orientation to correspond to the one of the `INGR_ORIENT` types defined in `image.h`.

Intergraph files allow one of eight possible orientations, either horizontal or vertical scan line orientations from each of the four corners of an image. However, image functions in V5 only recognized horizontal scan line orientations each of the four corners. If the *fileNameP* parameter is `NULL`, or names a non-Intergraph file, *orientationP* will be mapped to `INGR_ORIENT` style orientation naming.

originP is the origin of the image in world coordinates. If it is `NULL`, the image origin is not returned.

extentP is the extent of the image in world coordinates. If it is `NULL`, the image extent is not returned.

orientationP is a pointer to the orientation. If *fileNameP* names an Intergraph file the value returned is read from the file's header. If *fileNameP* names a non-Intergraph file or is `NULL`, the value of *orientationP* is translated to an Intergraph type as follows:

<i>orientationP</i> input	<i>orientationP</i> returned
TOP_LEFT	INGR_ORIENT_UpperLeftHorizontal
TOP_RIGHT	INGR_ORIENT_UpperRightHorizontal
LOWER_LEFT	INGR_ORIENT_LowerLeftHorizontal
LOWER_RIGHT	INGR_ORIENT_LowerRightHorizontal
other	INGR_ORIENT_UpperLeftHorizontal

fileNameP the full path name of a file or `NULL`.



This function was implemented in MicroStation 95.

Returns `mdlImage_extractIngrAttach` returns `SUCCESS` an Intergraph file is found, and the header can be read. `MDLERR_CANNOTOPENFILE` is returned if *fileNameP* is non-NULL and the file cannot be opened. `MDLERR_INSFINFO` is returned whenever *orientationP* remapping occurs.

See Also `mdlImage_updateIngrAttach`, `mdlImage_readFileInfo`.

mdlImage_extractPackByteSubImage

```
#include <image.h>
#include <mdlerrs.h>
#include <msimgdlm.fdf>

int mdlImage_extractPackByteSubImage
(
    byte    **outRowPP, /* <= output pack byte buffer (alloc by caller) */
    Point2d *outSizeP,  /* => output image size */
    byte    **inRowPP,  /* => input pack byte array */
    Point2d *inSizeP,   /* => input image size */
    Drectangle *rectP,  /* => sub rectangle to extract (or NULL)*/
    byte    *mapP       /* => redirection map (or NULL) */
);
```

Description The `mdlImage_extractPackByteSubImage` function extracts a packbyte subimage from a larger image and stretches or decimates it to fit a user-defined output image size.

outRowPP is a pointer to a buffer allocated by the caller. The format of this buffer is `IMAGEFORMAT_PackByte` and the size of this buffer is:

```
mdlImage_memorySize(outSizeP, IMAGEFORMAT_PackByte)
```

outSizeP defines the output image size in pixels.

inRowPP is a pointer to the source image buffer in `IMAGEFORMAT_PackByte` format.

inSizeP defines the size of the source image in pixels.

rectP defines a rectangular subimage in pixels. The rectangle defined by *rectP* should be a subset of the rectangle with origin at (0,0) and corner at (*inSizeP*->x - 1, *inSizeP*->y - 1). If *rectP* is `NULL`, the subimage rectangle is the entire image.

mapP defines an array of color indices used to remap the colors. *mapP* is a pointer to an array of 256 elements, or it is `NULL` if no color mapping is to occur.



This function was implemented in MicroStation 95.

Returns `mdlImage_extractPackByteSubImage` returns `SUCCESS` if the operation is completed successfully. `MDLERR_INSMEMORY` is returned if insufficient memory is available.

See Also `mdlImage_extractSubImage`, `mdlImage_extractByteMapSubImage`,
`mdlImage_extractBitMapSubImage`, `mdlImage_extractRGBSubImage`,
`mdlImage_extractRLEBitMapSubImage`, `mdlImage_memorySize`.

mdlImage_extractRGBSubImage

```
#include <image.h>
#include <msimgdlm.fdf>

int mdlImage_extractRGBSubImage
(
    byte    *outP,      /* <= output rgb image (allocated by caller) */
    Point2d *outSizeP, /* => output image size */
    byte    *inP,       /* => input rgb image */
    Point2d *inSizeP,   /* => input image size */
    Drectangle *rectP   /* => sub rectangle to extract (or NULL) */
);
```

Description The `mdlImage_extractRGBSubImage` function extracts a RGB subimage from a larger image and stretches or decimates it to fit a user-defined output image size.

outP is a pointer to a buffer allocated by the caller. The format of this buffer is `IMAGEFORMAT_RGBSeperate` and the size of this buffer is:

```
mdlImage_memorySize(outSizeP, IMAGEFORMAT_RGBSeperate)
```

outSizeP defines the output image size in pixels.

inP is the source image buffer in `IMAGEFORMAT_RGBSeperate` format.

inSizeP defines the size of the source image in pixels.

rectP defines a rectangular subimage in pixels. The rectangle defined by *rectP* should be a subset of the rectangle with origin at (0,0) and corner at (*inSizeP*->x - 1, *inSizeP*->y - 1). If *rectP* is NULL, the subimage rectangle is the entire image.



This function was implemented in MicroStation 95.

Returns `mdlImage_extractRGBSubImage` returns `SUCCESS`.

See Also `mdlImage_extractSubImage`, `mdlImage_extractByteMapSubImage`,
`mdlImage_extractPackByteSubImage`, `mdlImage_extractBitMapSubImage`,
`mdlImage_extractRLEBitMapSubImage`, `mdlImage_memorySize`.

mdlImage_extractSubImage

```
#include <image.h>
#include <mdlerrs.h>
#include <msimgd1m.fdf>

int mdlImage_extractSubImage
(
    byte    *outP,          /* <= extracted subimage */
    Point2d *outSizeP,      /* => pixel dimension of subimage */
    byte    *inP,          /* => source image */
    Point2d *inSizeP,       /* => source image size */
    Drectangle *rectP,      /* => source subimage rectangle (or NULL)*/
    int     imageFormat     /* => format of source image */
);
```

Description The mdlImage_extractSubImage function extracts a subimage from a larger image and stretches or decimates it to fit a user-defined output image size.

outP is a pointer to a buffer allocated by the caller. The format of this buffer is *imageFormat* and the size of this buffer is:

```
mdlImage_memorySize(outSizeP, imageFormat)
```

outSizeP defines the output image size in pixels.

inP is a pointer to the source image buffer.

inSizeP defines the size of the source image in pixels.

rectP defines a rectangular subimage in pixels. The rectangle defined by *rectP* should be a subset of the rectangle with origin at (0,0) and corner at (*inSizeP*->x - 1, *inSizeP*->y - 1). If *rectP* is NULL, the subimage rectangle is the entire image.

imageFormat is an IMAGEFORMAT type defined in image.h.

Returns mdlImage_extractSubImage returns SUCCESS if the operation is completed successfully. MDLERR_INSFMEMORY is returned if insufficient memory is available.

See Also mdlImage_extractByteMapSubImage, mdlImage_extractPackByteSubImage, mdlImage_extractRGBSubImage, mdlImage_extractBitMapSubImage, mdlImage_extractRLEBitMapSubImage, mdlImage_memorySize.

mdlImage_extReadFileToMap

```

#include <image.h>
#include <mdlerrs.h>
#include <msimgdlm.fdf>

int mdlImage_extReadFileToMap
(
    byte          **imageMapPP, /* <= image map */
    Point2d       *imageSizeP,  /* <= image size */
    byte          *redMapP,     /* <= red palette entries */
    byte          *grnMapP,     /* <= green palette entries */
    byte          *bluMapP,     /* <= palette entries */
    int           *paletteSizeP, /* <= palette size */
    char          *fileNameP,   /* => input file name */
    int           fileType,     /* => input file type */
    boolean       packBytes,    /* => pack bytes (run length encode) */
    MdlFunctionP stopFunc      /* => stop function (or NULL) */
);

```

Description The `mdlImage_extReadFileToMap` function reads a byte-mapped image file. The image file contains a mapped image with no more than 265 color palette entries.

This function is an extension to the `mdlImage_readFileToMap` function provided with V5, except that the format of the data returned is run length encoded if the *packBytes* flag is set. The *packbytes* compression scheme is a compact encoding scheme that rarely increases the size required to maintain a byte-mapped image, although it does typically add somewhat to the time required to display the image data.

imageMapPP is the address of an image buffer allocated by `mdlImage_extReadFileToMap`. When this memory is no longer required, it should be freed by the calling routine by using `mdlImage_freeImage`.

imageSizeP is returned with the size of the image in pixels.

redMapP, *grnMapP*, *bluMapP* are the red, green and blue color palette for an image.

paletteSize is the number of palette entries.

fileNameP is the full path name of the file to be read.

fileType is one of the possible `IMAGEFILE` types contained in `image.h`, or -1 for automatic detection of the type by using the file name extension, or header information in the file itself.

packBytes is a flag indicating whether to compress the byte-map that is read. If *packBytes* is `TRUE`, data will have `IMAGEFORMAT_PackByte`. If *packBytes* is `FALSE`, data will have `IMAGEFORMAT_ByteMap`.

stopFunc is a pointer to a MDL function that is called for every row of the image. If *stopFunc* returns a non-zero value, processing is aborted. A stop function can be used to allow the user to abort what could be a fairly

lengthy process. `mdlImage_checkStop` is provided as a generic stop function. If no stop function is desired, `NULL` can be specified.



This function was implemented in MicroStation 95.

Returns `mdlImage_extReadFileToMap` returns `SUCCESS` if the image file could be successfully read into a memory buffer. `MDLERR_BADFILETYPE` is returned if file does not contain byte-map color data. `MDLERR_INSMEMORY` is returned if insufficient memory is available. `MDLERR_USERABORT` is returned if the user aborts the read by pressing the reset button.

See Also `mdlImage_readFileToMap`, `mdlImage_freeImage`, `mdlImage_checkStop` [`mdl1lib.m1`].

mdlImage_extRGBToMap

```
#include <image.h>
#include <mdlerrs.h>
#include <msimgd1m.fdf>

int mdlImage_extRGBToMap
(
    byte          **imageMapPP, /* <= image map */
    byte          *rgbBufferP,  /* => image buffer (RGB) */
    Point2d       *sizeP,       /* => image size */
    byte          *redMapP,      /* => red palette entries */
    byte          *grnMapP,      /* => green palette entries */
    byte          *bluMapP,      /* => blue palette entries */
    int           paletteSize,   /* => number of palette entries */
    boolean        packBytes,     /* => pack bytes (run length encode) */
    Md1FunctionP   stopFunc      /* => stop function (or NULL) */
);
```

Description The `mdlImage_extRGBToMap` function converts from a RGB image to a byte-mapped image. This function is similar to `mdlImage_RGBToMap`, except that it supports both compressed and uncompressed byte-mapped formats.

This function converts from an RGB image with format `IMAGEFORMAT_RGBSeperate` to a byte-mapped image with format `IMAGEFORMAT_ByteMap` or `IMAGEFORMAT_PackByte`.

imageMapPP is the address of an image buffer allocated by this function. When this memory is no longer required, it should be freed by the calling routine by using `mdlImage_freeImage`.

rgbBufferP is a pointer to the source RGB buffer in `IMAGEFORMAT_RGBSeperate` format.

sizeP is a pointer to the size of the image in pixels.

redMapP, *grnMapP*, *bluMapP* are the red, green and blue color palette for an image.

paletteSize is the number of palette entries.

packBytes is a flag that identifies whether to compress the byte-mapped image. If *packBytes* is `TRUE`, the output format is `IMAGEFORMAT_PackByte`. If *packBytes* is `FALSE`, the output format is `IMAGEFORMAT_ByteMap`.

stopFunc is a pointer to a MDL function that is called for every row of the image. If *stopFunc* returns a non-zero value, processing is aborted. A stop function can be used to allow the user to abort what could be a fairly lengthy process. `mdlImage_checkStop [mdl.lib.mdl]` is provided as a generic stop function. If no stop function is desired, `NULL` can be specified.



This function was implemented in MicroStation 95.

Returns `mdlImage_extRGBToMap` returns `SUCCESS` the conversion is successful. `MDLERR_INSFMEMORY` is returned if insufficient memory is available.

See Also `mdlImage_RGBToMap`, `mdlImage_RGBToPackByte`, `mdlImage_mapToRGB`, `mdlImage_extMapToRGB`, `mdlImage_checkStop [mdl.lib.mdl]`, `mdlImage_freeImage`.

mdlImage_extRGBToScreenMap

```
#include <image.h>
#include <mdlerrs.h>
#include <msimgd1m.fdf>

int mdlImage_extRGBToScreenMap
(
    byte          **imageMapPP, /* <= image map */
    byte          *rgbBufferP,  /* => image buffer (RGB) */
    Point2d       *sizeP,       /* => image size */
    int           screen,        /* => screen number */
    boolean       packBytes,     /* => pack bytes (run length encode) */
    MdIFunctionP   stopFunc      /* => stop function (or NULL) */
);
```

Description The `mdlImage_extRGBToScreenMap` function converts a RGB image to a mapped image that is gamma corrected and dithered to the palette of the graphics device indicated by the screen. The mapped image is then suitable for display by `mdlWindow_rasterDataDraw`. This function is similar to `mdlImage_RGBToScreenMap`, except that it supports both compressed and uncompressed byte-mapped formats.

imageMapPP is the address of an image buffer allocated by this function. When this memory is no longer required, it should be freed by the calling routine by using `mdlImage_freeImage`.

rgbBufferP is a pointer to the source RGB buffer in `IMAGEFORMAT_RGBSeparate` format.

sizeP is a pointer to the size of the image in pixels.

screen is the screen number on which the image data is to be displayed. The screen is 0 for the main screen, 1 for the secondary screen.

packBytes is a flag that identifies whether to compress the byte-mapped image. If *packBytes* is TRUE, the output format is IMAGEFORMAT_PackByte. If *packBytes* is FALSE, the output format is IMAGEFORMAT_ByteMap.

stopFunc is a pointer to a MDL function that is called for every row of the image. If *stopFunc* returns a non-zero value, processing is aborted. A stop function can be used to allow the user to abort what could be a fairly lengthy process. *mdlImage_checkStop* is provided as a generic stop function. If no stop function is desired, NULL can be specified.



This function was implemented in MicroStation 95.

Returns *mdlImage_extRGBToScreenMap* returns SUCCESS the conversion is successful. MDLERR_INSFMEMORY is returned if insufficient memory is available.

See Also *mdlImage_RGBToScreenMap*, *mdlImage_RGBToMap*, *mdlImage_mapToRGB*, *mdlImage_checkStop*, *mdlWindow_rasterDataDraw*, *mdlImage_freeImage*.

mdlImage_freeImage

```
#include <image.h>
#include <msimgd1m.fdf>

void mdlImage_freeImage
(
    void      *imageP,          /* => pointer of image to free */
    Point2d *sizeP,            /* => size of image (needed for some formats) */
    int      imageFormat        /* => memory image format from image.h */
);
```

Description The *mdlImage_freeImage* function frees the memory buffer identified by *imageP*.

Since some memory formats, such as run-length-encoded formats, allocate more than a single block of memory, the only safe way to free all memory associated with an image memory is to use *mdlImage_freeImage*.

In most situations, memory should be allocated by calling *malloc*, and freed using *mdlImage_freeImage*, as follows:

```
imageP=malloc(mdlImage_memorySize(sizeP, imageFormat));
...
mdlImage_freeImage(imageP, sizeP, imageFormat);
```

imageP is a pointer to the image memory to be freed.

sizeP defines the size of the image in pixels. The image size is needed for some formats, such as run-length-encoded formats, since memory is allocated on a row-by-row basis.

imageFormat is an `IMAGEFORMAT` type defined in `image.h`.



This function was implemented in MicroStation 95.

Returns `mdlImage_freeImage` is of type `void`; it returns no value.

See Also `mdlImage_memorySize`.

mdlImage_getMapUsage

```
#include <image.h>
#include <msimgdlm.fdf>

void mdlImage_getMapUsage
(
    byte    mapUsed[256], /* <= palette index used or available */
    byte    *mapP,        /* => byte mapped image */
    Point2d *sizeP,        /* => image size in pixels */
    boolean packBytes      /* => packbytes compression */
);
```

Description The `mdlImage_getMapUsage` function determines which palette entries are used or available in an image. This function is useful to determine an available index to use as an index for a transparent color.

mapUsed[256] is an array of non-zero or `FALSE` values returned. A non-zero *mapUsed* entry indicates that the palette entry is used. `FALSE` indicates that the palette entry is unused.

mapP is a pointer to the byte-mapped image data.

sizeP is a pointer to the size of the image in pixels.

packBytes is a flags. If `TRUE`, then the image is compressed with packbytes compression; the image format is `IMAGEFORMAT_PackByte`. If `FALSE`, then the image is uncompressed; the image format is `IMAGEFORMAT_ByteMap`.



This function was implemented in MicroStation 95.

Returns `mdlImage_getMapUsage` is of type `void`; it returns no value.

See Also `mdlImage_setMapIfRGBMatch`.

mdlImage_greyScaleToBitMap

```
#include <image.h>
#include <mdlerrs.h>
#include <msdefs.h>
#include <msimgd1m.fdf>

int mdlImage_greyScaleToBitMap
(
    byte          *bitMapP, /* <= bitmap image (caller mallocs) */
    byte          *greyP,   /* => source greyscale image */
    Point2d       *sizeP,   /* => size in pixels */
    int           minGrey,  /* => minimum greyscale value */
    int           maxGrey,  /* => maximum greyscale value */
    int           ditherMode, /* => DITHERMODE_Pattern or _ErrorDiffusion */
    MdlFunctionP stopFunc /* => user stop function */
);
```

Description The `mdlImage_greyScaleToBitMap` function converts an uncompressed byte-per-pixel greyscale image to an uncompressed bitmap by dithering the greyscale image.

bitMapP is a pointer to the resulting bitmap image buffer. The caller must allocate memory for this buffer.

greyP is a pointer to the greyscale source image buffer to be converted to monochrome.

sizeP is a pointer to the size of the image in pixels.

minGrey is the minimum greyscale value.

maxGrey is the maximum greyscale value.

ditherMode is either `DITHERMODE_Pattern` to apply a Bayer dither or `DITHERMODE_ErrorDiffusion` to apply a Floyd and Steinberg dithering algorithm. Both defines are in `msdefs.h`.

stopFunc is not used.



This function was implemented in MicroStation 95.

Returns `mdlImage_greyScaleToBitMap` returns `SUCCESS` if the operation is completed successfully. `MDLERR_BADARG` is returned if an invalid `DITHERMODE` is detected.

See Also `mdlImage_byteMapToBitMap`, `mdlImage_packByteToBitMap`, `mdlImage_rgbToBitMap`.

mdlImage_isAVIAvailable

```
#include <image.h>
#include <msimgd1m.fdf>

BoolInt mdlImage_isAVIAvailable
(
    void
);
```

Description The `mdlImage_isAVIAvailable` function is to be used in conjunction with `mdlImage_getExportFormat` to determine whether the `IMAGEFILE_AVI` type is supported on the current MicroStation platform.

Example

```
char formatName[256];
int exportType, i=0, enabled;
while (SUCCESS==mdlImage_getExportFormat(formatName,&exportType,i++))
{
    enabled=(importType!=IMAGEFILE_AVI) ?
        TRUE:mdlImage_isAVIAvailable();
    mdlDialog_optionButtonInsertItem(formatName,NULL,NULL,NULL,
        &exportType, NULL,&enabled, NULL,
        dimP->dialogItemP->rawItemP, -1);
}
```



This function was implemented in MicroStation 95.

Returns `mdlImage_isAVIAvailable` returns `SUCCESS` if AVI files can be created, otherwise it returns `ERROR`.

See Also `mdlImage_getExportFormat`.

mdlImage_memorySize

```
#include <image.h>
#include <mdlerrs.h>
#include <msimgd1m.fdf>

int mdlImage_memorySize
(
    Point2d *sizeP,          /* => image size in pixels */
    int imageFormat          /* => image format from image.h */
);
```

Description The `mdlImage_memorySize` function returns the amount of memory initially required to hold the image in the specified image format. If the image is uncompressed, this is the total size. If it is compressed, this is the amount necessary to hold an array of pointers to each image row. Memory for each row will be

allocated in subsequent operations once the size of the compressed row is determined.

sizeP is a pointer to the size of the image in pixels.

imageFormat is any of the `IMAGEFORMAT` types defined in `image.h`.



This function was implemented in MicroStation 95.

Returns `mdlImage_memorySize` returns the positive number of bytes required to hold the input image. `MDLERR_BADARG` is returned if an invalid image format is detected.

See Also `mdlImage_freeImage`.

mdlImage_mirror

```
#include <image.h>
#include <mdlerrs.h>
#include <msimgd1m.fdf>

int mdlImage_mirror
(
    byte          *inpBufferP,    /* <=> image buffer */
    Point2d       *imageSizeP,    /* => image size */
    int           imageFormat,    /* => IMAGEFORMAT_xxx see image.h */
    boolean       vertical,       /* => IMAGE_FLIP_xxx see image.h */
    MdlFunctionP  stopFunc       /* => stop function (or NULL) */
);
```

Description The `mdlImage_mirror` function mirrors an image by reversing the pixels, either horizontally or vertically.

inpBufferP is a pointer to the image buffer which does not change as a result of the mirror operation. Instead, the buffer is overwritten during the mirror operation.

imageSizeP is the size of the image buffer.

imageFormat is any of the `IMAGEFORMAT` types defined in `image.h`.

vertical is a flag. If *vertical* is `TRUE`, the image is mirrored vertically. If *vertical* is `FALSE`, the image is mirrored horizontally.

stopFunc is a pointer to a MDL function that is called for every row of the image. If *stopFunc* returns a non-zero value, processing is aborted. A stop function can be used to allow the user to abort what could be a fairly lengthy process. `mdlImage_checkStop` is provided as a generic stop function. If no stop function is desired, `NULL` can be specified.



This function was implemented in MicroStation 95.

Returns `mdlImage_mirror` returns `SUCCESS` if the operation is completed successfully. `MDLERR_INSFMEMORY` is returned if insufficient memory is available. `MDLERR_USERABORT` is returned if *stopFunc* returns a non-zero value.

See Also `mdlImage_rotate`, `mdlImage_warp`, `mdlImage_checkStop` [`mdl1lib.mll`].

mdlImage_packByteBuffer

```
#include <image.h>
#include <msimgd1m.fdf>

void mdlImage_packByteBuffer
(
    byte    *outP,          /* <= output buffer*/
    int     *encodeSizeP,   /* <= encoded size (can be NULL) */
    byte    *inP,          /* => input buffer */
    int     rawSize         /* => input buffer size */
);
```

Description The `mdlImage_packByteBuffer` function encodes a sequence of byte-mapped data into packbyte format. It performs the inverse of `mdlImage_unpackByteBuffer`.

outP is a buffer that holds the packbyte encoded data. The caller should allocate this buffer to be a minimum of $(4 * \text{rawSize} / 3 + 2)$ bytes in length.

encodeSizeP is actual encoded size required. This value is normally used to allocate space to hold a row of encoded image data.

inP is a pointer to the source uncompressed byte-mapped data in format `IMAGEFORMAT_ByteMap`.

rawSize is the number of pixels to encode.



This function was implemented in MicroStation 95.

Returns `mdlImage_packByteBuffer` is of type `void`; it returns no value.

See Also `mdlImage_unpackByteBuffer`.

mdlImage_packByteToBitMap

```
#include <image.h>
#include <msdefs.h>
#include <mdlerrs.h>
#include <msimgd1m.fdf>

int mdlImage_packByteToBitMap
(
    byte      *bitMapP,      /* <= bitmap */
    byte      *packByteP,    /* => pack byte image */
    byte      *redMapP,      /* => red palette values */
    byte      *grnMapP,      /* => green palette values */
    byte      *bluMapP,      /* => blue palette values */
    int        paletteSize,   /* => number of palette entries */
    Point2d    *sizeP,        /* => size */
    int        ditherMode,    /* => dither mode */
    MdlFunctionP stopFunc     /* => stop function */
);
```

Description The `mdlImage_packByteToBitMap` function converts a compressed byte-mapped color image to an uncompressed bitmap by dithering the color image.

bitMapP is a pointer to the resulting bitmap image buffer. The caller must allocate memory for this buffer. The format of this buffer is `IMAGEFORMAT_BitMap`.

packByteP is a pointer to the compressed byte-mapped buffer to be converted to monochrome. The format of the buffer is `IMAGEFORMAT_PackByte`.

redMapP, *grnMapP*, *bluMapP* are pointers to the red, green and blue color palette entries.

paletteSize is the number of palette entries.

sizeP is a pointer to the size of the image in pixels.

ditherMode is either `DITHERMODE_Pattern` to apply a Bayer dither or `DITHERMODE_ErrorDiffusion` to apply a Floyd and Steinberg dithering algorithm.

stopFunc is not used.



This function was implemented in MicroStation 95.

Returns `mdlImage_packByteToBitMap` returns `SUCCESS` if the operation is completed successfully. `MDLERR_INSFMEMORY` is returned if insufficient memory is available. `MDLERR_BADARG` is returned if an invalid `DITHERMODE` is detected.

See Also `mdlImage_byteMapToBitMap`, `mdlImage_greyScaleToBitMap`, `mdlImage_rgbToBitMap`.

mdlImage_packByteToGreyScale

```

#include <image.h>
#include <msimgdlm.fdf>

int mdlImage_packByteToGreyScale
(
    byte    *greyP,          /* <= greyscale image (caller mallocs) */
    int     *minGreyP,       /* <= minimum grey value (or NULL) */
    int     *maxGreyP,       /* <= maximum grey value (or NULL) */
    byte    *packByteP,      /* => source packbyte image */
    byte    *redMapP,        /* => red palette entries */
    byte    *grnMapP,        /* => green palette entries */
    byte    *bluMapP,        /* => blue palette entries */
    int     paletteSize,     /* => number of palette entries */
    Point2d *sizeP           /* => image size in pixels */
);

```

Description The `mdlImage_packByteToGreyScale` function converts a compressed byte-mapped color image with format `IMAGEFORMAT_PackByte` to a greyscale image based on the formula:

$$\text{grey} = 0.30 * \text{red} + 0.59 * \text{green} + 0.11 * \text{blue}$$

greyP is a pointer to the resulting greyscale image buffer. The caller must allocate memory for this buffer.

minGreyP is the minimum greyscale value. If the pointer is `NULL`, this value is not returned.

maxGreyP is the maximum greyscale value. If the pointer is `NULL`, this value is not returned.

packByteP is a pointer to the compressed byte-mapped color image data to be converted to greyscale.

redMapP, *grnMapP*, *bluMapP* are pointers to the red, green and blue color palette entries.

paletteSize is the number of palette entries.

sizeP is a pointer to the size of the image in pixels.



This function was implemented in MicroStation 95.

Returns `mdlImage_packByteToGreyScale` returns `SUCCESS`.

See Also `mdlImage_paletteToGreyScale`, `mdlImage_byteMapToGreyScale`, `mdlImage_rgbToGreyScale`.

mdlImage_paletteToGreyScale

```
#include <image.h>
#include <msimgd1m.fdf>

void mdlImage_paletteToGreyScale
(
    byte    *greyPaletteP, /* <= greyscale palette */
    int     *minGreyP,    /* <= minimum grey value (or NULL) */
    int     *maxGreyP,    /* <= maximum grey value (or NULL) */
    byte    *redMapP,     /* => red palette entries */
    byte    *grnMapP,     /* => green palette entries */
    byte    *bluMapP,     /* => blue palette entries */
    int     paletteSize   /* => number of palette entries */
);
```

Description The `mdlImage_paletteToGreyScale` function computes a greyscale palette from a color palette using the formula:

$\text{grey} = 0.30 * \text{red} + 0.59 * \text{green} + 0.11 * \text{blue}$

greyPaletteP is pointer to an array of greyscale values.

minGreyP is the minimum greyscale value. If the pointer is `NULL`, this value is not returned.

maxGreyP is the maximum greyscale value. If the pointer is `NULL`, this value is not returned.

redMapP, *grnMapP*, *bluMapP* are pointers to the red, green and blue color palette entries.

paletteSize is the number of palette entries.



This function was implemented in MicroStation 95.

Returns `mdlImage_paletteToGreyScale` is of type `void`; it returns no value.

See Also `mdlImage_byteMapToGreyScale`, `mdlImage_packByteToGreyScale`, `mdlImage_rgbToGreyScale`.

mdlImage_readFileToBitMap

```
#include <image.h>
#include <mdlerrs.h>
#include <msimgd1m.fdf>

int mdlImage_readFileToBitMap
(
    byte    **imageBitMap, /* <= image bit map */
    Point2d *imageSize,    /* <= image size */
    ...
);
```

```

char      *fileName,      /* => input file name */
int       fileType,       /* => input file type */
boolean   runLengthEncode, /* => run length encode data */
MdIFunctionP stopFunc     /* => stop function (or NULL) */
);

```

Description The `mdlImage_readFileToBitMap` function reads a bitmapped image file into an image buffer and optionally run length encodes the bitmap.

Generally, `mdlImage_readFileInfo` is first used to determine the file type, then `mdlImage_readFileToBitMap` is called if the image color mode is `COLORMODE_MONOCHROME`.

imageBitMap is the address of a pointer to a memory buffer that is allocated by `mdlImage_readFileToBitMap` to hold the bitmap. When the memory buffer is no longer required it should be freed by the caller by passing the pointer to the memory buffer to `mdlImage_freeImage`, together with the size and format of the image data. The pointer should not be freed directly.

imageSize is returned with the size of the image in pixels.

fileName is the full path name of the file to be read.

fileType is one of the possible `IMAGEFILE` types contained in `image.h`, or -1 for automatic detection of file type by using the file name extension, or header information in the file itself.

runLengthEncode indicates whether or not compression is to occur. Compression will generally save lots of memory, but will require slightly large amount of processing time. The following possible data formats are returned depending on the value of *runLengthEncode*:

<i>runLengthEncode</i> Value	Image Format Type	Description
TRUE	<code>IMAGEFORMAT_RLEBitMap</code>	Run length encoded (generally most compressed)
FALSE	<code>IMAGEFORMAT_BitMap</code>	Bits per pixel (generally least compressed)

stopFunc is a pointer to a MDL function that is called for every row of the image. If *stopFunc* returns a non-zero value, processing is aborted. A stop function can be used to allow the user to abort what could be a fairly lengthy process. `mdlImage_checkStop [mdl1ib.m1]` is provided as a generic stop function. If no stop function is desired, `NULL` can be specified.



This function was implemented in MicroStation 95.

Returns `mdlImage_readFileToBitMap` returns `SUCCESS` if the image file could be successfully read into a memory buffer. `MDLERR_BADFILETYPE` is returned if file does not contain monochrome bitmap data. `MDLERR_INSMEMORY` is returned if insufficient memory is available. `MDLERR_USERABORT` is returned if the user aborts the read by pressing the reset button.

See Also `mdlImage_readFileToMap`, `mdlImage_readFileToRGB`, `mdlImage_readFileInfo`, `mdlImage_checkStop` [`mdl.lib.mdl`], `mdlImage_freeImage`.

mdlImage_remapToScreenPalette

```
#include <image.h>
#include <msimgd1m.fdf>

void mdlImage_remapToScreenPalette
(
    byte          *outP,           /* <= remapped byte map (may be inP) */
    byte          *inP,           /* => input map */
    Point2d       *sizeP,         /* => image size */
    byte          *redMapP,        /* => input red map */
    byte          *grnMapP,        /* => input green map */
    byte          *bluMapP,        /* => input blue map */
    int           paletteSize,     /* => input palette size */
    int           screenNumber     /* => physical screen number */
);
```

Description The `mdlImage_remapToScreenPalette` function remaps a palettized image to the nearest colors in a screen's palette. This function matches the user-supplied palette entries to the screen's palette, then converts each palette index in the source image data to the corresponding index in the destination image data.

outP is a buffer that receives remapped image data. Both *inP* and *outP* can point to the same buffer.

inP is a buffer that contains the source image data which is remapped to the screen palette.

sizeP defines the size in bytes of the source and destination buffers. Both *inP* and *outP* must be buffers which are a minimum of $sizeP \rightarrow x * sizeP \rightarrow y$ bytes in length.

redMapP, *grnMapP*, *bluMapP* are the red, green and blue color palettes for the source image.

paletteSize is the number of palette entries.

screenNumber is the screen number on which the image data is to be displayed. The *screenNumber* is 0 for the main screen, 1 for the secondary screen.



This function was implemented in MicroStation 95.

Returns `mdlImage_remapToScreenPalette` is of type `void`; it returns no value.

See Also `mdlImage_computeScreenMap`, `mdlImage_getScreenPalette`.

mdlImage_rgbToBitMap

```
#include <image.h>
#include <msdefs.h>
#include <mdlerrs.h>
#include <msimgd1m.fdf>

int mdlImage_rgbToBitMap
(
    byte          *bitMapP,      /* <= bitmap */
    byte          *rgbP,         /* => rgb image */
    Point2d       *sizeP,       /* => size */
    int           ditherMode,    /* => dither mode */
    MdlFunctionP  stopFunc      /* => stop function */
);
```

Description The `mdlImage_rgbToBitMap` function converts a RGB image to an uncompressed bitmap by dithering the color image.

bitMapP is a pointer to the resulting bitmap image buffer. The caller must allocate memory for this buffer. The format of this buffer is

`IMAGEFORMAT_BitMap`.

rgbP is a pointer to the RGB buffer to be converted to monochrome. The format of this buffer is `IMAGEFORMAT_RGBSeperate`.

sizeP is a pointer to the size of the image in pixels.

ditherMode is either `DITHERMODE_Pattern` to apply a Bayer dither or `DITHERMODE_ErrorDiffusion` to apply a Floyd and Steinberg dithering algorithm.

stopFunc is not used.



This function was implemented in MicroStation 95.

Returns `mdlImage_rgbToBitMap` returns `SUCCESS` if the operation is completed successfully. `MDLERR_INSFMEMORY` is returned if insufficient memory is available. `MDLERR_BADARG` is returned if an invalid `DITHERMODE` is detected.

See Also `mdlImage_byteMapToBitMap`, `mdlImage_packByteToBitMap`, `mdlImage_greyScaleToBitMap`.

mdlImage_rgbToGreyScale

```
#include <image.h>
#include <msimgd1m.fdf>

int mdlImage_rgbToGreyScale
(
    byte    *greyP,          /* <= greyscale image (caller mallocs) */
    int     *minGreyP,       /* <= minimum grey value (or NULL) */
    int     *maxGreyP,       /* <= maximum grey value (or NULL) */
    byte    *rgbP,          /* => source RGB data */
    Point2d *sizeP           /* => image size in pixels */
);
```

Description The mdlImage_rgbToGreyScale function converts a RGB image to a byte-per-pixel greyscale image based on the formula:

$\text{grey} = 0.30 * \text{red} + 0.59 * \text{green} + 0.11 * \text{blue}$

greyP is a pointer to the resulting greyscale image buffer. The caller must allocate memory for this buffer.

minGreyP is the minimum greyscale value. If the pointer is NULL, this value is not returned.

maxGreyP is the maximum greyscale value. If the pointer is NULL, this value is not returned.

rgbP is a pointer to the RGB color image data to be converted to greyscale. The RGB data is in IMAGEFORMAT_RGBSeperate format.

sizeP is a pointer to the size of the image in pixels.



This function was implemented in MicroStation 95.

Returns mdlImage_rgbToGreyScale returns SUCCESS.

See Also mdlImage_paletteToGreyScale, mdlImage_byteMapToGreyScale, mdlImage_packByteToGreyScale.

mdlImage_RGBToPackByte

```
#include <image.h>
#include <mdlerrs.h>
#include <msimgd1m.fdf>

int mdlImage_RGBToPackByte
(
    byte    **imageMapPP,    /* <= image map */
    byte    *rgbBufferP,     /* => image buffer (RGB) */
    ...
);
```

```

Point2d      *sizeP,          /* => image size */
byte         *redMapP,        /* => red palette entries */
byte         *grnMapP,        /* => green palette entries */
byte         *bluMapP,        /* => blue palette entries */
int          paletteSize,     /* => number of palette entries */
MDlFunctionP stopFunc         /* => stop function (or NULL) */
);

```

Description The `mdlImage_RGBToPackByte` function converts a RGB image to a packbyte compressed byte-mapped image. This function is similar to `mdlImage_RGBToMap`, which converts from RGB to an uncompressed byte-mapped image.

imageMapPP is the address of an image buffer allocated by `mdlImage_RGBToPackByte`. When this memory is no longer required, it should be freed by the calling routine by using `mdlImage_freeImage`.

rgbBufferP is a pointer to the source RGB buffer in `IMAGEFORMAT_RGBSeperate` format.

sizeP is a pointer to the size of the image in pixels.

redMapP, *grnMapP*, *bluMapP* are the red, green and blue color palette for an image.

paletteSize is the number of palette entries.

stopFunc is a pointer to a MDL function that is called for every row of the image. If *stopFunc* returns a non-zero value, processing is aborted. A stop function can be used to allow the user to abort what could be a fairly lengthy process. `mdlImage_checkStop [mdl11b.m1]` is provided as a generic stop function. If no stop function is desired, `NULL` can be specified.



This function was implemented in MicroStation 95.

Returns `mdlImage_RGBToPackByte` returns `SUCCESS` the conversion is successful. `MDLERR_INSFMEMORY` is returned if insufficient memory is available.

See Also `mdlImage_RGBToMap`, `mdlImage_mapToRGB`, `mdlImage_extMapToRGB`, `mdlImage_freeImage`, `mdlImage_checkStop [mdl11b.m1]`.

mdlImage_RGBToPackByteWithGamma

```
#include <image.h>
#include <msimgd1m.fdf>

int mdlImage_RGBToPackByteWithGamma
(
    byte          **imageMapPP, /* <= image map to be allocated */
    byte          *rgbBufferP,  /* => image buffer (RGB) */
    Point2d       *sizeP,      /* => image size */
    byte          *redMapP,     /* => red palette entries */
    byte          *grnMapP,     /* => green palette entries */
    byte          *bluMapP,     /* => blue palette entries */
    int           paletteSize,  /* => number of palette entries */
    double        gamma,       /* => gamma correction value */
    MdIFunctionP  stopFunc     /* => stop function (or NULL) */
);
```

Description The mdlImage_RGBToPackByteWithGamma function converts a RGB image to a packbyte compressed byte-mapped image and applies the supplied gamma value.

imageMapPP is the address of an image buffer allocated by mdlImage_RGBToPackByte. When this memory is no longer required, it should be freed by the calling routine by using mdlImage_freeImage.

rgbBufferP is a pointer to the source RGB buffer in IMAGEFORMAT_RGBSeperate format.

sizeP is a pointer to the size of the image in pixels.

redMapP, *grnMapP*, *bluMapP* are the red, green and blue color palette for an image.

paletteSize is the number of palette entries.

gamma is a gamma correction value that is applied to the RGB image prior to conversion.

stopFunc is a pointer to a MDL function that is called for every row of the image. If *stopFunc* returns a non-zero value, processing is aborted. A stop function can be used to allow the user to abort what could be a fairly lengthy process. mdlImage_checkStop [mdl11ib.m1] is provided as a generic stop function. If no stop function is desired, NULL can be specified.



This function was implemented in MicroStation 95.

Returns mdlImage_RGBToPackByteWithGamma returns SUCCESS if image is successfully converted. MDLERR_INSFMEMORY is returned if insufficient memory is available. If *stopFunc* returns a non-zero value, this value is returned.

See Also mdlImage_applyGamma, mdlImage_checkStop [mdl11ib.m1], mdlImage_freeImage.

mdlImage_rotate

```

#include <image.h>
#include <mdlerrs.h>
#include <msimgdlm.fdf>

int mdlImage_rotate
(
    byte          **outBufferPP, /* <= image buffer (rotated) */
    Point2d       *outSizeP,    /* <= image size (after rotate) */
    byte          *inpBufferP,  /* => image buffer (original) */
    Point2d       *imageSizeP,  /* => image size (original) */
    int           imageFormat,   /* => IMAGEFORMAT_ (see image.h) */
    int           rotation,      /* => 90, 180, 270 */
    MdlFunctionP  stopFunc      /* => stop function */
);

```

Description The `mdlImage_rotate` function rotates the image about the origin in a counter clockwise direction by the number of degrees specified in `rotation`. Currently, only 90, 180 and 270 degree rotations are supported.

outBufferPP is the address of a pointer to the image buffer allocated by `mdlImage_rotate`. If *outBufferPP* points to *inpBufferP* and the rotation is successful, *inpBufferP* is freed, and the pointer returned in *outBufferPP* replaces the image buffer.

outSizeP is the size of the output image.

inpBufferP is a pointer to the input image buffer.

imageSizeP is the size of the input image buffer.

imageFormat is any of the `IMAGEFORMAT` types defined in `image.h`.

rotation is 90, 180 or 270 degrees.

stopFunc is a pointer to a MDL function that is called for every row of the image. If *stopFunc* returns a non-zero value, processing is aborted. A stop function can be used to allow the user to abort what could be a fairly lengthy process. `mdlImage_checkStop [mdl1ib.m1]` is provided as a generic stop function. If no stop function is desired, `NULL` can be specified.



This function was implemented in MicroStation 95.

Returns `mdlImage_rotate` returns `SUCCESS` if the operation is completed successfully. `MDLERR_INSFMEMORY` is returned if insufficient memory is available. `MDLERR_BADARG` is returned if an invalid image format is detected. `MDLERR_USERABORT` is returned if *stopFunc* returns a non-zero value.

See Also `mdlImage_mirror`, `mdlImage_warp`, `mdlImage_checkStop`.

mdlImage_runLengthDecodeBitMapRow

```
#include <image.h>
#include <msimgd1m.fdf>

void mdlImage_runLengthDecodeBitMapRow
(
    byte    *outP,      /* <= output bitmap */
    UShort  *inP,       /* => input RLE bitmap */
    int     nPixels     /* => pixels to decode */
);
```

Description The mdlImage_runLengthDecodeBitMapRow function decodes a bitmap row from an array of UShort integers into a row in format IMAGEFORMAT_BitMap.

outP is a pointer to the bitmap buffer. It should be BITMAP_ROWBYTES (*nPixels*) in length.

inP is a pointer to the first of an array of UShort integers that represent the compressed bitmap. The first UShort is the number of 0-bits, next the number of 1-bits, and so on.

nPixels is the number of pixels to decode. The number of pixels can be less than the total in the row.



This function was implemented in MicroStation 95.

Returns mdlImage_runLengthDecodeBitMapRow is of type void; it returns no value.

See Also mdlImage_stretchRLEToBitMap, mdlImage_runLengthEncodeBitMapRow.

mdlImage_runLengthEncodeBitMap

```
#include <image.h>
#include <mdlerrs.h>
#include <msimgd1m.fdf>

int mdlImage_runLengthEncodeBitMap
(
    UShort    ***rlePPP, /* <= run length encoded bitmap buffer */
    byte      *bitmapP,  /* => source bitmap */
    Point2d   *sizeP     /* => image size in pixels */
);
```

Description The mdlImage_runLengthEncodeBitMap function converts a bitmap in format IMAGEFORMAT_BitMap to its run-length-encoded format IMAGEFORMAT_RLEBitMap.

rlePPP is the address of a pointer to a RLE bitmap buffer allocated by mdlImage_runLengthEncodeBitMap.

bitmapP is the source bitmap in IMAGEFORMAT_BitMap.

sizeP defines the size of the image in pixels.



This function was implemented in MicroStation 95.

Returns `mdlImage_runLengthEncodeBitMap` returns `SUCCESS` if the operation is completed successfully. `MDLERR_INSMEMORY` is returned if insufficient memory is available.

See Also `mdlImage_runLengthDecodeBitMapRow`.

mdlImage_runLengthEncodeBitMapRow

```
#include <image.h>
#include <msimgdlm.fdf>

void mdlImage_runLengthEncodeBitMapRow
(
  UShort *outP,           /* <= output encoded bitmap */
  int *compressedSizeP,   /* <= output size */
  byte *inP,              /* => input bitmap */
  int nPixels              /* => pixels to encode */
);
```

Description The `mdlImage_runLengthEncodeBitMapRow` function encodes a bitmap row in format `IMAGEFORMAT_BitMap` into any array of `UShort` integers. Encoding starts with the number of continuous 0-bits, then alternates from 1-bits to 0-bits and back.

The worst case compression is a sequence of alternating 0 and 1 bits, in which case an entire `Ushort` (16-bits) will be required to hold each bit of information. Since this case is possible, although highly improbable, the user is responsible for allocating an output buffer `sizeP->x * sizeof (UShort)` bytes. For most bitmaps, the compression sizes will be considerably less than this.

outP is a pointer to a buffer that receives the compressed bitmap.

compressedSizeP is a pointer to the length of the output buffer in bytes. Normally, this length is used to allocate memory to hold the compressed image.

inP is a pointer to the bitmap image buffer that is compressed.

nPixels is the number of pixels to encode.



This function was implemented in MicroStation 95.

Returns `mdlImage_runLengthEncodeBitMapRow` is of type `void`; it returns no value.

See Also `mdlImage_runLengthEncodeBitMap`, `mdlImage_runLengthDecodeBitMapRow`.

mdlImage_setMapIfRGBMatch

```
#include <image.h>
#include <msimgdlm.fdf>

void mdlImage_setMapIfRGBMatch
(
    byte      *mapP,      /* <=> mapped image change on RGB match */
    int       value,      /* => index to set in mapped image */
    byte      *rgbP,      /* => RGB image from which match is found */
    Point2d   *sizeP,     /* => size in pixels of both images */
    RGBColorDef *matchP,  /* => match RGB value */
    boolean    packBytes /* => TRUE == image is packbyte compressed */
);
```

Description The `mdlImage_setMapIfRGBMatch` function modifies a corresponding byte mapped image every time pixels in a corresponding RGB image match a specified value.

This function is useful where a RGB value identifies a transparent color, but the image is displayed on a palettized display. It allow all pixels that match a given RGB value in the RGB representation of the image to change the corresponding pixels in the byte mapped representation of the image.

mapP is a pointer to the mapped image.

value is a palette index that will be placed in the mapped image whenever a match occurs between the RGB image and *matchP*.

rgbP is a pointer to the RGB image in `IMAGEFORMAT_RGBSeperate` format.

sizeP is a pointer to the size of the image in pixels.

matchP is the RGB value to be matched in the RGB image.

packBytes is a flag. If `TRUE`, the mapped image is in `IMAGEFORMAT_PackByte` compressed with *packbytes* compression. If `FALSE`, the mapped image is in `IMAGEFORMAT_ByteMap` uncompressed format.



This function was implemented in MicroStation 95.

Returns `mdlImage_setMapIfRGBMatch` is of type `void`; it returns no value.

See Also `mdlImage_getMapUsage`.

mdlImage_setMapPolygon

```
#include <image.h>
#include <mdlerrs.h>
#include <msimgdlm.fdf>

int mdlImage_setMapPolygon
(
    byte      *dataP,      /* <=> image data */
```

```

boolean      *dataModifiedP, /* <= TRUE == changed occurred (or NULL) */
Point2d      *sizeP,        /* => image size in pixels */
int          imageFormat,   /* => image format */
Dpoint2d     *pointP,       /* => closed array of points */
int          nPoints,       /* => number of points in array */
boolean      outside,       /* => TRUE == fill outside with value */
int          value,         /* => fill value */
MdlFunctionP stopFunc       /* => user stop function */
);

```

Description The `mdlImage_setMapPolygon` function masks a polygonal area within an image. It fills the interior or exterior of a polygon with a user specified value. `mdlImage_setMapPolygon` works with byte-mapped, and bitmapped images, in both compressed and uncompressed formats.

dataP is a pointer to the image buffer.

dataModifiedP is pointer to a boolean. This value will be set to `TRUE` if the image is modified because the mask polygon falls somewhere within the image.

sizeP defines the size of the image in pixels.

imageFormat is an `IMAGEFORMAT` type identified in `image.h`. RGB formats are not allowed.

pointP is a pointer to the first element of an array of *nPoints* points. This array defines a closed masking polygon in pixel coordinates.

nPoints is the number of elements in the array starting at *pointP*.

outside is a flag that identifies whether values inside or outside of the polygon are filled. If *outside* is `TRUE`, values outside the polygon are filled with value. If *outside* is `FALSE`, values inside the polygon are filled with value.

value is the fill value.

stopFunc is a pointer to a MDL function that is called for every row of the image. If *stopFunc* returns a non-zero value, processing is aborted. A stop function can be used to allow the user to abort what could be a fairly lengthy process. `mdlImage_checkStop [mdl1ib.m1]` is provided as a generic stop function. If no stop function is desired, `NULL` can be specified.



This function was implemented in MicroStation 95.

Returns `mdlImage_setMapPolygon` returns `SUCCESS` if the operation is completed successfully. `MDLERR_INSFMEMORY` is returned if insufficient memory is available. `MDLERR_BADARG` is returned if an invalid image format is detected. `MDLERR_USERABORT` is returned if *stopFunc* returns a non-zero value.

See Also `mdlImage_setRGBPolygon`, `mdlImage_checkStop [mdl1ib.m1]`.

mdlImage_setRGBPolygon

```
#include <image.h>
#include <mdlerrs.h>
#include <msimgd1m.fdf>

int mdlImage_setRGBPolygon
(
    byte          *dataP,          /* <=> image data */
    boolean       *dataModifiedP, /* <= TRUE == changed occurred */
    Point2d       *sizeP,          /* => image size in pixels */
    Dpoint2d      *pointP,         /* => closed array of points */
    int           nPoints,         /* => number of points in array */
    boolean       outside,         /* => TRUE == fill outside with value */
    RGBColorDef   *colorP,         /* => fill value */
    MdlFunctionP  stopFunc        /* => user stop function */
);
```

Description The `mdlImage_setRGBPolygon` function masks a polygonal area within an image. It fills the interior or exterior of a polygon with a user specified color. `mdlImage_setRGBPolygon` works with images in `IMAGEFORMAT_RGBSeperate` format.

dataP is a pointer to the image buffer.

dataModifiedP is pointer to a boolean. This value will be set to `TRUE` if the image is modified because the mask polygon falls somewhere within the image.

sizeP defines the size of the image in pixels.

pointP is a pointer to the first element of an array of *nPoints* points. This array defines a closed masking polygon in pixel coordinates.

nPoints is the number of elements in the array starting at *pointP*.

outside is a flag that identifies whether values inside or outside of the polygon are filled. If *outside* is `TRUE`, values outside the polygon are filled with value. If *outside* is `FALSE`, values inside the polygon are filled with value.

colorP is a pointer to a `RGBColorRef` structure that identifies the red, green and blue values that will be used as the fill value.

stopFunc is a pointer to a MDL function that is called for every row of the image. If *stopFunc* returns a non-zero value, processing is aborted. A stop function can be used to allow the user to abort what could be a fairly lengthy process. `mdlImage_checkStop [mdl11b.m1]` is provided as a generic stop function. If no stop function is desired, `NULL` can be specified.



This function was implemented in MicroStation 95.

Returns `mdlImage_setRGBPolygon` returns `SUCCESS` if the operation is completed successfully. `MDLERR_INSFMEMORY` is returned if insufficient memory is available. `MDLERR_USERABORT` is returned if *stopFunc* returns a non-zero value.

See Also `mdlImage_setMapPolygon`, `mdlImage_checkStop` [`mdl1lib.m1`].

mdlImage_stretchRLEToBitMap

```
#include <image.h>
#include <mdlerrs.h>
#include <msimgdlm.fdf>

int mdlImage_stretchRLEToBitMap
(
    byte    **imagePP, /* <=> pointer to RLE bitmap (which is replaced) */
    Point2d *sizeP     /* => size of bitmap */
);
```

Description The `mdlImage_stretchRLEToBitMap` function replaces a RLE bitmap representation of an image with its corresponding bitmap representation. It converts from `IMAGEFORMAT_RLEBitMap` to `IMAGEFORMAT_BitMap`.

imagePP is a pointer to the image buffer. On entry it points to an array of pointers to `UShort` integers (`Ushort**`), the RLE representation. On exit it points to a bitmap buffer.

sizeP is a pointer to the size of the image in pixels.



This function was implemented in MicroStation 95.

Returns `mdlImage_stretchRLEToBitMap` returns `SUCCESS` the conversion is successful. `MDLERR_INSFMEMORY` is returned if insufficient memory is available.

See Also `mdlImage_runLengthEncodeBitMap`, `mdlImage_runLengthDecodeBitMapRow`.

mdlImage_subByteMapFromBitMap

```
#include <image.h>
#include <mdlerrs.h>
#include <msimgd1m.fdf>

int mdlImage_subByteMapFromBitMap
(
    byte    *outMapP,      /* <= output byte map (allocated by caller) */
    Point2d *outSizeP,     /* => output image size */
    byte    *inBitMapP,    /* => input bit map */
    Point2d *inSizeP,      /* => input image size */
    Drectangle *rectP,     /* => sub rectangle to extract (or NULL)*/
    int     foreground,    /* => foreground index */
    int     background,    /* => background index */
    boolean rle            /* => bitmap is run length encoded */
);
```

Description The mdlImage_subByteMapFromBitMap function converts a bitmapped image to a byte mapped image and stretches or decimates it to fit a user-defined output image size.

outMapP is a pointer to a buffer allocated by the caller. The format of this buffer is IMAGEFORMAT_ByteMap and the size of this buffer is:

mdlImage_memorySize(*outSizeP*, IMAGEFORMAT_ByteMap)

outSizeP defines the output image size in pixels.

inBitMapP is a pointer to the source image buffer in IMAGEFORMAT_BitMap or IMAGEFORMAT_RLEBitMap format, depending on the value of the *rle* flag.

inSizeP defines the size of the source image in pixels.

rectP defines a rectangular subimage in pixels. The rectangle defined by *rectP* should be a subset of the rectangle with origin at (0,0) and corner at (*inSizeP*->x - 1, *inSizeP*->y - 1). If *rectP* is NULL, the subimage rectangle is the entire image.

foreground is the color index of the foreground color. All 1-bits in the source image will be mapped to this color index.

background is the color index of the background color. All 0-bits in the source image will be mapped to this color index.

rle is a flag that identifies if the source bitmap is compressed. If *rle* is TRUE, a compressed bitmap in format IMAGEFORMAT_RLEBitMap is expected; if *rle* is FALSE, an uncompressed bitmap in format IMAGEFORMAT_BitMap is expected.



This function was implemented in MicroStation 95.

Returns mdlImage_subByteMapFromBitMap returns SUCCESS if the operation is completed successfully. MDLERR_INSFMEMORY is returned if insufficient memory is available.

See Also `mdlImage_subByteMapFromRLEBitMap`, `mdlImage_subByteMapFromPackByte`, `mdlImage_memorySize`.

mdlImage_subByteMapFromPackByte

```
#include <image.h>
#include <msimgdlm.fdf>
#include <mdlerrs.h>

int mdlImage_subByteMapFromPackByte
(
    byte          *outMapP, /* <= output byte map (allocated by caller) */
    Point2d       *outSizeP, /* => output image size */
    byte          **inRowPP, /* => input pack byte array */
    Point2d       *inSizeP, /* => input image size */
    Drectangle     *rectP, /* => sub rectangle to extract (or NULL) */
    byte          *mapP /* => redirection map */
);
```

Description The `mdlImage_subByteMapFromPackByte` function converts a packbyte image to a byte mapped image and stretches or decimates it to fit a user-defined output image size.

outMapP is a pointer to a buffer allocated by the caller. The format of this buffer is `IMAGEFORMAT_ByteMap` and the size of this buffer is:
`mdlImage_memorySize(outSizeP, IMAGEFORMAT_ByteMap)`

outSizeP defines the output image size in pixels.

inRowPP is a pointer to the source image buffer in `IMAGEFORMAT_PackByte` format.

inSizeP defines the size of the source image in pixels.

rectP defines a rectangular subimage in pixels. The rectangle defined by *rectP* should be a subset of the rectangle with origin at (0,0) and corner at (*inSizeP*->x - 1, *inSizeP*->y - 1).). If *rectP* is `NULL`, the subimage rectangle is the entire image.

mapP is an optional array of 256 bytes used to remap the palette indices during the conversion. If *mapP* is `NULL`, this parameter is ignored and no palette mapping occurs.



This function was implemented in MicroStation 95.

Returns `mdlImage_subByteMapFromPackByte` returns `SUCCESS` if the operation is completed successfully. `MDLERR_INSFMEMORY` is returned if insufficient memory is available.

See Also `mdlImage_subByteMapFromBitMap`, `mdlImage_subByteMapFromRLEBitMap`

mdlImage_subByteMapFromRLEBitMap

```
#include <image.h>
#include <mdlerrs.h>
#include <msimgd1m.fdf>

int mdlImage_subByteMapFromRLEBitMap
(
    byte      *outMapP,      /* <= output byte map (allocated by caller) */
    Point2d *outSizeP,      /* => output image size */
    UShort **inBitMapPP,    /* => input bit map */
    Point2d *inSizeP,       /* => input image size */
    Drectangle *rectP,      /* => sub rectangle to extract (or NULL)*/
    int      foreground,    /* => foreground index */
    int      background     /* => background index */
);
```

Description The mdlImage_subByteMapFromRLEBitMap function converts a RLE bitmapped image to a byte mapped image and stretches or decimates it to fit a user-defined output image size.

outMapP is a pointer to a buffer allocated by the caller. The format of this buffer is IMAGEFORMAT_ByteMap and the size of this buffer is:

mdlImage_memorySize(*outSizeP*, IMAGEFORMAT_ByteMap)

outSizeP defines the output image size in pixels.

inBitMapPP is a pointer to the source image buffer in IMAGEFORMAT_RLEBitMap.

inSizeP defines the size of the source image in pixels.

rectP defines a rectangular subimage in pixels. The rectangle defined by *rectP* should be a subset of the rectangle with origin at (0,0) and corner at (*inSizeP*->x - 1, *inSizeP*->y - 1). If *rectP* is NULL, the subimage rectangle is the entire image.

foreground is the color index of the foreground color. All 1-bits in the source image will be mapped to this color index.

background is the color index of the background color. All 0-bits in the source image will be mapped to this color index.



This function was implemented in MicroStation 95.

Returns mdlImage_subByteMapFromRLEBitMap returns SUCCESS if the operation is completed successfully. MDLERR_INSFMEMORY is returned if insufficient memory is available.

See Also mdlImage_subByteMapFromBitMap, mdlImage_subByteMapFromPackByte, mdlImage_memorySize.

mdlImage_tintImage

```
#include <image.h>
#include <basetype.h>
#include <msimgd1m.fdf>

void mdlImage_tintImage
(
    byte          *imageP,          /* <= tinted RGBSeperate image */
    Point2d       *imageSizeP,      /* => image size in pixels */
    RGBColorDef   *tintRGBP,        /* => RGB value of tint color */
    MdlFunctionP  stopFunc          /* => user stop function */
);
```

Description The `mdlImage_tintImage` function tints a RGB image in format `IMAGEFORMAT_RGBSeperate` by adjusting values as follows:

```
*imageP=((UInt) *imageP * tintRGBP->red)/255u;
imageP+=imageSizeP->x;
*imageP=((UInt) *imageP * tintRGBP->green)/255u;
imageP+=imageSizeP->x;
*imageP=((UInt) *imageP * tintRGBP->blue)/255u;
```

imageP is a pointer to an image buffer in form `IMAGEFORMAT_RGBSeperate`.

imageSizeP defines the size of the image in pixels.

tintRGBP defines the RGB value that will be used to tint the image.

stopFunc is a pointer to an MDL function that is called for every row of the image. If *stopFunc* returns a non-zero value, processing is aborted. A stop function can be used to allow the user to abort what could be a fairly lengthy process. `mdlImage_checkStop` [`mdl1lib.m1`] is provided as a generic stop function. If no stop function is desired, `NULL` can be specified.



This function was implemented in MicroStation 95.

Returns `mdlImage_tintImage` is of type `void`; it returns no value.

See Also `mdlImage_tintPalette`, `mdlImage_checkStop` [`mdl1lib.m1`].

mdlImage_tintPalette

```
#include <image.h>
#include <basetype.h>
#include <msimgd1m.fdf>

void mdlImage_tintPalette
(
    byte      *redP,          /* <=> red palette entry */
    byte      *grnP,          /* <=> green palette entry */
    byte      *bluP,          /* <=> blue palette entry */
    int        paletteSize,    /* => palette size */
    RGBColorDef *tintRGBP      /* => RGB value of tint color */
);
```

Description The mdlImage_tintPalette function tints each entry of an image palette by adjusting values as follows:

```
*redP=((UInt) * redP * tintRGBP->red)/255;
*grnP=((UInt) * grnP * tintRGBP->green)/255;
*bluP=((UInt) * bluP * tintRGBP->blue)/255;
```

redP, *grnP*, *bluP* are the red, green and blue color palette for the source image.

paletteSize is the number of palette entries.

tintRGBP defines the RGB value that will be used to tint the palette.



This function was implemented in MicroStation 95.

Returns mdlImage_tintPalette is of type void; it returns no value.

See Also mdlImage_tintImage.

mdlImage_typeFromFile

```
#include <image.h>
#include <mdlerrs.h>
#include <msimgd1m.fdf>

int mdlImage_typeFromFile
(
    char      *fileNameP      /* => full path name of file */
);
```

Description The mdlImage_typeFromFile function attempts to determine the type of file from header information within the file itself. If it can successfully validate the file type, it returns one of the supported IMAGEFILE formats defined in image.h.

Currently, this function recognizes the following types of files: FLI, GIF, IMG, IMG (24-bit), all Intergraph files and TIFF files. These files have

magic number and special check bytes that easily identify the type of image file.

The `mdlImage_typeFromFile` function is the preferred way to determine the format of a source (import) image file. It should be checked before calling `mdlImage_typeFromExtension`, since image types returned by this function are more likely to be correct.

fileNameP is full path name of the image file.



This function was implemented in MicroStation 95.

Returns `mdlImage_typeFromFile` returns an `IMAGEFILE` type identified in `image.h`. `MDLERR_NOMATCH` is returned if no valid image format can be determined.

See Also `mdlImage_typeFromExtension`, `mdlImage_getImportFormat`.

mdlImage_unpackByteBuffer

```
#include <image.h>
#include <msimgdlm.fdf>

void mdlImage_packByteBuffer
(
    byte    *outP,          /* <= output buffer */
    byte    *inP,           /* => input buffer */
    int     numDecode       /* => number of bytes to decode */
);
```

Description The `mdlImage_unpackByteBuffer` function unpacks a buffer in packbyte format to a sequence of bytes. It performs the inverse of `mdlImage_packByteBuffer`.

outP is a buffer that holds the decoded byte-mapped data. The caller should allocate *numDecode* bytes for this buffer.

inP is a pointer to the source compressed packbyte byte-mapped data in format `IMAGEFORMAT_PackByte`

numDecode is the number of pixels to decode.



This function was implemented in MicroStation 95.

Returns `mdlImage_unpackByteBuffer` is of type `void`; it returns no value.

See Also `mdlImage_packByteBuffer`.

mdlImage_updateIngrAttach

```
#include <image.h>
#include <mdlerrs.h>
#include <msimgd1m.fdf>

int mdlImage_updateIngrAttach
(
    Dpoint3d    *originP,      /* => origin of Ingr. file */
    Dpoint2d    *extentP,      /* => extent of Ingr. file */
    char        *fileNameP     /* => Ingr. filename to modify */
);
```

Description The mdlImage_updateIngrAttach function updates the origin and extent of an Intergraph file by reading, modifying and rewriting the image header.

originP is the origin of the image in world coordinates.

extentP is the extent of the image in world coordinates.

fileNameP is the full path name of an Intergraph file.



This function was implemented in MicroStation 95.

Returns mdlImage_updateIngrAttach returns SUCCESS if it could successfully modify the image header. MDLERR_CANNOTOPENFILE is returned if the file could not be opened. MDLERR_BADRASTERFORMAT is returned if the file is not an Intergraph file.

See Also mdlImage_extractIngrAttach.

mdlImage_warp

```
#include <image.h>
#include <mdlerrs.h>
#include <msimgd1m.fdf>

int mdlImage_warp
(
    byte        **outPP,       /* <= warped image buffer pointer */
    Point2d     *outSizeP,     /* <= warped image size in pixels */
    byte        *inP,          /* => input image buffer */
    Point2d     *inSizeP,      /* => input image size in pixels */
    int         imageFormat,    /* => image format */
    Transform    *inverseTransformP, /* => transform output to input pixel */
    byte        *transparentP,  /* => fill value if no input pixel */
    MdlFunctionP stopFunc      /* => stop function (or NULL) */
);
```

Description The mdlImage_warp function warps an image by using an inverse transform from each pixel of the output image to find the corresponding pixel in the source image.

If source point cannot be found, the output image is filled with a user-specified fill value.

outPP is a pointer to the warped image buffer allocated by `mdlImage_warp`. It will have the same format as the source image.

outSizeP is pointer to the size of the warped image in pixels. To warp the entire image, the size should be the maximum extents of the forward transform from the source to the destination image.

inP is a pointer to the source image buffer.

inSizeP is a pointer to the size of the source image.

imageFormat is any of the `IMAGEFORMAT` types defined in `image.h`, except `IMAGEFORMAT_RGB` or `IMAGEFORMAT_RGBA`. The only valid RGB format supported is `IMAGEFORMAT_RGBSeperate`.

inverseTransformP is a 2-D Transform from each pixel of the output image used to compute the corresponding pixel of the source image.

transparentP is the fill value. If the image is a bitmap and *transparentP* points to a non-zero value, the fill value is 1 bits; otherwise it is zero bits. If the image is mapped, *transparentP* points to a palette index. If the image is RGB, *transparentP* points to a 3-byte array of red, green and blue values.

stopFunc is a pointer to a MDL function that is called for every row of the image. If *stopFunc* returns a non-zero value, processing is aborted. A stop function can be used to allow the user to abort what could be a fairly lengthy process. `mdlImage_checkStop` is provided as a generic stop function. If no stop function is desired, `NULL` can be specified.



This function was implemented in MicroStation 95.

Returns `mdlImage_warp` returns `SUCCESS` if the operation is completed successfully. `MDLERR_INSFMEMORY` is returned if insufficient memory is available. `MDLERR_BADARG` is returned if an invalid image format is detected. `MDLERR_USERABORT` is returned if *stopFunc* returns a non-zero value.

See Also `mdlImage_rotate`, `mdlImage_mirror`, `mdlImage_checkStop`.

Material Table Functions

The material table functions are used to obtain material definitions describing the surface characteristics of objects in a design file. These material definitions may then be used in conjunction with an independent rendering software application to render MicroStation 3D design files.

The following table lists the material table functions:

Function	Used to
mdlMaterial_initialize	allocate and initialize structures needed by MicroStation to parse a material table.
mdlMaterial_cleanup	free up storage allocated by mdlMaterial_initialize.
mdlMaterial_load	parse a material table.
mdlMaterial_useTable	get a pointer to a material definition from a parsed material table which describes the surface characteristics of an object in a design file.

mdlMaterial_initialize

```
int mdlMaterial_initialize(void);
```

Description The mdlMaterial_initialize function allocates and initializes table and buffer space to be used by subsequent calls to the mdlMaterial_load function. This function must be called before any other material table functions are called. Once it has been called, the mdlMaterial_load and mdlMaterial_useTable functions may be called as many times as needed. When all material table information has been obtained, a call should be made to mdlMaterial_cleanup to free the memory allocated by mdlMaterial_initialize.

Returns mdlMaterial_initialize returns SUCCESS or a non-zero value if there was insufficient memory to complete the initialization.

See Also mdlMaterial_cleanup, mdlMaterial_load.

mdlMaterial_cleanup

```
void mdlMaterial_cleanup(void);
```

Description The mdlMaterial_cleanup function frees up all memory allocated and used by the mdlMaterial_initialize and mdlMaterial_load functions. The mdlMaterial_cleanup function should only be called after all material table information has been obtained using the mdlMaterial_load and mdlMaterial_useTable functions.

Returns mdlMaterial_cleanup has no return value.

See Also mdlMaterial_initialize.

mdlMaterial_load

```
int mdlMaterial_load
(
int      fileName,      /* => 0=Master dgn file, 1-255=ref files */
char     *filename      /* => ASCII file containing material table */
);
```

Description The mdlMaterial_load function reads an ASCII material table from the file specified by *filename* and parses the information into a table of material definitions in memory. Each material definition is uniquely identified by a level and color number pair and obtained through calls to the mdlMaterial_useTable function.

The *fileName* parameter identifies *filename* as a material table belonging to either the master design file (*fileName* equal to 0), or one of the attached reference files (*fileName* equal to 1-255).

The *filename* parameter is an ASCII file containing material table statements to be parsed into material definitions in memory.

Returns mdlMaterial_load returns SUCCESS or a non-zero value if the material table file could not be completely parsed.

See Also mdlMaterial_useTable.

mdlMaterial_useTable

```
#include <material.h>

int mdlMaterial_useTable
(
MaterialDefinition *materialDataP,      /* <= characteristics */
int      fileName,      /* => 0=Master dgn file, 1-255=ref files. */
int      level,         /* => Object's level. */
int      color          /* => Object's color. */
);
```

Description The mdlMaterial_useTable function is used to obtain a pointer to a material definition structure. This information may be used with independent rendering software to produce rendered images of 3D MicroStation design files. The material

definition structure includes the following information about the material characteristics of the specified object:

Material Definition Structure Member Name	Description
flags.color	Indicates whether the color structure member currently contains valid data (TRUE or FALSE).
color	The color of the material stated in Red, Green, Blue components. Values for each component range from 0.0 to 1.0.
flags.finish	Indicates whether the finish structure member currently contains valid data (TRUE or FALSE).
finish	The degree of polish of an object's surface. Value ranges from 0.0 to 1.0. For example, a ball bearing would have a value near 1.0; a piece of felt would have a value near 0.
flags.reflect	Indicates whether the reflect structure member currently contains valid data (TRUE or FALSE).
reflect	The share of incident light which is reflected with values ranging from 0.0 to 1.0. For example, a mirror would have a value near 1.0; a piece of felt would have a value near 0.
flags.transmit	Indicates whether the transmit structure member currently contains valid data (TRUE or FALSE).
transmit	The amount of incident light which passes through an object with values between 0.0 and 1.0. For example, a piece of glass would have a value near 1.0; a lead sheet would have a value near 0.
flags.pattern	Indicates whether the pattern structure member currently contains valid data (TRUE or FALSE).
pattern	The name of a raster file containing an image which can be applied to any surface on an object.
flags.shader	Indicates the number of valid entries in the <code>shader</code> array structure member. There can be up to 8 shader names in the array. For example, if <code>flags.shader</code> equals 3, then <code>shader[0]</code> , <code>shader[1]</code> and <code>shader[2]</code> contain shader names.
shader	An array of up to 8 surface shader names.
flags.metallic	Indicates that spectral highlights should be displayed in the base color of the material rather than white. This gives the surface a metallic appearance (TRUE or FALSE).

materialDataP is a pointer to a material definition structure to be filled out by `mdlMaterial_useTable`. The material definition returned is determined by the remaining parameters described below.

fileNumber refers to the master design file's material table (*fileNumber* equal to 0), or one of the attached reference file material tables (*fileNumber* equal to 1-255). It is from this indicated material table that the material definition pointed to by *materialDataP* will be obtained.

level and *color* reflect the level and color of the object whose material definition is being sought. These two parameters are used to uniquely identify a material definition in the material table identified by *fileNumber*.

Returns `mdlMaterial_useTable` returns `SUCCESS` if the material definition could be found corresponding to the given input parameters. Otherwise it returns a non-zero value.

See Also `mdlMaterial_load`.

Raster Reference Functions

The following table lists the Raster Reference functions.

Function	Used to
<code>mdlRastRef_attach [rastlib.msl]</code>	attach a raster reference file.
<code>mdlRastRef_detach [rastlib.msl]</code>	detach a raster reference file.
<code>mdlRastRef_extractReferences [rastlib.msl]</code>	scan design file for raster reference attachments.
<code>mdlRastRef_fileNameFromFileSpec [rastlib.msl]</code>	insert configuration variable into full path name specification so that path becomes relative.
<code>mdlRastRef_fileSpecFromFileName [rastlib.msl]</code>	find full path name specification from file name with embedded configuration variable.
<code>mdlRastRef_freeImage [rastlib.msl]</code>	free image memory associated with raster reference attachment.
<code>mdlRastRef_updateAttachment [rastlib.msl]</code>	change a raster reference file attachment.

mdlRastRef_attach [rastlib.msl]

```
#include <msrastrf.fdf>
#include <rastref.h>

int mdlRastRef_attach
(
    int          *indexP,          /* <= index of attached reference */
    RasterUpdateInfo *infoP,       /* => ptrs to attachment info */
    int          tintColor,        /* => color index (fgd or tint) */
    int          binaryBackground, /* => for monochrome only */
    int          contToneTransparent, /* => for greyscale */
    RGBColorDef *colorTransparentP, /* => for color only */
    int          level,            /* => element level */
    int          updateFlags       /* => bits ON to update display file */
);
```

Description The mdlRastRef_attach function creates a raster reference (Type 90) attachment and writes it to the design file. Optionally, it also adds a RasterReferenceFile structure to the global array of raster references. This array is maintained by two global variables, and you can access them by including the following lines in your program:

```
extern RasterReferenceFile *rasterRefFileP; /* array start */
extern int nRasterRefs; /* array count */
```

Optionally, mdlRastRef_attach will send a message to REF.MA so that it refreshes the dialog or display.

indexP is a pointer to the index in the *rasterRefFileP* array of structure(s). If *indexP* is NULL, *rasterRefFileP* and *nRasterRefs* are not changed. If *indexP* is not NULL, a new RasterReferenceFile structure will be inserted in raster layer order into the array starting at address *rasterRefFileP* and the count of the number of raster references, *nRasterRefs*, will be increased by one. The index in the array where the new structure is inserted is then returned to the caller.

infoP is a pointer to a *RasterUpdateInfo* structure. The caller provides this information to identify the type of raster reference. All pointers in this structure must be initialized to valid non-NULL values.

tintColor is the color index of the foreground color for a mapped or palettized color image.

binaryBackground is the color index of the background for a monochrome image.

contToneTransparent is the color index of the transparent color of a continuous tone or palettized color image.

colorTransparentP is a pointer to the color index of the transparent color of a RGB image.

level is element level that this raster is associated with.

updateFlags are a sequence of one or more flags connected by logical or operation. RASTREF_UPDATE_DISPLAY causes ref.ma to repaint the screen with the new reference file. RASTREF_UPDATE_FILE causes the raster reference attachment to be written to the master design file. RASTREF_UPDATE_DIALOG causes ref.ma to update the Raster Reference File Dialog. RASTREF_UPDATE_ALL does all of the above. If ref.ma is not loaded, operations performed by it are ignored.



This function was implemented in MicroStation 95.

Returns mdlRastRef_attach returns SUCCESS if the operation is completed successfully.

See Also mdlRastRef_updateAttachment [rastlib.msl], mdlRastRef_detach [rastlib.msl].

mdlRastRef_detach [rastlib.msl]

```
#include <msrastref.fdf>
#include <rastref.h>

int mdlRastRef_detach
(
    int      slot,           /* => slot to change */
    int      updateFlags    /* => bits ON to update display, file, dlg */
);
```

Description The mdlRastRef_detach function deletes a raster reference attachment.

slot is an index from the internal list of raster references, and must be a valid array index to this list. The following variables define the internal list of raster references:

```
extern RasterReferenceFile *rasterRefFileP; /* array start */
extern int nRasterRefs;                    /* array count */
```

updateFlags are a sequence of one or more flags connected by a logical or operation. RASTREF_UPDATE_DISPLAY causes ref.ma to repaint the screen with the new reference file. RASTREF_UPDATE_FILE causes the raster reference attachment to be written to the master design file. RASTREF_UPDATE_DIALOG causes ref.ma to update the Raster Reference File Dialog. RASTREF_UPDATE_ALL does all of the above. If ref.ma is not loaded, operations performed by it are ignored.



This function was implemented in MicroStation 95.

Returns mdlRastRef_detach returns SUCCESS if the operation is completed successfully.

See Also mdlRastRef_attach [rastlib.msl], mdlRastRef_updateAttachment [rastlib.msl].

mdlRastRef_extractReferences [rastlib.msl]

```
#include <msrastrf.fdf>
#include <mdlerrs.h>

int mdlRastRef_extractReferences
(
  RasterReferenceFile  **refPP,          /* <=> address of ptr to array to
                                          be malloced and filled */
  int    *nRefsP,          /* <=> ptr to number of array elements
                              allocated */
  int    fileName          /* => file to search (e.g. MASTERFILE) */
);
```

Description The `mdlRastRef_extractReferences` function scans the file indicated by *fileName*, and builds an array of information about Type 90 raster reference attachments. The array returned has already been sorted by raster layer number.

refPP is a pointer to the start of an array that will be allocated by `mdlRastRef_extractReferences`. Each index in this array is a single `RasterReferenceFile` structure. It is the caller's responsibility to free this array when it is no longer needed. This should be done by a sequence of `mdlRastRef_freeImage [rastlib.msl]` calls for each valid raster reference, followed by a call to free with the value of this pointer.

nRefsP is a pointer to the number of raster references that are found from the scan. If a zero value is returned, the *refPP* pointer has never been allocated, so it should not be freed.

fileName is an index to an open Incrustation master or reference file, such as MASTERFILE.



This function was implemented in MicroStation 95.

Returns `mdlRastRef_extractReferences` returns SUCCESS if it could successfully scan the file.

See Also `mdlRastRef_freeImage [rastlib.msl]`.

mdlRastRef_fileNameFromFileSpec [rastlib.msl]

```
#include <msrastrf.fdf>

int mdlRastRef_fileNameFromFileSpec
(
  char    *fileName,          /* <= file name with (possible) envVar */
  char    *fileSpec,          /* => file spec */
  char    *envVar             /* => environment variable nameto insert */
);
```

Description The `mdlRastRef_fileNameFromFileSpec` function creates a file name in the form:

`configname:name.ext`

where *configname* is an environment string, *name* is the base file name, and *ext* is the extension of the file.

fileName is the file name with a prefixed configuration variable.

fileSpec is the base file specification. If it contains a path, the path must point to the same location as that identified by the *envVar* configuration variable. Otherwise, it should simply be the base file name and extension.



This function was implemented in MicroStation 95.

Returns `mdlRastRef_fileNameFromFileSpec` returns `SUCCESS` the file name can be successfully constructed. Otherwise, it returns a non-zero value.

See Also `mdlRastRef_fileSpecFromFileName` [`rastlib.msl`].

mdlRastRef_fileSpecFromFileName [`rastlib.msl`]

```
#include <msrastrf.fdf>
```

```
int mdlRastRef_fileSpecFromFileName
```

```
(
```

```
char    *fileSpec,      /* <= translated filespec or NULL */
```

```
char    *fileName      /* => file name with (possible) envVar */
```

```
);
```

Description The `mdlRastRef_fileSpecFromFileName` function computes the full path name from a file specification of the form:

`configname:name.ext`

where *configname* is an environment string, *name* is the base file name, and *ext* is the extension of the file.

fileSpec is the returned file specification. This buffer should be `MAXFILELENGTH` in length.

fileName is the source file name, which includes the embedded configuration variable.



This function was implemented in MicroStation 95.

Returns `mdlRastRef_fileSpecFromFileName` returns `SUCCESS` if the environment string could be found. Otherwise, it returns a non-zero value.

See Also `mdlRastRef_fileNameFromFileSpec` [`rastlib.msl`].

mdlRastRef_freeImage [rastlib.msl]

```
#include <msrastrf.fdf>
#include <rastref.h>

void mdlRastRef_freeImage
(
  RasterReferenceFile *refP    /* => pointer to reference data */
);
```

Description The mdlRastRef_freeImage function frees the image data associated with a raster reference attachment. It does not resize the list of internal raster references, and does not free *rasterRefFileP*. If the caller is finished with this memory, the caller should call free, and set *rasterRefFileP* to NULL and *nRasterRefs* to 0. The internal list of raster references can be accessed by linking with the following declaration:

```
extern RasterReferenceFile *rasterRefFileP; /* array start */
extern int nRasterRefs;                    /* array count */
```

refP is a pointer to a RasterReferenceFile element whose image data is to be freed.



This function was implemented in MicroStation 95.

Returns mdlRastRef_freeImage is of type void; it returns no value.

See Also mdlRastRef_extractReferences [rastlib.msl].

mdlRastRef_updateAttachment [rastlib.msl]

```
#include <msrastrf.fdf>
#include <rastref.h>

int mdlRastRef_updateAttachment
(
  int          slot,                /* => slot to change */
  RasterUpdateInfo *changeP,        /* => ptrs to changed elements */
  int          *tintColorP,         /* => New color index (or NULL) */
  int          *colorModeP,         /* => New color mode (or NULL) */
  int          *binaryBackgroundP, /* => monochrome only (or NULL) */
  int          *contToneTransparentP, /* => for greyscale (or NULL) */
  RGBColorDef  *colorTransparentP, /* => for color only (or NULL) */
  int          updateFlags          /* => bits ON upd view, file, dlg */
);
```

Description The mdlRastRef_updateAttachment function changes a raster reference attachment.

slot is an index from the internal list of raster references, and must be a valid array index to this list. The following variables define the internal list of raster references:

```
extern RasterReferenceFile *rasterRefFileP; /* array start */
extern int nRasterRefs;                    /* array count */
```

changeP is a pointer to a `RasterUpdateInfo` structure. The caller provides this information to identify the type of raster reference. If *changeP* is `NULL` or any pointers in this structure are `NULL`, the associated item remains unchanged.

tintColorP is a pointer to the color index of the foreground color for a mapped or palettized color image. If the value is `NULL`, this item remains unchanged.

colorModeP is a pointer to the color mode (from `image.h`) of the image. If the value is `NULL`, this item remains unchanged.

binaryBackgroundP is a pointer to the color index of the background for a monochrome image. If the value is `NULL`, this item remains unchanged.

contToneTransparentP is a pointer to the color index of the transparent color of a continuous tone or palettized color image. If the value is `NULL`, this item remains unchanged.

colorTransparentP is a pointer to the color index of the transparent color of a RGB image.

updateFlags are a sequence of one or more flags connected by a logical or operation. `RASTREF_UPDATE_DISPLAY` causes `ref.ma` to repaint the screen with the new reference file. `RASTREF_UPDATE_FILE` causes the raster reference attachment to be written to the master design file.

`RASTREF_UPDATE_DIALOG` causes `ref.ma` to update the Raster Reference File Dialog. `RASTREF_UPDATE_ALL` does all of the above. If `ref.ma` is not loaded, operations performed by it are ignored.



This function was implemented in MicroStation 95.

Returns `mdlRastRef_updateAttachment` returns `SUCCESS` if the operation is completed successfully.

See Also `mdlRastRef_attach [rastlib.msl]`, `mdlRastRef_detach [rastlib.msl]`.

19

Settings Functions

MicroStation settings control element attributes and other system behavior. MDL applications should use `mdlParams_` functions to query or change these settings if needed.

This chapter discusses:

- Settings manipulation functions
- Level functions

Settings Manipulation Functions

Some settings are available to MDL applications as built-in variables. However, when possible, applications should use `mdlParams_setActive` to set them, because using this function lets MicroStation guarantee those settings' values are valid and helps make applications compatible across releases of MicroStation.

The following table lists the system parameter functions:

Function	Used to
<code>mdlParams_getActive</code>	get the value of an active setting.
<code>mdlParams_setActive</code>	change the value of an active setting.
<code>mdlParams_storeType9Variable</code>	save settings in the active design file.
<code>mdlParams_saveMasterLevelSymbology</code>	save current level symbology information.
<code>mdlParams_getLock</code>	retrieve the state of a MicroStation lock.
<code>mdlParams_setLock</code>	set the state of a MicroStation lock.

Example

See `params.mc`.

`mdlParams_getActive`, `mdlParams_setActive`

```
#include <mdl.h>

int mdlParams_getActive
(
    void    *param,          /* <= parameter value */
```

```
int      paramName      /* => parameter name */
);

int mdlParams_setActive
(
void      *param,        /* => parameter value */
int      paramName      /* => parameter name */
);
```

Description mdlParams_getActive returns the value of a MicroStation setting defined below. The parameter returned in *param* is determined by the value of *paramName*.

The mdlParams_setActive function is used to change the value of a setting. The new value is given in *param*. The setting to be changed is determined by the value of *paramName* as defined in the table below:

When passing values listed below to mdlParams_setActive, pay special attention to whether you should be passing a pointer or a value in *param*. mdlParams_getActive will always require a pointer for *param*.

paramName	Setting (equivalent MicroStation key-in)	Type for <i>param</i> (range)
ACTIVELOCK_ASSOCIATION	association lock (LOCK ASSOCIATION)	int * (0-1)
ACTIVELOCK_SNAPMODE	active snap mode (LOCK SNAP)	int * (0-1)
ACTIVEPARAM_ANGLE	active angle (AA=)	double * (0-360)
ACTIVEPARAM_AREAMODE	active area (solid/hole) (ACTIVE AREA)	int (0-1)
ACTIVEPARAM_AXISANGLE	axis increment (ACTIVE AXIS)	double *
ACTIVEPARAM_AXISORIGIN	axis origin (ACTIVE AXORIGIN)	double *
ACTIVEPARAM_CLASS	active class (ACTIVE CLASS)	int (0-15)
ACTIVEPARAM_CAPMODE	3D type (surface or solid) (ACTIVE CAPMODE)	int (0-1)
ACTIVEPARAM_CELLNAME	set active cell (AC=)	char *
ACTIVEPARAM_COLOR	active Color (CO=)	int (0-255)
ACTIVEPARAM_COLOR_BY_NAME	active color (CO=)	char *
ACTIVEPARAM_DIMCOMPAT	associative dimensioning compatibility mode (SET COMPATIBLE DIMENSION)	int * (0-1)
ACTIVEPARAM_FILLCOLOR	active fill color (ACTIVE FILLCOLOR)	int (0-6)
ACTIVEPARAM_FILLMODE	active fill (ACTIVE FILL)	int (0-1)

paramName	Setting (equivalent MicroStation key-in)	Type for <i>param</i> (range)
ACTIVEPARAM_FONT	active font (FT=)	int (0-255)
ACTIVEPARAM_GRIDMODE	grid configuration (ACTIVE GRIDMODE)	int (0=orthogonal, 1=isometric)
ACTIVEPARAM_GRIDRATIO	grid aspect ratio (ACTIVE GRIDRATIO)	double *
ACTIVEPARAM_GRIDREF	reference grid (GR=)	int
ACTIVEPARAM_GRIDUNITS	master grid (GU=)	double *
ACTIVEPARAM_KEYPOINT	snap divisor (KY=)	int (1-255)
ACTIVEPARAM_LEVEL	active level (LV=)	int (1-63)
ACTIVEPARAM_LINELENGTH	active line length (text node) (LL=)	int (1-255)
ACTIVEPARAM_LINESPACING	active line spacing (LS=)	double *
ACTIVEPARAM_LINestyle	active line style (LC=)	int (0-7)
ACTIVEPARAM_LINestyleNAME	active line style (LC=)	char *
ACTIVEPARAM_LINestylePARAMS	active modifiers	StyleParam * (defined in mslstyle.h)
ACTIVEPARAM_MLINECOMPAT	multi-line compatibility (SET COMPATIBLE MLINE)	int * (0-1)
ACTIVEPARAM_LINEWEIGHT	active line weight (WT=)	int (0-31)
ACTIVEPARAM_NODEJUST	active text node justification (ACTIVE TNJ)	int (see mdl.h for TXTJUST_ values)
ACTIVEPARAM_PATTERNANGLE	active pattern angle(s) (PA=)	Dpoint3d *
ACTIVEPARAM_PATTERNCELL	active pattern cell (AP=)	char *
ACTIVEPARAM_PATTERNDELTA	active pattern delta(s) (PD=)	Dpoint3d *
ACTIVEPARAM_PATTERNSCALE	active pattern scale (PS=)	Dpoint3d *
ACTIVEPARAM_POINT	active point (ACTIVE POINT)	char *
ACTIVEPARAM_SNAPOVERRIDE	one-time snap mode override (SNAP)	int * (0-7) (values in msdefs.h)
ACTIVEPARAM_TAB	space characters substituted for each tab	int (1-31)
ACTIVEPARAM_STREAMANGLE	active stream angle (ACTIVE STREAM ANGLE)	double *
ACTIVEPARAM_STREAMAREA	active stream area (ACTIVE STREAM AREA)	double *
ACTIVEPARAM_STREAMDELTA	active stream delta (SD=)	double *

paramName	Setting (equivalent MicroStation key-in)	Type for <i>param</i> (range)
ACTIVEPARAM_STREAMTOLERANCE	active stream tolerance (ST=)	double *
ACTIVEPARAM_TAGINCREMENT	tag increment (INCREMENT TEXT) (TI=)	int (1-1023)
ACTIVEPARAM_TERMINATOR	active line terminator (LT=)	char *
ACTIVEPARAM_TERMINATORSCALE	active line terminator scale	double *
ACTIVEPARAM_TEXTHEIGHT	active text height (TH=)	double *
ACTIVEPARAM_TEXTJUST	active text justification (ACTIVE TXJ)	int (see <code>mdl.h</code> for TXTJUST_ values)
ACTIVEPARAM_TEXTWIDTH	active text width (TW=)	double *
ACTIVEPARAM_UNITROUNDOFF	active unitround (UR=)	double *

For more information about these settings, see the menu reference chapters of the *MicroStation Reference Guide*.



The values for:

```

..._GRIDUNITS
..._TEXTHEIGHT
..._TEXTWIDTH
..._UNITROUNDOFF
..._LINESPACING
..._STREAMDELTA
..._STREAMTOLERANCE
..._STREAMAREA
..._PATTERNDELTA

```

specify distances (or areas) and are scaled by the current transform if one exists.

```

..._SCALE
..._PATTERNDELTA
..._PATTERNANGLE
..._PATTERNSCALE

```

require multiple values taken from the members of the `Dpoint3d` structure.

Returns `mdlParams_getActive` returns `SUCCESS` if the active parameter value is returned in *param*. It returns `ERROR` if *paramName* is not one of the values listed above.

mdlParams_setActive returns SUCCESS if the active parameter is changed and ERROR if either *paramName* is not one of the values listed above or the value in *param* was out of range (or a cell of the given name could not be found in the library).



In versions of MicroStation and MDE prior to 4.4, mdlParams_setActive returned a value of 1 instead of ERROR.

mdlParams_storeType9Variable

```
#include <mdl.h>

int mdlParams_storeType9Variable
(
    void    *variable,      /* => new value to store */
    int     varSize,        /* => size in bytes */
    ULONG   offset          /* => byte offset in type 9 element */
);
```

Description When the FILE DESIGN command is issued, several active settings are saved in the design file and are stored in the first element in the file, the Type 9 element. The mdlParams_storeType9Variable function stores one variable of the Type 9 element without saving all of them.

variable points to the new value of the Type 9 parameter to be saved.

varSize is the size, in bytes, of the Type 9 parameter.

offset is the byte offset in the Type 9 element to store the value.



file for the active parameters that can be stored with this function and for the proper values for offset.

The mdlParams_storeType9Variable function is provided mainly for compatibility with the User Command statement STO.



The use of mdlParams_storeType9Variable is discouraged because it has a strong potential to corrupt design files and is rarely necessary. Programs using mdlParams_storeType9Variable should be thoroughly tested using scratch design files before being released.

Returns mdlParams_storeType9Variable returns the number of bytes actually written to the file. This should match *varSize* if the function was successful.

See Also mdlSystem_fileDesign.

mdlParams_saveMasterLevelSymbology

```
int mdlParams_saveMasterLevelSymbology(void);
```

Description The `mdlParams_saveMasterLevelSymbology` function rewrites the Type 10 element with the current level symbology information. Level symbology information is stored in a Type 10 element in the design file.

The `mdlParams_saveMasterLevelSymbology` function has no parameters.

Returns The `mdlParams_saveMasterLevelSymbology` function returns `SUCCESS` if the Type 10 element was found. Otherwise, the function is rewritten and returns `ERROR`.

See Also `mdlSystem_fileDesign`.

mdlParams_getLock, mdlParams_setLock

```
#include <mdl.h>
#include <msmisc.fdf>

int mdlParams_getLock
(
    int    paramName
);

int mdlParams_setLock
(
    int    lockVal,
    int    paramName
);
```

Description The `mdlParams_getLock` function retrieves the state of one of MicroStation's locks, and the `mdlParams_setLock` sets the state of a MicroStation lock to the value of *lockVal*. The only valid values for *lockVal* are 0 or 1.

paramName is an integer that specifies which MicroStation lock is retrieved or set. The possible values for *paramName* and the effect they have are tabulated below:

paramName	Description	MicroStation Command
ACTIVELOCK_ANGLE	When on, the active angle can only be set only to unit multiples of the angle roundoff value.	LOCK ANGLE ON OFF
ACTIVELOCK_ASSOCIATION	When on, association lock is enabled.	LOCK ASSOCIATION ON OFF
ACTIVELOCK_AXIS	When on, inputs are constrained to multiple of axis angle.	LOCK AXIS ON OFF
ACTIVELOCK_BORESITE	When on, MicroStation locates elements at any depth along the sight line in 3D file.	LOCK BORESITE ON OFF

paramName	Description	MicroStation Command
ACTIVELOCK_CELLSTRETCH	When on, fence stretch operation stretches cells.	LOCK CELLSTRETCH ON OFF
ACTIVELOCK_CONSTRUCTION	When on, input is locked to the construction plane defined by the auxiliary coordinate system.	LOCK ACS ON OFF
ACTIVELOCK_DEPTH	When on, tentative points are locked to the active depth in a view.	LOCK DEPTH ON OFF
ACTIVELOCK_FENCECLIP	When on, elements are clipped at fence boundaries by fence commands.	LOCK FENCE CLIP
ACTIVELOCK_FENCEOVERLAP	When on, elements overlapping the fence are operated on by fence commands.	LOCK FENCE OVERLAP
ACTIVELOCK_FENCEVOID	When on, elements outside the fence are operated on by fence commands, rather than those inside.	LOCK FENCE VOID
ACTIVELOCK_GRAPHGROUP	When on, manipulations affect every element in the graphic group of the element chosen.	LOCK GGROUP ON OFF
ACTIVELOCK_GRID	When on, input datapoints lock to nearest grid point.	LOCK GRID ON OFF
ACTIVELOCK_ISOMETRIC	When on, input is locked to the active isometric plane.	LOCK ISOMETRIC ON OFF
ACTIVELOCK_LEVEL	When on, only elements on the active level can be manipulated.	LOCK LEVEL ON OFF
ACTIVELOCK_SCALE	When on, the active scale can be set only to unit multiple of scale roundoff value.	LOCK SCALE ON OFF
ACTIVELOCK_SELECTION	When on (the default), modification commands act on a selection set.	LOCK SELECTION ON OFF

paramName	Description	MicroStation Command
ACTIVELOCK_SNAP	When on, tentative points snap to elements.	LOCK SNAP ON OFF
ACTIVELOCK_TEXTNODE	When on, text can only be placed by attaching to a text node.	LOCK TEXTNODE ON OFF
ACTIVELOCK_UNIT	When on, input datapoints lock to nearest multiple of unit round value.	LOCK UNIT ON OFF



These functions were implemented in MicroStation 95.

Returns `mdlParams_getLock` returns the current state of the lock if *paramName* is one of the above constants, or `ERROR` otherwise. `mdlParams_setLock` returns `ERROR` if the *paramName* is not one of the above constants or *lockVal* is not 0 or 1, and returns `SUCCESS` otherwise.

Level Functions

MicroStation provides a way to name it's drawing levels and group related levels into level groups that can be manipulated as units for display purposes. Levels and groups can be arranged in a hierarchical order much like a file system.

Level names and level group names can have 16 characters. Level names can have optional comments of up to 32 characters. Level names are assigned and edited interactively with the Level Names dialog box.

Level names are stored in the design file. Therefore, different level name structures are possible for different design files, depending on the purposes of the design files. Most of the `mdlLevel` functions operate only on the level name definitions for the main design file. The only exception is `mdlLevel_getLevelNameByFile`. The master file provides the level name definitions.

MicroStation maintains an internal data structure called `designLevels`, which contains the currently named level definitions. This structure contains pointers to an array of `NamedLevel` for the named levels and to an array of `LevelGroup` for the level groups.

The MDL include file `levels.h` contains the definitions needed for using named levels.

The following table lists the level names functions:

Function	Used to
mdlLevel_getLevelMask	get level mask from numeric level.
mdlLevel_getLevelMaskFromLevel	get level mask from level name.
mdlLevel_getLevelMaskFromGroup	get level mask from level group.
mdlLevel_getLevelMaskFromPath	get level mask from level path.
mdlLevel_addLevel	add level name to level structure.
mdlLevel_deleteLevel	delete level name from level structure.
mdlLevel_editLevel	edit level name in level structure.
mdlLevel_getNextGroupID	get next unique group ID.
mdlLevel_addGroup	add level group to level structure.
mdlLevel_deleteGroup	delete level group from level structure.
mdlLevel_editGroup	edit level group in level structure.
mdlLevel_getDescendentLevels	get levels descendent from level group.
mdlLevel_getDescendentGroups	get level groups descendent from level group.
mdlLevel_getGroupLevels	get immediate levels belonging to level group.
mdlLevel_freeGroupSet	free memory from level group set.
mdlLevel_freeLevelSet	free memory from level name set.
mdlLevel_buildLevelPath	build level path specifying level group.
mdlLevel_getLevelIndex	get <i>designLevels</i> index of level name.
mdlLevel_getLevelIndexFromName	get <i>designLevels</i> index of level name from only level name.
mdlLevel_getParentGroup	get parent level group of level name.
mdlLevel_getGroupIndex	get <i>designLevels</i> index of level group.
mdlLevel_getGroupIndexFromName	get <i>designLevels</i> index of level group from only level group name.
mdlLevel_getGroupNameFromID	get level group name from level group ID.
mdlLevel_getGroupIDFromNameAndParent	get group ID from level group name and parent level group.
mdlLevel_freeAll	free <i>designLevels</i> level structure.
mdlLevel_freeLevels	free <i>designLevels</i> level names.
mdlLevel_freeGroups	free <i>designLevels</i> level groups.
mdlLevel_loadAllFromResource	load level names and level groups from external resource file.

Function	Used to
<code>mdlLevel_loadGroupsFromResource</code>	load only level groups from external resource file.
<code>mdlLevel_loadLevelsFromResource</code>	load only level names from external resource file.
<code>mdlLevel_saveAllToResource</code>	save level names and level groups to external resource file.
<code>mdlLevel_saveGroupsToResource</code>	save only level groups to external resource file.
<code>mdlLevel_saveLevelsToResource</code>	save only level names to external resource file.
<code>mdlLevel_getLevelName</code>	retrieve name associated with the specified level.
<code>mdlLevel_getLevelNameByFile</code>	find a level name defined for a file number.
<code>mdlLevelSymbology_extract</code>	extract the level symbology associated with a given level and file.

Example

See `lvlnames.mc`.

mdlLevel_getLevelMask

```
#include <mdl.h>
#include <levels.h>

void mdlLevel_getLevelMask
(
    short  *offset, /* <= offset of mask in level mask array */
    short  *mask,   /* <= 16 bit level mask */
    int    level,   /* => level number */
);
```

Description MicroStation frequently uses level masks to represent a set of levels for display operations. The level mask is an array of `short` integers. The first element of the level mask array represents levels 1-16, the second element of the array represents levels 17-32, and so on through `MAX_LEVELS`.

The `mdlLevel_getLevelMask` function sets the appropriate bit of *mask* corresponding to the numeric level modulo 16 passed in *level*. The lowest-order bit of *mask* corresponds to level 1, while the highest-order bit

represents level 16. The value *mask* should be inserted in a level mask array at index *offset*.



mask points to a single `short` integer, not to an array of `short` integers.

Returns The `mdlLevel_getLevelMask` function is of type `void`. It returns no value.

See Also `mdlLevel_getLevelMaskFromLevel`, `mdlLevel_getLevelMaskFromGroup`, `mdlLevel_getLevelMaskFromPath`.

mdlLevel_getLevelMaskFromLevel

```
#include <mdl.h>
#include <levels.h>

int mdlLevel_getLevelMaskFromLevel
(
    short  *levelMask,    /* <= level mask */
    char   *levelName,    /* => name of level */
    int    groupID        /* => groupID of level */
);
```

Description MicroStation frequently uses level masks to represent a set of levels for display operations. The level mask is an array of `short` integers. The first element of the level mask array represents levels 1-16, the second element of the array represents levels 17-32 and so on through `MAX_LEVELS`.

`mdlLevel_getLevelMaskFromLevel` sets the appropriate bit in the level mask array *levelMask*. *levelName* contains the level name and *groupID* contains the level's unique groupID. Only the bit corresponding to *levelName* is set in the *levelMask* array.

Returns `mdlLevel_getLevelMaskFromLevel` returns `SUCCESS` if the level was found in the level structure. Otherwise, it returns `INVALID_LEVELNAME`.

See Also `mdlLevel_getLevelMask`, `mdlLevel_getLevelMaskFromGroup`, `mdlLevel_getLevelMaskFromPath`.

mdlLevel_getLevelMaskFromGroup

```
#include <mdl.h>
#include <levels.h>

int mdlLevel_getLevelMaskFromGroup
(
    short  *levelMask,    /* <= level mask */
    int    groupID        /* => groupID of level */
);
```

Description MicroStation frequently uses level masks to represent a set of levels for display operations. The level mask is an array of `short` integers. The first element of the

level mask array represents levels 1-16, the second element of the array represents levels 17-32 and so on through `MAX_LEVELS`.

The `mdlLevel_getLevelMaskFromGroup` function sets the appropriate bits in the level mask array *levelMask* corresponding to all of the levels descendent from the group *groupID*. *groupID* contains the unique group ID of a group in the level structure. Only the bits corresponding to the descending levels of *groupID* are set in the level mask array *levelMask*.

Returns The `mdlLevel_getLevelMaskFromGroup` function returns `INVALID_GROUP` if the level was found in the level structure. Otherwise, it returns `ERROR`.

See Also `mdlLevel_getLevelMask`, `mdlLevel_getLevelMaskFromLevel`.

mdlLevel_getLevelMaskFromPath

```
#include <mdl.h>
#include <levels.h>

int mdlLevel_getLevelMaskFromPath
(
    short  *levelMask,          /* <= level mask for levelString */
    char   *errorMessage,      /* <= error message */
    int    places,             /* => number of bits in levelMask */
    char   *levelString,       /* => named level specification */
    int    parentID            /* => starting group */
);
```

Description MicroStation frequently uses level masks to represent a set of levels for display operations. The level mask is an array of short integers. The first element of the level mask array represents levels 1-16, the second element of the array represents levels 17-32 and so on through `MAX_LEVELS`.

The `mdlLevel_getLevelMaskFromPath` function sets the appropriate bits in the level mask array *levelMask* corresponding to the level path specified by *levelPath*. *levelPath* contains a level or group specification of the form:

```
<:groupName ...>levelName | groupName
```

levelPath can specify a single level or a level group. The complete level path specification does not need to be supplied if the level name or level group name is unique across the level structure.

errorText, possibly `NULL`, receives the text of any error generated during the call. *maxLevels*, usually `MAX_LEVELS`, specifies the size, in bits, of the level mask array *levelMask*. *parentID* contains the starting group ID for the level path. In this way partial paths relative to any group can be specified. The root of the level structure is zero.

Only the bits corresponding to the level path *levelPath* are set in the level mask array *levelMask*.

Returns The `mdlLevel_getLevelMaskFromPath` function returns `SUCCESS` if the level path was successfully parsed. `INVALID_GROUP` or `INVALID_LEVELNAME` is returned if the level path contains a level name or level group not found in the level structure. `DUPLICATE_GROUP` or `DUPLICATE_LEVEL` is returned if the level path did not uniquely resolve a level group or level name.

See Also `mdlLevel_getLevelMask`, `mdlLevel_getLevelMaskFromLevel`, `mdlLevel_getLevelMaskFromGroup`.

mdlLevel_addLevel

```
#include <mdl.h>
#include <levels.h>

int mdlLevel_addLevel
(
    char    *levelName,          /* => name of level */
    char    *levelComment,      /* => comment for this level */
    int     groupID,            /* => group ID of level */
    int     levelNumber         /* => IGDS level number */
);
```

Description The `mdlLevel_addLevel` function adds a new level name to the level structure. *levelName* contains the level name which must be less than 16 characters. The level description is passed in *levelComment*. Level comments are limited to 32 characters. The new level's unique group ID is passed in *groupID*. The *group ID* of the level structure's root is zero. The MicroStation number of the new level is passed in *levelNumber*.

Returns The `mdlLevel_addLevel` function returns `SUCCESS` if the level name was added successfully. `DUPLICATE_LEVEL` is returned if a level of name *levelName* already exists in group *groupID*. `MDLERR_INSFMEMORY` is returned if memory is insufficient to complete the operation.

See Also `mdlLevel_deleteLevel`, `mdlLevel_editLevel`, `mdlLevel_addGroup`, `mdlLevel_deleteGroup`, `mdlLevel_editGroup`.

mdlLevel_deleteLevel

```
#include <mdl.h>
#include <levels.h>

int mdlLevel_deleteLevel
(
    char    *levelName,          /* => name of level */
    int     groupID              /* => group ID of level */
);
```

- Description** The `mdlLevel_deleteLevel` function deletes a level from the level structure. The level name is passed in *levelName* and the level's unique *groupID* is passed in *groupID*.
- Returns** The `mdlLevel_deleteLevel` function returns `SUCCESS` if the level was added successfully. `INVALID_LEVELNAME` is returned if the level was not found in the level structure. `ERROR` is returned if a level structure is not loaded.
- See Also** `mdlLevel_addLevel`, `mdlLevel_editLevel`, `mdlLevel_addGroup`, `mdlLevel_deleteGroup`, `mdlLevel_editGroup`.

mdlLevel_editLevel

```
#include <mdl.h>
#include <levels.h>

int mdlLevel_editLevel
(
    char    *levelName,    /* => level name of existing level */
    int     groupID,       /* => groupID of existing level */
    char    *newName,      /* => new name of level */
    char    *newComment,   /* => new comment for this level */
    int     newNumber      /* => new IGDS level # for this level */
);
```

- Description** The `mdlLevel_editLevel` function retrieves the attributes of an existing level name in the level structure. *levelName* identifies the name of the level and *groupID* contains the level's unique *groupID*.

newName, *newComment*, and *newNumber* supply the level name, descriptive comment and group ID of the modified level.

- Returns** The `mdlLevel_editLevel` function returns `SUCCESS` if the level was modified successfully. `INVALID_LEVELNAME` is returned if the level name was not located in the level structure. `MDLERR_INSFMEMORY` is returned if memory is insufficient to complete the operation.
- See Also** `mdlLevel_addLevel`, `mdlLevel_deleteLevel`, `mdlLevel_addGroup`, `mdlLevel_deleteGroup`, `mdlLevel_editGroup`.

mdlLevel_getNextGroupID

```
#include <mdl.h>
#include <levels.h>

int mdlLevel_getNextGroupID
(
    int     *groupID /* <= next available group ID */
);
```

Description Every level group in the level structure is identified by a unique integer ID. The mdlLevel_getNextGroupID function retrieves the next available group ID. This ID can then be used to create and add a new level group to the level structure.

Returns The mdlLevel_getNextGroupID function returns SUCCESS if the next group ID could be determined. ERROR is returned if a level structure is not loaded.

See Also mdlLevel_addGroup.

mdlLevel_addGroup

```
#include <mdl.h>
#include <levels.h>

int mdlLevel_addGroup
(
    char    *groupName,    /* => name of level group */
    int     groupID,       /* => unique ID of group */
    int     parentID       /* => ID of parent group */
);
```

Description The mdlLevel_addGroup function adds new level groups to the level structure. *groupName* contains the name of new level group. *groupID* contains the level group's unique group ID. The level group is inserted into the level structure such that *parentID* is the group ID of the level group which owns the new group. The group ID of the root of the level structure is zero.

Returns The mdlLevel_addGroup function returns SUCCESS if the level group was added successfully. DUPLICATE_GROUP is returned if *groupName* already exists in group, *groupID*. MDLERR_INSFMEMORY is returned if memory is insufficient to complete the operation.

See Also mdlLevel_getNextGroupID, mdlLevel_deleteGroup, mdlLevel_editGroup, mdlLevel_addLevel, mdlLevel_deleteLevel, mdlLevel_editLevel.

mdlLevel_deleteGroup

```
#include <mdl.h>
#include <levels.h>

int mdlLevel_deleteGroup
(
    int     groupID,       /* => groupID of level group */
    boolean deleteSubGroups /* => delete descendent groups */
);
```

Description The `mdlLevel_deleteGroup` function removes level groups from the level structure. The level group's unique *groupID* is passed in *groupID*. Levels and groups descendent from *groupID* are not deleted if *deleteSubGroups* is `FALSE`.

Returns The `mdlLevel_deleteGroup` function returns `SUCCESS` if the level group was deleted. `INVALID_GROUP` is returned if the level group was not located in the level structure. `ERROR` is returned if a level structure is not loaded.

See Also `mdlLevel_addGroup`, `mdlLevel_editGroup`, `mdlLevel_addLevel`, `mdlLevel_deleteLevel`, `mdlLevel_editLevel`.

mdlLevel_editGroup

```
#include <mdl.h>
#include <levels.h>

int mdlLevel_editGroup
(
    int      groupID,          /* => groupID of existing group */
    char     *groupName       /* => new group name */
);
```

Description The `mdlLevel_editGroup` function modifies the name of an existing level group. *groupID* contains an existing level group's unique group identification number. *newName* contains the level group's new name.

Returns `mdlLevel_editGroup` returns `SUCCESS` if the level group was modified successfully. `INVALID_GROUP` is returned if the level group was not found in the level structure. `MDLERR_INSFMEMORY` is returned if memory is insufficient to complete the operation.

See Also `mdlLevel_addGroup`, `mdlLevel_deleteGroup`, `mdlLevel_addLevel`, `mdlLevel_deleteLevel`, `mdlLevel_editLevel`.

mdlLevel_getDescendentLevels

```
#include <mdl.h>
#include <levels.h>

int mdlLevel_getDescendentLevels
(
    LevelSet *levelSetP,      /* <= levels descendent from group */
    int      groupID         /* => groupID of group */
);
```

Description The MicroStation structure *designLevels* contains the current level structure. The member *designLevels.levels* points to an array of type `NamedLevel` containing the level names. The *designLevels.groups* member points to an array of type `LevelGroup` containing the level groups.

The `mdlLevel_getDescendentLevels` function obtains a structure of type `LevelSet` containing an array of indices into *designLevels.levels* for all levels descending from the group. These levels have a unique *groupID*.

levelSetP points to a structure of type `LevelSet`. This structure will be completed with the *designLevels.levels* array indices.



The `mdlLevel_getDescendentLevels` function allocates memory for an array of level indices in *levelSetP->indexArray*. You should free this memory when you are finished processing the levelSet with a call to `mdlLevel_freeLevelSet`.

Returns The `mdlLevel_getDescendentLevels` function returns `SUCCESS` if the level group was found in the level structure and `INVALID_GROUP` if it was not located. `ERROR` is returned if the function is unable to access sublevels.

See Also `mdlLevel_getDescendentGroups`, `mdlLevel_getGroupLevels`.

mdlLevel_getDescendentGroups

```
#include <mdl.h>
#include <levels.h>

int mdlLevel_getDescendentGroups
(
    GroupSet    *groupSetP,    /* <= groups descendent from group */
    int         groupID        /* => groupID of group */
);
```

Description The MicroStation structure *designLevels* contains the current level structure. The *designLevels.levels* member points to an array of type `NamedLevel` containing the level names. The member *designLevels.groups* points to an array of type `LevelGroup` containing the level groups.

`mdlLevel_getDescendentGroups` obtains a structure of type `GroupSet` containing an array of indices into *designLevels.groups* for all groups descending from the group. These groups have a unique ID of *groupID*. *groupSetP* points to a structure of type `GroupSet`, which will be completed with the *designLevels.groups* array indices.



The `mdlLevel_getDescendentGroups` function allocates memory for an array of level group indices in *groupSetP->indexArray*. You should free this memory when you are finished processing the `GroupSet` with a call to `mdlLevel_freeGroupSet`.

Returns The `mdlLevel_getDescendentGroups` function returns `SUCCESS` if the level group was found in the level structure and `INVALID_GROUP` if it was not located. `MDLERR_INSMEMORY` is returned if memory is insufficient to complete the operation.

See Also `mdlLevel_getDescendentLevels`, `mdlLevel_getGroupLevels`.

mdlLevel_getGroupLevels

```
#include <mdl.h>
#include <levels.h>

int mdlLevel_getGroupLevels
(
    LevelSet      *levelSetP,    /* <= levels belonging to group */
    int           groupID       /* => groupID of group */
);
```

Description The MicroStation designLevels structure contains the current level structure. The member *designLevels.levels* points to an array of type `NamedLevel` containing the level names. The *designLevels.groups* member points to an array of type `LevelGroup` containing the level groups.

The `mdlLevel_getGroupLevels` function obtains a structure of type `LevelSet` containing an array of indices into *designLevels.levels* for all levels immediately belonging to the group. These members have a unique ID of *groupID*. *levelSetP* points to a structure of type `LevelSet`. This structure will be completed with the *designLevels.levels* array indices.



The `mdlLevel_getGroupLevels` function allocates memory for an array of level indices in *levelSetP->indexArray*. You should free this memory when you are finished processing the `LevelSet` with a call to `mdlLevel_freeLevelSet`.

Returns The `mdlLevel_getGroupLevels` function returns `SUCCESS` if the level group was found in the level structure and `INVALID_GROUP` if it was not located. `MDLERR_INSFMEMORY` is returned if memory is insufficient to complete the operation.

See Also `mdlLevel_getDescendentLevels`, `mdlLevel_getDescendentGroups`.

mdlLevel_freeGroupSet

```
#include <mdl.h>
#include <levels.h>

void mdlLevel_freeGroupSet
(
    GroupSet      *groupSetP     /* => group set to release */
);
```

Description The `mdlLevel_freeGroupSet` function frees memory allocated by MicroStation during a call to `mdlLevel_getDescendentGroups`. This function allocates memory to build an array of level group indices into *designLevels.groups*. *groupSetP* points to a structure of type `GroupSet` previously passed to `mdlLevel_getDescendentGroups`.

Returns The `mdlLevel_freeGroupSet` function is of type `void`. It returns no value.

See Also `mdlLevel_getDescendentGroups`.

mdlLevel_freeLevelSet

```
#include <mdl.h>
#include <levels.h>

void mdlLevel_freeLevelSet
(
    LevelSet    *levelSetP    /* => level set to release */
);
```

Description The mdlLevel_freeLevelSet function frees memory allocated by MicroStation during a call to mdlLevel_getDescendentLevels or mdlLevel_getGroupLevels. These functions allocate memory to build an array of level name indices into *designLevels.levels*. *levelSetP* points to a structure of type LevelSet. This structure previously passed to either mdlLevel_getDescendentLevels or mdlLevel_getGroupLevels.

Returns The mdlLevel_freeLevelSet function is of type void. It returns no value.

See Also mdlLevel_getDescendentLevels, mdlLevel_getGroupLevels.

mdlLevel_buildLevelPath

```
#include <mdl.h>
#include <levels.h>

int mdlLevel_buildLevelPath
(
    char    *levelPath,    /* <= full path specification */
    int     groupID,       /* => group ID */
    char    *groupName     /* => group name */
);
```

Description The mdlLevel_buildLevelPath function constructs a level path specification for the level group whose unique group ID is *groupID*. This level path consists of a colon-delimited list of level groups from the root of the level structure to *groupID*. *groupName* can be NULL.

Returns mdlLevel_buildLevelPath returns SUCCESS if the level path was successfully built. INVALID_GROUP is returned if the level group was not found in the level structure.

See Also mdlLevel_getLevelMaskFromPath.

mdlLevel_getLevelIndex

```
#include <mdl.h>
#include <levels.h>

void mdlLevel_getLevelIndex
(
    int      *levelIndex,    /* <= index of level in designLevels */
    char     *levelName,     /* => name of level */
    int      groupID         /* => group ID of level */
);
```

Description The MicroStation *designLevels* structure contains the current level structure. The *designLevels.levels* member points to an array of type `NamedLevel`. This array contains the level names. The *designLevels.groups* member points to an array of type `LevelGroup`. This array contains the level groups.

The `mdlLevel_getLevelIndex` function returns the *designLevels.levels* index in *levelIndex*. This index has the level name specified by *levelName*. The level's unique group ID is passed in *groupID*.

Returns The `mdlLevel_getLevelIndex` function returns `SUCCESS` if the level name was found in the level structure. Otherwise, it returns `INVALID_LEVELNAME`. `ERROR` is returned if a level structure is not loaded.

See Also `mdlLevel_getLevelIndexFromName`.

mdlLevel_getLevelIndexFromName

```
#include <mdl.h>
#include <levels.h>

void mdlLevel_getLevelIndexFromName
(
    int      *index,         /* <= designLevels index of level */
    char     *levelName      /* => level name */
);
```

Description The MicroStation *designLevels* structure contains the current level structure. The *designLevels.levels* member points to an array of type `NamedLevel` containing the level names. The member *designLevels.groups* points to an array of type `LevelGroup` containing the level groups.

The `mdlLevel_getLevelIndexFromName` function returns the *designLevels.levels* index in *levelIndex*. This index has the level name specified by *levelName*.



A level name is not always unique across a level structure. `mdlLevel_getLevelIndex` uniquely specifies a level name if the level's group ID is known.

Returns The mdlLevel_getLevelIndexFromName function returns SUCCESS if the level name was found in the level structure. DUPLICATE_LEVEL is returned if the level name is not unique and INVALID_LEVELNAME is returned if the level is not found. ERROR is returned if a level structure is not loaded.

See Also mdlLevel_getLevelIndex.

mdlLevel_getParentGroup

```
#include <mdl.h>
#include <levels.h>

int mdlLevel_getParentGroup
(
    int      *parentID,      /* <= ID of parent group */
    char     *parentName,    /* <= name of parent group */
    int      groupID,        /* => ID of group */
    char     *groupName      /* => name of group */
);
```

Description The mdlLevel_getParentGroup function finds the level group (possibly the root) that owns another level group. The parent's unique group ID is returned in *parentID*. The ID of the root is zero. The name of the parent group is returned in *parentName*.

The desired level group can be specified either by the *groupID* or *groupName*. You can pass NULL for the unused argument. If *groupID* is passed it specifies the level group's unique group ID. If *groupName* is used, it specifies the level group name.



The name of a level group is not necessarily unique across a level structure but the group ID is. Always use the group ID if it is known.

Returns The mdlLevel_getParentGroup function returns SUCCESS if the parent group was located. DUPLICATE_GROUP is returned if *groupName* did not specify a unique group. INVALID_GROUP is returned if the level group was not found.

See Also mdlLevel_getDescendentGroups.

mdlLevel_getGroupIndex

```
#include <mdl.h>
#include <levels.h>

int mdlLevel_getGroupIndex
(
    int      *index,         /* <= designLevels index of group */
    char     *groupName      /* => group name */
);
```

Description The MicroStation structure *designLevels* contains the current level structure. The *designLevels.levels* member points to an array of type `NamedLevel`. This array contains the level names. The *designLevels.groups* member points to an array of type `LevelGroup`. This array contains the level groups.

The `mdlLevel_getGroupIndex` function returns the *designLevels.groups* index in *levelIndex*. This index has the level group specified by *groupID*. This argument contains the unique groupID of the level group.

Returns The `mdlLevel_getGroupIndex` function returns `SUCCESS` if the level group was found in the level structure. Otherwise, it returns `INVALID_GROUP`. `ERROR` is returned if a level structure is not loaded.

See Also `mdlLevel_getGroupIndexFromName`.

mdlLevel_getGroupIndexFromName

```
#include <mdl.h>
#include <levels.h>

int mdlLevel_getGroupIndexFromName
(
    int      *index,          /* <= designLevels index of group */
    char     *groupName      /* => group name */
);
```

Description The MicroStation *designLevels* structure contains the current level structure. The member *designLevels.levels* points to an array of type `NamedLevel` containing the level names. The *designLevels.groups* member points to an array of type `LevelGroup`. This array contains the level groups.

The `mdlLevel_getGroupIndexFromName` function returns the *designLevels-groups* index in *groupIndex*. This index has the level group specified by *groupName*.



A group name is not always unique across a level structure. `mdlLevel_getGroupIndex` uniquely specifies a level group if the level group's group ID is known.

Returns The `mdlLevel_getGroupIndexFromName` function returns `SUCCESS` if the level group was found in the level structure. `DUPLICATE_GROUP` is returned if the level group name is not unique, and `INVALID_GROUP` is returned if the level group is not found. `ERROR` is returned if a level structure is not loaded.

See Also `mdlLevel_getGroupIndex`.

mdlLevel_getGroupNameFromID

```
#include <mdl.h>
#include <levels.h>

int mdlLevel_getGroupNameFromID
(
    char    *groupName,    /* <= name of group */
    int     groupID        /* => ID of group */
);
```

Description The mdlLevel_getGroupNameFromID function finds the descriptive name of a level group. *groupName* receives the level group name. The level group's unique groupID is passed in *groupID*.

Returns The mdlLevel_getGroupNameFromID function returns SUCCESS if the level group was found. INVALID_GROUP is returned if the level group was not found in the level structure.

See Also mdlLevel_getGroupIDFromNameAndParent.

mdlLevel_getGroupIDFromNameAndParent

```
#include <mdl.h>
#include <levels.h>

int mdlLevel_getGroupIDFromNameAndParent
(
    int     *groupID,      /* <= ID of group */
    char    *groupName,    /* => name of group */
    int     parentID       /* => ID of parent group */
);
```

Description The mdlLevel_getGroupIDFromNameAndParent function finds the level group's unique group ID if the level group's name and the parent group's group ID are known. *groupID* receives the group ID of the level group. *groupName* and *parentID* specify the level group name and parent group's group ID.

Returns The mdlLevel_getGroupIDFromNameAndParent function returns SUCCESS if the group ID was found. INVALID_GROUP is returned if the level group corresponding to *groupName* and *parentID* could not be found.

See Also mdlLevel_getGroupNameFromID.

mdlLevel_freeAll

```
#include <mdl.h>
#include <levels.h>

void mdlLevel_freeAll(void);
```

Description mdlLevel_freeAll clears the current level structure and releases all memory allocated by MicroStation for the level structure. mdlLevel_freeAll has no

arguments. No level names or level groups are defined after `mdlLevel_freeAll` is called.



`mdlLevel_freeAll` calls `mdlLevel_freeLevels` and `mdlLevel_freeGroups` to free the current level structure.

Returns The `mdlLevel_freeAll` function is of type `void`. It returns no value.

See Also `mdlLevel_freeLevels`, `mdlLevel_freeGroups`.

mdlLevel_freeLevels

```
#include <mdl.h>
#include <levels.h>

void mdlLevel_freeLevels(void);
```

Description The `mdlLevel_freeLevels` function clears the current level names structures and releases all memory allocated by MicroStation for the level names. `mdlLevel_freeLevels` has no arguments. No level names are defined after `mdlLevel_freeLevels` is called.



The level group definitions are not cleared; they remain active.

Returns The `mdlLevel_freeLevels` function is of type `void`. It returns no value.

See Also `mdlLevel_freeAll`, `mdlLevel_freeGroups`.

mdlLevel_freeGroups

```
#include <mdl.h>
#include <levels.h>

void mdlLevel_freeGroups(void);
```

Description The `mdlLevel_freeGroups` function clears the current level group structures and releases all memory allocated by MicroStation for the level groups. `mdlLevel_freeGroups` has no arguments. No level groups are defined after `mdlLevel_freeGroups` is called.



The level names definitions are not cleared; they remain active.

Returns The `mdlLevel_freeGroups` function is of type `void`. It returns no value.

See Also `mdlLevel_freeAll`, `mdlLevel_freeLevels`.

mdlLevel_loadAllFromResource

```
#include <mdl.h>
#include <levels.h>

void mdlLevel_loadAllFromResource
(
char    *resourceFile /* => resource file name */
);
```

Description The `mdlLevel_loadAllFromResource` function loads the level name and level group definitions from an external resource file. The full resource file specification is passed in *resourceFile*.

Loading a level structure from an external resource file overrides the level structure contained in the design file.



`mdlLevel_loadAllFromResource` calls `mdlLevel_loadLevelsFromResource` and `mdlLevel_loadGroupsFromResource` to load the level structure.

Returns The `mdlLevel_loadAllFromResource` function returns `SUCCESS` if the level structures were loaded successfully. Otherwise, it returns `ERROR`.

See Also `mdlLevel_loadLevelsFromResource`, `mdlLevel_loadGroupsFromResource`.

mdlLevel_loadGroupsFromResource

```
#include <mdl.h>
#include <levels.h>

void mdlLevel_loadGroupsFromResource
(
char    *resourceFile /* => resource file name */
);
```

Description The `mdlLevel_loadGroupsFromResource` function loads only the level group definitions from an external resource file. The full file specification of the resource file is passed in *resourceFile*.

Loading level group definitions from an external resource file overrides the level group definitions contained in the design file.



The `mdlLevel_loadGroupsFromResource` function loads only the level group definitions from the resource file. The existing level name definitions remain active.

Returns The `mdlLevel_loadGroupsFromResource` function returns `SUCCESS` if the level group definitions were loaded successfully. Otherwise, it returns `ERROR`.

See Also `mdlLevel_loadAllFromResource`, `mdlLevel_loadLevelsFromResource`.

mdlLevel_loadLevelsFromResource

```
#include <mdl.h>
#include <levels.h>

void mdlLevel_loadLevelsFromResource
(
char    *resourceFile /* => resource file name */
);
```

Description The `mdlLevel_loadLevelsFromResource` function loads only the level name definitions from an external resource file. The full resource file specification is passed in *resourceFile*.

Loading level name definitions from an external resource file overrides the level name definitions contained in the design file.



The `mdlLevel_loadLevelsFromResource` function loads only the level name definitions from the resource file. The existing level group definitions remain active.

Returns The `mdlLevel_loadLevelsFromResource` function returns `SUCCESS` if the level name definitions were loaded successfully. Otherwise, it returns `ERROR`.

See Also `mdlLevel_loadAllFromResource`, `mdlLevel_loadGroupsFromResource`.

mdlLevel_saveAllToResource

```
#include <mdl.h>
#include <levels.h>

void mdlLevel_saveAllToResource
(
char    *resourceFile /* => resource file name */
);
```

Description The `mdlLevel_saveAllToResource` function writes the level name and level group definitions to an external resource file. The full resource file specification is passed in *resourceFile*.



The `mdlLevel_saveAllToResource` function calls `mdlLevel_saveLevelsToResource` and `mdlLevel_saveGroupsToResource` to save the level structure.

Returns The `mdlLevel_saveAllToResource` function returns `SUCCESS` if the level structures were written successfully. Otherwise, it returns `ERROR`.

See Also `mdlLevel_saveLevelsToResource`, `mdlLevel_saveGroupsToResource`.

mdlLevel_saveGroupsToResource

```
#include <mdl.h>
#include <levels.h>

void mdlLevel_saveGroupsToResource
(
char    *resourceFile /* => resource file name */
);
```

Description The `mdlLevel_saveGroupsToResource` function writes only the level group definitions to an external resource file. The full resource file specification is passed in *resourceFile*.



The `mdlLevel_saveGroupsToResource` function writes only the level group definitions to the resource file. The existing level name definitions are not saved.

Returns The `mdlLevel_saveGroupsToResource` function returns `SUCCESS` if the level group definitions were written successfully. Otherwise, it returns `ERROR`.

See Also `mdlLevel_saveAllToResource`, `mdlLevel_saveLevelsToResource`.

mdlLevel_saveLevelsToResource

```
#include <mdl.h>
#include <levels.h>

void mdlLevel_saveLevelsToResource
(
char    *resourceFile /* => resource file name */
);
```

Description The `mdlLevel_saveLevelsToResource` function writes only the level name definitions to an external resource file. The full resource file specification is passed in *resourceFile*.



The `mdlLevel_saveLevelsToResource` function writes only the level name definitions to the resource file. The existing level group definitions are not saved.

Returns The `mdlLevel_saveLevelsToResource` function returns `SUCCESS` if the level name definitions were written successfully. Otherwise, it returns `ERROR`.

See Also `mdlLevel_saveAllToResource`, `mdlLevel_saveGroupsToResource`.

mdlLevel_getLevelName, mdlLevel_getLevelNameByFile

```

#include <mslevel.fdf>
#include <levels.h>

int mdlLevel_getLevelName
(
    char    *name,          /* <= name of level */
    int     level           /* => IGDS level number */
);

int mdlLevel_getLevelNameByFile
(
    char    *name,          /* <= name of level */
    int     level,          /* => IGDS level number */
    int     fileNumber      /* file number */
);

```

Description The `mdlLevel_getLevelName` function retrieves a name associated with the specified level. Using this function is superior to the other techniques for finding this information both because of the simplicity, and because it is the only technique that takes into consideration the names used by the user. If a level has multiple names, it can be very confusing to the user if the software randomly uses the available names for that level. For example, consider what happens if a level has the names *mud* and *water*. If the user sets the active level to *mud*, places an element, and then tries to move that element, MicroStation's description of the element should say the element is on level *mud*, not level *water*. Likewise, if the user has set the active level to *water* prior to placing the element, MicroStation's description of the element should say it is on level *water*. To provide this functionality, MicroStation remembers the most recently used name for each level. `mdlLevel_getLevelName` always returns the most recently used name for any level. The `mdlLevel_getLevelNameByFile` functions operates like `mdlLevel_getLevelName`, except it uses the level names from the specified file if the names are available. `mdlLevel_getLevelName` is implemented as a call to `mdlLevel_getLevelNameByFile` passing 0 for `fileNumber`.

`mdlLevel_getLevelNameByFile` first tries to use level names defined for the specified file. If the specified file does not have a name specified for that level, `mdlLevel_getLevelNameByFile` tries to find a name in the definitions associated with the main design file.

name points to a buffer to receive the level name. The buffer must be at least `LEVEL_NAME_WIDTH + 1` bytes long.

level is a level number.

fileNumber is a file number. It is either 0 for the master design file or a reference file number.



These functions were implemented in MicroStation 95.

Returns mdlLevel_getLevelName and mdlLevel_getLevelNameByFile return SUCCESS if there is a name defined for that level or ERROR if there is not a name defined for that level.

mdlLevelSymbology_extract

```
#include <levels.h>
#include <mslevel.fdf>
#include <mdlerrs.h>

int mdlLevelSymbology_extract
(
    int      *colorP,          /* <= color (-1 if no override) */
    int      *styleP,         /* <= style (-1 if no override) */
    int      *weightP,        /* <= weight (-1 if no override) */
    int      level,           /* => level */
    int      fileNumber       /* => file number */
);
```

Description The mdlLevelSymbology_extract function extracts the level symbology for a given level of the master design file or a reference file.

If *colorP* is not NULL, the location it points to receives either the color specified via level symbology, or -1 if override is not set for the color level symbology.

If *styleP* is not NULL, the location it points to receives either the style specified via level symbology, or -1 if override is not set for the style level symbology.

If *weightP* is not NULL, the location it points to receives either the line weight specified via level symbology, or -1 if override is not set for the weight level symbology.

level specifies a MicroStation level.

fileNumber is 0 for the main design file, or the number of a reference file.



This function was implemented in MicroStation 95.

Returns mdlLevelSymbology_extract returns SUCCESS if *fileNumber* is valid. MDLERR_BADFILENAME is returned otherwise.

20

Math

This chapter describes functions with a mathematical orientation - those used for changing the orientation of objects in space, converting between coordinate systems, and computing vector distances, cross-products and dot-products.

The sections in this chapter are:

- Rotation matrix functions
- Transformation matrix functions
- Vector functions

Rotation Matrix Functions

Rotation matrices are used extensively in MicroStation to describe the orientation of objects in space. In MDL, they are referenced using the `typedef RotMatrix` defined in `basetype.h`. `RotMatrix` is a square 2 x 2 matrix for the 2D files and a 3 x 3 matrix for 3D files.

Many internal MDL functions that expect a `RotMatrix` as a parameter expect the matrix to be **orthonormal**. An orthonormal matrix is referred to as a **pure** rotation matrix; a matrix whose transpose is equal to its inverse. Applications should ensure that the matrix is orthonormal where noted, by using the `mdlRMMatrix_isOrthonormal [mdl.lib.m1]` function.

Some internal MDL functions expect `RotMatrix` to be **orthogonal**. An orthogonal matrix is rotation matrix that has been scaled but not skewed. In other words, perpendicular lines in an orthogonal matrix maintain a 90° angle, though their lengths can be altered.

MDL functions that require both a rotation and a scaling component use `Transform` rather than `RotMatrix`.

Function	Used to
<code>mdlRMMatrix_getIdentity</code>	set a rotation matrix to the identity matrix.
<code>mdlRMMatrix_isIdentity</code>	query if a rotation matrix is the identity matrix.
<code>mdlRMMatrix_isOrthonormal [mdlLib.m]</code>	query if a rotation matrix is orthonormal.
<code>mdlRMMatrix_multiply</code>	set the rotation matrix to a product.
<code>mdlRMMatrix_invert</code>	set the rotation matrix to the transpose of the rotation matrix.
<code>mdlRMMatrix_getInverse</code>	get the inverse of a square matrix.
<code>mdlRMMatrix_rotate</code>	rotate the rotation matrix.
<code>mdlRMMatrix_multiplyByTMatrix</code>	set the rotation matrix to the product of the transformation matrix and the rotation matrix.
<code>mdlRMMatrix_normalize</code>	normalize the columns of a rotation matrix and put the result in another rotation matrix.
<code>mdlRMMatrix_from3Points</code>	generate a rotation matrix from three points that define a plane.
<code>mdlRMMatrix_fromView</code>	copy the rotation matrix for a view to the rotation matrix pointed to by <i>outRMMatrix</i> .
<code>mdlRMMatrix_getColumnVector</code>	extract a column from the rotation matrix and store it in a vector.
<code>mdlRMMatrix_getRowVector</code>	extract a row from the rotation matrix and store it in a vector.
<code>mdlRMMatrix_fromColumnVectors</code> <code>mdlRMMatrix_fromRowVectors</code>	create a rotation matrix based on vectors that define either columns or rows.
<code>mdlRMMatrix_fromAngle</code>	generate a rotation matrix from the angle given by <i>angle</i> in radians.
<code>mdlRMMatrix_fromQuat</code>	generate a rotation matrix from the quaternion pointed to by <i>quat</i> .
<code>mdlRMMatrix_toAngle</code>	return the rotation angle for a 2D rotation matrix
<code>mdlRMMatrix_toQuat</code>	generate a quaternion matrix from a 3D rotation matrix

Function	Used to
<code>mdlRMatrix_fromTMatrix</code>	extract the rotation and scaling portions of a transformation matrix and copy them to <i>rMatrix</i> .
<code>mdlRMatrix_fromNormalVector</code>	generate a (somewhat arbitrary) rotation matrix based on direction vector.
<code>mdlRMatrix_rotatePoint</code>	rotate a point by the rotation matrix pointed to by <i>rMatrix</i> .
<code>mdlRMatrix_rotatePointArray</code>	rotate an array of points by the rotation matrix pointed to by <i>rMatrix</i> .
<code>mdlRMatrix_unrotatePoint</code>	unrotate a point.
<code>mdlRMatrix_unrotatePointArray</code>	unrotate an array of points.

Example

See `dynamic.mc`.

mdlRMatrix_getIdentity

```
#include <mdl.h>

void mdlRMatrix_getIdentity
(
    RotMatrix    *rMatrix      /* <= rotation matrix */
);
```

Description `mdlRMatrix_getIdentity` sets the rotation matrix pointed to by *rMatrix* to the identity matrix.

Returns `mdlRMatrix_getIdentity` is of type `void`. It returns no value.

See Also `mdlRMatrix_isIdentity`.

mdlRMatrix_isIdentity

```
int mdlRMatrix_isIdentity
(
    RotMatrix    *rotMatrixP,  /* => rotation matrix */
    int          dimension     /* => dimension (2 or 3) */
);
```

Description The `mdlRMatrix_isIdentity` function returns `TRUE` if the rotation matrix represents identity. An identity rotation matrix specifies no rotation.

rotMatrixP is a pointer to a rotation matrix.

dimension is either the rotation matrix dimension (either 2 or 3).

Returns `mdlRMMatrix_isIdentity` returns `TRUE` if the rotation matrix is identity.

See Also `mdlRMMatrix_getIdentity`.

mdlRMMatrix_isOrthonormal [mdl.lib.mdl]

```
boolean mdlRMMatrix_isOrthonormal
(
  RotMatrix  *rotMatrixP,    /* => Rotation matrix */
  int        dimension       /* => matrix dimension (2 or 3) */
);
```

Description The `mdlRMMatrix_isOrthonormal` function queries if the rotation matrix pointed to by *rotMatrixP* is orthonormal. An orthonormal rotation matrix specifies only rotation, no scaling or shear is present. *dimension* is the rotation matrix dimension; it should be either 2 or 3.

Returns `mdlRMMatrix_isOrthonormal` returns `TRUE` if the rotation matrix is orthonormal.

mdlRMMatrix_multiply

```
#include <mdl.h>

void mdlRMMatrix_multiply
(
  RotMatrix  *outRMMatrix,    /* <= */
  RotMatrix  *rMatrix1,       /* => */
  RotMatrix  *rMatrix2        /* => */
);
```

Description `mdlRMMatrix_multiply` sets the rotation matrix pointed to by *outRMMatrix* to the product of *rMatrix1* and *rMatrix2*.



outRMMatrix can be the same as *rMatrix1* or *rMatrix2*.

Returns `mdlRMMatrix_multiply` is of type `void`. It returns no value.

mdlRMMatrix_invert, mdlRMMatrix_getInverse

```
#include <mdl.h>

void mdlRMMatrix_invert
(
  RotMatrix  *outRMMatrix,    /* <= */
  RotMatrix  *rMatrix1,       /* => */
  RotMatrix  *rMatrix2        /* => */
);
```

```
RotMatrix    *inRMMatrix1    /* => */
);

void mdlRMMatrix_getInverse
(
RotMatrix    *outRMMatrix,    /* <= */
RotMatrix    *inRMMatrix1    /* => */
);
```

Description `mdlRMMatrix_invert` sets the rotation matrix pointed to by *outRMMatrix* to the transpose of the rotation matrix pointed to by *inRMMatrix*.

`mdlRMMatrix_getInverse` sets the rotation matrix pointed to by *outRMMatrix* to the inverse of the rotation matrix pointed to by *inRMMatrix*.



`mdlRMMatrix_getInverse` is more computationally intense than `mdlRMMatrix_invert` is, so if a rotation matrix is orthonormal, the transpose is the inverse and `mdlRMMatrix_invert` should be used to obtain the inverse.

Returns `mdlRMMatrix_invert` is of type `void`. It returns no value.

See Also `mdlRMMatrix_isOrthonormal` [mdl1lib.m1].

mdlRMMatrix_rotate

```
#include <mdl.h>

void mdlRMMatrix_rotate
(
RotMatrix    *outRMMatrix,    /* <= */
RotMatrix    *inRMMatrix,    /* => original RMatrix */
double       xAngle,          /* => x rotation angle */
double       yAngle,          /* => y rotation angle */
double       zAngle           /* => z rotation angle */
);
```

Description The `mdlRMMatrix_rotate` function rotates the rotation matrix pointed to by *inRMMatrix* by the angles specified in *xAngle*, *yAngle* and *zAngle* in radians. The result is placed in the rotation matrix pointed to by *outRMMatrix*.

In 3D files, the rotations are applied first about the Z-axis, then about the Y-axis, and finally about the X-axis. If this is not the desired order of rotation, make separate calls to `mdlRMMatrix_rotate`. In 2D files, only *zAngle* is used.



outRMMatrix and *inRMMatrix* can be the same.

Returns The `mdlRMMatrix_rotate` function is of type `void`. It returns no value.

mdlRMatrix_multiplyByTMatrix

```
#include <mdl.h>

void mdlRMatrix_multiplyByTMatrix
(
    RotMatrix    *outRMatrix,    /* <= */
    Transform    *tMatrix,       /* => transformation matrix */
    RotMatrix    *inRMatrix      /* => original RMatrix */
);
```

Description The `mdlRMatrix_multiplyByTMatrix` function sets the rotation matrix pointed to by *outRMatrix* to the product of the transformation matrix pointed to by *tMatrix* and the rotation matrix pointed to by *inRMatrix*.



Since transforms need not be pure orthonormal rotation matrices, `mdlRMatrix_multiplyByTMatrix` can create non-orthonormal rotation matrices. The `mdlRMatrix_normalize` function often needs to be called so *outRMatrix* can be used as a rotation matrix after `mdlRMatrix_multiplyByTMatrix` is called.

outRMatrix and *inRMatrix* can be the same.

Returns The `mdlRMatrix_multiplyByTMatrix` function is of type `void`. It returns no value.

See Also `mdlRMatrix_normalize`, `mdlRMatrix_isOrthonormal` [`mdl.lib.mdl`].

mdlRMatrix_normalize

```
#include <mdl.h>

void mdlRMatrix_normalize
(
    RotMatrix    *outRMatrix,    /* <= */
    RotMatrix    *inRMatrix      /* => */
);
```

Description `mdlRMatrix_normalize` normalizes each column of the rotation matrix pointed to by *inRMatrix* and puts the result in the rotation matrix pointed to by *outRMatrix*.



`mdlRMatrix_normalize` basically negates the effect of any scaling that has been applied to *inRMatrix*. This function is needed only if your rotation matrix becomes scaled.

outRMatrix and *inRMatrix* can be the same.

Returns `mdlRMatrix_normalize` is of type `void`. It returns no value.

mdlRMMatrix_from3Points

```
#include <mdl.h>

void mdlRMMatrix_from3Points
(
    RotMatrix    *outRMMatrix,    /* <= */
    Dpoint3d     *origin,         /* => point 1 */
    Dpoint3d     *xAxis,          /* => point 2 */
    Dpoint3d     *point3          /* => point 3 */
);
```

Description mdlRMMatrix_from3Points generates a rotation matrix from three points that define a plane. The vector from *origin* to *xAxis* defines the plane's X-axis and *point3* defines its orientation.

Returns mdlRMMatrix_from3Points returns SUCCESS if the points define a plane and a rotation matrix is created. It returns ERROR if the three points are co-linear.

mdlRMMatrix_fromView

```
#include <mdl.h>

void mdlRMMatrix_fromView
(
    RotMatrix    *outRMMatrix,    /* <= */
    int          viewNumber,       /* => view number */
    int          allowAuxCoord     /* => if TRUE, use aux coords */
);
```

Description The mdlRMMatrix_fromView function copies the rotation matrix for view *viewNumber* to the rotation matrix pointed to by *outRMMatrix*. If *allowAuxCoord* is TRUE and there is an active ACS, the function copies the ACS's rotation matrix.

Returns The mdlRMMatrix_fromView function is of type void. It returns no value.

mdlRMatrix_getColumnVector, mdlRMatrix_getRowVector

```
#include <mdl.h>

void mdlRMatrix_getColumnVector
(
    Dpoint3d    *outVec,          /* <= column vector */
    RotMatrix    *rMatrix,        /* => rotation matrix */
    int          col              /* => column number */
);

void mdlRMatrix_getRowVector
(
    Dpoint3d    *outVec,          /* <= row vector */
    RotMatrix    *rMatrix,        /* => rotation matrix */
    int          row              /* => row number */
);
```

Description `mdlRMatrix_getColumnVector` extracts column *col* from the rotation matrix pointed to by *rMatrix*. The function then stores *col* in the vector pointed to by *outVec*.

`mdlRMatrix_getRowVector` extracts *row* from the rotation matrix pointed to by *rMatrix*. The function then stores *row* in the vector pointed to by *outVec*.

Returns The `mdlRMatrix_getColumnVector` and `mdlRMatrix_getRowVector` functions are of type `void`. They return no values.

mdlRMatrix_fromColumnVectors, mdlRMatrix_fromRowVectors

```
#include <mdl.h>

void mdlRMatrix_fromColumnVectors
(
    RotMatrix    *outRMatrix,     /* <= */
    Dpoint3d    *col1,           /* => column 1 */
    Dpoint3d    *col2,           /* => column 2 */
    Dpoint3d    *col3            /* => column 3 */
);

void mdlRMatrix_fromRowVectors
(
    RotMatrix    *outRMatrix,     /* <= */
    Dpoint3d    *row1,           /* => row 1 */
    Dpoint3d    *row2,           /* => row 2 */
    Dpoint3d    *row3            /* => row 3 */
);
```

Description `mdlRMatrix_fromColumnVectors` and `mdlRMatrix_fromRowVectors` create a rotation matrix in *outRMatrix* based on vectors that define either the columns or rows.

`mdlRMatrix_fromColumnVectors` creates a rotation matrix with the following format:

$$\text{outRMatrix} = \begin{bmatrix} \text{col1} \rightarrow x & \text{col2} \rightarrow x & \text{col3} \rightarrow x \\ \text{col1} \rightarrow y & \text{col2} \rightarrow y & \text{col3} \rightarrow y \\ \text{col1} \rightarrow z & \text{col2} \rightarrow z & \text{col3} \rightarrow z \end{bmatrix}$$

`mdlRMatrix_fromRowVectors` creates a rotation matrix with the following format:

$$\text{outRMatrix} = \begin{bmatrix} \text{row1} \rightarrow x & \text{row1} \rightarrow y & \text{row1} \rightarrow z \\ \text{row2} \rightarrow x & \text{row2} \rightarrow y & \text{row2} \rightarrow z \\ \text{row3} \rightarrow x & \text{row3} \rightarrow y & \text{row3} \rightarrow z \end{bmatrix}$$

For *outRMatrix* to be a valid (orthonormal) rotation matrix, either the column or row vectors should be normalized before these routines are called or `mdlRMatrix_normalize` should be called for the resultant matrix.

Returns The `mdlRMatrix_fromColumnVectors` and `mdlRMatrix_fromRowVectors` functions are of type `void`. They return no values.

See Also `mdlRMatrix_isOrthonormal` [mdl1lib.m1].

mdlRMatrix_fromAngle, mdlRMatrix_fromQuat

```
#include <mdl.h>

void mdlRMatrix_fromAngle
(
    RotMatrix    *outRMatrix,          /* <= */
    double       angle                 /* => rotation angle */
);

void mdlRMatrix_fromQuat
(
    Dpoint3d     *outRMatrix,          /* <= */
    long         *quat                 /* => quaternion */
);
```

Description The `mdlRMMatrix_fromAngle` function generates a rotation matrix in *outRMMatrix* from the angle given by *angle* in radians.

The `mdlRMMatrix_fromQuat` function generates a rotation matrix in *outRMMatrix* from the quaternion pointed to by *quat*. Quaternions are stored in some 3D elements as a more compact way of representing rotation matrices.



The `mdlRMMatrix_fromAngle` function generates a rotation matrix in the z=0 plane in 3D. `mdlRMMatrix_fromQuat` is valid only in 3D.

Returns The `mdlRMMatrix_fromAngle` and `mdlRMMatrix_fromQuat` functions are of type `void`. They return no values.

mdlRMMatrix_toAngle, mdlRMMatrix_toQuat

```
#include <mdl.h>

double mdlRMMatrix_toAngle
(
    RotMatrix    *rMatrix      /* => rotation matrix */
);

void mdlRMMatrix_toQuat
(
    long         *quat,        /* <= quaternion */
    Dpoint3d     *rMatrix      /* => rotation matrix */
);
```

Description `mdlRMMatrix_toAngle` returns the rotation angle for a 2D rotation matrix.

`mdlRMMatrix_toQuat` generates a quaternion matrix from a 3D rotation matrix.



The `mdlRMMatrix_toAngle` function is valid only in 2D.

Returns The `mdlRMMatrix_toAngle` function returns the angle in radians.
The `mdlRMMatrix_toQuat` function is of type `void`. It returns no value.

mdlRMMatrix_fromTMatrix

```
#include <mdl.h>

void mdlRMMatrix_fromTMatrix
(
    RotMatrix    *outRMMatrix, /* <= rotation matrix */
    Transform     *tMatrix     /* => transformation matrix */
);
```


Description `mdlRMMatrix_fromTMatrix` extracts the rotation and scaling portions of the transformation matrix pointed to by *tMatrix*. The function then copies the rotation and scaling portions to *rMatrix*.



Since transforms need not be pure orthonormal rotation matrices, `mdlRMMatrix_fromTMatrix` can create non-orthonormal rotation matrices or matrices with a non-unity (!=1) determinant. The `mdlRMMatrix_normalize` function often needs to be called so that *outRMMatrix* can be used as a rotation matrix after `mdlRMMatrix_fromTMatrix` is called.

Returns The `mdlRMMatrix_fromTMatrix` function is of type `void`. It returns no value.

See Also `mdlRMMatrix_isOrthonormal` [`mdl1lib.mdl`].

mdlRMMatrix_fromNormalVector

```
#include <mdl.h>

void mdlRMMatrix_fromNormalVector
(
    RotMatrix    *outRMMatrix,      /* <= rotation matrix */
    Dpoint3d     *normalVector      /* => normal vector */
);
```

Description The `mdlRMMatrix_fromNormalVector` function generates a rotation matrix based on a direction vector defining its Z-axis. Since the directions for the X- and Y-axes are not fully constrained, MicroStation arbitrarily assigns them using an algorithm designed to align them as closely with the world coordinate system as possible. The algorithm ensures that while the axes are arbitrary, they are consistent.

Returns The `mdlRMMatrix_fromNormalVector` function is of type `void`. It returns no value.

mdlRMMatrix_rotatePoint, mdlRMMatrix_rotatePointArray, mdlRMMatrix_unrotatePoint, mdlRMMatrix_unrotatePointArray

```
#include <mdl.h>

void mdlRMMatrix_rotatePoint
(
    Dpoint3d     *point,            /* <=> point to rotate */
    RotMatrix     *rMatrix          /* => rotation matrix */
);

void mdlRMMatrix_rotatePointArray
(
    Dpoint3d     *point,            /* <=> array to rotation */
    RotMatrix     *rMatrix,         /* => rotation matrix */
    int          numPoints          /* => number of points */
);

void mdlRMMatrix_unrotatePoint
```

```
(
Dpoint3d      *point,          /* <=> point to rotate */
RotMatrix     *rMatrix         /* => rotation matrix */
);

void mdlRMMatrix_unrotatePointArray
(
Dpoint3d      *point,          /* <=> array to rotation */
RotMatrix     *rMatrix,        /* => rotation matrix */
int           numPoints        /* => number of points */
);
```

Description The `mdlRMMatrix_rotatePoint` function rotates *point* by the rotation matrix pointed to by *rMatrix*.

The `mdlRMMatrix_rotatePointArray` function is identical to `mdlRMMatrix_rotatePoint` except that it rotates an array of *numPoints* points.

The `mdlRMMatrix_unrotatePoint` function rotates *point* by the inverse of the rotation matrix pointed to by *rMatrix*.

The `mdlRMMatrix_unrotatePointArray` function is identical to `mdlRMMatrix_unrotatePoint` except that it unrotates an array of *numPoints* points.

Returns The `mdlRMMatrix_rotatePoint`, `mdlRMMatrix_rotatePointArray`, `mdlRMMatrix_unrotatePoint` and `mdlRMMatrix_unrotatePointArray` functions are of type `void`. They return no values.

Transformation Matrix Functions

Transformation matrices are used extensively in MicroStation to convert between coordinate systems. In MDL they are referenced with the typedef `Transform` defined in `basetype.h`. `Transform` is a union of a 2 x 3 matrix for 2D files and a 3 x 4 matrix for 3D files.

Transformation matrices are generally formed when a rotation, scaling, and translation are applied to the identity matrix. Any MDL functions that require both a rotation and a scaling component use `Transform` rather than `RotMatrix`.

The following table lists transformation matrix functions:

Function	Used to
<code>mdlTMatrix_getIdentity</code>	set the transformation matrix to the identity matrix.
<code>mdlTMatrix_multiply</code>	set the transformation matrix to a product.
<code>mdlTMatrix_rotateByRMatrix</code>	rotate a transformation matrix by a rotation matrix.
<code>mdlTMatrix_rotateByAngles</code>	rotate the transformation matrix by specific angles.
<code>mdlTMatrix_scale</code>	scale the transformation matrix.
<code>mdlTMatrix_translate</code>	shift the origin of the transformation matrix.
<code>mdlTMatrix_setOrigin</code>	set the origin of a transformation matrix to a known point.
<code>mdlTMatrix_setTranslation</code>	set the last column of a translation matrix to given values.
<code>mdlTMatrix_getTranslation</code>	get the last column of a translation matrix.
<code>mdlTMatrix_fromRMatrix</code>	set the square portion of a transformation matrix to a rotation matrix.
<code>mdlTMatrix_transformPoint</code>	transform a point by a transformation matrix.
<code>mdlTMatrix_transformPointArray</code>	transform an array of points by the transformation matrix pointed to by <i>tMatrix</i> .
<code>mdlTMatrix_rotateScalePoint</code>	rotate and scale a point by a transformation matrix.
<code>mdlTMatrix_transpose</code>	swap rows and columns of the transformation matrix.
<code>mdlTMatrix_referenceToMaster</code>	transform elements from reference file to master file coordinate system.
<code>mdlTMatrix_masterToReference</code>	get matrix to transform elements from master file to reference file coordinate system.

Example

See `math.mc` and `trumpet.mc`.

mdlTMatrix_getIdentity

```
#include <mdl.h>

void mdlTMatrix_getIdentity
(
    Transform    *tMatrix
);
```

Description The `mdlTMatrix_getIdentity` function sets the transformation matrix pointed to by *tMatrix* to the identity matrix.

Returns The `mdlTMatrix_getIdentity` function is of type `void`. It returns no value.

mdlTMatrix_multiply

```
#include <mdl.h>

void mdlTMatrix_multiply
(
    Transform    *outTMatrix,    /* <= T1 x T2 */
    Transform    *tMatrix1,      /* => T1 */
    Transform    *tMatrix2       /* => T2 */
);
```

Returns The `mdlTMatrix_multiply` function sets the transformation matrix pointed to by *outTMatrix* to the product of *tMatrix1* and *tMatrix2*.



outTMatrix can point to the same memory as *tMatrix1* or *tMatrix2*.

Returns The `mdlTMatrix_multiply` function is of type `void`. It returns no value.

mdlTMatrix_rotateByRMatrix

```
#include <mdl.h>

void mdlTMatrix_rotateByRMatrix
(
    Transform    *outTMatrix,    /* <= */
    Transform    *inTMatrix,     /* => orig transformation matrix */
    RotMatrix    *rMatrix        /* => rotation matrix */
);
```

Returns The `mdlTMatrix_rotateByRMatrix` function rotates the transformation matrix pointed to by *inTMatrix* by the rotation matrix pointed to by *rMatrix*. The result is placed in the transformation matrix pointed to by *outTMatrix*.



outTMatrix and *inTMatrix* can have the same values.

Returns The `mdlTMatrix_rotateByRMatrix` function is of type `void`. It returns no value.

mdlTMatrix_rotateByAngles

```
#include <mdl.h>

void mdlTMatrix_rotateByAngles
(
    Transform    *outTMatrix,    /* <= */
    Transform    *inTMatrix,     /* => original matrix */
    double       xAngle,         /* => x angle */
    double       yAngle,         /* => y angle */
    double       zAngle          /* => z angle */
);
```

Returns The mdlTMatrix_rotateByAngles function rotates the transformation matrix pointed to by *inTMatrix* by the angles specified in *xAngle*, *yAngle* and *zAngle* in radians. The result is placed in the transformation matrix pointed to by *outTMatrix*.

In 3D files, the rotations are applied first about the Z-axis, then about the Y-axis, and finally about the X-axis. If this is not the desired order of rotation, make separate calls to mdlTMatrix_rotateByAngles. In 2D files, only *zAngle* is used.



outTMatrix and *inTMatrix* can have the same values.

Returns The mdlTMatrix_rotateByAngles function is of type void. It returns no value.

mdlTMatrix_scale

```
#include <mdl.h>

void mdlTMatrix_scale
(
    Transform    *outTMatrix,    /* <= */
    Transform    *inTMatrix,     /* => original matrix */
    double       xScale,         /* => x scale */
    double       yScale,         /* => y scale */
    double       zScale          /* => z scale */
);
```

Returns The mdlTMatrix_scale function scales the transformation matrix pointed to by *inTMatrix* by the scale factors *xScale*, *yScale* and *zScale*. The result is placed in the transformation matrix pointed to by *outTMatrix*.



outTMatrix and *inTMatrix* can have the same values.

Returns The mdlTMatrix_scale function is of type void. It returns no value.

mdlTMatrix_translate, mdlTMatrix_setOrigin, mdlTMatrix_setTranslation, mdlTMatrix_getTranslation

```
#include <mdl.h>

void mdlTMatrix_translate
(
    Transform    *outTMatrix,    /* <= */
    Transform    *inTMatrix,     /* => original matrix */
    double       xDist,          /* => x distance */
    double       yDist,          /* => y distance */
    double       zDist           /* => z distance */
);

void mdlTMatrix_setOrigin
(
    Transform    *tMatrix,       /* <=> */
    Dpoint3d     *point          /* => new origin */
);

void mdlTMatrix_setTranslation
(
    Transform    *tMatrix,       /* <=> */
    Dpoint3d     *point          /* => translation */
);

void mdlTMatrix_getTranslation
(
    Dpoint3d     *point,         /* <= translation from tMatrix */
    Transform    *tMatrix        /* => transformation matrix */
);
```

Returns The `mdlTMatrix_translate` function shifts the origin of the transformation matrix pointed to by *inTMatrix* by the distances given by *xDist*, *yDist* and *zDist*. The result is placed in the transformation matrix pointed to by *outTMatrix*. *outTMatrix* and *inTMatrix* can have the same values.

The `mdlTMatrix_setOrigin` function sets the origin of the transformation matrix pointed to by *tMatrix* to the coordinate given by *point*.

The `mdlTMatrix_setTranslation` function sets the translation column of the transform *tMatrix* to the values specified in *point*.

The `mdlTMatrix_getTranslation` function returns the translation column of the transform *tMatrix* into *point*.

Returns The `mdlTMatrix_translate`, `mdlTMatrix_setOrigin`, `mdlTMatrix_setTranslation`, and `mdlTMatrix_getTranslation` functions are of type `void`. They return no values.

mdlTMatrix_fromRMatrix

```
#include <mdl.h>

void mdlTMatrix_fromRMatrix
(
    Transform    *outTMatrix,    /* <= */
    RotMatrix    *inRMatrix     /* => */
);
```

Returns The `mdlTMatrix_fromRMatrix` function sets the square portion of the transformation matrix pointed to by *outTMatrix* to the rotation matrix pointed to by *inRMatrix*.



The translation portion of *outTMatrix* is unaffected.

Returns The `mdlTMatrix_fromRMatrix` function is of type `void`. It returns no value.

mdlTMatrix_transformPoint, mdlTMatrix_transformPointArray, mdlTMatrix_rotateScalePoint

```
#include <mdl.h>

void mdlTMatrix_transformPoint
(
    Dpoint3d     *point,         /* <=> point to be transformed */
    Transform    *tMatrix       /* => transformation matrix */
);

void mdlTMatrix_transformPointArray
(
    Dpoint3d     *point,         /* <=> points to be transformed */
    Transform    *tMatrix,       /* => transformation matrix */
    int          numPoints       /* => number of points */
);

void mdlTMatrix_rotateScalePoint
(
    Dpoint3d     *point,         /* <=> point to be transformed */
    Transform    *tMatrix       /* => transformation matrix */
);
```

Returns The `mdlTMatrix_transformPoint` function transforms *point* by the transformation matrix pointed to by *tMatrix*.

The `mdlTMatrix_transformPointArray` function is identical to `mdlTMatrix_transformPoint` except it transforms an array of *numPoints* points.

The `mdlTMatrix_rotateScalePoint` function rotates and scales *point* by the transformation matrix pointed to by *tMatrix*. It does not translate *point* by the translation portion of *tMatrix*.

Returns The `mdlTMatrix_transformPoint`, `mdlTMatrix_transformPointArray`, and `mdlTMatrix_rotateScalePoint` functions are of type `void`. They return no values.

mdlTMatrix_transpose

```
#include <mdl.h>

void mdlTMatrix_transpose
(
    Transform    *outTMatrix,    /* <= */
    Transform    *inTMatrix     /* => */
);
```

Returns The `mdlTMatrix_transpose` function swaps rows and columns of the square portion of a transformation matrix *inTMatrix* into the transformation matrix *outTMatrix*. It does not change the translation column.



The transpose of a transformation matrix is rarely useful unless the transform is a pure orthonormal rotation matrix.

Returns The `mdlTMatrix_transpose` function is of type `void`. It returns no value.

mdlTMatrix_referenceToMaster, mdlTMatrix_masterToReference

```
#include <mdl.h>

void mdlTMatrix_referenceToMaster
(
    Transform    *outTMatrix,    /* <= */
    int          fileNumber      /* => file number */
);

void mdlTMatrix_masterToReference
(
    Transform    *outTMatrix,    /* <= */
    int          fileNumber      /* => file number */
);
```

Returns The `mdlTMatrix_referenceToMaster` function returns a transformation matrix in *outTMatrix*. This matrix can be used to transform elements in reference file *fileNumber* to the master design file's coordinate system.

The `mdlTMatrix_masterToReference` function returns a transformation matrix in *outTMatrix*. This matrix can be used to transform elements in the master file to the reference file *fileNumber*'s coordinate system.

Returns The `mdlTMatrix_referenceToMaster` and `mdlTMatrix_masterToReference` functions are of type `void`. They return no values.

Returns `mdlTMatrix_transformPoint`, `mdlTMatrix_transformPointArray`.

Vector Manipulation Functions

Vector manipulation functions provide general-purpose mathematical functions for computing distances, directions, cross products and dot products for vectors. Vector manipulation functions generally accept either points or direction vectors, both of which use the typedef `Dpoint3d` defined in `basetype.h`.

The following table lists vector manipulation functions:

Function	Used to
<code>mdlVec_distance</code>	return the magnitude of a vector.
<code>mdlVec_magnitude</code>	return the magnitude of a direction vector.
<code>mdlVec_areParallel</code>	test whether one direction vector is parallel to another
<code>mdlVec_arePerpendicular</code>	test whether one direction vector is perpendicular to another
<code>mdlVec_crossProduct</code>	set a direction vector to the cross product of two other direction vectors.
<code>mdlVec_dotProduct</code>	return the dot product of two direction vectors.
<code>mdlVec_pointEqual</code>	compare two points for exact equality
<code>mdlVec_pointEqualUOR</code>	compare two points for equality within one UOR.
<code>mdlVec_colinear</code>	test whether the three points in the array pointed to by <i>points</i> are colinear.
<code>mdlVec_addPoint</code>	add two points.
<code>mdlVec_addPointArray</code>	add a point to each point in an array.
<code>mdlVec_subtractPoint</code>	subtract two points.
<code>mdlVec_subtractPointArray</code>	subtract a point from each point in an array.
<code>mdlVec_normalize</code>	normalize a direction vector so that its magnitude is 1.
<code>mdlVec_computeNormal</code>	calculate a normalized direction vector.
<code>mdlVec_extractPolygonNormal</code>	get the normal vector and average position of a polygon.
<code>mdlVec_scale</code>	scale a direction vector.
<code>mdlVec_intersect</code>	calculate the intersection point of two infinite lines.
<code>mdlVec_pointOffDesignPlane</code>	query whether a given point is on or off of the design plane.

Function	Used to
<code>mdlVec_forcePlanarity</code>	project a point to the closest point on a plane defined by a series of points.
<code>mdlVec_projectPoint</code>	project a point in a given direction for a specified distance.
<code>mdlVec_linePlaneIntersect</code>	get intersection of line and plane.
<code>mdlVec_planePlaneIntersect</code>	get intersection of two planes.
<code>mdlVec_projectPointToPlaneInView</code>	projects a point to a plane in a given view.
<code>mdlVec_projectPointToLineInView</code>	projects a point to a line in a given view.

Example

See `math.mc`.

mdlVec_distance, mdlVec_magnitude

```
#include <mdl.h>

double mdlVec_distance
(
    Dpoint3d    *point1,      /* => point 1 */
    Dpoint3d    *point2      /* => point 2 */
)

double mdlVec_magnitude
(
    Dpoint3d    *dVec         /* => direction vector */
);
```

Description `mdlVec_distance` returns the magnitude of the vector from *point1* to *point2*.

`mdlVec_magnitude` returns the magnitude of the direction vector *dVec*.

Returns The `mdlVec_distance` and `mdlVec_magnitude` functions return a `double`.

mdlVec_areParallel, mdlVec_arePerpendicular

```
#include <mdl.h>

boolean mdlVec_areParallel
(
```

```

Dpoint3d    *dVec1,    /* => direction vector 1 */
Dpoint3d    *dVec2    /* => direction vector 2 */
);

boolean mdlVec_arePerpendicular
(
Dpoint3d    *dVec1,    /* => direction vector 1 */
Dpoint3d    *dVec2    /* => direction vector 2 */
);

```

Description `mdlVec_areParallel` tests to determine whether the direction vector pointed to by *dVec1* is parallel to the direction vector pointed to by *dVec2*.

`mdlVec_arePerpendicular` tests to determine whether the direction vector pointed to by *dVec1* is perpendicular to the direction vector pointed to by *dVec2*.

Returns The `mdlVec_areParallel` and `mdlVec_arePerpendicular` functions return `TRUE` if the test is satisfied and `FALSE` if not.

mdlVec_crossProduct, mdlVec_dotProduct

```

#include <mdl.h>

void mdlVec_crossProduct
(
Dpoint3d    *product, /* <= cross product direction vector */
Dpoint3d    *dVec1,   /* => direction vector 1 */
Dpoint3d    *dVec2    /* => direction vector 2 */
);

double mdlVec_dotProduct
(
Dpoint3d    *dVec1,   /* => direction vector 1 */
Dpoint3d    *dVec2    /* => direction vector 2 */
);

```

Description The `mdlVec_crossProduct` function sets the direction vector pointed to by *product* to the cross product of the direction vectors pointed to by *dVec1* and *dVec2*.

The `mdlVec_dotProduct` function returns the dot product of the direction vectors pointed to by *dVec1* and *dVec2*.

Returns The `mdlVec_crossProduct` function is of type `void`. It returns no value.

The `mdlVec_dotProduct` function returns the dot product as a `double`.

mdlVec_pointEqual, mdlVec_pointEqualUOR, mdlVec_colinear

```
#include <mdl.h>

boolean mdlVec_pointEqual
(
  Dpoint3d    *point1,      /* => point 1 */
  Dpoint3d    *point2,      /* => point 2 */
);

boolean mdlVec_pointEqualUOR
(
  Dpoint3d    *point1,      /* => point 1 */
  Dpoint3d    *point2,      /* => point 2 */
);

boolean mdlVec_colinear
(
  Dpoint3d    points[3]     /* => points */
);
```

Description The `mdlVec_pointEqual` function compares *point1* and *point2* for exact equality. In 2D it ignores the Z components.

The `mdlVec_pointEqualUOR` function compares *point1* and *point2* for equality within one UOR.

The `mdlVec_colinear` function tests to determine whether the three points in the array pointed to by *points* are colinear.

Returns The `mdlVec_pointEqual`, `mdlVec_pointEqualUOR` and `mdlVec_colinear` functions return TRUE if the test is satisfied and FALSE if not.

mdlVec_addPoint, mdlVec_addPointArray, mdlVec_subtractPoint, mdlVec_subtractPointArray

```
#include <mdl.h>

void mdlVec_addPoint
(
  Dpoint3d    *sum,          /* <= direction vector */
  Dpoint3d    *point1,       /* => direction vector 1 */
  Dpoint3d    *point2,       /* => direction vector 2 */
);

void mdlVec_addPointArray
(
  Dpoint3d    *array,        /* <=> */
  Dpoint3d    *point,        /* => point array */
  int         numPoints      /* => number of points */
);

void mdlVec_subtractPoint
```

```
(
Dpoint3d    *diff,          /* <= */
Dpoint3d    *point1,        /* => point 1 */
Dpoint3d    *point2         /* => point 2 */
);

void mdlVec_subtractPointArray
(
Dpoint3d    *array,          /* <=> */
Dpoint3d    *point,          /* => point array */
int          numPoints       /* => number of points */
);
```

Description The `mdlVec_addPoint` function adds *point1* and *point2* and stores the result in *sum*.

The `mdlVec_addPointArray` function adds *point* to each point in the array pointed to by *array*. *numPoints* indicates the number of points in the array.

The `mdlVec_subtractPoint` function subtracts *point2* from *point1* and stores the result in *diff*.

The `mdlVec_subtractPointArray` function subtracts *point* from each point in the array pointed to by *array*. *numPoints* indicates the number of points in the array.

Returns The `mdlVec_addPoint`, `mdlVec_addPointArray`, `mdlVec_subtractPoint` and `mdlVec_subtractPointArray` functions are of type `void`. They return no values.

mdlVec_normalize, mdlVec_computeNormal

```
#include <mdl.h>

void mdlVec_normalize
(
Dpoint3d    *dVec           /* <=> direction vector */
);

void mdlVec_computeNormal
(
Dpoint3d    *outDVec,        /* <= normal vector */
Dpoint3d    *point1,         /* => point 1 */
Dpoint3d    *point2          /* => point 2 */
);
```

Description The `mdlVec_normalize` function normalizes the direction vector pointed to by *dVec* so that its magnitude is 1.0.

The `mdlVec_computeNormal` function calculates a normalized direction vector from *point2* to *point1*. The result is stored in *outDVec*.

Returns The `mdlVec_normalize` and `mdlVec_computeNormal` functions are of type `void`. They return no values.

mdlVec_extractPolygonNormal

```

Public void mdlVec_extractPolygonNormal
(
Dpoint3d      *normalP,      /* <= normal of polygon */
Dpoint3d      *positionP,    /* <= center of polygon */
Dpoint3d      *vertices,
int           nVertices
);

```

Description The `mdlVec_extractPolygonNormal` function returns the normal vector (in *normalP*) and average position (in *positionP*) for the polygon specified by *vertices* and *nVertices*.

Returns `mdlVec_extractPolygonNormal` is of type `void`; it returns no value.

mdlVec_scale

```

#include <mdl.h>

void mdlVec_scale
(
Dpoint3d      *outDVec,      /* <= scaled vector */
Dpoint3d      *inDVec,      /* => direction vector */
double        scale          /* => scale factor */
);

```

Description The `mdlVec_scale` function scales the direction vector pointed to by *inDVec* by *scale*. The result is stored in the direction vector pointed to by *outDVec*.

Returns The `mdlVec_scale` function is of type `void`. It returns no value.

mdlVec_intersect

```

#include <mdl.h>

int mdlVec_intersect
(
Dpoint3d      *intersectPoint, /* <= intersection point */
DVector3d     *vector1,       /* => vector 1 */
DVector3d     *vector2       /* => vector 2 */
);

```

Description The `mdlVec_intersect` function calculates the intersection point of two infinite lines defined by vectors *vector1* and *vector2*. The intersection is returned in *intersectPoint* and is not necessarily within the given line segments.

Returns `mdlVec_intersect` returns `SUCCESS` if the two lines intersect and `-1` if they do not.

mdlVec_pointOffDesignPlane [mdlLib.mll]

```
boolean mdlVec_pointOffDesignPlane
(
  Dpoint3d    *point    /* => point to check */
);
```

Description The mdlVec_pointOffDesignPlane function queries whether a given point is on or off of the design plane. *point* is the point being queried.

Returns mdlVec_pointOffDesignPlane returns TRUE if *point* is off the design plane.

mdlVec_forcePlanarity, mdlVec_projectPoint

```
#include <mdl.h>

void mdlVec_forcePlanarity
(
  Dpoint3d    *outPoint,    /* <= point on plane */
  Dpoint3d    *inPoint,    /* => original point */
  Dpoint3d    *planePoints, /* => points defining plane */
  int         numPoints    /* => # of points in planePoints */
);

void mdlVec_projectPoint
(
  Dpoint3d    *outPoint,    /* <= projected point */
  Dpoint3d    *inPoint,    /* => original point */
  Dpoint3d    *dVec,       /* => direction vector */
  double      distance     /* => distance to project */
);
```

Description The mdlVec_forcePlanarity function projects the point *inPoint* to the closest point on the plane defined by the array of *numPoints* in *planePoints*. mdlVec_forcePlanarity steps through the array until it finds three non-colinear points to define the plane. The new point on the plane is returned in *outPoint*.



If *numPoints* is less than 3 or all *numPoints* of *planePoints* are colinear, *outPoint* is set to *inPoint*.

mdlVec_projectPoint projects the point *inPoint* along the unit direction vector by the distance *distance*. The resulting point is returned in *outPoint*. In other words, the distance from *inPoint* to *outPoint* is *distance* and the direction is *dVec*.

Returns The mdlVec_forcePlanarity and mdlVec_projectPoint functions are of type void. They return no values.

mdlVec_linePlaneIntersect

```
void mdlVec_linePlaneIntersect
(
Dpoint3d      *intersectPt,      /* <= intersection point */
Dpoint3d      *linePt,           /* => point on line */
Dpoint3d      *lineNormal,       /* => normal vector to line */
Dpoint3d      *planePt,          /* => point on plane */
Dpoint3d      *planeNormal,      /* => normal vector to plane */
int            perpendicular      /* => TRUE=return closest pt. */
);
```

Description The `mdlVec_linePlaneIntersect` function returns in *intersectPt* the intersection between the line specified by *linePt* and *lineNormal* and the plane specified by *planePt* and *planeNormal*.

If *perpendicular* is non-zero or the line is parallel to the plane, the closet point on the plane is returned rather than the line-plane intersection.

Returns `mdlVec_linePlaneIntersect` is of type `void`; it returns no value.

See Also `mdlVec_planePlaneIntersect`, `mdlVec_projectPointToPlaneInView`.

mdlVec_planePlaneIntersect

```
int mdlVec_planePlaneIntersect
(
Dpoint3d      *intPnt,           /* <= point on intersection line */
Dpoint3d      *intNorm,          /* <= direction of intersection line */
Dpoint3d      *p1,              /* => point on plane 1 */
Dpoint3d      *n1,              /* => normal to plane 1 */
Dpoint3d      *p2,              /* => point on plane 2 */
Dpoint3d      *n2,              /* => normal to plane 2 */
);
```

Description The `mdlVec_planePlaneIntersect` function computes the intersection line between the two planes defined by *p1*, *n1*, *p2* and *n2*. The intersection line is defined by *intPnt* and *intNorm*.

Returns `mdlVec_planePlaneIntersect` returns `SUCCESS` if the intersection is calculated successfully and `ERROR` otherwise.

See Also `mdlVec_linePlaneIntersect`.

mdlVec_projectPointToPlaneInView

```
void mdlVec_projectPointToPlaneInView
(
Dpoint3d    *projectedPoint,    /* <= projected point */
Dpoint3d    *pointOnPlane,      /* => point on plane */
Dpoint3d    *planeNormal,       /* => normal vector to plane */
Dpoint3d    *point,             /* => point to project */
int          view,              /* => view number */
boolean      useConstruction     /* => use ACS if constr. lock on */
);
```

Description The `mdlVec_projectPointToPlaneInView` function projects the point specified by *point* to the plane specified by *pointOnPlane* and *planeNormal* in the view specified by *view*.

If *useConstruction* is non-zero and ACS lock is enabled, the auxiliary coordinate system is used to determine the boresite vector rather than *view*. The projection direction in camera views is along a line from the camera position to point, for non-camera views it is the Z axis of the view. If the plane is parallel to the projection direction, the point is projected to the nearest point on the plane.

Returns `mdlVec_projectPointToPlaneInView` is of type `void`, it returns no value.

mdlVec_projectPointToLineInView

```
int mdlVec_projectPointToLineInView
(
Dpoint3d    *projectedPoint,    /* <= projected point */
Dpoint3d    *pointOnLine,       /* => point on line */
Dpoint3d    *lineNormal,        /* => normal vector to line */
Dpoint3d    *point,            /* => point to project */
int          view,              /* => view number */
boolean      useConstruction     /* => use ACS if constr. lock on */
);
```

Description The `mdlVec_projectPointToLineInView` function projects the point specified by *point* to the line specified by *pointOnLine* and *lineNormal*.

If *useConstruction* is non-zero, the projection direction is determined by the current ACS, otherwise the orientation of *view* is used.

Returns `mdlVec_projectPointToLineInView` is of type `void`, it returns no value.

21

Non-graphical Data

MDL applications can fully access the relational database interfaces that MicroStation supports. MDL applications can call functions to perform database queries: insert, delete and edit rows in tables; build database linkages; process elements for reporting; and filter elements against database criteria.

MicroStation provides interfaces to the following database products:

	MS-DOS	UNIX	Macintosh	WinNT
Oracle	✓	✓	✓	✓
Xbase	✓	✓	✓	✓
Informix		✓		✓
ODBC				✓
RIS	✓	*✓		✓

* Clipper only.

Regardless of the external database used, MicroStation functionality for dealing with non-graphical attributes is virtually the same. MDL database commands provide application developers with extensive tools for extending MicroStation's abilities to associate graphical and non-graphical data.

The MDL database interface is designed to be independent of the external database. The only deviation from this independence occurs in MDL functions that use SQL statements. While the Oracle and Informix interfaces support full SQL statement processing, the MicroStation interface to dBASE III/IV and FoxPro supports only a limited subset of SQL.

MDL applications may operate as MS_INITAPPS front ends and use database services. For more information on this topic, see "Using MicroStation Database Servers from Initapps" section of Part V of the MDL Programmer's Guide.

This chapter contains the following sections:

Database Interface Functions

- SQL Functions
- Database Settings Functions
- Tag Functions

Database Interface Functions

The following table lists database interface functions:

Function	Used to
mdlDB_attachActiveEntityElement	link a graphic element with the active entity.
mdlDB_attachActiveEntityDscr	link an element descriptor with the active entity.
mdlDB_detachAttributesElement	remove only attribute linkages recognized by the database server from an input element.
mdlDB_detachAttributesDscr	remove only attribute linkages recognized by the database server from an element descriptor.
mdlDB_openReport	open the report tables in the database to prepare for report generation.
mdlDB_closeReport	close the report tables in the database following report generation.
mdlDB_elementReport	process the database linkages for an input element.
mdlDB_reviewAttributes	interactively review the attribute linkages for an element.
mdlDB_decodeLink	extract the linkage parameters from a database linkage.
mdlDB_encodeLink	convert database linkage from internal to external format.
mdlDB_executeScreenForm	run a database screen form.
mdlDB_buildLink	construct a database linkage that can be appended to an element.
mdlDB_largestMslink	return the largest MSLINK key in a table.

Function	Used to
mdlDB_deleteElement	process element attribute linkages as if the element were deleted from the design file with the MicroStation DELETE ELEMENT command.
mdlDB_copyElement	process element attribute linkages as if the element were copied in the design file with the MicroStation COPY ELEMENT command.
mdlDB_elementFilter	test to determine whether an element contains attribute data linkages that belong to the database server.
mdlDB_defineFenceFilter	set active fence filter SQL SELECT statement.
mdlDB_fenceFilter	test to determine whether an element contains attribute data linkages that belong to the database server and satisfy the current DEFINE SEARCH criteria.
mdlDB_loadDADscr	load a displayable attribute text node with attribute text from the database.
mdlDB_buildDALinkFromLink	build displayable attribute linkage from existing database linkage.
mdlDB_defineAEBByLink	define active entity from existing database linkage.
mdlDB_defineAEBBySQLInsert	define active entity by SQL INSERT statement.
mdlDB_defineAEBBySQLSelect	define active entity from SQL SELECT statement.
mdlDB_displayActiveEntity	display active entity.
mdlDB_editActiveEntity	edit active entity using screen form.
mdlDB_findLinks	find element locations by database linkage type.
mdlDB_freeSimpleSelectResultXbase	free the results that were returned, as "brokenResults", from mdlDB_simpleSelectXbase.
mdlDB_getMscatalogName	get the value of the MS_CATALOG environment variable.
mdlDB_setLinkCacheInfo	set an application's database linkage interests.
mdlDB_simpleSelectXbase	select a single row from an Xbase database table.

Example

See database.mc.

mdlDB_attachActiveEntityElement

```
#include <mselems.h>
#include <dbdefs.h>
#include <dberrs.h>

int mdlDB_attachActiveEntityElement
(
  MSElementUnion  *out,      /* <= elm with active entity linkage */
  MSElementUnion  *in       /* => elm to be linked to active ent. */
);
```

Description The mdlDB_attachActiveEntityElement function links a graphic element with the active entity. The active entity must have been established with the MicroStation FIND, CREATE ENTITY, DEFINE AE or ACTIVE ENTITY commands. The database linkage that is created will be appended to existing attributes present on the input element *in*, creating the output element *out*. No existing linkages will be disturbed.

If the linkage mode is NEW, the database server will copy the active entity and add a new record to the database.

The MicroStation configuration variable MS_LINKTYPE controls the linkage format.

Returns mdlDB_attachActiveEntityElement returns SUCCESS if the active entity is successfully linked to the input graphic elements. Error codes are listed in dberrs.h.

See Also mdlDB_attachActiveEntityDscr, mdlDB_buildLink, mdlElement_appendAttributes.

mdlDB_attachActiveEntityDscr

```
#include <mselems.h>
#include <dbdefs.h>
#include <dberrs.h>

int mdlDB_attachActiveEntityDscr
(
  MSElementDscr  **elemDscrPP /* <=> comp elem linked w/ act. ent */
);
```

Description The mdlDB_attachActiveEntityDscr function links a complex element contained in an element descriptor with the active entity. The active entity must have been established with the MicroStation FIND, CREATE ENTITY, DEFINE AE or ACTIVE ENTITY command. The database linkage that is created will be appended to any

existing attributes present on the complex element pointed to by *elemDscrPP*. No existing linkages will be disturbed.

If the linkage generation mode is NEW, the database server copies the active entity and adds a new record to the database.

The MicroStation configuration variable MS_LINKTYPE controls the linkage format.

Returns mdlDB_attachActiveEntityDscr returns SUCCESS if the active entity is successfully linked to the input graphic elements. Error codes are listed in dberrs.h.

See Also mdlDB_attachActiveEntityElement, mdlDB_buildLink, mdlElement_appendAttributes.

mdlDB_detachAttributesElement

```
#include <mselems.h>
#include <dbdefs.h>
#include <dberrs.h>

int mdlDB_detachAttributesElement
(
    MSElementUnion    *out,      /* <= element with no database linkages */
    MSElementUnion    *in       /* => element with database linkages */
);
```

Description The mdlDB_detachAttributesElement function removes only attribute linkages recognized by the database server from the input element *in*. The element with no database linkages is returned in *out*. No other types of user data linkages will be disturbed on the element. The configuration variable MS_LINKTYPE sets the types of linkages that the server processes.

Returns The mdlDB_detachAttributesElement function returns SUCCESS if the attribute linkages were successfully removed. Error codes are listed in dberrs.h.

See Also mdlDB_detachAttributesDscr, mdlElement_stripAttributes, mdlElement_appendAttributes.

mdlDB_detachAttributesDscr

```
#include <mselems.h>
#include <dbdefs.h>
#include <dberrs.h>

int mdlDB_detachAttributesDscr
(
    MSElementDscr     **elemDscrPP /* <=> comp elem to det. attributes */
);
```

Description The mdlDB_detachAttributesDscr function removes only attribute linkages recognized by the database server from the complex element contained in the

element descriptor pointed to by *elemDscrPP*. No other types of user data linkages will be disturbed on the element. The configuration variable `MS_LINKTYPE` sets the types of linkages that the server processes.

Returns The `mdlDB_detachAttributesDscr` function returns `SUCCESS` if the attribute linkages were successfully removed. Error codes are listed in `dberrs.h`.

See Also `mdlDB_detachAttributesElement`, `mdlElement_stripAttributes`, `mdlElement_appendAttributes`.

mdlDB_openReport

```
#include <dbdefs.h>
#include <dberrs.h>

int mdlDB_openReport(void);
```

Description `mdlDB_openReport` initializes the report tables in the database to prepare for report generation. This function has no arguments. The contents of the report table defined by the MicroStation command `ACTIVE REPORT` are deleted.

Subsequent calls to `mdlDB_elementReport` will accumulate rows in the report tables according to the attribute linkages present on the elements.

Returns The `mdlDB_openReport` function returns `SUCCESS` if the report tables were successfully initialized. Error codes are listed in `dberrs.h`.

See Also `mdlDB_closeReport`, `mdlDB_elementReport`.

mdlDB_closeReport

```
#include <dbdefs.h>
#include <dberrs.h>

int mdlDB_closeReport(void);
```

Description The `mdlDB_closeReport` function closes report tables in the database after report generation. This function has no arguments. The MicroStation command `ACTIVE REPORT` defines the report table name.

Subsequent reporting operations must be preceded by `mdlDB_openReport`.

Returns The `mdlDB_closeReport` function returns `SUCCESS` if the report tables are successfully closed and report generation is complete. Error codes are listed in `dberrs.h`.

See Also `mdlDB_openReport`, `mdlDB_elementReport`.

mdlDB_elementReport

```
#include <mselems.h>
#include <dbdefs.h>
#include <dberrs.h>

int mdlDB_elementReport
(
  MSElementUnion   *in          /* => elm containing database linkages */
);
```

Description mdlDB_elementReport processes database linkages on the input element *in*. The database server will add one row to the corresponding report table for each occurrence of an attribute linkage. The MicroStation command ACTIVE REPORT defines the report table name.

Subsequent calls to mdlDB_elementReport will accumulate additional rows in the report tables according to the attribute linkages present on the elements.

Returns The mdlDB_elementReport function returns SUCCESS if the attribute linkages are successfully processed. Error codes are listed in dberrs.h.

See Also mdlDB_openReport, mdlDB_closeReport.

mdlDB_reviewAttributes

```
#include <mselems.h>
#include <dbdefs.h>
#include <dberrs.h>

int mdlDB_reviewAttributes
(
  MSElementUnion   *in          /* => elm containing database linkages */
);
```

Description mdlDB_reviewAttributes interactively reviews attribute linkages present on an element. The database server will execute the REVIEW MicroStation command for the element linkages.

If screen forms are activated, the entity's active screen form will be used. Otherwise, the SQL SELECT statement specified by the MicroStation command ACTIVE REVIEW will control attributes displayed for each linkage.

Returns The mdlDB_reviewAttributes function returns SUCCESS if the attribute linkages are successfully processed.

mdlDB_decodeLink

```
#include <mselems.h>
#include <dbdefs.h>
#include <dberrs.h>

int mdlDB_decodeLink
(
    DatabaseLink *link,    /* <= generic database linkage */
    short        p[]      /* => database linkage
);
```

Description The mdlDB_decodeLink function extracts linkage parameters from a database linkage. The database server decodes the *p* linkage, and the linkage parameters are returned in the link structure. Regardless of the linkage type (RIS, INFORMIX, DMRS, ORACLE, ODBC or XBASE), all MicroStation database linkages can be represented by the database-independent structure DatabaseLink.

The format of the structure DatabaseLink is as follows:

```
typedef struct
{
    unsigned short linkSize; /* # of BYTES in linkage */
    unsigned short linkType;
    unsigned short entity;
    char tablename[64];
    unsigned long mslink;
    LinkProps props;
    unsigned short dasType;
} DatabaseLink;
```

The structure member linkType is set to one of the following values to represent the linkage signature:

```
ORACLE_LINKAGE Oracle linkage
INFORMIX_LINKAGE Informix linkage
RIS_LINKAGE Relational Interface System linkage
DMRS_LINKAGE DMRS linkage
XBASE_LINKAGE dBASE III/IV linkage
ODBC_LINKAGE ODBC Linkage
```

Returns The mdlDB_decodeLink function returns SUCCESS if the linkage is successfully decoded. Error codes are listed in dberrs.h.

See Also mdlDB_buildLink.

mdlDB_encodeLink

```
#include <dbdefs.h>
#include <dberrs.h>

int mdlDB_encodeLink
(
```

```
short      *attributes,    /* <= attr linkage (file format) */
DatabaseLink *link        /* => linkage to convert from internal format */
);
```

Description The mdlDB_encodeLink function is used to convert a database linkage from internal format *link* to one in external file format *attributes*. The length, in bytes, of the attribute linkage is given by the *linkSize* member of *link*.

Returns mdlDB_encodeLink returns SUCCESS if the linkage was converted successfully. Other error codes are listed in dberrs.h.

See Also mdlDB_decodeLink.

mdlDB_executeScreenForm

```
#include <dberrs.h>

int mdlDB_executeScreenForm
(
char    *formname,        /* => screen form name */
char    *arguments        /* => command line arguments to form mgr */
);
```

Description The mdlDB_executeScreenForm function will run a database screen form. The *formname* argument should point to a string containing the name of the form to be executed. Unless a full pathname is given, the path indicated by the MS_DBASE configuration variable will be searched for the form.

arguments points to a string containing arguments to be passed to the forms package for the database being used. The value NULL can be passed if no arguments are needed.

Returns The mdlDB_executeScreenForm function returns SUCCESS if the form is located and successfully executed. Error codes are listed in dberrs.h.

mdlDB_buildLink

```
#include <mselems.h>
#include <dbdefs.h>
#include <dberrs.h>

int mdlDB_buildLink
(
short    *link,           /* <= database linkage */
int      *linkLength,     /* <= size (words) of linkage */
int      linkType,        /* => linkage type (ORACLE, XBASE, etc, ...) */
LinkProps *props,         /* => properties of linkage */
char     *tableName,      /* => table name of linkage */
unsigned long mslink,      /* => MSLINK key for linkage */
int      dasType          /* => displayable attribute type */
);
```

Description The `mdlDB_buildLink` function constructs a user data linkage according to the passed parameters. The returned linkage *link* can be attached to an element using `mdlElement_appendAttributes`. The linkage length, in words, is returned in *linkLength*.

The linkage type *linkType* is one of the following values:

```
ORACLE_LINKAGE Oracle linkage
INFORMIX_LINKAGE Informix linkage
RIS_LINKAGE Relational Interface System linkage
DMRS_LINKAGE VAX DMRS linkage
XBASE_LINKAGE dBASE III/IV linkage
ODBC_LINKAGE ODBC linkage
```

The format of `Linkprops` is as follows:

```
typedef struct
{
    unsigned short information;    /* informational linkage */
    unsigned short remote;        /* remote linkage */
    unsigned short modified;      /* data linkage modified */
    unsigned short user;          /* user data linkage */
    unsigned short class;         /* class of linkage */
};
```

tablename specifies the name of the database table to which the linkage will point. The table row's unique integer key is specified by *mslink*. The displayable attribute type is given by *dasType*.

Returns The `mdlDB_buildLink` function returns `SUCCESS` if a linkage is successfully constructed. Error codes are listed in `dberrs.h`.

See Also `mdlElement_appendAttributes`, `mdlDB_largestMslink`.

mdlDB_largestMslink

```
#include <mselems.h>
#include <dbdefs.h>
#include <dberrs.h>

int mdlDB_largestMSlink
(
    unsigned long    *mslink,        /* <= largest value of MSLINK column */
    char            *tableName      /* => table name */
);
```

Description The `mdlDB_largestMslink` function returns the highest MSLINK key of table *tableName* in *mslink*. The largest MSLINK key in a table must be known while new

rows are being added to the table. A call to `mdlDB_largestMslink` can be followed by `mdlDB_buildLink` to construct a new database linkage.

Returns The `mdlDB_largestMslink` function returns `SUCCESS` if the highest MSLINK key is successfully located. Error codes are listed in `dberrs.h`.

See Also `mdlDB_buildLink`.

mdlDB_deleteElement

```
#include <mselems.h>
#include <dbdefs.h>
#include <dberrs.h>

int mdlDB_deleteElement
(
  MSElementUnion  *p          /* => element w/ database linkages */
);
```

Description The `mdlDB_deleteElement` function processes an element's attribute linkages as if the element were deleted from the design file with a MicroStation command. This function should be called to maintain the database tables in synchronization with the design file. It should be called only when elements are not deleted with a MicroStation command or the `mdlElement_undoableDelete` function.

This function does not delete the graphic element from the design file. It processes only the attribute linkages. Normally, the state of the SET DELETE [ON|OFF] switch will control whether linked table rows are deleted from the database.

Returns The `mdlDB_deleteElement` function returns `SUCCESS` if the linkages are successfully processed. Error codes are listed in `dberrs.h`.

See Also `mdlDB_copyElement`, `mdlElement_undoableDelete`.

mdlDB_copyElement

```
#include <mselems.h>
#include <dbdefs.h>
#include <dberrs.h>

int mdlDB_copyElement
(
  MSElementUnion  *p          /* => element w/ database linkages */
);
```

Description The `mdlDB_copyElement` function processes the attribute linkages of an element as if the element were copied in the design file using a MicroStation command. This function should be called to maintain the database tables in synchronization with

the design file. It should be called only when elements are not copied with a MicroStation command.

This function does not copy the graphic element in the design file. It processes only the attribute linkages. Normally, the linkage generation mode (such as NEW or DUPLICATE) will control whether new rows are added to the database.

Returns The `mdlDB_copyElement` function returns `SUCCESS` if the linkages are successfully processed. Error codes are listed in `dberrs.h`.

See Also `mdlDB_deleteElement`.

mdlDB_elementFilter

```
#include <mselems.h>
#include <dbdefs.h>
#include <dberrs.h>

int mdlDB_elementFilter
(
  MSElementUnion    *p          /* => element w/ database linkages */
);
```

Description The `mdlDB_elementFilter` function tests to determine whether an element contains attribute data linkages that belong to the database server. Elements can contain many types of attribute data linkages that belong to either application programs or the database interface.

The `mdlDB_elementFilter` function is a convenient way to determine whether the element *p* contains at least one linkage that belongs to the database server.

Returns The `mdlDB_elementFilter` function returns `TRUE` if the element has one or more linkages that belong to the database server and `FALSE` if no recognized linkages are present.

See Also `mdlDB_fenceFilter`.

mdlDB_defineFenceFilter

```
#include <dbdefs.h>
#include <dberrs.h>

int mdlDB_defineFenceFilter
(
  char    *sqlSelect    /* => SQL SELECT statement for fence filter */
);
```

Description The `mdlDB_defineFenceFilter` function is used to establish a new fence filter for a database table. The argument *sqlSelect* is an SQL SELECT statement which specifies a subset of a table. The fence filter will limit all fence operations to only those

elements which meet both the geometric criteria of the fence *and* the database criteria expressed by the SELECT statement.

Each table may have its SELECT statement filter. The FENCEFILTER column of MSCATALOG is updated with the SELECT statement *sqlSelect*.

The mdlDB_defineFenceFilter function is equivalent to the MicroStation DEFINE SEARCH (DS=) key-in.

Returns mdlDB_defineFenceFilter returns SUCCESS if the fence filter is activated. Other error codes are listed in dberrs.h.

See Also mdlDB_fenceFilter.

mdlDB_fenceFilter

```
#include <mselems.h>
#include <dbdefs.h>
#include <dberrs.h>

int mdlDB_fenceFilter
(
  MSElementUnion  *p          /* => element w/ database linkages */
);
```

Description The mdlDB_fenceFilter function tests to determine whether an element contains attribute data linkages that belong to the database server and satisfy the current DEFINE SEARCH criteria. Elements can contain many types of attribute data linkages that belong to application programs or the database interface.

The mdlDB_fenceFilter function is a convenient way to determine whether the element *p* contains at least one linkage that belongs to the database server and satisfies the current fence filter.

Returns The mdlDB_fenceFilter function returns TRUE if the element has one or more linkages that satisfy the active fence filter and FALSE if no linkages satisfy the filter.

See Also mdlDB_elementFilter.

mdlDB_loadDADscr

```
#include <mselems.h>
#include <dbdefs.h>
#include <dberrs.h>

int mdlDB_loadDADscr
(
  MSElementDescr  **elemDscrPP /* <=> text node loaded w/ disp attr */
);
```

Description `mdlDB_loadDADscr` loads a displayable attribute text node with attribute information from the database. The element descriptor *elemDscrPP* must point to a text node element with a displayable attribute database linkage.

Existing text elements attached to the text node are replaced with the text strings reflecting the current database attribute contents. The screen form associated with the displayable attribute type of the text node's database linkage determines the text string format. The text string properties such as font, text size, line spacing and justification are taken from the text node.

Returns The `mdlDB_loadDADscr` function returns `SUCCESS` if the text node is successfully loaded. Error codes are listed in `dberrs.h`.

mdlDB_buildDALinkFromLink

```
#include <dbdefs.h>
#include <dberrs.h>

int mdlDB_buildDALinkFromLink
(
    short      *attributes,    /* <= displayable attribute linkage */
    int        *linkLength,    /* <= length (words) of attributes */
    DatabaseLink *link        /* => database linkage */
);
```

Description The `mdlDB_buildDALinkFromLink` function is used to build a displayable attribute linkage from an existing database linkage.

The displayable attribute linkage created is based on the linkage *link*. The displayable attribute linkage will reference the same entity number and `MSLINK` key contained in linkage *link*. The active displayable attribute type will be used to determine the formatting of the displayable attributes.

The argument *attributes* receives the completed displayable attribute linkage. The displayable attribute linkage is in external file format and may be directly appended to an element with a function such as `mdlElement_appendAttributes`. The argument *linkLength* receives the length, in words, of the displayable attribute linkage.

Returns `mdlDB_buildDALinkFromLink` returns `SUCCESS` if the displayable attribute linkage was created successfully. Other error codes are listed in `dberrs.h`.

See Also `mdlDB_activeDAType`, `mdlDB_loadDADscr`.

mdlDB_defineAEBByLink

```
#include <dbdefs.h>
#include <dberrs.h>
int mdlDB_defineAEBByLink
(
    DatabaseLink      *linkage      /* => linkage defining active entity */
);
```

Description The mdlDB_defineAEBByLink function is used to define a new active entity based on a database linkage.

The argument *linkage* is a pointer to a database linkage in external (file format) format. The database linkage contains an entity number which specifies the table (through MSCATALOG) and an MSLINK key which specifies the row within the table.

The mdlDB_defineAEBByLink function is equivalent to the MicroStation DEFINE AE command.

Returns mdlDB_defineAEBByLink returns SUCCESS if the active entity is successfully defined from the database linkage. Other error codes are listed in dberrs.h.

See Also mdlElement_extractAttributes, mdlElmdscr_extractAttributes, mdlDB_defineAEBBySQLInsert, mdlDB_defineAEBBySQLSelect.

mdlDB_defineAEBBySQLInsert

```
#include <dbdefs.h>
#include <dberrs.h>

int mdlDB_defineAEBBySQLInsert
(
    char      *sqlInsert      /* => SQL INSERT statement */
);
```

Description The mdlDB_defineAEBBySQLInsert function is used to define a new active entity from an SQL INSERT statement. It is used in NEW linkage mode to establish an active entity when there is no existing row in the database table which can be used as the basis for the active entity.

Do not supply an MSLINK value in the INSERT statement. MicroStation will automatically assign a value to MSLINK when a linkage is actually created.

The mdlDB_defineAEBBySQLInsert function is equivalent to the MicroStation ACTIVE ENTITY (AE=) command.

Returns mdlDB_defineAEBBySQLInsert returns SUCCESS if the active entity is successfully defined from the SQL INSERT statement. Other error codes are listed in dberrs.h.

See Also mdlDB_defineAEBByLink, mdlDB_defineAEBBySQLSelect.

mdlDB_defineAEBySQLSelect

```
#include <dbdefs.h>
#include <dberrs.h>

int mdlDB_defineAEBySQLSelect
(
    char    *sqlSelect      /* => SQL SELECT statement */
);
```

Description The mdlDB_defineAEBySQLSelect function is used to define an active entity based on an existing row in a database table. The argument *sqlSelect* defines an SQL SELECT statement. If the SELECT statement returns multiple rows the first one is used as the basis for the active entity.

The mdlDB_defineAEBySQLSelect function is equivalent to the MicroStation FIND (FI=) command.

Returns mdlDB_defineAEBySQLSelect returns SUCCESS if the active entity is successfully defined from the SQL SELECT statement. Other error codes are listed in dberrs.h.

See Also mdlDB_defineAEByLink, mdlDB_defineAEBySQLInsert.

mdlDB_displayActiveEntity

```
#include <dbdefs.h>
#include <dberrs.h>

int mdlDB_displayActiveEntity
(
    void
);
```

Description The mdlDB_displayActiveEntity function will display the active entity. The active entity is the target row to which graphics elements will be linked.

The mdlDB_displayActiveEntity function is equivalent to the MicroStation SHOW AE key-in.

Returns mdlDB_displayActiveEntity returns SUCCESS if the active entity is successfully displayed. Other error codes are listed in dberrs.h.

See Also mdlDB_defineAEBySQLInsert, mdlDB_defineAEBySQLSelect, mdlDB_defineAEByLink, mdlDB_editActiveEntity.

mdlDB_editActiveEntity

```
#include <dbdefs.h>
#include <dberrs.h>

int mdlDB_editActiveEntity
(
    void
);
```

Description The mdlDB_editActiveEntity function will display the active entity and allow editing of the individual fields.

The active entity is the target row to which graphics elements will be linked. If the active linkage mode is DUPLICATE, the active entity represents an actual table row and edits are written directly to the database. If the linkage mode is NEW, the active entity is a prototype row, and edits are written in-memory only.

The mdlDB_editActiveEntity function is equivalent to the MicroStation EDIT AE key-in.

Returns mdlDB_editActiveEntity returns SUCCESS if the active entity is defined. Other error codes are listed in dberrs.h.

See Also mdlDB_defineAEBBySQLInsert, mdlDB_defineAEBBySQLSelect, mdlDB_defineAEBByLink, mdlDB_displayActiveEntity.

mdlDB_findLinks

```
#include <msdb.fdf>
#include <dbdefs.h>

int mdlDB_findLinks /* <= error status */
(
    LinkInfo    **linksPP,      /* <= array of found database linkages */
    int         *numLinksP,     /* <= number of records in array */
    int         fileNum,        /* => file number to search; -1 is all */
    int         entity,         /* => entity to seek; -1 is all */
    ULong       mslink          /* => mslink to seek; 0 is all */
);
```

Description The mdlDB_findLinks function searches MicroStation's cache of element locations by *mslink* and *entity*, and returns an allocated array of information on found elements.

linksPP is the address of a pointer that will be set to point to the allocated array. It is the caller's responsibility to free the array when no longer needed. See dbdefs.h for the declaration of the LinkInfo typedef.

numLinksP is the address of an integer that will be set to indicate the number of records in the array.

fileNum is the file number in which to search, or -1 to indicate all files.

entity is the entity number to search by. Only elements with database linkages of this entity type will be found. Set *entity* to -1 to return information on all entity types.

mslink is the linkage number to search by. By definition, there is only one record in a database that has a given *mslink* value, for each entity type. Only elements with database linkages with this *mslink* number will be found. Set *mslink* to 0 to return information on all links.



This function was implemented in MicroStation 95.

Returns `mdlDB_findLinks` returns `MDLERR_NOTAVAILABLE` if the cache of element locations is empty, meaning either that there are no database linkages in any of the currently open files, or that the user has not turned on the “Use mslink caching” preference. The function returns `SUCCESS` if elements were found, and `!SUCCESS` if none were found.

See Also `mdlDB_setLinkCacheInfo`.

mdlDB_freeSimpleSelectResultXbase

```
#include <rdbmslib.fdf>

int mdlDB_freeSimpleSelectResultXbase
(
char    **result, /* => result returned from mdlDB_simpleSelectXbase */
int     numCols  /* => num of cols mdlDB_simpleSelectXbase returns */
);
```

Description `mdlDB_freeSimpleSelectResultXbase` frees the results that were returned (as “brokenResults” from `mdlDB_simpleSelectXbase`. There should be a call to `mdlDB_freeSimpleSelectResultXbase` for each call to `mdlDB_simpleSelectXbase`.

result is the pointer to the result list.

numCols is the number of columns' values that were returned from `mdlDB_simpleSelectXbase`.



This function was implemented in MicroStation 95.

Returns `mdlDB_freeSimpleSelectResultXbase` returns `SUCCESS` if the memory is successfully freed, and a non-zero value if an error occurred.

See Also `mdlDB_simpleSelectXbase`.

mdlDB_getMscatalogName

```
#include <rdbmslib.fdf>
#include <dbdefs.h>

void mdlDB_getMscatalogName
(
char    *mscatNameP /* <= control table name */
);
```

Description `mdlDB_getMscatalogName` gets the value of the `MS_MSCATALOG` environment variable. If no value is assigned, it returns the default control table name. For ODBC flat files, the default is ‘mscatlog’ and for all other platforms it is ‘mscatalog.’

mscatNameP points to a buffer to receive the control table name. The buffer must be at least `MAX_TABLE_LENGTH + 1` bytes.



This function was implemented in MicroStation 95.

Returns The mdlDB_getMscatalogName function is of type void, it returns no value.

mdlDB_setLinkCacheInfo

```
#include <msdb.fdf>

int mdlDB_setLinkCacheInfo /* <= SUCCESS or SUCCESS */
(
    int      *entitiesP,      /* => array of entities to declare interest in */
    int      numEntities      /* => number of entries in array */
);
```

Description The mdlDB_setLinkCacheInfo function is used to specify to MicroStation which entity numbers are of interest to the calling application. Only the locations of elements with links to those entity numbers will be cached, for later access through mdlDB_findLinks. The function must be called before a design file is opened.

No element locations are cached unless the user has turned on the “Use mslink caching” option in the Preferences dialog.

If mdlDB_setLinkCacheInfo is not called by any application, the locations of all elements with any database linkage will be cached.

If more than one application calls mdlDB_setLinkCacheInfo, the locations of all elements with links to entity numbers requested by any application will be cached.

entitiesP points to an array of integers indicating which entities the caller wants to have cached.

numEntities indicates the number of entries in the array. If *numEntities* is 0, it is taken to mean that all entity numbers are of interest and all elements with any database linkage should be cached.



This function was implemented in MicroStation 95.

Returns mdlDB_setLinkCacheInfo always returns SUCCESS unless the caller passes inappropriate parameters.

See Also mdlDB_findLinks.

mdlDB_simpleSelectXbase

```
#include <rdbmslib.fdf>

int mdlDB_simpleSelectXbase
(
    char      **brokenResult, /* <= results of query, if not NULL */
    char      *charResult,   /* <= results of query in 1 string, non-NULL */
    int       *numValues,    /* <= number of values returned */
);
```

```
void    *futureUse,    /* <= for future use, always set to NULL */
char    *stmt         /* => select statement */
);
```

Description `mdlDB_simpleSelectXbase` selects a single row from an Xbase database table. The SQL statement *stmt* can be any otherwise legal Xbase SQL statement.

Upon return, *numValues* contains the number of values that were specified in the select list of *stmt* (this is set whether any rows are found or not).

brokenResult, if not passed as `NULL`, is allocated by MicroStation: enough to hold *numValues* pointers. Each of these pointers points to a column's value. *brokenResult* should be passed to `mdlDB_freeSimpleSelectResultXbase` when the results are no longer needed.

charResult is an alternative way of receiving the results from the query. If not passed as `NULL`, the results of the query are placed into this location, and each result is terminated by a `NULL` character '\0'; The caller must allocate enough space for these values.

futureUse is presently not used, and should always be passed as `NULL`.



This function was implemented in MicroStation 95.

Returns The `mdlDB_simpleSelectXbase` function returns `QUERY_NOT_FINISHED` if at least one row satisfied the where clause. `QUERY_FINISHED` is returned if no rows satisfy the where clause.

See Also `mdlDB_freeSimpleSelectResultXbase`, `mdlDB_sqlQuery`, `mdlDB_readColumn`.

SQL Functions

The following table lists SQL database functions:

Function	Used to
<code>mdlDB_writeColumn</code>	update columns in table rows.
<code>mdlDB_readColumn</code>	read columns from table rows.
<code>mdlDB_deleteRow</code>	delete the table row that has a unique integer key.
<code>mdlDB_sqlQuery</code>	process single column SQL query.
<code>mdlDB_processSQL</code>	process non-SELECT SQL statement.
<code>mdlDB_getErrorText</code>	retrieve text of error message from error code.

Function	Used to
mdlDB_describeTable	obtain a description of the structure of a table in the database.
mdlDB_openCursor	open a cursor for a query so that rows can be retrieved from a table.
mdlDB_fetchRow	fill a descriptor with a row of data from the query for which the cursor was opened.
mdlDB_closeCursor	close a cursor to end a query.
mdlDB_freeSQLDADescriptor	free MS_sqllda descriptor memory allocated during mdlDB_openCursor and memory allocated by mdlDB_describeTable and mdlDB_describeColumn [rdbmslib.m].
mdlDB_insertRowByForm	add new row to database table using screen form.
mdlDB_addRowWithMslink [rdbmslib.m]	add a row to the database and return the new mslink.
mdlDB_databaseProfile [rdbmslib.m]	get a description of the capabilities of the current database.
mdlDB_openCursorWithID	open a database query from a pool of available cursors.
mdlDB_fetchRowByID	fetch a row for the identified cursor.
mdlDB_closeCursorByID	close the identified cursor.
mdlDB_copyTable [rdbmslib.m]	create a new table with the structure of an existing table.
mdlDB_extractLinkages [rdbmslib.m]	extract an elements database linkages.
mdlDB_describeDatabase [rdbmslib.m]	describe the database tables accessible by the current user.
mdlDB_describeColumn [rdbmslib.m]	describe a column in a particular table.
mdlDB_additionalRequest [rdbmslib.m]	extend the database server architecture with user defined requests.

Example

See database.mc.

mdlDB_writeColumn

```
#include <mselems.h>
#include <dbdefs.h>
#include <dberrs.h>

int mdlDB_writeColumn
(
    char          *tableName,    /* => database table name */
    unsigned long  mslink,       /* => key of row to update */
    char          *columnName,   /* => name of column to update */
    char          *columnValue   /* => new value of column */
);
```

Description The mdlDB_writeColumn updates columns in table rows. The column *columnName* in the row with an MSLINK key of *mslink* is updated in table *tableName*. The column with the name *columnName* is updated with the value *columnValue*.

Type conversion is automatically performed for columns with data types other than char.

Returns The mdlDB_writeColumn function returns SUCCESS if the update is successfully processed. Error codes are listed in dberrs.h.

See Also mdlDB_readColumn.

mdlDB_readColumn

```
#include <mselems.h>
#include <dbdefs.h>
#include <dberrs.h>

int mdlDB_readColumn
(
    char          columnValue,   /* <= value of column read */
    char          *tableName,    /* => name of table to read */
    unsigned long  mslink,       /* => MSLINK key of row to read */
    char          *columnName    /* => name of column to read */
);
```

Description The mdlDB_readColumn function reads columns from table rows. The row with an MSLINK key of *mslink* is read from table *tableName*. If using dBASE, *tablename* should be the alias.

The column contents of column *columnName* is returned in *columnValue*.

Type conversion is automatically performed for all columns.

Returns The mdlDB_readColumn function returns SUCCESS if the row is successfully processed. Error codes are listed in dberrs.h.

See Also mdlDB_writeColumn.

mdlDB_deleteRow

```
#include <mselems.h>
#include <dbdefs.h>
#include <dberrs.h>

int mdlDB_deleteRow
(
    unsigned long    mslink,          /* => MSLINK key of row to delete */
    char             *tableName       /* => database table name */
);
```

Description The mdlDB_deleteRow function deletes the row from table *tableName*. This table has the unique integer key *mslink*.

Returns The mdlDB_deleteRow function returns SUCCESS if the row is successfully deleted.

See Also mdlDB_deleteElement.

mdlDB_sqlQuery

```
#include <dbdefs.h>
#include <dberrs.h>

int mdlDB_sqlQuery
(
    char    *column,          /* <= column result */
    char    *query            /* => SQL query */
);
```

Description The mdlDB_sqlQuery function is designed for an SQL query that returns a single-column/single-row result. It should be used only for queries that meet these criteria. If more than one column is required or if the query will return more than one row, a cursor must be used.

The argument *column* is a character array that receives the result of the SQL SELECT statement *query*. All values are returned as character data. Appropriate conversions may be done on the result if numeric data is required.

Returns mdlDB_sqlQuery returns SUCCESS if the query is executed without errors. A syntactically correct query that returns no rows will return NO_ROWS_RETURNED (4040). If the query generated an SQL error, then SQL_LOOKUP (-100) is returned. Error codes are listed in dberrs.h.

See Also mdlDB_openCursor, mdlDB_fetchRow, mdlDB_closeCursor.

mdlDB_processSQL

```
#include <dbdefs.h>
#include <dberrs.h>

int mdlDB_processSQL
(
    char    *sqlStatement /* => SQL statement */
);
```

Description The `mdlDB_processSQL` function submits general non-SELECT SQL statements to the database server. *sqlStatement* represents any character string (usually an SQL statement) that the database server can process. In MicroStation, these statements can be interactively submitted to the server when the statement is preceded with the vertical bar '|'. Note that SQL relational databases like Oracle, Informix and Ingres frequently differ in their implementation of SQL by providing vendor-specific enhancements to ANSI SQL.

The database server must be able to process SQL statements.

Returns `mdlDB_processSQL` returns `SUCCESS` if the statement was processed without error. Positive error codes represent MicroStation errors and are listed in `dberrs.h`. Negative return values represent database specific error codes and may be found in the vendor's documentation.

See Also `mdlDB_getErrorText`, `mdlDB_sqlQuery`, `mdlDB_openCursor`, `mdlDB_fetchRow`, `mdlDB_closeCursor`.

mdlDB_getErrorText

```
#include <dbdefs.h>
#include <dberrs.h>

int mdlDB_getErrorText
(
    char    *errorText, /* <= text of error message */
    int     errorCode   /* => error code */
);
```

Description The `mdlDB_getErrorText` function will translate a database interface error code into a text message. It is particularly useful for obtaining error messages for errors which occurred in the database (non-MicroStation) processing of an MDL function. In general, MicroStation error return codes are positive. Errors from the RDBMS are usually negative. `mdlDB_getErrorText` will process both positive and negative error codes.

The argument *errorText* is a character array that will receive the error message corresponding to error code *errorCode*.

Returns `mdlDB_getErrorText` returns `SUCCESS` if the error code was successfully translated to a text message.

mdlDB_describeTable

```
#include <dbdefs.h>
#include <dberrs.h>

int mdlDB_describeTable
(
    MS_sqlda      *sqlda,          /* <=> table structure */
    char          *table_name      /* => name of database table */
);
```

Description mdlDB_describeTable describes the structure of a table in the database. An MS_sqlda structure, allocated by the caller, should be passed to the function. This structure is defined in the header file dbdefs.h as follows:

```
typedef struct
{
    short      numColumns;
    char       **name;
    char       **value;
    short      *type;
    short      *length;
    short      *scale;
    short      *prec;
    short      *null;
} MS_sqlda;
```

If successful, mdlDB_describeTable will complete the descriptor's members.



The memory allocated by this function must be freed by calling mdlDB_freeSQLDADescriptor.

numColumns indicates the number of columns found in the table. The remaining structure members are arrays that can be indexed to obtain column information.

name points to an array of NULL terminated strings. This array indicates the column names.

value is used only with the fetch operation as described for the mdlDB_fetchRow function.

type points to an array of column types as defined in the dbdefs.h header file.

length points to an array of column lengths. This field is significant only for character columns and is set to zero for all other types.

prec and *scale* point to arrays of numbers where *prec* is the maximum number of characters in the number and *scale* indicates the maximum number of digits after the decimal.

null points to an array of short integers that are set to `TRUE` if `NULL` values are allowed in this column and `FALSE` otherwise.

Returns The `mdlDB_describeTable` function returns `SUCCESS` if the table description is successfully obtained. Error codes are listed in `dberrs.h`.

See Also `mdlDB_openCursor`, `mdlDB_fetchRow`, `mdlDB_closeCursor`.

mdlDB_openCursor, mdlDB_fetchRow, mdlDB_closeCursor

```
#include <dbdefs.h>
#include <dberrs.h>

int mdlDB_openCursor
(
    char    *query      /* => SQL SELECT statement */
);

int mdlDB_fetchRow
(
    MS_sql_da    *sql_da    /* <= current row of query */
);

int mdlDB_closeCursor();
```

Description The `mdlDB_openCursor` function opens a cursor for an SQL query. It places the cursor before the first row matching the specified query. After a cursor is opened, rows can be retrieved with the `mdlDB_fetchRow` function.

`mdlDB_fetchRow` retrieves the next row of the query for which the cursor has been opened. It receives a pointer to an `MS_sql_da` structure. `MS_sql_da` is allocated by MicroStation, and its structure is defined in the `dbdefs.h` header file as follows:

```
typedef struct
{
    short    numColumns;
    char     **name;
    char     **value;
    short    *type;
    short    *length;
    short    *scale;
    short    *prec;
    short    *null;
} MS_sql_da;
```

If successful, `mdlDB_fetchRow` will complete the descriptor with the row in front of the cursor and move the cursor in front of the next row satisfying the query.

The members of the `MS_sql_da` are described with the `mdlDB_describeTable` function. The lengths pointed to by the *length*

member describe column lengths, not necessarily value lengths. For example, the value for a 40-character column with the last 20 characters blank will be a 40-character string.



If a cursor is already open when `mdlDB_openCursor` is called, `CURSOR_ALREADY_OPEN` is returned and the cursor remains open. This is a change from Version 4. MicroStation will no longer close the cursor if it is already open; the decision of when or if to use `mdlDB_closeCursor` is left up to the developer. To use multiple cursors refer to `mdlDB_openCursorWithID`.



Note that `mdlDB_openCursor`, `mdlDB_fetchRow` and `mdlDB_closeCursor` should be replaced by `mdlDB_openCursorWithID`, `mdlDB_fetchRowByID` and `mdlDB_closeCursorByID` respectively.

Returns `mdlDB_openCursor` returns `SUCCESS` if the cursor is successfully opened or `CURSOR_ALREADY_OPEN` if the cursor is already open.

`mdlDB_fetchRow` returns `QUERY_NOT_FINISHED` if a row is successfully fetched. `QUERY_FINISHED` is returned when no rows remain in the query.

`mdlDB_closeCursor` returns `SUCCESS` if the cursor is successfully closed.

See Also `mdlDB_openCursorWithID`, `mdlDB_fetchRowByID`, `mdlDB_closeCursorByID`, `mdlDB_describeTable`.

mdlDB_freeSQLDADescriptor

```
#include <dbdefs.h>
#include <dberrs.h>
#include <dbserver.h>

void mdlDB_freeSQLDADescriptor
(
    MS_sqlda      *sqlda          /* <=> descriptor to free */
);
```

Description `mdlDB_freeSQLDADescriptor` will free memory which has been allocated by MicroStation during `mdlDB_openCursor`, `mdlDB_describeTable` and `mdlDB_describeColumn` function calls. The structure `Ms_sqlda` contains pointers to arrays of character pointers for column names, values and other attributes. When a database cursor is opened with `mdlDB_openCursor` MicroStation examines the query and allocates sufficient memory in the `Ms_sqlda` for the columns referenced in the query.



After completing a query by calling `mdlDB_closeCursor` the memory MicroStation has allocated must be freed with `mdlDB_freeSQLDADescriptor`.

Returns `mdlDB_freeSQLDADescriptor` is of type `void`. It returns no value.

See Also `mdlDB_openCursor`, `mdlDB_fetchRow`, `mdlDB_closeCursor`.

mdlDB_insertRowByForm

```
#include <dbdefs.h>
#include <dberrs.h>

int mdlDB_insertRowByForm
(
char    *table          /* => table name */
);
```

Description The `mdlDB_insertRowByForm` function is used to insert a new row into a database table using a screen form. The argument *table* specifies a table listed in MSCATALOG. MicroStation will display the screen form listed for *table* in the MSCATALOG column SCREENFORM. Completing the screen form will add a new row to the database table.

The new row added will become the active entity. The `mdlDB_insertRowByForm` function is equivalent to the MicroStation CREATE ENTITY command.

Returns `mdlDB_insertRowByForm` returns `SUCCESS` if the screen form was successfully displayed. Other error codes are listed in `dberrs.h`.

See Also `mdlDB_executeScreenForm`.

mdlDB_addRowWithMslink [rdbmslib.ml]

```
#include <rdbmslib.fdf>

int mdlDB_addRowWithMslink
(
ULong    *mslink,        /* <= mslink for new row */
char     *tableName,     /* <= table to add row to, or NULL */
char     *insertStmt     /* => values for new row */
);
```

Description The `mdlDB_addRowWithMslink` function adds a new row to the indicated table and returns the new mslink number.

mslink specifies the new mslink for the added row.

tableName specifies the table to which the row should be added. If an insert statement is provided this argument is not necessary.

insertStmt specifies an insert statement to supply new values for the new row. Any columns but the MSLINK column can be specified. NOT NULL columns have to be specified values in the insert statement. The insert statement must have the form:

```
insert into table (column list) values (value list)
```

Returns mdlDB_addRowWithMslink returns SUCCESS if the row was successfully added. Otherwise, an error status is returned.

mdlDB_databaseProfile [rdbmslib.ml]

```
#include <rdbmslib.fdf>

int mdlDB_databaseProfile
(
    DatabaseProfile *profileP      /* <=> profile structure */
);
```

Description mdlDB_databaseProfile gets a description of the capabilities of the current database server.

profileP specifies a DatabaseProfile structure to be filled in by this function.

Returns mdlDB_databaseProfile returns SUCCESS if the database profile was successfully retrieved. Otherwise, an error status is returned.

mdlDB_openCursorWithID, mdlDB_fetchRowByID, mdlDB_closeCursorByID

```
#include <msdb.fdf>

int mdlDB_openCursorWithID
(
    CursorID *cursorID,      /* <= ID for cursor opened */
    char *query              /* => query for cursor */
);

int mdlDB_fetchRowByID
(
    MS_sqlida *sqlida,       /* <= current row of query */
    CursorID cursorID       /* => ID for cursor for fetch */
);

int mdlDB_closeCursorByID
(
    CursorID cursorID       /* => ID for cursor to close */
);
```

Description These functions operate exactly like mdlDB_openCursor, mdlDB_fetchRow and mdlDB_closeCursor cursor except that they allow the user to open more than one cursor at a time. A pool of cursors is available to the database server. Each time mdlDB_openCursorWithID is called one of these is allocated. The cursor is subsequently referenced using the identifier returned. When no more cursors are available mdlDB_openCursorWithID returns an error.

Returns mdlDB_openCursorWithID returns SUCCESS if an unused cursor was available and if the query was successfully opened. Otherwise, an error status is returned. mdlDB_fetchRowByID returns QUERY_NOT_FINISHED if a row was successfully

fetches for the indicated cursor. `QUERY_FINISHED` is returned when no rows remain in the query.

`mdlDB_closeCursorByID` returns `SUCCESS` if the indicated cursor was successfully closed. Otherwise, an error status is returned.

See Also `mdlDB_describeTable`.

mdlDB_copyTable [rdbmslib.mli]

```
#include <rdbmslib.fdf>

int mdlDB_copyTable
(
  char    *tableName,      /* => current name for the table */
  char    *newTableName    /* => new name for the table */
);
```

Description The `mdlDB_copyTable` creates a new table with the same structure as an existing table. Rows for the existing table are *not* copied into the new table.

tableName specifies the name of an existing database table.

newTableName specifies name for the new database table.

Returns `mdlDB_copyTable` returns `SUCCESS` if the new table is successfully created. Otherwise, an error status is returned. If the new table contains an `mslink` column, the column will be set to accept `NULL` values.

mdlDB_extractLinkages [rdbmslib.mli]

```
#include <rdbmslib.fdf>

int mdlDB_extractLinkages
(
  DatabaseLink **link,      /* <= array of linkages found */
  int          *linkCount,  /* <= number of linkages found */
  MSElement   *element     /* => element to extract from */
);
```

Description The `mdlDB_extractLinkages` extracts an element's database linkages. Only the linkage types identified by `MS_LINKTYPE` will be recognized by this function. The user is responsible for freeing it.

link specifies a pointer to a `DatabaseLink` which is set to point to an array of the linkages found. This function allocates the memory needed by the `DatabaseLink`.

linkCount specifies the number of linkages found on the element.

element specifies the element from which to extract linkages.

Returns `mdlDB_extractLinkages` returns `SUCCESS` if the any existing linkages were successfully extracted. Otherwise, an error status is returned.

mdlDB_describeDatabase [rdbmslib.ml]

```
#include <rdbmslib.fdf>

int mdlDB_describeDatabase
(
  StringList  **tableNames  /* <= string list of table names */
);
```

Description The mdlDB_describeDatabase returns a string list containing the names of all of the tables to which the user has access.

tableNames specifies a string list pointer which will point to a string list of the database table names. The user is responsible for calling mdlStringList_destroy on *tableNames*.

Returns mdlDB_describeDatabase returns SUCCESS if the database is successfully described. Otherwise, an error status is returned.

See Also mdlDB_copyTable [rdbmslib.ml], mdlDB_describeColumn [rdbmslib.ml].

mdlDB_describeColumn [rdbmslib.ml]

```
#include <rdbmslib.fdf>

int mdlDB_describeColumn
(
  MS_sqllda  *sqllda,      /* <= column sqllda descriptor */
  char       *tableName,   /* => name of the column's table */
  char       *columnName   /* => name of the column */
);
```

Description mdlDB_describeColumn returns a column descriptor for a column for a particular table in the database.

sqllda specifies the information about the column. Since only the information about one column is returned, just the first position of each array in *sqllda* is allocated and populated. The user is responsible for calling mdlDB_freeSQLDADescriptor on *sqllda*.

tableName specifies the name of the table containing the column.

columnName specifies the name of the column.

Returns mdlDB_describeColumn returns SUCCESS if the descriptor for the requested column is successfully retrieved. Otherwise, an error status is returned.

See Also mdlDB_copyTable [rdbmslib.ml], mdlDB_describeDatabase [rdbmslib.ml].

mdlDB_additionalRequest [rdbmslib.mtl]

```
#include <rdbmslib.fdf>

int mdlDB_additionalRequest
(
    void    **returnData,          /* <= buffer for the return data */
    int     *returnLength,        /* <= length of the return buffer */
    void    *data,                /* => data for the message */
    int     length                 /* => length of the data buffer */
);
```

Description The mdlDB_additionalRequest allows developers to define their own database request for the SQL Tool Kit.

returnData specifies a buffer that holds the return data. *returndata* is dynamically allocated in the function. The user is responsible for freeing *returndata*.

returnLength specifies the size of the return data.

data specifies the data making up the message.

length specifies the length of the data in the message.

Returns mdlDB_additionalRequest returns SUCCESS if the request was successfully processed. Otherwise, an error status is returned.

Database Settings Functions

The following table lists database settings functions:

Function	Used to
mdlDB_activeAutoCommitMode	control database commits.
mdlDB_activeDAType	set active displayable attribute type.
mdlDB_activeDatabase	attach database to design file.
mdlDB_activeDeleteMode	control deletion of linked rows during graphics delete operations.
mdlDB_activeFormsMode	selects screen forms or SQL Window for attribute review.
mdlDB_activeLinkageMode	set active linkage mode.
mdlDB_activeReportFile	name report table.
mdlDB_activeReviewTable	set SQL SELECT statement for attribute review.
mdlDB_activeRowConfirmMode	set confirmation mode for ATTACH DA and DEFINE AE commands.

Example

See database.mc.

mdlDB_activeAutoCommitMode

```
#include <dbdefs.h>
#include <dberrs.h>

int mdlDB_activeAutoCommitMode
(
    boolean      autoCommitMode /* => turn auto commit mode on */
);
```

Description The mdlDB_activeAutoCommitMode function is used to control when a transaction-capable database commits work. Most RDBMSs (Oracle, Informix, Ingres) support the concept of transactions. A transaction is a unit of work which must be committed in its entirety before being posted to the database.

Normally, the MicroStation database interfaces immediately commit each SQL statement as it is processed. Applications which require greater control over database operations can turn off automatic commits. In this mode, work is normally only committed when a COMMIT statement is submitted to the database through mdlDB_processSQL. Work performed since the last COMMIT may be undone by submitting a ROLLBACK.

Returns mdlDB_activeAutoCommitMode returns SUCCESS if the transaction mode was successfully changed.

See Also mdlDB_processSQL.

mdlDB_activeDAType

```
#include <dbdefs.h>
#include <dberrs.h>

int mdlDB_activeDAType
(
    int      daType /* => displayable attribute code */
);
```

Description mdlDB_activeDAType is used to set the active displayable attribute type. The displayable attribute type is used when MicroStation builds a displayable attribute linkage for a text node during the ATTACH DA command. The displayable attribute type is an index into the displayable attribute table defined for each entity in MSCATALOG.

The argument *daType* is an integer value between 1 and 255 that maps to a row in a displayable attributes table. The displayable attribute format may be either an SQL SELECT statement or a screen form.

mdlDB_activeDAType is equivalent to a MicroStation ACTIVE DATYPE (DA=) key-in.

Returns mdlDB_activeDAType returns SUCCESS if the displayable attribute type is successfully changed. Other error codes are listed in dberrs.h.

See Also mdlDB_buildDALinkFromLink.

mdlDB_activeDatabase

```
#include <dbdefs.h>
#include <dberrs.h>

int mdlDB_activeDatabase
(
  char      *databaseName /* => active database name */
);
```

Description The mdlDB_activeDatabase function attaches the database *databaseName* to the current design file. Most database products have the concept of a database which is a collection of tables, views, index files, etc., with user accounts and privileges associated with those objects. The argument *databaseName* defines which database (and MSCATALOG table) is associated with a design file.

The mdlDB_activeDatabase function is equivalent to a MicroStation ACTIVE DATABASE (DB=) key-in.



Oracle is an exception. Oracle collects tables and other database objects together on a user account basis. The function mdlDB_activeDatabase has no meaning for Oracle. Instead, the user must be connected to the database using the CONNECT statement.

Returns mdlDB_activeDatabase returns SUCCESS if the database is successfully attached. Other error codes are listed in dberrs.h.

See Also mdlDB_processSQL.

mdlDB_activeDeleteMode

```
#include <dbdefs.h>
#include <dberrs.h>

int mdlDB_activeDeleteMode
(
  boolean      deleteRows /* => delete linked rows */
);
```

Description mdlDB_activeDeleteMode is used to control whether database rows linked to graphics elements are deleted when the graphics element is deleted or the linkages are detached. The argument *deleteRows* is TRUE when linked rows are to be deleted

during graphics delete and detach linkage operations. A value of `FALSE` means delete and detach operations have no effect on linked rows.

`mdlDB_activeDeleteMode` is equivalent to the MicroStation SET DELETE (ON | OFF) key-in.

Returns `mdlDB_activeDeleteMode` returns `SUCCESS` if the deletion mode was successfully changed. Other error codes are listed in `dberrs.h`.

mdlDB_activeFormsMode

```
#include <dbdefs.h>
#include <dberrs.h>

int mdlDB_activeFormsMode
(
    boolean useForms /* => use forms */
);
```

Description The `mdlDB_activeFormsMode` function controls when MicroStation uses screen forms for attribute review and editing. If the argument *useForms* is `TRUE` then the screen forms defined in MSCATALOG for each entity will be used for attribute review. If *useForms* is `FALSE` then the SQL Window dialog box will be used.

`mdlDB_activeFormsMode` is equivalent to using the MicroStation SET FORMS (ON | OFF) key-in.

Returns `mdlDB_activeFormsMode` returns `SUCCESS` if the forms mode was successfully changed. Other error codes are listed in `dberrs.h`.

mdlDB_activeLinkageMode

```
#include <dbdefs.h>
#include <dberrs.h>

int mdlDB_activeLinkageMode
(
    char *mode /* => linkage mode */
);
```

Description `mdlDB_activeLinkageMode` sets the active database linkage mode. The linkage mode controls how and when MicroStation adds new rows to the database. When linkages are attached to graphics elements and when elements are copied,

MicroStation uses the linkage mode to determine when to add new rows to the database.

The argument *mode* is a character string which specifies the linkage mode. The string may be one of the following values:

Linkage Mode	Meaning
"new"	each element is linked to a different row
"duplicate"	many elements may be linked to the same row
"information"	like duplicate mode but don't contribute to reports
"none"	no linkages possible

mdlDB_activeLinkageMode is equivalent to the MicroStation key-in ACTIVE LINK (NEW | DUPLICATE | INFORMATION | NONE). Changing the linkage mode will clear the active entity.

Returns mdlDB_activeLinkageMode returns SUCCESS if the linkage mode was successfully changed. Other error codes are listed in dberrs.h.

mdlDB_activeReportFile

```
#include <dbdefs.h>
#include <dberrs.h>

int mdlDB_activeReportFile
(
  char    *reportSpecification    /* => report table spec. */
);
```

Description The mdlDB_activeReportFile function is used to define the report table to be created for an entity during a FENCE REPORT operation. During report operations MicroStation creates a new table having the same structure as the primary table and fills the table with each row linked to graphics elements within the fence.

The format of the *reportSpecification* argument is

```
table:reportTable
```

where *table* represents a table in MSCATALOG and *reportTable* is the name of the table to be created during a FENCE REPORT operation.

mdlDB_activeReportFile is equivalent to the MicroStation ACTIVE REPORT (RS=) key-in.

Returns mdlDB_activeReportFile returns SUCCESS if the report table is assigned successfully. Other error codes are listed in dberrs.h.

See Also mdlDB_activeReviewTable.

mdlDB_activeReviewTable

```
#include <dbdefs.h>
#include <dberrs.h>

int mdlDB_activeReviewTable
(
    char    *sqlReview    /* => SQL SELECT review statement */
);
```

Description mdlDB_activeReviewTable is used to specify an SQL SELECT statement to be used for attribute review operations. Every entity has an (optional) SQL SELECT statement which is saved in the SQLREVIEW column of MSCATALOG. This SELECT statement is used for attribute review when screen forms mode is turned off.

The argument *sqlReview* specifies an SQL SELECT statement. The first table referenced in the FROM clause is assumed to be the target table. The SQLREVIEW column of this table row in MSCATALOG is updated.

The mdlDB_activeReviewTable function is equivalent to the MicroStation ACTIVE REVIEW (RA=) key-in.

Returns mdlDB_activeReviewTable returns SUCCESS if the SQLREVIEW column is saved correctly in MSCATALOG. Other error codes are listed in dberrs.h.

See Also mdlDB_activeReportFile.

mdlDB_activeRowConfirmMode

```
#include <dbdefs.h>
#include <dberrs.h>

int mdlDB_activeRowConfirmMode
(
    boolean    confirmRows    /* => confirm rows if multiple linkages */
);
```

Description The mdlDB_activeRowConfirmMode function controls when MicroStation prompts for confirmation on a database operation which requires selecting a graphics element with an existing linkage. The DEFINE AE and ATTACH DA commands require selecting a graphics element which may contain more than one linkage. If there are multiple linkages and *confirmRows* is TRUE MicroStation will prompt for confirmation of the row in the SQL Window. If *confirmRows* is FALSE, the first linkage on the element will be selected.

mdlDB_activeRowConfirmMode is equivalent to the MicroStation SET CONFIRM (ON | OFF) key-in.

Returns mdlDB_activeRowConfirmMode returns SUCCESS if the confirmation mode is successfully set. Other error codes are listed in dberrs.h.

Database Dialog Functions

MicroStation DBForms is an MDL tool kit that provides an extensive set of functions required to create any MicroStation dialog box based database application. With these functions, you can generate multi-paged database forms on the fly, automating its every operation, save them in a resource file (so you can modify them later using the DBDIALOG or BUILDER applications) etc. The MDL application, DBDIALOG itself is developed using these functions.

The following table lists database dialog functions available with MicroStation DBForms tool kit. Unless otherwise noted, the DBForms functions were initially implemented in MicroStation SE.

Function	Used to
mdlDBDialog_attachCurrentRow	attach the current row to the element(s) selected.
mdlDBDialog_clearValues	clear the database form fields.
mdlDBDialog_deleteCurrentRow	delete the current record.
mdlDBDialog_detachCurrentRow	detach the current row from the selected element(s).
mdlDBDialog_firstRow	make the first row selected as the active record of the database form.
mdlDBDialog_generateRscFile	create a database dialog resource file using the specified table name, column list, criteria, item list and attributes.
mdlDBDialog_insertRow	insert a new row to the table with column values present on the database form.
mdlDBDialog_itemGetState	get value of the database form field.
mdlDBDialog_itemSetState	set value of the database form field.
mdlDBDialog_lastRow	make the last row selected as the active record of the database form.
mdlDBDialog_locateCurrentRow	locate elements having linkages to the current row.
mdlDBDialog_nextPage	scroll to next page in case of multi-paged database forms.
mdlDBDialog_nextRow	scroll one row forward and display the new record as the active one for the database form.
mdlDBDialog_openForm	open a database form using the specified table name, column list, criteria, item list and attributes.

Function	Used to
mdlDBDialog_openFormFromElement	open a form for each database linkage on the given element.
mdlDialog_openWithDBQuery	open database dialog box from the opened resource file.
mdlDBDialog_prevPage	scroll to previous page in case of multipaged database forms.
mdlDBDialog_prevRow	scroll one row backward and display the new record as the active one for the database form.
mdlDBDialog_processUserQuery	execute a user query on a database form.
mdlDBDialog_publishExtraHooks	publish hooks for the additional database dialog items.
mdlDBDialog_queryRscFile	retrieve the dialog resource ID and primary table name from the database form resource file.
mdlDBDialog_review	extract records from the element(s) selected and populate the database form.
mdlDBDialog_setPage	change page number in random in case of multipaged database forms.
mdlDBDialog_startQuery	start a new query based on values and operators entered on the database form.
mdlDBDialog_updateCurrentRow	update the current record with column values present on the database form.
mdlDBDialog_useFenceIfActive	set or get the use fence flag for a database form.

mdlDBDialog_attachCurrentRow

```
#include <dbdlglib.fdf>

int mdlDBDialog_attachCurrentRow
(
    DialogBox *dbP    /* => database form with the row to attach */
);
```

Description mdlDBDialog_attachCurrentRow prompts and attaches the current row displayed in the database form to the element(s) selected. This also works on fence elements if a fence is active and the Use Fence flag is not set to OFF by calling mdlDBDialog_useFenceIfActive.

Unlike the ATTACH AE command, this function does not require or use an AE table.

Returns `mdlDBDialog_attachCurrentRow` returns `SUCCESS` if database form contains enough information to create a database linkage.

See Also `mdlDBDialog_detachCurrentRow`, `mdlDBDialog_useFenceIfActive`.

mdlDBDialog_clearValues

```
#include <dbdlglib.fdf>

int mdlDBDialog_clearValues
(
    DialogBox *dbP    /* => database form to clear values */
);
```

Description `mdlDBDialog_clearValues` clears all column values from the database form, *dbP*.

Returns `mdlDBDialog_clearValues` returns `SUCCESS` if *dbP* is a valid database form.

See Also `mdlDBDialog_openForm`, `mdlDBDialog_startQuery`, `mdlDialog_openWithDBQuery`.

mdlDBDialog_deleteCurrentRow

```
#include <dbdlglib.fdf>

int mdlDBDialog_deleteCurrentRow
(
    DialogBox *dbP    /* => database form with the row to delete */
);
```

Description `mdlDBDialog_deleteCurrentRow` deletes the current record from the primary table. Delete is done based on the primary key value.

Returns `mdlDBDialog_deleteCurrentRow` returns `SUCCESS` if the record is successfully deleted from the primary table.

See Also `mdlDBDialog_insertRow`.

mdlDBDialog_detachCurrentRow

```
#include <dbdlglib.fdf>

int mdlDBDialog_detachCurrentRow
(
    DialogBox *dbP    /* => database form with the row to detach */
);
```

Description `mdlDBDialog_detachCurrentRow` prompts and detaches the current row displayed in the database form from the element(s) selected. This also works on fence elements if a fence is active and the Use Fence flag is not set to OFF by calling `mdlDBDialog_useFenceIfActive`.

Unlike MicroStation DETACH command, this function does not detach all the linkages but only the one displayed in the form.

Returns `mdlDBDialog_detachCurrentRow` returns `SUCCESS` if database form contains enough information to decide which database linkage to detach.

See Also `mdlDBDialog_attachCurrentRow`, `mdlDBDialog_useFenceIfActive`.

mdlDBDialog_firstRow, mdlDBDialog_lastRow, mdlDBDialog_nextRow, mdlDBDialog_prevRow

```
#include <dbdlglib.fdf>

int mdlDBDialog_firstRow
(
    DialogBox *dbP      /* => Database form to scroll to first row */
);

int mdlDBDialog_lastRow
(
    DialogBox *dbP      /* => Database form to scroll to last row */
);

int mdlDBDialog_nextRow
(
    DialogBox *dbP      /* => Database form to scroll next row */
);

int mdlDBDialog_prevRow
(
    DialogBox *dbP      /* => Database form to scroll previous row */
);
```

Description `mdlDBDialog_firstRow` synchronizes the database form, *dbP* with the first record of the query.

`mdlDBDialog_lastRow` synchronizes the database form, *dbP* with the last record of the query. `mdlDBDialog_nextRow` synchronizes the database form *dbP* with the next record of the query.

`mdlDBDialog_prevRow` synchronizes the database form, *dbP* with the previous record of the query. This function only works for key array based database forms.

Returns These functions return `SUCCESS` if *dbP* is a valid database form with an active query.

See Also `mdlDBDialog_openForm`, `mdlDialog_openWithDBQuery`.

mdlDBDialog_generateRscFile

```
#include <dbdlglib.fdf>

int mdlDBDialog_generateRscFile
(
    char *rscFileName, /* => resource file to generate */
    int dialogId,      /* => dialog box resource ID */
);
```

```
char *tableName, /* => primary table name */
char *columnList, /* => NULL means include all columns */
Ulong itemMask, /* => default items */
Ulong attributes, /* => not implemented, pass 0L always */
);
```

Description `mdlDBDialog_generateRscFile` generates (creates, writes and closes) a resource file containing the dialog and item resources of the database form using the specified primary tablename, column list and database features. The resource file created by this function can be edited using the DBDIALOG (Resource->Edit) or BUILDER applications and can be used with `mdlDialog_openWithDBQuery` or `mdlDBDialog_openFormFromElement`.

rscFileName specifies target file to be created.

dialogId specifies the unique resource Id to be assigned to the database dialog form.

tableName specifies the primary table name for which the database form is opened. This is also the title of the form.

columnList specifies the list of columns that should appear on the form. Column names are separated by commas. If *columnList* is NULL, all the columns of the primary table will appear on the database form.

itemMask specifies the standard database features the database form should come up with. See `mdlDBDialog_openForm` for more information.

Returns `mdlDBDialog_generateRscFile` returns SUCCESS if the resource file is generated successfully and ERROR otherwise.

See Also `mdlDBDialog_openForm`, `mdlDBDialog_queryRscFile`, `mdlDialog_openWithDBQuery`.

mdlDBDialog_insertRow

```
#include <dbdlglib.fdf>

int mdlDBDialog_insertRow
(
DialogBox *dbP /* => database form with values to insert */
);
```

Description `mdlDBDialog_insertRow` inserts a new row to the primary table with the column values present in the database form. If the primary key is MSLINK and a record is already existing with the given MSLINK value then `mdlDBDialog_insertRow` will attempt to insert the record with the next available MSLINK value.

Returns `mdlDBDialog_insertRow` returns SUCCESS if the record is successfully added to the primary table.

See Also `mdlDBDialog_deleteCurrentRow`.

mdlDBDialog_itemGetState, mdlDBDialog_itemSetState

```
#include <dbdlglib.fdf>

int mdlDBDialog_itemGetState
(
    char      **valuePP, /* <= item's external state */
    DialogBox *dbP,      /* => Database form to get item state */
    int       itemIndex /* => item index of item for which to get state */
);

int mdlDBDialog_itemSetState
(
    DialogBox *dbP,      /* => Database form to set item state */
    int       itemIndex, /* => item index of item for which to set state */
    char      *valueP    /* => state to set item to */
);
```

Description mdlDBDialog_itemGetState gets a pointer to the value of the item at *itemIndex* on the database form *dbP*.

mdlDBDialog_itemSetState sets the value of the item at *itemIndex* on the database form *dbP*.

valuePP specifies the pointer to the item value. Users should never modify this pointer.

valueP specifies the new value to set the item's state with.

Returns mdlDBDialog_itemGetState returns SUCCESS if the value is retrieved successfully.

mdlDBDialog_itemSetState returns SUCCESS if the value is set successfully.

See Also mdlDBDialog_insertRow, mdlDBDialog_updateCurrentRow.

mdlDBDialog_locateCurrentRow

```
#include <dbdlglib.fdf>

int mdlDBDialog_locateCurrentRow
(
    DialogBox *dbP /* => database form with the row to locate */
);
```

Description mdlDBDialog_locateCurrentRow locates one after the other, all elements linked to the current row from the map pointed by the MAPID field. If the primary table does not have a MAPID field, it looks at the master file for elements linked to the current row. Located elements are hilited and displayed with a red halo around them in the selected view. Until there are no more elements linked to the current

row to be displayed, it keeps prompting you to identify views to display the elements.

Returns `mdlDBDialog_locateCurrentRow` returns `SUCCESS` if database form contains enough information to decide which database linkage to locate.

See Also `mdlDBDialog_review`, `mdlDBDialog_attachCurrentRow`.

mdlDBDialog_nextPage, mdlDBDialog_prevPage, mdlDBDialog_setPage

```
#include <dbdlglib.fdf>

int mdlDBDialog_nextPage
(
    DialogBox *dbP      /* => Database form to scroll to next page */
);

int mdlDBDialog_prevPage
(
    DialogBox *dbP      /* => Database form to scroll to previous page */
);

int mdlDBDialog_setPage
(
    DialogBox *dbP,      /* => Database form to set page */
    int      pageNum    /* => page number to set */
);
```

Description `mdlDBDialog_nextPage` scrolls the multi-paged database form *dbP* to the next page.

`mdlDBDialog_prevPage` scrolls the multi-paged database form *dbP* to the previous page.

`mdlDBDialog_setPage` scrolls the multi-paged database form *dbP* to *pageNum*.

Returns These functions return `SUCCESS` if *dbP* is a multi-paged database form with a suitable current state.

See Also `mdlDBDialog_openForm`, `mdlDialog_openWithDBQuery`.

mdlDBDialog_openForm

```
#include <dbdlglib.fdf>
#include <dbform.h>

DialogBox* mdlDBDialog_openForm
(
    int      dialogId,      /* => unique id for the database dialog form */
    char     *tableName,    /* => primary table name */
    char     *columnList,   /* => NULL means include all columns */
    char     *condition,    /* => WHERE clause used at opening execution */
    ULONG    itemMask,      /* => default items */
    ...
);
```

```
ULong  attributes      /* => form attributes */
);
```

Description mdlDBDialog_openForm dynamically creates and displays an MDL based multi page database form using the specified primary tablename, column list, query conditions, database features and attributes.

dialogId specifies the unique id to be assigned to the new database form. Users must make sure that this id is different from other dialog box resource Ids present in their application.

tableName specifies the primary table name for which the database form is opened. This is also the title of the form.

columnList specifies the list of columns that should appear on the form. Column names are separated by commas. If *columnList* is NULL, all of the columns of the primary table will appear on the database form.

condition specifies the WHERE clause used while creating the SELECT statement which is getting executed on opening the form. If *condition* is NULL, all rows of the primary table will be selected.

itemMask specifies the standard database features the database form should come up with. This must be a bitmapped ULong that may contain any combinations of the following constants.

Constant	Description
DBDIALOG_BUTTON_ATTACH	Include Attach Linkage push button.
DBDIALOG_BUTTON_DEFAULT	Add all the default database features which includes First, Next and Last buttons for record scrolling, Query, Clear, Insert, Update and Delete buttons for record modification.
DBDIALOG_BUTTON_DETACH	Include Detach Linkage push button.
DBDIALOG_BUTTON_LOCATE	Include Locate Linkage push button.
DBDIALOG_BUTTON_NOCLEAR	Do not include Clear push button.
DBDIALOG_BUTTON_NODELETE	Do not include Delete push button.
DBDIALOG_BUTTON_NOFIRST	Do not include First push button for record scrolling.
DBDIALOG_BUTTON_NOINSERT	Do not include Insert push button.
DBDIALOG_BUTTON_NOLAST	Do not include Last push button for record scrolling.
DBDIALOG_BUTTON_NONEXT	Do not include Next push button for record scrolling.
DBDIALOG_BUTTON_NOQUERY	Do not include Query push button
DBDIALOG_BUTTON_NOUPDATE	Do not include Update push button.
DBDIALOG_BUTTON_PREVROW	Include Previous push button for record scrolling. This is valid only for the default key array based database forms. (=> if DBDIALOGATTR_CURSORBASED bit is not set in attributes)

Constant	Description
DBDIALOG_BUTTON_REVIEW	Include Review Linkage push button.
DBDIALOG_GRAPHICS_ITEMS	Add all the default database linkage features which includes Attach, Review, Detach and Locate buttons.
DBDIALOG_LABEL_COUNTROW	Include the active and total record count labels at the top of the form. This is valid only for the default key array based database forms.
DBDIALOG_MLTEXT_QUERY	Include a multiline text box where user can enter SQL queries.
DBDIALOG_TOGGLE_USEFENCE	Include 'Use Fence if Active' toggle button to decide whether to use fence while doing graphic operation.

The *attributes* parameter specifies various attributes for the database form. Default is 0L. At present there are only two options supported. This must be a bitmapped ULong that may contain any combinations of the following constants.

Constant	Description
DBDIALOGATTR_CURSORBASED	Use the native Database Cursor based form instead of the default key array based. Cursor based form does not support the Previous record feature or displaying record Counts.
DBDIALOGATTR_NOQUERYEXECUTE	Do not execute queries or fetch column values on opening the form. This makes the form come up without any delay.

Returns mdlDBDialog_openForm returns a pointer to the opened dialog box. NULL is returned if the table or column list specified is not valid or the given dialog id is already in use.

See Also mdlDBDialog_publishExtraHooks, mdlDialog_openWithDBQuery, mdlDBDialog_openFormFromElement.

mdlDBDialog_openFormFromElement

```
#include <dbdlglib.fdf>

int mdlDBDialog_openFormFromElement
(
  MSElementUnion *el, /* =>linked element */
  int *dialogId /* =>id of first form if dynamic creation */
);
```

Description mdlDBDialog_openFormFromElement opens database form for each linkage on the element, *el*. If *dialogId* is NULL, it looks at the directory pointed by MS_DBDIALOGRSC

to check whether dialog resource files with the names matching to the entity number (i.e., parcel.rsc) exist there. If found, it uses these resources to create the database forms.

If *dialogId* is not NULL, it calls mdlDBDialog_openForm for each linkage with the entity table as primary table listing all columns and using form Id starting from the *dialogId* value.

Returns mdlDBDialog_openFormFromElement returns SUCCESS if database forms are opened successfully.

See Also mdlDBDialog_generateRscFile, mdlDialog_openWithDBQuery.

mdlDialog_openWithDBQuery

```
#include <msdialog.fdf>
#include <dlogbox.h>

DialogBox *mdlDialog_openWithDBQuery
(
RscFileHandle rFileH,      /* => NULL means search opened rsc files */
void          *ownerMD,    /* => who will owner mdl task of dialog */
int           resourceId,  /* => id of dialog to open */
char          *primaryTableName, /* => name of primary table */
char          *dbQueryStrP, /* => SQL SELECT statement */
BoolInt       executeQuery, /* => TRUE to execute at open */
ULong        attributes    /* => additional form attributes if any */
);
```

Description mdlDialog_openWithDBQuery is a low level function which opens a database form from a resource file. Database form is a dialog box with database attribute bit DIALOGATTR_DATABASE set.

rFileH specifies the resource file to search for the database form resource. If NULL, all the calling application's open resource files and then MicroStation's open resource files will be searched.

resourceId specifies the resource ID of the dialog box resource that is used to create the database form. The dialog resource for a database form can contain items hooked to database fields. For example, users may hook the field 'owner' from 'property' table to the text item TEXTID_Owner in either of the two methods listed below:

```
...
{{7*XC, GENY(3), 45*XC, 0}, Text, TEXTID_Owner, ON, 0,
TXT_TextStreet, "dbaccess=\"property.owner\"",
...
```

or

```
DItem_TextRsc TEXTID_Owner=
{
    NOCMD, MCMD, NOSYNONYM, NOHELP, MHELP, NOHOOK, NOARG,
```

```

        64, "%s", "%s", "", "", NOMASK, 0, TXT_TextOwner, ""
    }

    extendedAttributes
    {
        {
            {EXTATTR_DBACCESS, "property.owner"},
            {EXTATTR_DBPAGENUMBER, "2"} /* appear on second page */
        }
    };

```

The extended attribute, `EXTATTR_DBPAGENUMBER` specifies the page number where the item should appear on the database form.

primaryTableName specifies the primary table name for which the database form is opened.

dbQueryStrP specifies the SELECT statement to be executed when the database form comes up if *executeQuery* is TRUE.

attributes specifies various attributes for the database form. Default is 0L. Currently only one attribute is supported.

Constant	Description
DBDIALOGATTR_CURSORBASED	Use the native Database Cursor based form instead of the default key array based. Cursor based form does not support the Previous record feature or displaying record Counts.

Returns `mdlDialog_openWithDBQuery` returns a pointer to the opened dialog box. NULL is returned if the dialog box resource was not found or database server is not active or primary table name is not valid or errors occurred while loading the items contained in the dialog box.

See Also `mdlDBDialog_openForm`, `mdlDBDialog_openFormFromElement`.

mdlDBDialog_processUserQuery

```

#include <dbdlglib.fdf>

int mdlDBDialog_processUserQuery
(
    DialogBox *dbP,          /* => database form to execute the query */
    char *userQueryP /* => user query, NULL => all rows */
);

```

Description `mdlDBDialog_processUserQuery` selects all the rows satisfying the WHERE clause specified by *userQueryP* and synchronizes the form with the query results.

userQueryP can be any valid SQL WHERE clause (i.e., "WHERE MSLINK<= 10"). If NULL is passed, it selects all the rows.

Returns mdlDBDialog_processUserQuery returns SUCCESS if no errors occur.

See Also mdlDBDialog_startQuery.

mdlDBDialog_publishExtraHooks

```
#include <dbdlglib.fdf>

void mdlDBDialog_publishExtraHooks(void);
```

Description mdlDBDialog_publishExtraHooks publishes the hooks for some of the database items created by mdlDBDialog_openForm functions. This needs to be called once in the application after publishing the user hooks. It is advisable to call this function in the following order if your application uses one of the mdlDBDialog_openForm functions.

```
mdlDialog_publishHooks(..);
mdlDBDialog_publishExtraHooks();
```

Returns mdlDBDialog_publishExtraHooks is of type void and does not return value.

See Also mdlDBDialog_openForm, mdlDBDialog_openFormFromElement.

mdlDBDialog_queryRscFile

```
#include <dbdlglib.fdf>

int mdlDBDialog_queryRscFile
(
    int          *dialogId,    /* <= dialog form id */
    char         *tableName,   /* <= primary table name */
    RscFileHandle rFileH      /* => handler for form resource file*/
);
```

Description mdlDBDialog_queryRscFile retrieves the database form id and table name from the resource file created by mdlDBDialog_generateRscFile.

dialogId specifies the unique resource id assigned to the database dialog form. If NULL, it is not retrieved.

tableName specifies the primary table name of the database form present in the resource file. If NULL, table name is not retrieved.

rFileH specifies the resource file to search for the database form resource.

Returns mdlDBDialog_queryRscFile returns SUCCESS if the requested information is retrieved successfully and ERROR otherwise.

See Also mdlDBDialog_generateRscFile, mdlDialog_openWithDBQuery.

mdlDBDialog_review

```
#include <dbdlglib.fdf>

int mdlDBDialog_review
```

```
(
DialogBox *dbP    /* => database form to output review */
);
```

Description `mdlDBDialog_review` prompts and reviews the database linkages of the element(s) linked to the primary table. It initializes current query and synchronizes the database form with the review results. This also works on fence elements if a fence is active and the Use Fence flag is not set OFF by calling `mdlDBDialog_useFenceIfActive`.

Unlike MicroStation REVIEW command, this function does not review all linkages, but only the ones to the primary table of the database form *dbP*.

Returns `mdlDBDialog_review` returns `SUCCESS` if the primary table of the database form has `MSLINK` column and an entry in `MSCATALOG` table.

See Also `mdlDBDialog_openFormFromElement`, `mdlDBDialog_useFenceIfActive`.

mdlDBDialog_startQuery

```
#include <dbdlglib.fdf>

int mdlDBDialog_startQuery
(
DialogBox *dbP    /* => database form to generate the query */
);
```

Description `mdlDBDialog_startQuery` creates and executes the SQL query generated from the database form and displays the first query result. If found, it first attempts to form a SELECT statement from the multiline query box item `MLTEXTID_DBQuery`. Users can enter any valid WHERE clause in this field before calling `mdlDBDialog_startQuery`.

In case of no or empty multiline query item, `mdlDBDialog_startQuery` attempts to create and execute the SELECT statement using the values and operators available with the database field items. The default arithmetical operator is EQUAL TO (=). Other operators recognized by `mdlDBDialog_startQuery` are listed below:

Operator	Description
%	like
!	not equal to
<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to

AND is the default logical operator. Users need to add a pipe ‘|’ character at the end of the values to use OR as the connection operator instead.

If all the fields are cleared before calling `mdlDBDialog_startQuery`, it selects all the rows from primary table.

Returns `mdlDBDialog_startQuery` returns `SUCCESS` if no errors occur.

See Also `mdlDBDialog_processUserQuery`, `mdlDBDialog_clearValues`.

mdlDBDialog_updateCurrentRow

```
#include <dbdlglib.fdf>

int mdlDBDialog_updateCurrentRow
(
    DialogBox *dbP      /* => database form with the row to update */
);
```

Description `mdlDBDialog_updateCurrentRow` updates the current record with column values present in the database form. Update is done based on the primary key value.

Returns `mdlDBDialog_updateCurrentRow` returns `SUCCESS` if the record is successfully updated.

See Also `mdlDBDialog_insertRow`.

mdlDBDialog_useFenceIfActive

```
#include <dbdlglib.fdf>

int mdlDBDialog_useFenceIfActive
(
    DialogBox *dbP,      /* => database form to set/get use fence flag */
    int useFence         /* => use fence flag, TRUE, FALSE or -1 */
);
```

Description `mdlDBDialog_useFenceIfActive` sets or gets the USE FENCE flag associated with the database form *dbP* to decide whether to use the fence elements with the linkage functions if a fence is active.

If *useFence* is either `TRUE` or `FALSE`, it sets the USE FENCE flag to ON or OFF. If `-1`, is passed it retrieves the current state of the USE FENCE flag.

Returns `mdlDBDialog_useFenceIfActive` returns `SUCCESS` if *useFence* is either `TRUE` or `FALSE` and the flag is set successfully. If *useFence* is `-1`, it returns `TRUE` or `FALSE` depending on USE FENCE flag is ON or OFF.

See Also `mdlDBDialog_review`, `mdlDBDialog_attachCurrentRow`, `mdlDBDialog_detachCurrentRow`.

Tag Functions

Tag functions are used to manipulate tag data in a design file. Tag data is non-graphical data which is associated with a graphical element. Tag data is not the same as the user data, which can be appended to any MicroStation element. Tag data is in a special element (type 37).

Tags are logically grouped into tag **sets**. The first step taken to attach tags to an element is to define the set of tags to be used. The name of the tag set must be unique within the design file. The tag set definition contains information such as the data type of the tag, default value, and default display properties for each tag in the set. Each tag in a particular set must have a unique name. However, tags in two different sets can have the same name. After tags are defined they can be placed or attached to a MicroStation element. When a tag is attached to a base element a new tag element is created and associated to the base element.



Tag and tag set names are not case sensitive.

Tag set libraries are resource files that contain a resource class designed to hold tag set definitions. The resource class is `TgLb` and its type is defined in the published header file `tagdata.h`. Tag set libraries have a default file extension of `.tlb`. All of the tag set definitions in a library have the same resource ID (currently 1). The resource's alias is the tag set name and is used to distinguish between tags sets in a tag set library. Because tag set definitions are stored in the design file in Type 66 elements, the import and export options in the Tag Sets settings box convert tag set definitions between Type 66 format and the new external resource format.

The structures used by these functions to define a tag are defined in the file `tagdata.h` as follows:

```
typedef struct tagsetspec
{
    char setName[TAG_SET_NAME_MAX];
    char reportName[TAG_MAX_RPT_NAME];
    int fileNo;
    int futureUse;
} TagSetSpec;

typedef struct tagspec
{
    TagSetSpec set;
    char tagName[TAG_NAME_MAX];
} TagSpec;

typedef struct tagvalue
{
    UShort type;
    UShort size;          /* in bytes; valid for userVal & binary */
}
```

```
union
{
    double doubleVal;
    long longVal;
    short shortVal;
    char *stringVal;
    void *binaryVal;
    void *userVal;
} val;
} TagValue;

typedef struct tagDefinition
{
    char name[TAG_NAME_MAX];
    UShort id;
    char prompt[TAG_PROMPT_MAX];
    UShort propsMask;
    char styleName[TAG_MAX_STYLE_LEN];
    TagValue value;
} TagDef;
```

The `id` member of the `TagDef` structure indicates the index of the tag definition in the set. This value is used internally and should be ignored.

The following table lists Tag Data functions:

Function	Used to
<code>mdlTag_createSetDef</code>	create the definition of a tag set.
<code>mdlTag_deleteSetDef</code>	delete the definition of a set.
<code>mdlTag_setSetDef</code>	update the definition of a set.
<code>mdlTag_getSetNames</code>	get a list of names of the defined sets.
<code>mdlTag_getSetDef</code>	get the definition of a set.
<code>mdlTag_addTagDefToSet</code>	add a new tag definition to a set.
<code>mdlTag_getTagDef</code>	get the definition of a tag.
<code>mdlTag_setTagDef</code>	update the definition of a tag.
<code>mdlTag_create</code>	create a tag instance.
<code>mdlTag_extract</code>	extract information about a tag.
<code>mdlTag_deleteTagInstances</code>	delete all instances of a given tag.
<code>mdlTag_deleteTagDef</code>	delete a tag definition from a set.
<code>mdlTag_freeTagDef</code>	free a tag definition.
<code>mdlTag_freeTagDefArray</code>	free an array of tag definitions.
<code>mdlTag_getElementTags</code>	get all tags for a given element.

Function	Used to
mdlTag_generateReport	generate a report for a tag set.
mdlTag_getAssocElement	get the base element to which a tag is attached.

mdlTag_createSetDef

```
#include <mstagdat.fdf>

int mdlTag_createSetDef
(
  char    *setName,      /* => name of set to create */
  int     numTags,      /* => number of tags in set */
  TagDef  *tagDefs,     /* => array of tag definitions */
  char    *reportName,  /* => name of report file */
  boolean futureArg1,   /* => for future use, always pass TRUE */
  int     futureArg2    /* => for future use, always pass 0 */
);
```

Description The mdlTag_createSetDef function creates a definition for a tag set.

setName specifies the name of the set to be created. Set names must be unique within a design file. The maximum size of a tag set name is defined by TAG_SET_NAME_MAX.

numTags specifies the number of tags in the set to be created.

tagDefs specifies an array of tag definitions.

reportName is the name of the report file for the set. If the name of the report file is not being set, pass NULL.

Returns mdlTag_createSetDef returns SUCCESS if the tag set was successfully created. Otherwise, an error status is returned.

See Also mdlTag_deleteSetDef, mdlTag_getSetDef.

mdlTag_deleteSetDef

```
#include <mstagdat.fdf>

int mdlTag_deleteSetDef
(
  char    *setName,      /* => name of set to delete */
  int     futureArg      /* => for future use, always pass 0 */
);
```

Description The mdlTag_deleteSetDef function deletes the definition for a tag set.

setName specifies the name of the set to be deleted. The maximum size of a tag set name is defined by TAG_SET_NAME_MAX.

Returns mdlTag_deleteSetDef returns SUCCESS if the tag set was successfully deleted. Otherwise, an error is returned.

See Also mdlTag_createSetDef, mdlTag_getSetDef.

mdlTag_setSetDef

```
#include <mstagdat.fdf>

int mdlTag_setSetDef
(
    char    *setName,      /* => name of the set */
    int     futureArg,     /* => for future use, always pass 0 */
    char    *newSetName,   /* => new name for the set */
    char    *reportName    /* => name for report file */
);
```

Description The mdlTag_setSetDef updates the definition of a tag set.

setName specifies the name of the set to update.

newSetName specifies the new name for the set. If the name of the set is not being changed, pass NULL.

reportName is the name of the report file for the set. If the name of the report file is not being changed, pass NULL.

Returns mdlTag_setSetDef returns SUCCESS if the set definition is successfully updated. Otherwise, an error is returned.

See Also mdlTag_createSetDef, mdlTag_addTagDefToSet.

mdlTag_getSetNames

```
#include <mstagdat.fdf>

int mdlTag_getSetNames
(
    StringList **stringList, /* <= string list of set names */
    int         *numSets,    /* <= number of sets found */
    int         fileNo,      /* => file number to search in */
    int         futureArg    /* => for future use, always use 0 */
);
```

Description The mdlTag_getSetNames function retrieves a sorted string list of names of available tag sets in a given design file.

stringList specifies a pointer to a string list to be set by this function.

numSets specifies a pointer to an integer which is set to the number of sets found.

fileNo specifies the file which is to be searched for the tag sets.

Returns `mdlTag_getSetNames` returns `SUCCESS` if the design file contains at least one tag set and the string list was successfully created. Otherwise, an error is returned.

See Also `mdlTag_createSetDef`, `mdlTag_getSetDef`, `mdlTag_deleteSetDef`.

mdlTag_getSetDef

```
#include <mstagdat.fdf>

int mdlTag_getSetDef
(
    int      *numTags,          /* <= # of tags in the set */
    TagDef   **tagDefs,        /* <= array of tag definitions */
    char     *reportName,      /* <= report file name for set */
    boolean  *futureArg1,      /* <= for future use, always pass NULL */
    char     *setName,         /* => set to get definition for */
    int      fileNo,           /* => file number to search in */
    int      futureArg2        /* => for future use, always use 0 */
);
```

Description `mdlTag_getSetDef` retrieves an array of tag definitions for a given set.

numTags specifies a pointer to an integer which is set to the number of tags in the set.

tagDefs specifies a tag definition pointer which points to the array of tag definitions found. MicroStation allocates the memory for *tagDefs*, and the calling application should free it by calling `mdlTag_freeTagDefArray` when it is no longer needed.

reportName is the name of the report file for the set.

setName specifies the set whose tag definitions should be retrieved.

fileNo specifies the file number in which to search for the set definition.

Returns `mdlTag_getSetDef` returns `SUCCESS` if the definition of the tag set was found. Otherwise, an error is returned.

See Also `mdlTag_createSetDef`, `mdlTag_deleteSetDef`.

mdlTag_addTagDefToSet

```
#include <mstagdat.fdf>

int mdlTag_addTagDefToSet
(
    TagDef   *tagDef,          /* => definition of new tag */
    char     *setName,         /* => set to add definition to */
    int      futureArg         /* => for future use, always use 0 */
);
```

Description The mdlTag_addTagDefToSet function adds a new tag definition to an existing set.
tagDef specifies a pointer to the definition of the new tag.

setName specifies the name of the tag set where the new tag should be added.

Returns mdlTag_addTagDefToSet returns SUCCESS if the set definition was found and the new tag definition was added successfully. Otherwise, an error is returned.

See Also mdlTag_getTagDef, mdlTag_setTagDef, mdlTag_deleteTagDef.

mdlTag_getTagDef

```
#include <mstagdat.fdf>

int mdlTag_getTagDef
(
    TagDef **tagDef,      /* <= definition of the tag */
    char *setName,        /* => set name containing tag */
    char *tagName,        /* => tag name to get definition for */
    int fileNo,           /* => file number to search in */
    int futureArg         /* => for future use, always use 0 */
);
```

Description The mdlTag_getTagDef function retrieves the definition of a particular tag within a particular tag set.

tagDef specifies a pointer to a tag definition which points to the tag definition found. MicroStation allocates memory for *tagDef*, so the calling application should free this memory using mdlTag_freeTagDef when no longer needed.

setName specifies the name of the set for the tag desired.

tagName specifies the name of the tag whose definition should be retrieved.

fileNo specifies the file number in which to search for the tag.

Returns mdlTag_getSetNames returns SUCCESS if the specified tag definition was found. Otherwise, an error is returned.

See Also mdlTag_setTagDef, mdlTag_addTagDefToSet, mdlTag_freeTagDef.

mdlTag_setTagDef

```
#include <mstagdat.fdf>

int mdlTag_setTagDef
(
    TagDef *tagDef,          /* => new definition of tag */
    char *setName,           /* => set to set the tag in */
    char *tagName,           /* => name of the tag */
    int futureArg            /* => for future use, always use 0 */
);
```

Description The `mdlTag_setTagDef` function changes the definition of an existing tag in a given set. The tag set and the tag definition must already exist.

tagDef specifies a pointer to the new definition for the tag.

setName specifies the name of the tag set containing the tag.

tagName specifies the name of the tag to be set.

Returns `mdlTag_setTagDef` returns `SUCCESS` if the tag definition is found and successfully set. Otherwise, an error is returned.

See Also `mdlTag_getTagDef`, `mdlTag_addTagDefToSet`.

mdlTag_create

```
#include <mstagdat.fdf>

int mdlTag_create
(
    MSElement *pElmOut,      /* <= tag element created */
    MSElement *pElmIn,      /* => template element */
    TagSpec *tagSpec,         /* => tag definition */
    UShort *tagProps,         /* => properties of tag */
    TagValue *value,          /* => value for tag, or NULL */
    ULong *targetId,          /* => target element's tag, or NULL */
    DPoint3d *dPoint,         /* => origin, offset, or NULL */
    TextSizeParam *txSizeWd,  /* => size for text, or NULL */
    RotMatrix *rMatrix,        /* => rotation matrix, or NULL */
    TextParamWide *txtParams, /* => parameters, or NULL */
    char *styleName           /* => text style name, or NULL */
);
```

Description `mdlTag_create` creates a tag element. *pElmOut* specifies a pointer to an `MSElement` structure where the output element will be stored.

pElmIn specifies a pointer to an `MSElement` to be used as a template for the new tag. Pass `NULL` if no template element is being supplied.

tagSpec specifies a pointer to a specification for the definition of the tag.

tagProps specifies a mask containing information about the characteristics of the tag. See tagdata.h for possible values for this mask.

value specifies a pointer to a value for the tag. If this pointer is NULL, the default value for the tag is retrieved from the tags definition.

targetId specifies the id value for the element to which the tag will be attached, if it is attached.

dPoint specifies a pointer to a DPoint3d structure which contains the origin of the tag element if it is not being attached or the offset from the base element if it is being attached. NULL can be passed if there is to be no offset from the base element.

txSize specifies a pointer to a TextSizeParam structure containing the size for the text used to display the tags value. NULL can be passed to indicate that these values should come from the style name or the active settings.

rMatirx specifies a pointer to a rotation matrix which gives the rotation of the display text. NULL can be passed to indicate that these values should come from the style name or the active settings.

txtParams specifies a pointer to a structure containing additional parameters for the text display of the tag's value. NULL can be passed to indicate that these values should come from the style name or the active settings.

styleName specifies a pointer to a text style name. NULL can be passed if the text characteristics are to come from the *txSizeWd* and *txtParams* arguments or from the active settings.

Returns mdlTag_create returns SUCCESS if the tag was created, and an error otherwise.

See Also mdlTag_createSetDef, mdlTag_extract, mdlTag_deleteTagInstances.

mdlTag_extract

```
#include <mstagdat.fdf>

int mdlTag_extract
(
    DPoint3d      *origin,          /* <= origin of associated element */
    Dpoint3d      *userOrigin,      /* <= snap point */
    TagSpec       *tagSpec,         /* <= tag specification */
    boolean       *displayable,     /* <= TRUE for displayable */
    TagValue      *value,           /* <= value for tag */
    ULong         *targetId,        /* <= target element's id */
    DPoint3d      *offset,          /* <= offset */
    TextSizeParam *textSize,        /* <= size for text */
    RotMatrix     *rMatrix,         /* <= rotation matrix */
    TextParamWide *txtParams,       /* <= parameters */
    MSElement     *element,        /* => tag element */

```

```
int      fileNo      /* => file number for the tag */
);
```

Description `mdlTag_extract` extracts information about a tag element. Any of the output parameters can be `NULL`, indicating the caller does not care about that piece of information.

origin specifies the origin of the tag element if it is not associated (attached) to another element. If it is attached to another element, *origin* is the origin of that element.

userOrigin specifies the tag element's snap point.

tagSpec specifies a pointer to a specification for the definition of the tag.

displayable specifies whether or not the tag value gets displayed.

tagValue specifies the value for the tag.

targetId specifies the id value for the element to which the tag is attached, if it is attached.

offset specifies the offset of the tag element from its base element if the tag is associated to an element. In this case, *origin* + *offset* equals the actual position of the origin of the tag.

textSize specifies text size characteristics for the tag when it is displayed.

rMatirx specifies the rotation element for the tag when its value is displayed.

txtParams specifies text parameters for the value of the tag when it is displayed.

element is the input element from which the values should be extracted.

fileNo is the file number of the design file where the input element is located.

Returns `mdlTag_extract` returns `SUCCESS` if the values are successfully extracted. Otherwise, an error is returned.

See Also `mdlTag_create`.

mdlTag_deleteTagInstances

```
#include <mstagdat.fdf>

int mdlTag_deleteTagInstances
(
char    *setName,      /* => name of set */
char    *tagName,      /* => tag name */
int     futureArg      /* => for future use, always pass 0 */
);
```

Description mdlTag_deleteTagInstances deletes all tag instances, (type 37 elements), for a given tag definition from the master file. mdlTag_deleteTagInstances should always be called before a call to mdlTag_deleteTagDef since tags should be deleted when their definition is deleted.

setName specifies the name of the set for the tag.

tagName specifies the name of the tag.

Returns mdlTag_deleteTagInstances returns SUCCESS if the tag element(s) are found and successfully deleted. Otherwise, an error is returned.

See Also mdlTag_getTagDef, mdlTag_deleteTagDef, mdlTag_setTagDef, mdlTag_addTagDefToSet.

mdlTag_deleteTagDef

```
#include <mstagdat.fdf>

int mdlTag_deleteTagDef
(
    TagSpec      *tagSpec /* => tag to delete */
);
```

Description The mdlTag_deleteTagDef function deletes a tag definition from an existing tag set. mdlTag_deleteTagInstances should always be called before this function, since tags should be deleted when their definition is deleted.

tagSpec specifies a pointer to the specification for the tag.

Returns mdlTag_deleteTagDef returns SUCCESS if the tag definition was found and successfully deleted. Otherwise, an error is returned.

See Also mdlTag_getTagDef, mdlTag_setTagDef, mdlTag_addTagDefToSet, mdlTag_deleteTagInstances.

mdlTag_freeTagDef

```
#include <mstagdat.fdf>

void mdlTag_freeTagDef
(
    TagDef *tagDefP /* <=> tag definition to free */
);
```

Description mdlTag_freeTagDef frees a tag definition and any dynamically allocated contents.

tagDefP specifies the tag definition to free.

Returns mdlTag_freeTagDef is of type void. It returns no value.

See Also mdlTag_freeTagDefArray, mdlTag_getTagDef.

mdlTag_freeTagDefArray

```
#include <mstagdat.fdf>

void mdlTag_freeTagDefArray
(
  TagDef *tagArrayP,    /* <=> tag definition array */
  int     numtags       /* => number of tags in array */
);
```

Description mdlTag_freeTagDefArray frees an array of tag definitions and any dynamically allocated contents.

tagArrayP specifies the array of tag definitions to free.

numtags specifies the number of tags in the array.

Returns mdlTag_freeTagDefArray is of type void. It returns no value.

See Also mdlTag_freeTagDef, mdlTag_getSetDef.

mdlTag_getElementTags

```
#include <mstagdat.fdf>

int mdlTag_getElementTags
(
  MSElementDescr **elemDescrPP, /* <= linked list of tags */
  int               *numElementDescrs, /* <= number of tags */
  MSElement        *element,          /* => element to get tags for */
  int               fileNo,            /* => file number for element */
  int               futureArg          /* => future arg, always pass 0 */
);
```

Description The mdlTag_getElementTags function gets an element descriptor containing all the tags attached to a particular element.

elemDescrPP specifies an element descriptor containing all the tag elements attached to a base element. The *userData1* member of the element descriptor contains the file position for the elements.

numElementDescrs specifies the number of tags retrieved.

element specifies a pointer to the element for which to get tags.

fileNo is the file number of the input element for which to get tags.

Returns mdlTag_getElementTags returns SUCCESS if the tags are successfully collected or no tags existed. Otherwise, an error is returned.

See Also mdlTag_create.

mdlTag_generateReport

```
#include <mstagdat.fdf>

int mdlTag_generateReport
(
    char          *fileName,      /* => fully qualified report file name */
    char          *setName,       /* => name of the tag set to report on */
    StringList    *tagList,       /* => stringlist with tags to report */
    ULong         exportType,     /* => export type for report */
    int           futureUse       /* for future use, always pass 0 */
);
```

Description mdlTag_generateReport generates an ASCII report file for the indicated tag set.

fileName the file name with path where the report will be generated. Even though the filename is specified in the tag set, *fileName* is the fully-qualified filename, specifying more information than the tag set contains. If the name specified in *fileName* disagrees with the name in the tag set, *fileName* takes priority.

setName specifies the name of the tag set which should be reported on.

tagList specifies a string list containing the names of the columns for the report. The columns of the report will appear in the same order as the strings in the string list.

exportType is either TAG_EXPORT_TAGGED or TAG_EXPORT_ALL, indicating whether only tagged elements are reported on or all elements are reported on, respectively.

Returns mdlTag_generateReport returns SUCCESS if the report is successfully generated. Otherwise, an error is returned.

See Also mdlTag_setSetDef.

mdlTag_getAssocElement

```
#include <mstagdat.fdf>

int mdlTag_getAssocElement
(
    MSElement    *assocElmP,     /* <= associated element */
    int           *fileNum,       /* <= file number for element */
    ULong         *filePos,       /* <= file position for element */
    MSElement    *tagElm        /* => base tag element */
);
```

Description mdlTag_getAssocElement gets the base element to which a tag is attached.

assocElmP specifies the base element for the tag.

fileNum specifies the file number of the tag element whose base element should be obtained. File number of the base element is returned back if it is different from the tag element.

filePos specifies the file position for the base element. `NULL` can be passed if the file position is not needed.

tagElm specifies the tag element whose base element should be obtained.



This function was implemented in MicroStation 95.

Returns `mdlTag_getAssocElement` returns `SUCCESS` if the base element is successfully retrieved. Otherwise, an error is returned.

See Also `mdlTag_create`.

22

External Program Communication Functions

External program communication functions enable MDL programs to start external programs and communicate with the programs through message queues and shared memory. External programs are completely separate from MicroStation. They are not MDL programs. An external program can use MicroCSL, but MicroCSL is not required.

External Program Communication Functions



Please read devtools.txt, installed in the MDL directory, for the latest version information on tools specific to your development platform.

The following table lists external program communication functions used by MDL programs:

Function	Used to
mdlExternal_terminateProgram	terminate an external program that the MDL application started.
mdlExternal_startProgram	start an external program.
mdlExternal_startRealModeProgram (DOS only)	start a real mode program.
mdlExternal_startMCSLProgram	start an external program and create a MicroCSL session to be used by the external program.
mdlExternal_queueGet	get a message queue for communicating with an external program.
mdlExternal_messageSend	send a message from an MDL application to an external program.
mdlExternal_messageReceive	receive a message from an external program.
mdlExternal_queueRemove	remove a message queue, freeing all of its associated memory.
mdlExternal_shmemGet	get a shared memory region that lets an MDL application and external program share data.

Function	Used to
mdlExternal_shmemRemove	remove a shared memory region.
mdlExternal_wait	wait for an external program to terminate.
mdlExternal_sendSignal (UNIX platforms only)	send a signal to an external program.
mdlExternal_setFunction	designate user functions for interacting with external programs.

The following table lists external program communication functions used by external programs to communicate with MDL programs:

Function	Used to
extprg_queueAttach	attach program to a message queue that was created by an MDL application.
extprg_queueDetach	detach program from the message queue.
extprg_messageSend	send messages to an MDL application.
extprg_messageReceive	receive messages from an MDL application.
extprg_shmemAttach	provide access to a shared memory region that was created by an MDL application
extprg_shmemDetach	tell MicroStation that the external program is no longer using the shared memory region.
extprg_signalUstn (UNIX platforms only)	perform platform-specific initialization for an external program.
extprg_initializePlatform	remove a message queue, freeing all of its associated memory.
extprg_getCommandLineArguments	get the arguments that were passed to the external program by mdlExternal_startProgram.
extprg_shutdownPlatform	perform platform specific shutdown for an external program.

The functions listed above are located in a library in the .../mdl/library directory. The filename of the library is platform-specific as follows:

Platform	Library name
UNIX (all)	libexpt.a
DOS	libexpt.lib
Mac	libexpt.o
Windows NT	libexpt.lib

The following table lists external program communication user functions. MDL calls these user-supplied functions when certain events occur within MicroStation. The programmer determines the user function name. (The names below are for illustration only). These functions are designated to MDL through function pointer arguments to MDL routines.

Function	MicroStation calls when
userExternal_messageReceived	a message is received from an external program.
userExternal_programTerminated	an external program terminates.

Windows NT-specific notes

Prior to Windows NT, MicroStation would terminate an MDL application's external programs when MicroStation unloaded the MDL application. Under Windows NT, MicroStation cannot terminate the external program. Therefore, the MDL program must have a mechanism to allow it to tell the external program that the external program should terminate. Typically, this is handled as follows:

1. The MDL program establishes an unload hook.
2. MicroStation then calls the MDL program's unload hook, which sends a message to the external program telling it that the MDL application is being unloaded.
3. The MDL application calls mdlExternal_messageReceive to wait for the external program to acknowledge the message.
4. The MDL application calls mdlExternal_wait to wait for the external process to terminate.

If the external program is a MicroCSL program, then the MDL program should also have a userSystem_exitDesignFileState. It should terminate the external program when the userSystem_exitDesignFileState is called.

Debugging external programs under Windows NT

External programs can be debugged using the Visual C++ tools.

Add a `DebugBreak` call to the `main` of the external program and recompile and relink the external program. Make sure that `DebugBreak` is prototyped correctly. (See the header files delivered with Visual C++).

Then, when the `DebugBreak` call is hit in the external program, an error is generated and the Visual C++ debugger will be started. Step past the `int 3` instruction and debug as if you had started the program normally.

DOS-specific notes

When an MDL application starts a program to run in protected mode, the external program can return to MicroStation while still remaining loaded. In this sense, it simulates multi-tasking. The external program runs in its own address space under MicroStation control.

Simulated message queues and shared memory enable MDL applications to communicate with their protected mode external programs. Control is exchanged between MicroStation and the external programs during message queue operations. The following rules govern this exchange:

- As soon as the external program is started (before returning from `mdlExternal_startProgram` or `mdlExternal_startMCSLProgram`) control is transferred.
- During a send operation (`mdlExternal_messageSend`, `extprg_messageSend`), control is transferred. For example, if the external program calls `extprg_messageSend` to send a message, `extprg_messageSend` transfers control to MicroStation, which resumes the MDL application.
- During a receive operation `extprg_messageReceive` transfers control if no message is outstanding. If a message is outstanding, the receive operation returns immediately.

The external program remains loaded after the MDL application returns to MicroStation. However, it is not dispatched again until the MDL program calls `mdlExternal_messageSend`.



For specific information about OS requirements and compiler versions for your particular platform, you must refer to the `platform.txt` file.

Protected mode external programs run under MicroStation control. They use MicroStation's copy of the DOS extender, so they do not need their own copy.

As protected mode programs run under MicroStation control and share MicroStation's DOS PSP the initialization module provided with the High C compiler module needs to be replaced with one provided by Bentley Systems Incorporated (BSI). This module, contains initialization logic that runs before the program's `main` is called. The initialization file to use depends on the version of the compiler used to compile the external.

Since protected mode external programs share MicroStation's copy of the DOS extender, they also share the virtual memory manager. As the protected mode external program's need for memory exceeds real memory capacity, a portion of MicroStation or the external program is swapped to satisfy the requirements.

The external program and MDL program have different address spaces. Therefore, they cannot share pointers.

An example of an external program is provided. The example consists of the following files:

Example	Required files
externpg.mc	the MDL source file for the MDL application that starts the external program and communicates with it.
externpg.c	the C source file for the external program. This is compiled with the MetaWare High C compiler.
externpg.h	a file containing definitions shared by the MDL and C sources.
externpg.mke	a makefile used with BSI's make utility. This makefile creates the MDL application and the protected mode program that the application uses.

The link step that appears in the makefile is as follows:

```
$(mdlapps)externpg.exp : $(externpgObjs) $(mdlLibs)libextp.lib
$(msg)
> $(objectDir)link.cmd
$(MWlopt)
$(NOSTUB)
$(externpgObjs)
-lib $(mdlLibs)libextp.lib
$(pmLibOpts)
$(naLibOpts)
-publist byvalue
-MINDATA 0
-MAXDATA 0
-80386
-nostub
```

```
-exe $@
<
$(pLinkCmd) @$(objectDir)temp.cmd
```

where:

```
externpg0bjs = $(initexp0bjs) \
               $(objectDir)externpg$(oext)
```

`$(MWlopt)`, `$(pmLib0pts)`, `$(naLib0pts)` and `$(initexp0bjs)` are all defined in the file `mdl.mki`.

`MWlopt` normally defines a linker parameter specifying what kind of symbols are to be included with the external program. The value for this flag depends on the type of debugger being used. Specify `-cvsymbols` for the MetaWare debugger, `-fullsym` for the Phar Lap 386|SRCBug debugger, or `symbols` for the Phar Lap 386DEBUG debugger. The include file `mdl.mki` sets `MWlopt` appropriate depending on what makefile macros are defined. For example, if `DEBUG` and `SB386_DEBUG` are defined, it defines `MWlopt` as `-fullsym`. Check `mdl.mki` for more information on this.

`initexp0bjs` specifies an initialization object file. `$(initexp0bjs)` must appear before the compiler's libraries to avoid using an initialization module from one of the compiler's libraries. The definition of `initexp0bjs` varies depending on the compiler being used. Some of the possibilities are `initcexp.obj` for release 1.62 of HighC, and `init320g.obj` for release 3.20 of the High C compiler. For the Lahey Fortran compiler, use both `hc310.obj` and `lfininit.obj`. Check `mdl.mki` for the standard definitions.

`MINDATA` and `MAXDATA` are zero so that no extra memory is allocated for the program when the program is loaded. All memory required by the code, data and stack is allocated when the program is loaded, but no memory is allocated for the heap. As the program requests more memory through `malloc` calls, memory is allocated for the heap.

Any real mode program can run if memory is sufficient. Real mode programs cannot use `mdlExternal_...` functions. They cannot communicate with MDL programs.



Beginning with Phar Lap version 5.0, the `-nostub` switch is required so that a `.exp` file is created rather than a `.exe` file.

Debugging protected mode external programs

Protected mode external programs can be debugged using Phar Lap's 386DEBUG, Phar Lap's 386|SRCBug, or MetaWare's `mdb`. Both 386|SRCBug and `mdb` are source level debuggers; 386DEBUG is not. The following comments apply regardless of the debugger used.

Set `MS_TRAP=NONE`. Without this, MicroStation tries to catch all faults and recover or at least report the source. If your program causes a fault, MicroStation intercepts it and reports what line in the MDL program caused the external program to be dispatched. If `MS_TRAP` is set to `NONE`, the fault is passed to the debugger.

Using mdb

Source files should be compiled with the `-g` option. The program to be debugged should be linked with the `-cvsymbols` option.

You may need to use the virtual memory driver `vmmdrv.exp`. When MicroStation is started by one of the debuggers, MicroStation does not have access to virtual memory. To provide access to virtual memory specify the virtual memory driver when starting the debugger. The virtual memory driver `vmmdrv.exp` is available as part of the Phar Lap developer's tool kit.

To debug, start MicroStation with the debugger. To do this, enter the following commands at the DOS prompt:

```
SET R386=-VM \PATH\VMMDRV.EXP -CALLBUFS 8
\PATH\MDB MGDS
```

Omit `-VM \PATH\VMMDRV.EXP` if you are not using virtual memory.

Use `load <application> nocode` to load the symbol table. This tells the debugger to use the symbol table from the external program.

After your program is loaded, the debugger must be invoked as a result of an `INT 3` in your program. `INT 3` is a debug interrupt. When it occurs, the debugger starts debugging the external program. To cause the `INT 3` to occur in your program, you can explicitly code it in your program or you can direct MicroStation to make it occur. To tell MicroStation to make it occur, you can set `debugLevel` to 5, or use the MicroStation command `set debug 5`. To code it in your program, include the statement `_inline (0xCC);` in your C code.

When the `INT 3` is executed, the debugger is invoked and your program is stopped at the `INT 3` instruction. Enter the command `EIP++` in the debugger. This causes the PC's instruction pointer to be advanced beyond the `INT 3` instruction. If you do not do this, the `INT 3` is executed again.

Debug as if you had started the program normally.

Using 386DEBUG or 386|SRCBug

This section contains a few suggestions for using Phar Lap's debuggers to debug protected mode external programs.

To be able to use either of these debuggers for the external program, MicroStation must be loaded by the debugger, but the debugger must be directed to load the symbol file from the external program. Use the `SYMFILE` option to direct the debugger to load the symbol table from the external program. For example, to load the symbol table from `externpg.exp`, start MicroStation with the following DOS command:

```
386DEBUG -SYMFILE EXTERNPG.EXP MGDSPM DGNFILE
```

Even though the symbol table is loaded at initialization, it is not possible to enter breakpoints or display data until the external program is loaded. Just enter `GO` in response to the initial prompt from the debugger.

Once the external program is installed, the debugger should be invoked via an `INT 3` instruction contained in the external program.

If MicroStation was started by one of these debuggers, then an `INT 3` instruction causes the debugger to be invoked. If MicroStation was not started by one of these debuggers, then the `INT 3` instruction just returns to the external program and the `INT 3` has no effect.

Once an `INT 3` instruction in the external program has invoked one of these debuggers, the following steps must be taken:

- Enter `R EIP EIP+1` to increment the instruction pointer. If this is not done, the `INT 3` instruction is executed again.
- Enter `XR C 4C`. This command tells 386DEBUG to use the value of `4C` as the `CS` register for determining which symbol is associated with a given address. Typically, the first external program is loaded with `4C` in the `CS` value. If your program is loaded with another value for the `CS` register, specify that value in the `XR` command. If this `XR` command is not executed, the debugger does not display the name of the functions being referenced.

After taking these steps, it is possible to use the debugger normally. That is, it is possible to use the debugger exactly as if it had started the external program.

Using External Programs Under UNIX

Portability Issues

Some aspects of the `mdlExternal_...` functions are unique to the specific platform. This section discusses some of the aspects that are unique to UNIX implementation.

mdlExternal_terminateProgram

On Unix workstations, `mdlExternal_terminateProgram` tries to terminate the external program by sending a `SIGQUIT` signal. If the external program does not have a `SIGQUIT` signal handler and a `SIGQUIT` signal is sent to it, a fatal error message displays and a core file is created. To prevent this, the external program should define a signal handler using `sigset`. The signal handler should perform any cleanup required by the external program. Then it should call `exit`.

If the external program is using MicroCSL and has called `indfpi` or `ms_app_login`, then MicroCSL has already defined a signal handler. An external program that uses MicroCSL does not need to define a signal handler unless it needs to perform some cleanup on its own. If an external program defines its own signal handler, it should not call `indfpi` or `ms_app_login` after setting up the signal handler. Code similar to what follows should be used to set up the signal handler for `SIGQUIT`:

```
void (*sigset())();  
microCSLHandlerP = sigset(SIGQUIT, appFunction);
```

When the signal occurs, the application signal handler should perform its own processing, and then call the MicroCSL signal handler, passing in the signal number as the only parameter:

```
(*microCSLHandlerP)(SIGQUIT);
```



Due to operating system restrictions, `mdlExternal_terminateProgram` only cleans up MicroStation data structures under Windows NT; it does not terminate the external program. Consequently, MDL programs that run under Windows NT should use another mechanism for synchronizing with the external program. The MDL program should use `mdlExternal_wait` to wait until the external program terminates.

mdlExternal_messageReceive

If `mdlExternal_messageReceive` is called with a time-out value, the time-out interval is restarted after every MicroCSL request. Therefore, when deciding what time-out interval to use, you do not need to consider how long the MicroCSL tasks will take.

Debugging external programs under UNIX

Generally, you can debug as you're accustomed. Use `mdlExternal_startMCSLProgram` or `mdlExternal_startProgram` to start the debugger specifying your program as the first parameter. Start your program with a command similar to:

```
mdlExternal_startMCSLProgram("/usr/bin/dbg", "yourapp",
                             arguments to your application);
```

Some debuggers do not pass through command line arguments. If you use a debugger that does not pass through the command line arguments, you may want to make the information available to the external program by an environment variable or by writing them to a file for your application to read.

When the debugger starts, tell it to ignore the `SIGUSR1` and `SIGALRM` signals. These may be used in communicating between MicroStation and the external program. The debugger cannot interfere with these.

The focus will not automatically switch between MicroStation and the debugger. You must always click in the window that is to receive the input.

Function definitions

The remainder of this section describes function calls. Those beginning with `mdlExternal_...` are MDL built-in functions and are available to MDL applications. Those beginning with `extprg_...` are available to external programs to use in communicating with MDL applications. Those beginning with `userExternal_...` are user functions that MDL provides to MicroStation through calls to `mdlExternal_setFunction`.

mdlExternal_terminateProgram

```
#include <msextern.h>

void mdlExternal_terminateProgram
(
    MSExternalProgramDescr *progDescrP    /* => program descr */
);
```

Description The `mdlExternal_terminateProgram` function terminates an external program that an MDL application started. The MDL application specifies the program to terminate by passing in *progDescrP*. The value of *progDescrP* would have been returned to the MDL application when it started the external program with `mdlExternal_startProgram`.

`mdlExternal_terminateProgram` merely terminates the external program and frees memory used by the external program. On the PC running the Protected Mode MicroStation, the external program's files are not automatically flushed or closed. `mdlExternal_terminateProgram` should not be used until after the external program exits. Once the external program exits, `mdlExternal_terminateProgram` should be called to clean up data structures local to MicroStation.

mdlExternal_terminateProgram frees the memory that *progDescrP* points to. Therefore, the MDL application must not use the pointer again.

A userExternal_programTerminated user function should not call mdlExternal_terminateProgram.

If an external program is terminated with mdlExternal_terminateProgram, MicroStation does not call the userExternal_programTerminated function.

Returns The mdlExternal_terminateProgram function is of type void. It returns no value.

See Also mdlExternal_startProgram, mdlExternal_wait, userExternal_programTerminated.

mdlExternal_startProgram

```
#include <msextern.h>

MSExternalProgramDescr *mdlExternal_startProgram
(
    char    *pName,          /* => name of the program */
    char    *pargument,      /* => argument list */
    ...
);
```

Description The mdlExternal_startProgram function starts external programs. *pName* is the only required argument. It points to the name of the program to run. If no directory path is specified with the name, the function tries to find the file in the current directory and the directories specified with the MS_MDL environment variable.

The arguments to be passed to the external program can be specified in one string, separated by blanks, or specified separately with an additional *pargument* parameter provided for each argument. Once the external program terminates, MicroStation retains the MSExternalProgramDescr structure until MicroStation knows the external program has terminated. That is, MicroStation removes the MSExternalProgramDescr structure in any of the following cases:

- The MDL application's program terminated hook has just returned to MicroStation.
- The built-in function mdlExternal_terminateProgram is about to return to the MDL program.
- The built-in function mdlExternal_wait is about to return to the MDL program.
- MicroStation terminates the MDL program that started the external program.

When MDL terminates an MDL application, it also terminates any active external programs.

Returns The `mdlExternal_startProgram` function returns a pointer to an `MSEExternalProgramDescr` structure if the function successfully started the external program. Otherwise, it returns `NULL`. If `mdlExternal_startProgram` returns `NULL`, more detailed information is provided in `mdlErrno`. The error codes are described in `mdlerrs.h`.

See Also `mdlExternal_terminateProgram`, `mdlExternal_startRealModeProgram` (DOS only), `mdlExternal_startMCSLProgram`, `mdlExternal_wait`, `userExternal_programTerminated`.

mdlExternal_startRealModeProgram (DOS only)

```
int mdlExternal_startRealModeProgram
(
    char    *pName,          /* => program name */
    char    *pArgs,          /* => program argument list */
    int     *pExitStatus,    /* <= receives exit status */
    int     enterTextMode,   /* => TRUE puts screen in text mode */
    int     pauseAfter       /* => TRUE means wait for user input */
);
```

Description The `mdlExternal_startRealModeProgram` function starts a real mode program and waits for it to complete.

pName points to the name of the program to run.

pArgs points to the arguments to be passed into the program.

pExitStatus points to an integer to receive the program's exit status.

A non-zero value for *enterTextMode* puts the PC monitor in text mode after MicroStation clears graphics.

A non-zero value for *pauseAfter* displays a prompt and waits for the operator to enter a keystroke before redisplaying the graphics and returning to MicroStation.

Depending on the real mode program size, MicroStation may need to temporarily free memory it is using in the real mode address space. MicroStation uses the task size setting in *tcb->task_size* when deciding whether to free memory. The task size is set with the SET TASKSIZE key-in, the "External Progs. (Conv)" field of Memory Usage in User Preferences, or by an MDL program explicitly setting *tcb->task_size*. To tell MicroStation to free as much memory as possible, set *tcb->task_size* to -1. If your MDL program explicitly sets *tcb->task_size*, then it should restore the value when `mdlExternal_startRealModeProgram` returns.

Returns The `mdlExternal_startRealModeProgram` function returns `SUCCESS` if it could run the program. Otherwise, it returns a non-zero value. If the program runs successfully, the program's exit status is stored in the area pointed to by *pExitStatus*.

See Also `mdlExternal_startProgram`.

mdlExternal_startMCSLProgram

```
MSExternalProgramDescr *mdlExternal_startMCSLProgram
(
    char    *nameP,          /* => name of the program to start */
    char    *argumentP,     /* => argument list */
    ...
);
```

Description The `mdlExternal_startMCSLProgram` function starts external programs and creates a MicroCSL session to be used by the external program. The `mdlExternal_startMCSLProgram` is very similar to `mdlExternal_startProgram`, and all of the documentation for `mdlExternal_startProgram` also applies to `mdlExternal_startMCSLProgram`.

With the DOS version of MicroStation, there is no difference between `mdlExternal_startMCSLProgram` and `mdlExternal_startProgram`.

With non-DOS versions of MicroStation, external programs that do not use MicroCSL should always be started with `mdlExternal_startProgram`. External programs that do use MicroCSL must be started with `mdlExternal_startMCSLProgram`.

Returns The `mdlExternal_startMCSLProgram` function returns a pointer to an `MSExternalProgramDescr` structure if the function successfully started the external program. Otherwise, it returns `NULL`. If `mdlExternal_startMCSLProgram` returns `NULL`, more detailed information is provided in `mdlErrno`. The error codes are described in `mdlerrs.h`.

If the file cannot be found, `mdlExternal_startMCSLProgram` returns with `mdlErrno` set to `MDLERR_CANNOTOPENFILE`. If the file can be found, but cannot be executed `mdlExternal_startMCSLProgram` returns with `mdlErrno` set to `MDLERR_BADFILETYPE`.

If MicroStation cannot run the MCSL program because the maximum number of MicroCSL tasks are active, `mdlExternal_startMCSLProgram` returns `NULL` with `mdlErrno` set to `MDLERR_MAXMCSLTASKS`.

If the program is already active as an MCSL program, the start fails with `mdlErrno` set to `MDLERR_ALREADYLOADED`.

For other errors, `mdlErrno` is set to `MDLERR_SYSTEMERROR` and more information is in `errno`. The value in `errno` is not provided by MDL, but is provided by the underlying software. For example, on the Intergraph workstation, it is provided by UNIX. The values for `errno` are defined in a header file provided with the system. Typically, this is `errno.h`.

See Also `mdlExternal_startProgram`.

mdlExternal_queueGet

```
#include <msextern.h>

MSMsgqDescr *mdlExternal_queueGet
(
    key_t    key, /* => EXTP_KEY_GENERATE, or actual key */
    int      op  /* => EXTP_QUEUE_ATTACH or EXTP_QUEUE_CREATE */
);
```

Description The `mdlExternal_queueGet` function gets a message queue for communicating with an external program.

key can be a value that the programmer assigns to the key, or it can be `EXTP_KEY_GENERATE`. Using `EXTP_KEY_GENERATE` is recommended.

op can be `EXTP_QUEUE_ATTACH` or `EXTP_QUEUE_CREATE`. Using `EXTP_QUEUE_CREATE` is recommended.

A given message queue should be used for an MDL program to communicate with an external program.

Returns The `mdlExternal_queueGet` function returns a pointer to an `MSMsgqDescr` structure if the function successfully satisfied the request. Otherwise, it returns `NULL` and more detailed information is provided in `mdlErrno`.

See Also `mdlExternal_messageSend`, `mdlExternal_messageReceive`, `userExternal_messageReceived`, `extprg_queueAttach`.

mdlExternal_messageSend

```
#include <msextern.h>

int mdlExternal_send
(
    ExternalMessage *pMessageBuffer, /* => msg buffer */
    MSMsgqDescr     *pDescr,         /* => queue to use */
    MSExternalProgramDescr *progDescrP /* => program descr */
);
```

Description The `mdlExternal_messageSend` function sends a message from an MDL application to an external program.

pMessageBuffer points to the message to be sent. MDL fills in *pMessageBuffer->mtype* and *pMessageBuffer->sendpid*. The MDL application must fill in *pMessageBuffer->msglength*, *pMessageBuffer->reqtype* and *pMessageBuffer->mtext*.

pMessageBuffer->mtext gets the application's portion of the message. *pMessageBuffer->msglength* is set to the message size in *pMessageBuffer->mtext*.

The programmer determines how *pMessageBuffer->reqtype* is used.

progDescrP points to the program descriptor of the external program to which the message is sent. If the MDL application uses only one external program, it can pass `NULL` for this parameter.

Under DOS, the external program typically is stopped at a call to `extprg_messageReceive`. When `mdlExternal_messageSend` is called, MDL gives control to the external program.

Under DOS, `mdlExternal_messageSend` and `mdlExternal_messageReceive` are not really using message queues. They use one buffer. If the external program or the MDL application uses consecutive sends and the message recipient also tries to send, buffer contents will be overwritten.

Returns The `mdlExternal_messageSend` function returns `SUCCESS` if the message is sent successfully. Otherwise, it returns a non-zero value and more detailed information is provided in `mdlErrno`.

See Also `extprg_messageReceive`, `mdlExternal_queueGet`.

mdlExternal_messageReceive

```
#include <msextern.h>

int mdlExternal_messageReceive
(
    ExternalMessage *pmessageBuffer,    /* <=> message buffer */
    MSMsgqDescr    *pMsgq,             /* => message queue */
    int             mtextSize,          /* => max text size */
    int             timeout,            /* => max time to wait */
);
```

Description The `mdlExternal_messageReceive` function receives messages from external programs. `mdlExternal_messageReceive` blocks MicroStation until a message is received or the timeout expires. Messages can also be received asynchronously with the `userExternal_messageReceived` function.

The message is received in the buffer that *pmessageBuffer* points to. *mtextSize* gives the maximum size of a message to be received in *pmessageBuffer->mtext*. The actual size of the received message is stored in *pmessageBuffer->msglength*.

pmessageBuffer->sendpid identifies the external program that sent the message. This program identification should match the `pid` field in an `MSExternalProgramDescr` structure for an external program started by the MDL application. *pmessageBuffer->mtype* contains `EXTP_FOR_MDL`.

timeout is ignored under DOS. On other platforms, it is used to specify how long `mdlExternal_messageReceive` should wait for a message before returning to the calling application. It is specified in terms of seconds. On UNIX platforms, the timeout counter is reset every time MicroStation services a MicroCSL call for the external program.

Returns mdlExternal_messageReceive returns SUCCESS if it successfully received a message. Otherwise, it returns a non-zero value. Under DOS, it returns 2 if the queue is empty.

See Also userExternal_messageReceived, extprg_messageSend, mdlExternal_queueGet.

mdlExternal_queueRemove

```
#include <msextern.h>

void mdlExternal_queueRemove
(
    MSMsgqDescr *pDescr    /* => message queue descriptor */
);
```

Description The mdlExternal_queueRemove function removes a message queue, freeing all of the queue's associated memory.

pDescr points to the MSMsgqDescr for the queue to be removed. The value for *pDescr* must be a value returned by mdlExternal_queueGet.

Once mdlExternal_queueRemove returns, all data structures associated with the message queue are freed. No structure should be accessed after mdlExternal_queueRemove returns.

Returns The mdlExternal_queueRemove function is of type void. It returns no value.

See Also mdlExternal_queueGet.

mdlExternal_shmemGet

```
#include <msextern.h>

MSShmemDescr *mdlExternal_shmemGet
(
    key_t    key,          /* => EXPT_GENERATE_KEY or a key value */
    int      size          /* => maximum size for data area */
);
```

Description The mdlExternal_shmemGet function gets a shared memory region that can be used to share data between an MDL application and an external program.

key specifies an identifier to be associated with the shared memory region. EXPT_GENERATE_KEY will generally be used.

size specifies the size of the shared memory region. The shared memory region's size is not restricted.

Returns The mdlExternal_shmemGet function returns a pointer to an MSShmemDescr structure. The *psharedMem* field of this structure points to the shared memory. The *externalName* field of the structure contains the name that external programs use to attach the shared memory region.

See Also mdlExternal_shmemRemove, extprg_shmemAttach, extprg_shmemDetach.

mdlExternal_shmemRemove

```
#include <msextern.h>

void mdlExternal_shmemRemove
(
    MShmemDescr *pDescr    /* => pointer to shared memory descr */
);
```

Description The `mdlExternal_shmemRemove` function removes a shared memory region specified by *pDescr*. Neither the MDL application nor the external program can access the shared memory or the descriptor after this function is called.

Returns The `mdlExternal_shmemRemove` function is of type `void`. It returns no value.

See Also `mdlExternal_shmemGet`.

mdlExternal_wait

```
int mdlExternal_wait
(
    int                *statusP,    /* <= receives pgm exit sts */
    MSExternalProgramDescr *programP /* => program to wait for */
);
```

Description The `mdlExternal_wait` function is used to wait for an external program to terminate. When the program terminates, `mdlExternal_wait` stores the exit status in the `int` pointed to by *statusP*. Then it removes all data structures used to keep track of the program.

While MicroStation is in `mdlExternal_wait`, most processing in MicroStation is blocked. MicroStation can do little more than process MicroCSL requests. While an MDL application is in a call to `mdlExternal_wait`, the external program should not use MicroCSL calls that use the MicroStation input queue. Generally, calls described in any section of the MicroCSL manual that pertains to a specific platform should not be used. Calls for retrieving user input and simulating user input will not work. `prdfpi` is the one exception to this; calls to `prdfpi` will work.

Returns Under DOS, `mdlExternal_wait` returns `SUCCESS` if the program has already terminated. It returns `ERROR` otherwise.

On other platforms, `mdlExternal_wait` always returns `SUCCESS`.

See Also `mdlExternal_terminateProgram`, `userExternal_programTerminated`.

mdlExternal_sendSignal (UNIX platforms only)

```
mdlExternal_sendSignal
(
    int    pid,    /* external program */
    ...
);
```

```
int      signalNumber    /* signal number to send */
);
```

Description The mdlExternal_sendSignal function sends a signal to an external program. *pid* identifies the external program. It does not have to identify one started by MicroStation, nor does MicroStation need to have any knowledge of the external program. If the external program was started by MicroStation, then the *pid* field in the corresponding MSExternalProgramDescr can be supplied as the *pid* parameter. *signalNumber* provides the number of the signal to send.

Applications that rely on mdlExternal_sendSignal are not portable. The function is not available in DOS and may not be available on other platforms.

Returns mdlExternal_sendSignal returns SUCCESS if it successfully sends the signal. Otherwise, it returns -1 and an appropriate error code is stored in errno. On the workstation, the error codes are defined in errno.h.

mdlExternal_setFunction

```
MdlFunctionP mdlExternal_setFunction
(
int          type,        /* => EXTERNAL_MESSAGE_RECEIVED */
MdlFunctionP function    /* => MDL function address */
);
```

Description mdlExternal_setFunction designates user functions for interacting with external programs.

type can be EXTERNAL_MESSAGE_RECEIVED or EXTERNAL_PROGRAM_TERMINATED.

pFunction must be NULL or a pointer to a valid MDL function.

Returns mdlExternal_setFunction returns a pointer to the user function (of the same type) that was previously set using mdlExternal_setFunction. If *type* is invalid, mdlExternal_setFunction returns -1.

See Also userExternal_messageReceived, userExternal_programTerminated.

extprg_queueAttach

```
#include <msextern.h>

MSMsgqDescr *extprg_queueAttach
(
char      *pExternalName /* => name of the queue */
);
```

Description The `extprg_queueAttach` function attaches the program to a message queue that was created by an MDL application.

pExternalName points to the name that `mdlExternal_queueGet` placed in the *externalName* field of the `MSMsgqDescr` that `mdlExternal_queueGet` returned.

Returns The `extprg_queueAttach` function returns a pointer to an `MSMsgqDescr` structure if no errors occur. This pointer can be used as an argument in calls to `extprg_messageSend`, `extprg_messageReceive`, and `extprg_queueDetach`. If an error occurs, `extprg_queueAttach` returns `NULL`.

See Also `mdlExternal_queueGet`, `extprg_messageSend`, `extprg_messageReceive`, `extprg_queueDetach`.

extprg_queueDetach

```
#include <msextern.h>

int extprg_queueDetach
(
    MSMsgqDescr *pMsgqDescr    /* => pointer to queue descriptor */
);
```

Description The `extprg_queueDetach` function detaches the program from the message queue described by *pMsgqDescr*.

Returns The `extprg_queueDetach` function always returns `SUCCESS`.

See Also `extprg_queueAttach`.

extprg_messageSend

```
#include <msextern.h>

int extprg_messageSend
(
    ExternalMessage *pmessage,    /* => message to send */
    MSMsgqDescr *pMsgqDescr      /* => ptr to queue descriptor */
);
```

Description The `extprg_messageSend` function is used by an external program to send messages back to an MDL application.

pMsgqDescr points to a message queue descriptor. This argument should be a value returned by `extprg_queueAttach`.

pmessage points to a message to be sent. The `ExternalMessage` structure describes the message format.

`extprg_messageSend` fills in *pmessage->sendpid* and *pmessage->mtype*.

The data portion of the message should be in *pmessage->mtext*. *pmessage->msglength* must be set equal to the size of the data in *pmessage->mtext*.

The user determines how *pmessage->reqtype* is used.

`ExternalMessage` is defined in `msextern.h` with *mtext* defined as a one-byte array. Programs should have their own definition of `ExternalMessage` with *mtext* defined with the required size.

Returns The `extprg_messageSend` function returns a non-zero value if an error occurs.

See Also `mdlExternal_messageReceive`, `extprg_queueAttach`.

extprg_messageReceive

```
#include <msextern.h>

int extprg_messageReceive
(
    ExternalMessage *pmessage,      /* => message receiving buffer */
    MSMsgqDescr    *pMsgqDescr,    /* => message queue descriptor */
    int             maxSize,        /* => maximum message size */
    int             flag            /* => message type */
);
```

Description The `extprg_messageReceive` function receives messages from an MDL application.

pMsgqDescr designates the message queue that receives the message.

pMsgqDescr should have a value returned by `extprg_queueAttach`.

pmessage is the buffer that receives the message. The data portion of the message is stored in *pmessage->mtext*. The value of *pmessage->msglength* gives the size of data in *pmessage->mtext*. *pmessage->reqtype* is filled in with the *reqtype* from the message sent.

maxSize gives the size of *pmessage->mtext*.

flag is meaningless under DOS.

Under DOS, if a message is already in the message queue, `extprg_messageReceive` returns immediately. Otherwise, it returns control to MicroStation and the MDL application resumes executing. On other platforms, *flag* can be `IPC_NOWAIT` or 0. In UNIX, `IPC_NOWAIT` is defined in the header file `ipc.h`.

Returns The `extprg_messageReceive` function returns `SUCCESS` when a message is received. Otherwise, it returns a non-zero value.

See Also `mdlExternal_messageSend`, `extprg_queueAttach`.

extprg_shmemAttach

```
#include <msextern.h>

SHMEM_SC char *extprg_shmemAttach
(
    char    *pExternalName /* => name of shared memory segment */
);
```

Description The `extprg_shmemAttach` function provides access to a shared memory region that an MDL application created using `mdlExternal_shmemGet`.

pExternalName points to the external name that `mdlExternal_shmemGet` supplied in the *externalName* field of the `MSShmemDescr` it created.

Under DOS, `SHMEM_SC` is defined as `_far`. `SHMEM_SC` is a 48-bit far pointer.

If `msextern.h` is compiled without having `pm386` defined for the compiler, then `SHMEM_SC` is defined to expand to nothing. If that occurs, then the segment portion of the 48-bit pointer returned by `extprg_shmemAttach` is lost. This causes very serious bugs. Verify that the profile used by the High C compiler defines the constant `pm386`.

Returns The `extprg_shmemAttach` function returns a pointer to the shared memory area if no errors occur. If an error occurs, it returns `NULL`. Under DOS, the pointer is a 48-bit address.

See Also `mdlExternal_shmemGet`, `extprg_shmemDetach`.

extprg_shmemDetach

```
#include <msextern.h>

void extprg_shmemDetach
(
    SHMEM_SC    *shmemP /* => pointer to shared memory region */
);
```

Description The `extprg_shmemDetach` function notifies MicroStation that the external program is no longer using the shared memory region. *shmemP* is a shared memory pointer returned from `mdlExternal_shmemGet`.

An external program that uses shared memory should call `extprg_shmemDetach` when the shared memory is no longer needed.

Returns The `extprg_shmemDetach` function is of type `void`. It returns no value.

See Also `mdlExternal_shmemGet`, `extprg_shmemAttach`.

extprg_signalUstn (UNIX platforms only)

```
void extprg_signalUstn();
```

Description The `extprg_signalUstn` function tells MicroStation to examine its message queues looking for messages. If it finds a message queue that has a message, and the

application that created the message queue has a `userExternal_messageReceived` user hook, then MicroStation receives the message and passes it to the `userExternal_messageReceived` user hook.

Applications that rely on `extprg_signalUstn` are not portable. The function is not available under DOS and may not be available on other platforms.

Returns The `extprg_signalUstn` function is of type `void`.

extprg_initializePlatform

```
int extprg_initializePlatform
(
    int      stackSize      /* => Size of stack in bytes */
);
```

Description `extprg_initializePlatform` does platform-specific initialization for an external program. Because an external program can do its own initialization; this function is not required.

On the Macintosh, this function sets the application heap limit (`ApplLimit`) based on the given `stackSize` (see Macintosh documentation for further information) and initializes the Toolbox managers required for communication with the MDL program.

Returns `extprg_initializePlatform` always returns `SUCCESS`.

See Also `extprg_shutdownPlatform`.

extprg_getCommandLineArguments

```
int extprg_getCommandLineArguments
(
    int      argc,          /* <= Number of arguments */
    char     ***argv        /* <= Array of string pointers to arguments */
);
```

Description On the Macintosh, `extprg_getCommandLineArguments` returns the arguments that were passed to the external program by `mdlExternal_startProgram` as command line arguments really don't exist.

On other platforms, this function does nothing.

Returns `extprg_initializePlatform` returns either `SUCCESS` or `MDLERR_INSMEMORY` if unable to allocate space for the strings (Macintosh only).

extprg_shutdownPlatform

```
void extprg_shutdownPlatform(void);
```


Description *extprg_shutdownPlatform* does platform specific shutdown for an external program. An external program can do its own shutdown/cleanup; this function is not required.

On the Macintosh, this function frees memory allocated for the command line arguments.

Returns *extprg_shutdownPlatform* is of type `void`; it returns no value.

See Also *extprg_initializePlatform*.

userExternal_messageReceived

```
#include <msextern.h>

void userExternal_messageReceived
(
    ExternalMessage *pMessage      /* => the received message */
);
```

Description MicroStation calls *userExternal_messageReceived* when a message is received from an external program and the MDL program with the attached message queue is not waiting for a message. This function does not cause a modal, or wait state such as *mdlExternal_messageReceive*.

The *userExternal_messageReceived* user function is designated to MicroStation when the user function name is specified in a call to *mdlExternal_setFunction*. *userExternal_messageReceived* does not need to be the function name. The actual function name does not matter to MicroStation.

Under DOS, *userExternal_messageReceived* serves no real purpose since an asynchronous message cannot be received. Under DOS, the external program runs only when the MDL application calls *mdlExternal_messageSend* or *mdlExternal_messageReceive*.

However, *userExternal_messageReceived* can be valuable in UNIX because the external program runs asynchronously and can make asynchronous requests. When an external program asynchronously sends a message, it should call *extprg_signalUstn* (UNIX platforms only) after calling *extprg_messageSend* to notify MicroStation that it should poll the message queues.

Returns MDL ignores the *userExternal_messageReceived* return value.

See Also *mdlExternal_messageReceive*, *mdlExternal_setFunction*.

userExternal_programTerminated

```
void userExternal_programTerminated
(
    MSExternalProgramDescr*pDescr      /* => external pgm descr */
);
```

Description The *userExternal_programTerminated* function can be called when an external program terminates.

pDescr points to an `MSEExternalProgramDescr` structure. The `mdlExternal_startProgram` function creates the descriptor.

The external program's exit status is provided in *pDescr->exitStatus*.

When the *userExternal_programTerminated* function returns to MDL, the descriptor and all associated data structures are freed. The MDL application must not use the descriptor and its structures again.

Returns The *userExternal_programTerminated* function is of type `void`. It returns no value.

See Also `mdlExternal_setFunction`, `mdlExternal_startProgram`, `mdlExternal_terminateProgram`, `mdlExternal_wait`.

23

Digitizer Functions

The digitizer functions are used to interpret data received from tablet input devices.

The following table lists digitizer functions:

Function	Used to
<code>mdlDigitize_setFunction</code>	designate a user-supplied function called by MicroStation when it needs the application to control input.
<code>mdlDigitize_getTabletPoint</code>	designate a user-supplied function, <code>userDigitize_getTabletPoint</code> , be called the next time a data, command, or reset button is entered on the input device.

Function	MicroStation calls when
<code>userDigitize_getTabletPoint</code>	the next time a data, command or reset button is entered on the input device.
<code>userDigitize_setup</code>	the user finishes entering monument point pairs.
<code>userDigitize_transform</code>	MicroStation has a point in tablet coordinates that it needs to be transformed to design file coordinates (UORs).
<code>userDigitize_uorInput</code>	whenever MicroStation needs to know what the user-controlled input position is.

mdlDigitize_setFunction

```
#include <userfnc.h>

MdlFunctionP mdlDigitize_setFunction
(
    int          type,      /* => type of events to be handled */
    MdlFunctionP functionP /* => function to be called */
);
```

Description MicroStation provides two methods for an application to affect the interpretation of input coordinates from tablet input devices. The first method allows the application to calculate the linear transformation matrix that MicroStation uses internally to transform coordinates from the tablet to design file coordinates. The second method allows the application to perform the transformation, thus allowing for an arbitrary (and possibly nonlinear) transformation.

MicroStation also allows an application to supply user input directly in design file coordinates, allowing arbitrary input devices to be supported.

`mdlDigitize_setFunction` is used to designate a user-supplied function that is called by MicroStation when it needs the application to take action or supply information to control the input. The *type* parameter must be either `DIGITIZE_SETUPHOOK`, `DIGITIZE_TRANSFORMHOOK`, or `DIGITIZE_UORINPUT`, and it specifies which of the types of events the MDL function designated by *functionP* handles:

<i>type</i>	<i>functionP</i> called when
<code>DIGITIZE_SETUPHOOK</code>	the user finishes entering monument points pairs during <code>DIGITIZER SETUP</code> command (see <code>userDigitize_setup</code>).
<code>DIGITIZE_TRANSFORMHOOK</code>	MicroStation needs a point on the digitizer (outside the screen partition) transformed to design file coordinates, (see <code>userDigitize_transform</code>).
<code>DIGITIZE_UORINPUT</code>	MicroStation wants position and/or input device information from the user application.

An application that simply wants to control the characteristics of the linear transformation that MicroStation uses to go from tablet coordinates to design file coordinates need only set up a `DIGITIZE_SETUPHOOK` function.

An application that wants a nonlinear transformation between tablet coordinates and design file coordinates would set up both a `DIGITIZE_SETUPHOOK` function to get the user defined monument point pairs and a `DIGITIZE_TRANSFORMHOOK` function to get control whenever MicroStation is transforming input coordinates.

An application that supports a non-tablet input device would use a `DIGITIZE_UORINPUT` function.

To stop MicroStation from calling an application function for any of the events described above, call `mdlDigitize_setFunction` with the appropriate value of *type* and `NULL` for the *functionP* parameter.

Returns `mdlDigitize_setFunction` returns a pointer to the user function (of the same type) that was previously set using `mdlDigitize_setFunction`. If *type* is invalid, `mdlDigitize_setFunction` returns -1.

See Also `userDigitize_setup`, `userDigitize_transform`, `userDigitize_uorInput`.

mdlDigitize_getTabletPoint

```
void mdlDigitize_getTabletPoint
(
MdlFunctionP functionP      /* => function to be called */
);
```

Description This function requests that the designated user-supplied function, `userDigitize_getTabletPoint`, be called the next time a data, command, or reset button is entered on the input device. The user function is only called once; `mdlDigitize_getTabletPoint` must be called each time a raw tablet point is desired by the application. Calling this function with a `NULL` argument will cancel a previous request to call the user function.

Returns This function is of type `void`, it does not return a value.

See Also `userDigitize_getTabletPoint`, `mdlDigitize_setFunction`.

userDigitize_getTabletPoint

```
#include <userfnc.h>
#include <mdl.h>

void userDigitize_getTabletPoint
(
Point3d pointP,    /* => tablet position */
int    region,     /* => not used */
int    button      /* => tablet button (DATAPNT, CMDPNT, RESET) */
);
```

Description This function called when a data, command, or reset button is entered on the input device following a call to `mdlDigitize_getTabletPoint`. The raw tablet position is specified by *pointP* and *button* specifies the input device button that was entered.

Returns This function does not return a value.

See Also `mdlDigitize_getTabletPoint`.

userDigitize_setup

```

#include <userfnc.h>
#include <mdl.h>

int userDigitize_setup
(
    Transform    *digTransP,          /* <= digitizer transform */
    DPoint3d     *monumentPointsP,    /* => monument points, uors */
    DPoint3d     *tabletPointsP,      /* => corresponding points on tablet */
    int          numPoints            /* => number of points */
);

```

Description If an MDL application designates it as a digitize setup function, *userDigitize_setup* is called when the user finishes entering monument point pairs in the DIGITIZER SETUP command. The application programmer determines the function name; *userDigitize_setup* is used merely as an example.

digTransP supplies the address where the application program should put the output digitizer transformation. MicroStation transforms tablet coordinates as follows:

```

2D design file (mgds_modes.three_d == 0)
outPoint.x = digTransP->form2d[0][0] * inPoint.x +
digTransP->form2d[0][1] * inPoint.y + digTransP->form2d[0][2];

outPoint.y = digTransP->form2d[1][0] * inPoint.x +
digTransP->form2d[1][1] * inPoint.y + digTransP->form2d[1][2];

3D design file (mgds_modes.three_d == 0)
outPoint.x = digTransP->form3d[0][0] * inPoint.x +
digTransP->form3d[0][1] * inPoint.y + digTransP->form3d[0][3];

outPoint.y = digTransP->form3d[1][0] * inPoint.x +
digTransP->form3d[1][1] * inPoint.y + digTransP->form2d[1][3];

outPoint.z = digTransP->form3d[2][0] * inPoint.x +
digTransP->form3d[2][1] * inPoint.y + digTransP->form2d[2][3];

```

In the 3D case, since the Z coordinate on input is inherently zero, the third column of the 3×4 matrix is effectively unused.

monumentPointsP is the address of an array of input points that the user identified as the monument points in the design file.

tabletPointsP is the address of an array of input points that the user identified as the points on the tablet that correspond to the monument points in the design file. There is one tablet point for each monument point.

numPoints is the number of points in the *monumentPointsP* and *tabletPointsP* arrays.

Returns *userDigitize_setup* must return zero if it successfully calculates a digitizer transform, or any non-zero value if the calculation cannot be performed.

See Also mdlDigitize_setFunction.

userDigitize_transform

```
#include <userfnc.h>
#include <mdl.h>

void userDigitize_transform
(
    DPoint3d    *outPointP,    /* <= output point, design coordinates */
    DPoint3d    *inPointP     /* => input point */
);
```

Description If an MDL application designates it as a digitize transform function, *userDigitize_transform* is called whenever MicroStation has a point in tablet coordinates that it needs to be transformed to design file coordinates (UORs). The application programmer determines the function name; *userDigitize_transform* is used merely as an example.

outPointP supplies the address where the application program should put the transformed point.

inPointP is the address of the input point in tablet coordinates.

Returns The return value from *userDigitize_transform* is not used.

See Also mdlDigitize_setFunction.

userDigitize_uorInput

```
#include <userfnc.h>
#include <mdl.h>
#include <msdefs.h>
#include <msbutton.h>

void userDigitize_uorInput
(
    Point3d *uorP,           /* <= output point, design coordinates */
    int     *newDataP,       /* <= new data indicator */
    int     *buttonCodeP,    /* <= button code */
    int     *buttonTransP,   /* <= button transition */
    short   *shiftStatusP,   /* <= button shift status */
    ULONG   *timeP,          /* <= button time */
    int     shouldBuffer     /* => tells app if buffering input is worth it */
);
```

Description If an MDL application designates it as a UOR input function, *userDigitize_uorInput* is called whenever MicroStation needs to know what the

user-controlled input position is. The application programmer determines the function name; *userDigitize_uorInput* is used merely as an example.

outPointP supplies the address where the application program should put the current user-controlled input position.



The application is responsible for transforming whatever coordinates are returned from the input device into design file coordinates.

newDataP is the address where the application program returns whether there is new position information in *outPointP* or not, and if whether there is more buffered position information available. If *newDataP* is set to NO_UORDATA, the application is indicating that the input position is the same as it was the previous time the *userDigitize_uorInput* function was called. If *newDataP* is set to NEW_UORDATA, the application is indicating that there is new position information, and if *newDataP* is set to MORE_UORDATA, the application is indicating that there is no information available, and there is more position information that has been buffered as well. The application should buffer position information if it can when it gets called with the *shouldBuffer* parameter set to TRUE. This occurs only when the user is using one of the stream input MicroStation commands.

If the application wants to indicate that the user has pushed a button or other actuator on the input device, it should return a non-zero value in the address supplied in the *buttonCodeP* parameter. The value should either DATAPNT, TENTPNT or RESET as defined in msbutton.h. Whenever *buttonCodeP* is set non-zero, *buttonTransP* must be set to either BUTTONTRANS_DOWN or BUTTONTRANS_UP as defined in msdefs.h. Also, *timeP* should be set to the time at which the button transition occurred (as returned by mdlSystem_getTicks) and *shiftStatusP* should be set to either 0 (for no modifiers pressed) or some OR'ed combination of SHIFTKEY and CTRLKEY as defined in keys.h.

Returns *userDigitize_uorInput* should always return 0.

See Also mdlDigitize_setFunction.

24

Data Conversion

The data conversion functionality of MicroStation provides a standard way to transfer data between formats and platforms.

The following sections comprise this chapter:

- Conversion functions
- Binary portability functions

Conversion Functions

Data conversion functions provide techniques for converting values between various MicroStation formats. Most of these functions are rarely needed.

The following table lists data conversion functions:

Function	Used to
mdlCnv_IPointToPoint	convert Point3d to Dpoint3d.
mdlCnv_IPointToPointArray	convert array of Point3ds to array of Dpoint3ds.
mdlCnv_DPointToPoint	convert Dpoint3d to Point3d.
mdlCnv_DPointToPointArray	convert array of Dpoint3ds to array of Point3ds.
mdlCnv_fromR50ToAscii	convert Radix50 string to ASCII string.
mdlCnv_fromAsciiToR50	convert ASCII string to Radix50 string.
mdlCnv_toRational	determines if a double precision floating point value is a rational fraction.
mdlCnv_fromScanFormat	convert unsigned long number in scan format to signed long number in internal format.
mdlCnv_fromScanFormatArray	convert an array of unsigned long numbers in scan format to signed long numbers in internal format.
mdlCnv_toScanFormat	convert signed long number in internal format to unsigned long number in scan format.

Function	Used to
mdlCnv_toScanFormatArray	convert an array of signed long numbers in internal format to unsigned long numbers in scan format.
mdlCnv_doubleFromFileFormat	convert floating point number in file format to internal format.
mdlCnv_doubleToFileFormat	convert floating point number in internal format to file format.
mdlCnv_swapWord	convert a long integer from MIDDLE_ENDIAN.
mdlCnv_swapWordArray	format to native long integer format.
mdlCnv_doubleToNativeFloat mdlCnv_nativeFloatToDouble	convert a double to a float or a float to a double.
mdlCnv_roundDoubleToLong mdlCnv_roundDoubleToULong	convert a double to a long or unsigned long.
mdlCnv_masterToUOR	convert a length in Master Units to a length in UORs.
mdlCnv_UORToMaster	convert a length in UORs to a length in Master Units.

mdlCnv_IPointToPoint, mdlCnv_IPointToPointArray, mdlCnv_DPointToPoint, mdlCnv_DPointToPointArray

```
#include <mdl.h>

void mdlCnv_IPointToPoint
(
    Dpoint3d      *dPoint,          /* <= Dpoint representation */
    Point3d       *iPoint          /* => Ipoint value */
);

void mdlCnv_IPointToPointArray
(
    Dpoint3d      *dPoint,          /* <= Dpoint3d array */
    Point3d       *iPoint,          /* => Point3d array */
    int           numPoints         /* => number of points in array */
);

void mdlCnv_DPointToPoint
(
    Point3d       *iPoint,          /* <= Point3d representation */
    Dpoint3d      *dPoint          /* => Dpoint3d value */
);

void mdlCnv_DPointToPointArray
(
```

```

Point3d      *iPoint,          /* <= Point3d array */
Dpoint3d     *dPoint,          /* => Dpoint3d array */
int          numPoints         /* => number of points in array */
);

```

Description The mdlCnv_IPointToDPoint function converts the Point3d in *iPoint* to a Dpoint3d in *dPoint*.

The mdlCnv_IPointToDPointArray function converts the array of Point3ds in *iPoint* to an array of Dpoint3ds in *dPoint*. *numPoints* gives the number of points.

The mdlCnv_DPointToIPoint function converts the Dpoint3d in *dPoint* to a Point3d in *iPoint*.

The mdlCnv_DPointToIPointArray function converts the array of Dpoint3ds in *dPoint* to an array of Point3ds in *iPoint*. *numPoints* gives the number of points.

See Also The mdlCnv_IPointToDPoint, mdlCnv_IPointToDPointArray, mdlCnv_DPointToIPoint, and mdlCnv_DPointToIPointArray functions are of type void. They return no values.

mdlCnv_fromR50ToAscii, mdlCnv_fromAsciiToR50

```

#include <mdl.h>

void mdlCnv_fromR50ToAscii
(
    int      length,          /* => max len of output str (in mult of 3) */
    short    *r50chars,       /* => radix 50 to be converted */
    char     *asciiString     /* <= output string */
);

void mdlCnv_fromAsciiToR50
(
    int      length,          /* => length of string to be converted */
    char     *asciiString,     /* => string to be converted */
    short    *r50chars        /* <= output array */
);

```

Description Radix50 is a compact format for storing character strings. It can store three characters for each 16-bit word. The disadvantage of Radix50 is that it is not valid for all ASCII characters. MicroStation uses Radix50 in many cases, such as for cell names.

mdlCnv_fromR50ToAscii converts the Radix50 string *r50chars* to an ASCII string in *asciiString*. *length* is the size of the output buffer. Its size should be a multiple of 3.

The `mdlCnv_fromAsciiToR50` function converts the ASCII string in *asciiString* to a Radix50 string in *r50Chars*. *length* is the length of the input string.



Radix50 is valid for capital letters (A-Z), numbers (0-9), the dollar sign '\$', space ' ', period '.' and underscore '_'. All other ASCII characters are translated to a space.

Returns `mdlCnv_fromR50ToAscii` and `mdlCnv_fromAsciiToR50` are of type `void`. They return no values.

mdlCnv_toRational

```
boolean mdlCnv_toRational
(
double input,          /* => floating point value to check */
int *numeratorP,       /* <= numerator if input is rational */
int *denominatorP      /* <= denominator if input is rational */
);
```

Description `mdlCnv_toRational` determines whether a double precision floating point value is a rational fraction with a denominator of 2, 3, 4, 5, 6, 8, 10 or 12. The algorithm is really designed for numbers less than 1.0, but if *input* is greater than 1.0, the number is first divided into 1.0, the same algorithm is applied, and the numerator and denominator are reversed on output, so in that case the numerator is in the above set.

Returns If the number is not rational, the function returns `FALSE`, in which case *numeratorP* and *denominatorP* are meaningless. If the number is rational, *numeratorP* is set to the numerator and *denominatorP* is set to the denominator of the rational fraction. Numbers less than 0.2 or greater than 5.0 are not considered, and the function will return `FALSE`.

mdlCnv_fromScanFormat, mdlCnv_fromScanFormatArray, mdlCnv_toScanFormat, mdlCnv_toScanFormatArray

```
#include <mdl.h>

long mdlCnv_fromScanFormat
(
ULong input           /* => value changed to internal format */
);

ULong mdlCnv_toScanFormat
(
long input            /* => value changed to scan format */
);

long mdlCnv_fromScanFormatArray
(
```

```

ULong  *inputP, /* => pointer to array to convert */
int     count   /* => number to convert */
);

ULong mdlCnv_toScanFormatArray
(
long    *inputP, /* => pointer to array to convert */
int     count   /* => number to convert */
);

```

Description For historical reasons, the MicroStation design file scanner uses a coordinate value format that differs from the format used elsewhere in the program. `mdlCnv_fromScanFormat` and `mdlCnv_toScanFormat` exist to convert numbers to and from internal to scan format. Range values in element headers are also stored in scan format.

The `mdlCnv_fromScanFormat` function converts an unsigned long number, *input*, in scan format, to a signed long number in internal format.

The `mdlCnv_toScanFormat` function converts a signed long number, *input*, in internal format, to an unsigned long number in scan format.

The `mdlCnv_toScanFormatArray` and `mdlCnv_fromScanFormatArray` functions perform the same conversion, but they take a pointer to an array of values to be converted (in place) in *inputP* and the number of values to be converted in the array in *count*. These two functions are not built-ins; to use them link with `mdl.lib`.

Returns `mdlCnv_fromScanFormat` and `mdlCnv_toScanFormatArray` return signed long values that are the internal format equivalents of the input values. `mdlCnv_toScanFormat` and `mdlCnv_fromScanFormatArray` return unsigned long values that are the scan format equivalents of the input values.

mdlCnv_doubleFromFileFormat, mdlCnv_doubleToFileFormat

```

#include <mdl.h>

void mdlCnv_doubleFromFileFormat
(
double *dValue /* <=> value converted to internal format */
);

void mdlCnv_doubleToFileFormat
(
double *dValue /* <=> value converted to internal format */
);

```

Description Different computers use different formats for storing floating point numbers. MicroStation and MDL always use the native format for the computer on which they are operating. However, since this format varies from computer to computer,

floating point values saved in the file sometimes need to be converted to the internal format.

The `mdlCnv_doubleFromFileFormat` function converts the floating point number in the file format pointed to by *dValue* to internal format.

The `mdlCnv_doubleToFileFormat` function converts the floating point number in internal format pointed to by *dValue* to file format.



MicroStation automatically converts all floating point values that it knows about to internal format. These functions are needed only for floating point information saved manually (in attribute information at the end of elements).

Returns `mdlCnv_doubleFromFileFormat` and `mdlCnv_doubleToFileFormat` are of type `void`.

mdlCnv_swapWord, mdlCnv_swapWordArray

```
void mdlCnv_swapWord
(
    long    *word      /* <=> 4-byte integer to be swapped */
);

void mdlCnv_swapWordArray
(
    long    *words,    /* <=> 4-byte integer to be swapped */
    int     numLongs   /* => number of longs in array */
);
```

Description The `mdlCnv_swapWord` function converts a long integer from the `MIDDLE_ENDIAN` (PDP-11) format that is common in the MicroStation file format to the native `long` integer format. On a `LITTLE_ENDIAN` machines, such as the Intel and Intergraph Clipper processors, the two 2-byte words are swapped. On `BIG_ENDIAN` machines such as the mc68000, SPARC and HP 700 processors, the bytes within each of the two 2-byte words swapped. The swapping is done in-place on the `long` integer whose address is specified by the *word* parameter.



The function also performs the inverse conversion, i.e., from native `long` format to the `MIDDLE_ENDIAN` format.

The `mdlCnv_swapWordArray` function is similar, except that the *words* parameter specifies an address that is the start of an array of *numLongs* longs, each of which is converted.

Returns Both `mdlCnv_swapWord` and `mdlCnv_swapWordArray` are of type `void` and return no value.

See Also “Differing Data Representations” in the *Resources* chapter of the *MDL Programmer’s GuideStartHelpCode*.

mdlCnv_doubleToNativeFloat, mdlCnv_nativeFloatToDouble

```
#include <mdl.h>

void mdlCnv_doubleToNativeFloat
(
    float    *fValue,          /* <= float representation */
    double   *dValue          /* => double representation */
);

void mdlCnv_nativeFloatToDouble
(
    double   *dValue,          /* <= double representation */
    float    *fValue          /* => float representation */
);
```

Description MDL does not support the `float` C data type. Thus, machines on which the size of floats differs from the size of doubles (generally four bytes verses eight bytes) can have trouble reading or writing external data files with floats in them. The `mdlCnv_doubleToNativeFloat` and `mdlCnv_nativeFloatToDouble` functions convert between the native float and double formats.



The data type for *fValue* is declared as `char` since MDL does not have float. The data type should be large enough to hold a float.

Returns The `mdlCnv_doubleToNativeFloat` and `mdlCnv_nativeFloatToDouble` functions are of type `void`. They return no values.

mdlCnv_roundDoubleToLong, mdlCnv_roundDoubleToULong

```
#include <mdl.h>

long mdlCnv_roundDoubleToLong
(
    double   dValue           /* => double representation */
);

ULong mdlCnv_roundDoubleToULong
(
    double   dValue           /* => double representation */
);
```

Description The `mdlCnv_roundDoubleToLong` and `mdlCnv_roundDoubleToULong` functions round (rather than truncate) a double value to a long or unsigned long value.

Returns `mdlCnv_roundDoubleToLong` returns a long integer representation of *dValue*.
`mdlCnv_roundDoubleToULong` returns an unsigned long integer representation of *dValue*.

mdlCnv_masterToUOR, mdlCnv_UORToMaster

```

int mdlCnv_masterToUOR
(
    double *UORsP,          /* <= length in UORs */
    double masterUnits,     /* => length in Master Units */
    int     fileNum         /* => MASTERFILE or ref. file number */
);

int mdlCnv_UORToMaster
(
    double *masterUnitsP,   /* <= length in Master Units */
    double UORs,           /* => length in UORs */
    int     fileNum         /* => MASTERFILE or ref. file number */
);

```

Description mdlCnv_masterToUOR converts a length in Master Units, specified by the *masterUnits* parameter, to a length in UORs. The result is stored in the address specified by the *UORsP* parameter. The mdlCnv_UORToMaster performs the inverse conversion. The *fileNum* parameter is MASTERFILE if *masterUnits* is specified for the master file, or 1 through *tcb->maxRefs* if it is specified for one of the reference files.

Returns mdlCnv_masterToUOR and mdlCnv_UORToMaster return MDLERR_BADSLOT if the *fileNum* is out of range or if it specifies a reference file slot that is not in use. A value of -1 is returned if the output pointer is NULL. Otherwise the return code is SUCCESS.

Binary Portability Functions

The binary portability functions allow an application to read data from a file that was generated on a platform with a different architecture from the “current” platform. The “current” platform is defined as the platform where the calling application is running. Conversely, these functions also allow an application to write data to a file that will be read on a different platform.

These functions will be useful to applications that need to:

- Convert user data in application elements
- Convert resources to/from version 4.0 files
- Convert data to/from an application-specific file

Resources from version 5.0 files are automatically converted by the MicroStation Resource Manager. For converting user data stored in element linkages, see the mdlLinkage_... functions.

How the Binary Portability Functions Work

The Binary portability functions use **data definition resources** to convert data from one machine format to another. A data definition corresponds to a data structure, or in terms of the C language, a typedef.

There must be one data definition resource for every structure type that will need to be written to or read from file. Each data definition resource is assigned a unique resource ID which is passed in as an argument to a binary portability function.

The data definition is the foundation for converting data to or from a file, whether the data comes from a Type 66 element, a user attribute linkage on another element type, a resource file, or an application-defined data file.



To completely understand how to use the binary portability functions, you should read the “Binary Portability” section of the “Resources” chapter in the MDL Programmer’s Guide.

The following table lists Binary Portability functions:

Function	Used to
mdlCnv_compileDataDefBlock	convert a data definition into a set of conversion rules. This is automatically called by mdlCnv_bufferToFileFormat and mdlCnv_bufferFromFileFormat if necessary.
mdlCnv_bufferToFileFormat	convert data from native machine format to platform independent format.
mdlCnv_bufferFromFileFormat	convert data from platform independent format to native machine format.
mdlCnv_calcMemSizeFromDataDef	calculate the size of an output buffer needed for mdlCnv_bufferFromFileFormat.
mdlCnv_calcFileSizeFromDataDef	calculate the size of an output buffer needed for mdlCnv_bufferToFileFormat

mdlCnv_compileDataDefBlock

```
#include <msbnrypo.fdf>

int mdlCnv_compileDataDefBlock /* <= SUCCESS or error. */
(
    boolean *inBufferCnvPossibleP, /* <= same buffer conv poss? */
```

```

int      *sizeofConvRulesP,          /* <= compiled cnv rules size */
void     **convRulesPP,              /* <= save compiled info here */
ULONG    dataDefBlockID              /* => RscId of datadef block */
);

```

Description The `mdlCnv_compileDataDefBlock` function converts a data definition resource to a set of conversion rules. The conversion rules are used to convert data of a particular structure from native “in memory” format to platform independent (file) format, or vice versa. The conversion rules are passed in as a parameter to one of the other Binary Portability routines or one of the `mdlLinkage_...` routines.

inBufferCnvPossibleP is a pointer to an integer which indicates whether the *extFmtDataP* and *intrnlFmtDataP* parameters may be one and the same in the `mdlCnv_bufferToFileFormat` and `mdlCnv_bufferFromFileFormat` function calls. This will be the case if the in-memory and file format alignment restrictions for the structure are identical. This argument may be `NULL`.

sizeofConvRulesP is a pointer to an integer which tells the size of the generated conversion rules so the application can determine if it should free the conversion rules after each use (and regenerate it as needed), or keep it around to speed up the performance of the `mdlCnv_bufferToFileFormat` and `mdlCnv_bufferFromFileFormat` functions. If this parameter is `NULL`, it is ignored.

convRulesPP should be the address of a pointer with the value of `NULL` before calling the `mdlCnv_compileDataDefBlock` function. Afterwards, the `NULL` pointer will have been set to point to the `malloced` conversion rules. This memory is allocated out of the calling application’s heap, so it is up to the calling application to free the memory. As stated above, these conversion rules are passed in as a parameter to one of the other Binary Portability routines or one of the `mdlLinkage_...` routines.

dataDefBlockID is the resource ID of a data definition block resource. A data definition block resource functions as a “road map” for a particular data structure, providing the information necessary to convert it to or from file format. The correlation between a data definition resource ID and a structure definition is made in an application’s `.mt` files. The *dataDefBlockID* resource is searched for and loaded from the application’s open resource files. Once loaded, it is “compiled” into a set of conversion rules appropriate for the current platform.

mdlCnv_compileDataDefBlock returns SUCCESS or one of the following error codes:

Error Code	Description
MDLERR_BADARG	<i>convRulesPP</i> must be the address of a pointer to NULL.
MDLERR_DATADEFNOTFOUND	The data definition corresponding to <i>dataDefBlockID</i> could not be found in any of the resource files opened by the application.
MDLERR_BADDATADEF	The data definition resource is corrupt.
MDLERR_INSMEMORY	Insufficient memory to allocate the conversion rules.

See Also mdlCnv_bufferToFileFormat, mdlCnv_bufferFromFileFormat, mdlLinkage_extractFromElement, mdlLinkage_deleteFromElement, mdlLinkage_appendToElement.

mdlCnv_bufferToFileFormat

```
#include <msbnrpo.fdf>

int mdlCnv_bufferToFileFormat /* <= SUCCESS or error. */
(
    int      *extSizeP,          /* <= Size of converted data */
    byte     *extFmtDataP,      /* <= Where to store converted data */
    void     **convRulesPP,     /* <=> Conversion rules */
    byte     *intrnlFmtDataP,   /* => Data to be converted */
    ULong    dataDefBlockID     /* => Data def block rsc id */
);
```

Description mdlCnv_bufferToFileFormat converts in-memory (native machine format) data to file format data. The calling program must have opened a resource file that contains a data definition resource corresponding to the structure of the data to be converted.

extSizeP is a pointer to an integer that indicates the size of the data converted. Pass NULL if this information is not needed.

extFmtDataP points to a buffer to receive the converted (file format) data. The size of the converted data pointed to by *extFmtDataP* will never be greater than the size of the data pointed to by *intrnlFmtDataP*.

convRulesPP is used to save and reuse the data conversion rules. When conversion rules need to be generated, the data definition resource indicated by *dataDefBlockID* is loaded from one of the application's open resource files and "compiled" into the set of conversion rules. Conversion rules are bi-directional, that is, one set of rules for a particular structure may be used for calls to mdlCnv_bufferToFileFormat and mdlCnv_bufferFromFileFormat.

If *convRulesPP* is `NULL`, the rules will be regenerated for the conversion and discarded afterwards. If (**convRulesPP*) is `NULL`, the rules will still be regenerated, but their address will be saved at that location. If (**convRulesPP*) is non-`NULL` (i.e., contains a pointer saved from a previous call), the overhead of loading and compiling the conversion rules will be skipped and the conversion will take place immediately.

intrnlFmtDataP points to the buffer that needs to be converted to file format.

dataDefBlockID identifies the resource to be used in generating the conversion rules. See the description of the *dataDefBlockID* parameter under `mdlCnv_compileDataDefBlock` for more information.

dataDefBlockID resource could not be found in any of the application's open resource files.

Returns `mdlCnv_bufferToFileFormat` returns `SUCCESS`, `MDLERR_DATADEFNOTFOUND`, `MDLERR_BADDATADEF`, `MDLERR_INSMEMORY` (described on the previous page) or one of the following additional error codes:

Error Code	Description
<code>MDLERR_NOTCONVRULES</code>	The conversion rules pointed to by (<i>convRulesPP</i>) were invalid.
<code>MDLERR_BUFFERALIGNMENTSDIFFER</code>	<i>intrnlFmtDataP</i> and <i>extFmtDataP</i> were equal when an in-buffer conversion could not be performed. See the description of the <i>inbufferCnvPossibleP</i> parameter under <code>mdlCnv_compileDataDefBlock</code> for more information.

See Also `mdlCnv_compileDataDefBlock`, `mdlCnv_bufferFromFileFormat`, `mdlCnv_calcMemSizeFromDataDef`, `mdlCnv_calcFileSizeFromDataDef`.

mdlCnv_bufferFromFileFormat

```
#include <msbncpyo.fdf>

int mdlCnv_bufferFromFileFormat /* <= SUCCESS or error */
(
    boolean *doubleAlignP, /* <= converted data dbl aligned? */
    int      *intrnlSizeP,  /* <= Size of converted data */
    byte     *intrnlFmtDataP, /* <= store converted data here */
    void     **convRulesPP, /* <=> Conversion rules */
    byte     *extFmtDataP,  /* => Data to be converted */
    ULONG    dataDefBlockID /* => Data definition block rsc id */
);
```

Description The `mdlCnv_bufferFromFileFormat` function is used to convert file format data to in-memory (native machine format) data. The calling program must have opened a resource file that contains a data definition resource corresponding to the structure of the data to be converted.

doubleAlignP points to a boolean flag which will indicate whether the converted data (in native machine format) needs to be kept aligned on a `double` boundary. This parameter can usually be `NULL` since the converted data will always be aligned to a double boundary. (This flag would be useful to the caller if the data had to be copied to another location in memory at a later time).

intrnlSizeP is used to record the size of the converted data. This parameter can be `NULL`.

intrnlFmtDataP points to a buffer to receive the converted (native machine format) data.

convRulesPP is used to save and reuse the data conversion rules. See the description of the *convRulesPP* parameter under `mdlCnv_bufferToFileFormat` for more information.

extFmtDataP points to the buffer that needs to be converted to native machine format.

dataDefBlockID identifies the resource to be used in generating the conversion rules. See the description of the *dataDefBlockID* parameter under `mdlCnv_compileDataDefBlock` for more information.

`mdlCnv_bufferFromFileFormat` returns `SUCCESS` or one of the following error codes:

Error Code	Description
MDLERR_DATADEFNOTFOUND	The <i>dataDefBlockID</i> resource could not be found in any of the application's open resource files.
MDLERR_BADDATADEF	The data definition resource is corrupt.
MDLERR_INSMEMORY	Insufficient memory to allocate the conversion rules.
MDLERR_NOTCONVRULES	The conversion rules pointed to by (<i>*convRulesPP</i>) were invalid.
MDLERR_BUFFERALIGNMENTSDIFFER	<i>intrnlFmtDataP</i> and <i>extFmtDataP</i> were equal when an in-buffer conversion could not be performed. See the description of the <i>inbufferCnvPossibleP</i> parameter under <code>mdlCnv_compileDataDefBlock</code> for more information.

See Also `mdlCnv_compileDataDefBlock`, `mdlCnv_bufferToFileFormat`,
`mdlCnv_calcMemSizeFromDataDef`, `mdlCnv_calcFileSizeFromDataDef`.

mdlCnv_calcMemSizeFromDataDef, mdlCnv_calcFileSizeFromDataDef

```
#include <msbncpyo.fdf>

int mdlCnv_calcMemSizeFromDataDef /* <= SUCCESS or error. */
(
    boolean *doubleAlignP,      /* <= converted data dbl aligned? */
    int      *intrnlSizeP,      /* <= Size of converted data */
    void      **convRulesPP,    /* <=> Conversion rules */
    byte      *extFmtDataP,     /* => Data to be converted */
    ULONG     dataDefBlockID    /* => Data definition block rsc id */
);

int mdlCnv_calcFileSizeFromDataDef /* <= SUCCESS or error. */
(
    int      *extSizeP,         /* <= Size of converted data */
    void      **convRulesPP,    /* <=> Conversion rules */
    byte      *intrnlFmtDataP,  /* => Data to be converted */
    ULONG     dataDefBlockID    /* => Data def block rsc id */
);
```

Description *mdlCnv_calcMemSizeFromDataDef* steps through the input buffer *extFmtDataP* in exactly the same way that *mdlCnv_bufferFromFileFormat* would, but without actually converting any data. This allows an application to determine what a suitably sized output buffer would be for *mdlCnv_bufferFromFileFormat* if the size is not known in advance.

mdlCnv_calcFileSizeFromDataDef steps through the input buffer *intrnlFmtDataP* in exactly the same way that *mdlCnv_bufferToFileFormat* would, but without actually converting any data. This allows an application to determine what a suitably sized output buffer would be for *mdlCnv_bufferToFileFormat* if the size is not known in advance.

The arguments for *mdlCnv_calcMemSizeFromDataDef*, *doubleAlignP*, *intrnlSizeP*, *convRulesPP*, *extFmtDataP* and *dataDefBlockID*, have the same meaning as they do for *mdlCnv_bufferFromFileFormat*.

The arguments for *mdlCnv_calcFileSizeFromDataDef*, *extSizeP*, *convRulesPP*, *intrnlFmtDataP* and *dataDefBlockID*, have the same meaning as they do for *mdlCnv_bufferToFileFormat*.

Returns *mdlCnv_calcMemSizeFromDataDef* and *mdlCnv_calcFileSizeFromDataDef* return SUCCESS or one of the error codes that are also returned by *mdlCnv_bufferFromFileFormat* and *mdlCnv_bufferToFileFormat*.

See Also *mdlCnv_bufferFromFileFormat*, *mdlCnv_bufferToFileFormat*.

25

File Manipulation

This chapter contains functions that manipulate files, directories and filenames.

In this chapter, the following topics are discussed:

- File utility functions
- Work file functions
- File list functions
- Text file functions
- Low Level I/O Support

File Utility Functions

Because conventions for handling filenames and directories vary across operating systems, writing portable code to handle these issues is difficult. The file functions alleviate the problem by hiding the differences.

The following table lists file functions:

Function	Used to
<code>mdlFile_create</code>	derive a filename that can be used to create a file.
<code>mdlFile_find</code>	find a file based on an incomplete filename specification.
<code>mdlFile_parseName</code>	derive the components of a filename.
<code>mdlFile_buildName</code>	create a filename based on the components of the name.
<code>mdlFile_getcwd</code>	retrieve the name of the current working directory.

Function	Used to
<code>mdlFile_mkdir</code>	create a new directory and, unlike <code>mkdir</code> , create the path to that directory if necessary.
<code>mdlFile_getDiskFree</code>	determine the number of bytes free on the specified file system.
<code>mdlFile_isValidDrive</code> (PC only)	check for the existence of a drive.
<code>mdlFile_setDrive</code> (PC only)	set the current drive.
<code>mdlFile_getDrive</code> (PC only)	retrieve the number of the current drive.
<code>mdlFile_findFiles</code>	create a list of files that satisfy the given requirements.
<code>mdlFile_setDefaultShare</code> (PC only)	modify the file sharing mode used by <code>fopen</code> to open a file.
<code>mdlFile_getFileAttributes</code>	return the attributes of the given name.
<code>mdlFile_checkDesignFile</code>	query whether a file is a valid design file and if it is two or three dimensional.
<code>mdlFile_setFileType</code> (Macintosh only)	set a file's type and creator on Macintosh platforms.
<code>mdlFile_copy</code>	copy a file.
<code>mdlFile_abbreviateName</code>	abbreviates a filename by successively replacing leading directory names with ellipses.
<code>mdlFile_isUntitledDesign</code>	determine if active file is untitled.
<code>mdlFile_findFirst</code>	
<code>mdlFile_findNext</code>	

Example

See `file.mc`.

mdlFile_create, mdlFile_find

```
#include <mdefs.h>

int mdlFile_find
(
    char    *outnameP,      /* <=> name of file */
    char    *innameP,      /* => usually root filename */
    char    *envvarP,      /* => usually environment variable */

```



```
char    *extP          /* => usually file extension */
);

int mdlFile_create
(
char    *outnameP,      /* <=> name of file */
char    *innameP,       /* => usually root filename */
char    *envvarP,       /* => usually environment variable */
char    *extP          /* => usually file extension */
);
```

Description The `mdlFile_find` function finds a file based on the names provided in *innameP*, *envvarP*, and *extP*. It returns the full file path specified in the buffer *outnameP*.

The `mdlFile_create` function creates a full file specification based on the names provided in *innameP*, *envvarP* and *extP*. It returns the full file path specified in the buffer *outnameP*. If a file with that name exists before the call, the file is deleted.

The string pointed to by *innameP* is a filename that has a format a user would enter. It can contain a logical filename and a path. It can contain an extension. The information in *innameP* is used first to find the file.

The string pointed to by *envvarP* usually provides the name of an environment variable. The information in the environment variable generally provides a list of paths to examine when searching for the file. *envvarP* can be `NULL`.

The string pointed to by *extP* usually provides a file extension (example `.dgn`). *extP* can be `NULL`.

outnameP points to a buffer to receive the full filename. The size of the buffer must be at least `MAXFILELENGTH` bytes. *outnameP* can be `NULL` if `mdlFile_find` is being used to check for the existence of a file.

Returns The `mdlFile_find` function returns `SUCCESS` if it finds the file. Otherwise, it returns a non-zero value. If `mdlFile_find` returns `SUCCESS` and *outnameP* is not `NULL`, the buffer that *outnameP* points to receives the full file specification.

The `mdlFile_create` function returns `SUCCESS` if a valid file could be created. If so, *outnameP* contains a valid file path. Otherwise, the value in *outnameP* is invalid. If `mdlFile_create` returns `SUCCESS` and *outnameP* is not `NULL`, the buffer *outnameP* points to receives the full file specification.

See Also `mdlFile_parseName`.

mdlFile_parseName

```
#include <msdefs.h>

void mdlFile_parseName
(
    char    *inputNameP,    /* => filename to be parsed */
    char    *devP,          /* <=> receives device name */
    char    *dirP,          /* <=> receives dir specification */
    char    *nameP,         /* <=> receives root filename */
    char    *extP           /* <=> receives file extension */
);
```

Description *mdlFile_parseName* parses the filename pointed to by *inputNameP* and places the components in the buffers the other variables point to. Any pointers can be `NULL`.

If *mdlFile_parseName* does not find a component in the input name, the corresponding buffer is not modified. For example, if the input name does not contain a device specification, the buffer *devP* points to is not modified.

devP points to a buffer to receive the device name. The buffer must be at least `MAXDEVICELENGTH` bytes.

dirP points to a buffer to receive the directory name. The buffer must be at least `MAXDIRLENGTH` bytes.

nameP points to a buffer to receive the root filename. The buffer must be at least `MAXNAMELENGTH` bytes.

extP points to a buffer to receive the extension name. The buffer must be at least `MAXEXTENSIONLENGTH` bytes.

Returns The *mdlFile_parseName* function is of type `void`. It returns no value.

See Also *mdlFile_buildName*, *mdlFile_create*, *mdlFile_find*.

mdlFile_buildName

```
#include <msdefs.h>

void mdlFile_buildName
(
    char    *outputNameP,   /* <=> receives filename */
    char    *devP,          /* => device name */
    char    *dirP,          /* => directory specification */
    char    *nameP,         /* => root filename */
    char    *extP           /* => file extension */
);
```

Description mdlFile_buildName creates a filename in the buffer pointed to by *outputNameP* using the components specified by the other variables. Any pointers can be NULL.

Any components can be NULL, or they can point to an empty string. If a component is NULL or points to an empty string, mdlFile_buildName does not place the corresponding separator into the output string. For example, on a PC if *devP* is NULL, mdlFile_buildName does place a colon in the output string before the directory specification.

devP points to the device name; *dirP* points to the directory name. The final character of the directory name should be the directory separator. If it is not, mdlFile_buildName appends a directory separator, but if it is, mdlFile_buildName does not add another directory separator.

nameP points to the root filename. *extP* points to the extension name.

outputNameP points to the buffer to receive the full filename. The buffer must have at least MAXFILELENGTH bytes.

Returns The mdlFile_buildName function is of type void. It returns no value.

See Also mdlFile_parseName.

mdlFile_getcwd

```
void mdlFile_getcwd
(
    char    *workingDirectoryP, /* <=> receives value */
    int     bufferSize         /* => size of buffer */
);
```

Description The mdlFile_getcwd function stores the name of the current working directory in the buffer pointed to by *workingDirectoryP*.

Returns The mdlFile_getcwd function is of type void. It returns no value.

mdlFile_mkdir

```
int mdlFile_mkdir
(
    char    *path /* <=> receives value */
);
```

Description mdlFile_mkdir creates the new directory and path specified in *path*. Unlike the standard C function mkdir, mdlFile_mkdir creates the path to the resulting directory if it doesn't already exist.

A sample value for *path* would be d:\d1\d2\d3\d4.

Returns The mdlFile_mkdir function returns SUCCESS (zero) if successful, and ERROR if the path and/or directory were not created. The ERROR values correspond to those returned by the implementation of mkdir on the platform being used.

mdlFile_getDiskFree

```
int mdlFile_getDiskFree
(
    unsigned long    *freeSpaceP,    /* <= receives bytes free */
    int              fileSystem      /* => drive number */
);
```

Description The `mdlFile_getDiskFree` function reports the number of bytes free on the file system specified by *fileSystem*.

freeSpaceP receives the number of free bytes available on *fileSystem*.

fileSystem specifies the operating system file system to examine for free storage. Where 0 means the “current” drive, 1 means A:\, 2 means B:\, 3 means C:\, etc. This function returns meaningful information on DOS, Microsoft Windows and IBM OS/2.

Returns The `mdlFile_getDiskFree` function returns `SUCCESS` if no errors occurred. Otherwise, it returns `ERROR`.

mdlFile_isValidDrive (PC only)

```
boolean mdlFile_isValidDrive
(
    char    *drive    /* <= drive to test for existence */
);
```

Description `mdlFile_isValidDrive` checks to see if the logical drive specified by *drive* is a valid drive (a physical device). It is meaningful only for the PC.

drive specifies a drive and must be terminated by a path separator.

Returns `mdlFile_isValidDrive` returns `TRUE` if *drive* is a valid drive. Otherwise, it returns `FALSE`.

On platforms other than the PC, `mdlFile_isValidDrive` always returns `FALSE`.

mdlFile_setDrive (PC only)

```
int mdlFile_setDrive
(
    int    *logicalDrivesP,    /* <= num drive devices in system */
    char    *defaultDrive      /* => new default drive number */
);
```

Description The `mdlFile_setDrive` function establishes a new default drive for file system operations. It is meaningful only for the PC. Performing an `mdlFile_setDrive` has

the same effect as interactively changing the DOS default drive with a drive letter key-in such as A:.

logicalDrivesP receives the number of logical DOS drives in the system. The DOS CONFIG.SYS parameter `LASTDRIVE` controls this value. The logical drives begin with drive letter A: and continue consecutively through the alphabet up to the `LASTDRIVE` value. For example, if **logicalDrivesP* is 10, the system contains drive devices A: through J:.

drive specifies the new default drive. It is an integer corresponding to the drive number. Drive A: is 1, drive B: is 2, and so on.



Each returned logical drive is not necessarily valid. The logical drives represent only space that DOS has allocated internally for drive devices. A given logical device cannot be readable because no physical device is associated with it. Or, in the case of a floppy diskette drive, no media is in the drive.

Returns The `mdlFile_setDrive` function returns `SUCCESS` if no errors occurred. Otherwise, it returns `ERROR`. The current default drive is unchanged if the function failed.

On platforms other than the PC, `mdlFile_setDrive` always returns `ERROR`.

See Also `mdlFile_getDrive` (PC only), `mdlFile_isValidDrive` (PC only).

mdlFile_getDrive (PC only)

```
int mdlFile_getDrive
(
    int      *driveNumberP /* => default drive number */
);
```

Description The `mdlFile_getDrive` function returns the current default drive number. It is meaningful only for the PC.

driveNumberP receives the default drive. It is an integer corresponding to the DOS drive number, where drive A: is 1, drive B: is 2, and so on.

Returns The `mdlFile_getDrive` function returns `SUCCESS` if no errors occurred. Otherwise, it returns `ERROR`.

On platforms other than the PC, `mdlFile_getDrive` always returns `ERROR`.

See Also `mdlFile_setDrive` (PC only), `mdlFile_isValidDrive` (PC only).

mdlFile_findFiles

```
#include <system.h>

int mdlFile_findFiles
(
    FindFileInfo **fileInfoP,      /* <= receives ptr to an array */
    int             *fileCount,    /* <= receives count of files */
    char           *fileSpec,      /* => file specification */
    int             attributeFilter /* => attributes for find */
);
```

Description The `mdlFile_findFiles` function collects information on files matching the specifications provided by *fileSpec* and *attributeFilter*. Information on files and directories can be obtained.

`mdlFile_findFiles` allocates the memory to hold an array of `FindFileInfo` structures using the `malloc` function. Each entry in the array describes a file or directory matching the specifications.

fileInfoP points to a variable that receives a pointer to the array of `FindFileInfo` structures.

fileCount points to a variable set to a count of the number of entries in the array of `FindFileInfo` structures.

fileSpec points to a file specification describing the requested files or directories. It is an expression that consists of any combination of constant characters and the special wild card characters `?` and `*`. The wild card characters are expanded by `mdlFile_findFiles` exactly as the operating system expands these pattern-matching characters in interactive key-ins. The wild card character `?` matches any single character and the wild card character `*` matches any sequence of characters.

attributeFilter is set to any of the following constants, which can be combined with the bitwise OR operator `|`:

Constant	Description
FF_NORMAL	files
FF_READONLY	read-only attribute
FF_SUBDIR	directories

When the information is no longer needed, the MDL application must call `free` to free the memory pointed to by *fileInfoP*.

Returns `mdlFile_findFiles` returns `SUCCESS` if no errors occurred. It returns `MDLERR_INSMEMORY` if memory could not be allocated for the array of `FindFileInfo` structures.

mdlFile_setDefaultShare (PC only)

```
#include <system.h>

int mdlFile_setDefaultShare
(
    int    shareMode    /* => default file sharing mode */
);
```

Description On the PC, `mdlFile_setDefaultShare` sets a file sharing mode used by `fopen` when opening a file for the MDL task. On platforms other than the PC, it does nothing.

shareMode may be `MDL_SHARE_COMPATIBILITY`, `MDL_SHARE_DENY_READ_WRITE`, `MDL_SHARE_DENY_WRITE`, `MDL_SHARE_DENY_READ` or `MDL_SHARE_DENY_NONE`. By default, `MDL_SHARE_DENY_NONE` is used.

DOS ignores the sharing mode unless the DOS utility SHARE is installed.

The effect of the sharing mode flags is as follows:

`MDL_SHARE_COMPATIBILITY` allows unlimited access to the file from the same machine. Programs running on other machines cannot access the file while it is open unless the file has the read-only attribute. Any attempt to open the file in compatibility mode fails if the file has already been opened with any other sharing mode.

`MDL_SHARE_DENY_READ_WRITE` provides exclusive access to the file. Any subsequent attempts to open the file fail. This mode fails if the file has already been opened in any mode other than `MDL_SHARE_COMPATIBILITY` mode.

`MDL_SHARE_DENY_WRITE` permits subsequent opens for read-only access. This mode fails if the file has already been opened with `MDL_SHARE_COMPATIBILITY` or `MDL_SHARE_DENY_WRITE` mode.

`MDL_SHARE_DENY_READ` permits subsequent opens for write-only access. This mode fails if the file has already been opened with `MDL_SHARE_COMPATIBILITY` or `MDL_SHARE_DENY_READ` mode.

`MDL_SHARE_DENY_NONE` is similar to `MDL_SHARE_COMPATIBILITY`, but it does not allow the file to be opened in compatibility mode. This mode fails if the file has already been opened in compatibility mode.

Returns The `mdlFile_setDefaultShare` function returns the previous file sharing mode.

mdlFile_getFileAttributes

```
#include <msdefs.h>
#include <system.h>

int mdlFile_getFileAttributes
(
```

```
int      *attributesP, /* <= attributes of nameP */
char     *nameP        /* => file name or directory path */
);
```

Description The `mdlFile_getFileAttributes` function returns the attributes of the given name in *attributesP*.

nameP points to a string specifying either a complete directory path or complete file name specification.

Returns The `mdlFile_getFileAttributes` function returns `SUCCESS` if it finds the directory or file. Otherwise, it returns a non-zero value.

See Also `mdlFile_find`.

mdlFile_checkDesignFile

```
int mdlFile_checkDesignFile
(
boolean *isThreeD,      /* <= TRUE if file is 3d */
char    *fileName      /* => filename */
);
```

Description The `mdlFile_checkDesignFile` function checks whether a file is a valid design file and if it is a design file whether it three or two dimensional.

isThreeD points to a boolean that is set to `TRUE` if *fileName* is a three dimensional design file.

fileName is the name of the file to be checked.

Returns `mdlFile_checkDesignFile` returns `SUCCESS` if the file is a design file and `ERROR` otherwise.

mdlFile_setFileType (Macintosh only)

```
#include <msfile.fdf>
#include <msdefs.h>

int mdlFile_setFileType /* <= SUCCESS or error. */
(
char      *filename,    /* => set type of this file */
int       volume,       /* => volume ref. number (usually 0) */
unsigned long type,     /* => file type */
unsigned long creator    /* => file creator */
);
```

Description `mdlFile_setFileType` is used to set the file type and creator of a file. This function is typically called for newly created files. Currently, this function only has meaning on the Macintosh platform, however, it will compile successfully on all platforms.

filename specifies the file to have its type information set.

volume is used to indicate the volume reference number where *filename* resides. If *filename* is specified with an absolute path, *volume* can be set to zero.

type indicates the file type of the file. MicroStation defines the following types (applications may also provide their own values for files not in this list):

Type Macro	Value	File Type
DGNFILE_2DTYPE	Dgn2	2D Design
DGNFILE_3DTYPE	Dgn3	3D Design
CELFILE_2DTYPE	Cel2	2D Cell Library
CELFILE_3DTYPE	Cel3	3D Cell Library
FONTFILETYPE	uFnt	Fonts
HELPIFILETYPE	uHlp	Help
TEXTFILETYPE	uTxt	MicroStation Text
PLTFCGFILETYPE	TEXT	Plotter Configuration
USERPREFFILETYPE	uPrf	User Preferences
PLOTSTREAM	uPfl	Plotter Stream
PLOTEPSF	EPSF	Extended PostScript
FUNCKEYFILETYPE	uFky	Function Key Definitions
CTBLFILETYPE	uCtb	Color Table
RSCFILETYPE	uRsc	Resource File
DWGFILETYPE	DWG	DWG Format File
DXFFILETYPE	DXF	DXF File

creator is the signature of the application to whom the file should belong. For the file types listed in the description of the *type* parameter, *creator* should be USTNSIGNATURE.

Returns mdlFile_setFileType returns SUCCESS or an operating system-specific error code.

See Also mdlImage_getOSFileType.

mdlFile_copy

```
int mdlFile_copy
(
    char    *destinationP, /* => name of destination file */
    char    *sourceP      /* => name of source file */
);
```

Description The `mdlFile_copy` function creates the file specified by *destinationP* and copies to it the contents of the file specified by *sourceP*.

If the file specified by *destinationP* does not exist before the call, `mdlFile_copy` creates it. If it does exist before the call, `mdlFile_copy` deletes its contents.

Returns The `mdlFile_copy` function returns `SUCCESS` if no errors occurred. Otherwise, it returns a non-zero value. The variable `errno` will contain additional information.

mdlFile_abbreviateName

```
void mdlFile_abbreviateName
(
char    *fileName,      /* <=> file name to be abbreviated */
int     maxLength      /* => maximum length of abbreviated name */
);
```

Description The `mdlFile_abbreviateName` function abbreviates a filename to at most *maxLength* characters. It abbreviates by successively removing leading directory names, replacing them with ellipses (...).

fileName points to the string holding the file name. It is modified with the resultant file name.

maxLength is the maximum number of characters desired in the resultant name.

Returns `mdlFile_abbreviateName` is of type `void`; it does not return a status.

mdlFile_findFirst

mdlFile_findNext

mdlFile_isUntitledDesign

```
#include <msfile.fdf>

int mdlFile_isUntitledDesign
(
char    *dgnFileNameP
);
```

Description The `mdlFile_isUntitledDesign` function is typically used to determine if the active design file is an untitled file.

dgnFileNameP points to the string holding the file name to check.



This function was implemented in MicroStation 95.

Returns mdlFile_isUntitledDesign returns TRUE if *dgnFileNameP* is the name of an untitled file. Otherwise, FALSE is returned.

See Also mdlSystem_saveDesignFile, mdlSystem_newDesignFile.

Work File Functions

It may often be desirable to perform design file operations in a temporary file or to create a new design file or cell library while keeping the current one open. The functions discussed in this section allow users to perform design file operations on files not tracked or recognized directly by MicroStation. The file operations are performed using element descriptors and, in most cases, element descriptor functions can be used to manipulate the elements read from or written to a work file.

Before performing any file operations with these functions, the work file must be opened with the `fopen` library function. This function opens the file and returns a FILE pointer which is used in all work file functions performing file operations.



A file which is to be used as a MicroStation file (design file or cell library) should not be used for normal file operations (`fwrite`, `fread`, etc.). This is important because MicroStation file operations are performed on an element basis and assume that the data has been properly formatted and stored in the file. That is, elements are converted on file I/O operations and the EOF marker is not the same in MicroStation files as the operating/file system EOF marker.

In addition to properly opening and accessing the work file as described above, work file functions require that the file/element type be specified. The file/element type indicates whether the data is to be treated as a 2D or 3D element definition. This allows performing operations on 2D and 3D files at the same time. Since 2D and 3D elements can be handled by the MDL application at the same time using these functions, it is important that the programmer not use certain element descriptor functions when the element descriptor describes a 2D element and the master file is 3D or vice-versa. The functions to be avoided in these cases are:

- all the file I/O operations (cannot be used on work files).
- all the display functions (`mdlElmdscr_displayFromFile` cannot be used on work files).
`mdlElmdscr_undoableDelete`
`mdlElmdscr_igdsSize`
`mdlElmdscr_transform`
`mdlElement_stroke`
`mdlElmdscr_validate`
`mdlElmdscr_markElement`

As with all element descriptor operations, *it is up to the programmer to free the memory allocated* by the MDL functions when element descriptors are created by calling the `mdlElmdscr_freeAll` function. For more information concerning element descriptors, refer to the *Element Descriptor Functions* section in this manual.



All `mdlWorkDgn_...` operations are permanent and are not “remembered” by MicroStation. Care should be taken when modifying files through these functions.

The following table lists work file functions:

Function	Used to
<code>mdlWorkDgn_read</code>	read element descriptor from a work file.
<code>mdlWorkDgn_write</code>	write element descriptor to a work file.
<code>mdlWorkDgn_findEof</code>	locate the design file EOF position for the work file.
<code>mdlWorkDgn_delete</code>	delete the element descriptor from the work file.
<code>mdlWorkDgn_igdsSize</code>	determine the file size of the element descriptor.
<code>mdlWorkDgn_validate</code>	update the complex header of the element descriptor.

mdlWorkDgn_read

```
#include <stdio.h>
#include <mselems.h>

ULong mdlWorkDgn_read
(
  MSElementDescr  **elemDescrPP, /* <= Element dscr return area */
  ULong            filePos,        /* => File read position */
  FILE             *fileP,         /* => File pointer */
  ULong            *startFilePos,  /* <= actual file pos */
  int              fileType        /* => File type (2d or 3d) */
);
```

Description The `mdlWorkDgn_read` function reads the element at *filePos* in file *fileP* and creates a new element descriptor. A pointer to the element descriptor is returned in *elemDescrPP* (the “PP” indicates the it is a pointer to a pointer). The *fileP* parameter must be a file opened by the `fopen` library function.

If the element at address *filePos* is a complex header, `mdlWorkDgn_read` reads, and allocates memory for, the header and all of its components.

The *fileType* parameter specifies the dimension type of the file being read. A value of 0 means the file contains 2D elements and a value of 1 means the file contains 3D elements.



mdlWorkDgn_read never returns deleted elements; they are skipped. Therefore, the file position for the element descriptor *elemDscrPP* can differ from *filePos* if *filePos* points to a deleted element. The actual file position from which the elements that comprise the element descriptor were read is returned in *startFilePos*. If the actual file position is not needed, pass NULL for *startFilePos*.

Returns mdlWorkDgn_read returns the file position of the next element in the file. If the value returned is 0, an error occurred and the reason is returned in mdlErrno.

See Also mdlElmdscr_freeAll.

mdlWorkDgn_write

```
#include <stdio.h>
#include <mselems.h>

ULong mdlWorkDgn_write
(
    MSElementDescr    *elemDscrP,    /* => Element dscr to write */
    ULong              filePos,        /* => File write position */
    FILE               *fileP,        /* => File pointer */
    int                cacheSlot,      /* => Unused */
    int                fileType        /* => File type (2d or 3d) */
);
```

Description mdlWorkDgn_write writes the element(s) pointed to by *elemDscrP* at the file position *filePos* in file *fileP*. *fileP* must be a file opened by the fopen library function.

If *filePos* has a value of -1, the element(s) are appended to the end of the file. If *filePos* does not specify the EOF position and the element(s) at *filePos* are the same size as those pointed to by *elemDscrP* then the old one is overwritten, otherwise the old one is deleted and the new one is appended to the end of the file.

fileType specifies the dimension type of the file being written to. A value of 0 means the file contains 2D elements and a value of 1 means the file contains 3D elements.

The *cacheSlot* parameter is currently unused.

Returns mdlWorkDgn_write returns the file position of the next element in the file. If the value returned is 0, an error occurred and the reason is returned in mdlErrno.

See Also mdlWorkDgn_findEof.

mdlWorkDgn_findEof

```
#include <stdio.h>
#include <mselems.h>

ULong mdlWorkDgn_findEof
(
    FILE    *fileP,          /* => File pointer */
    int     cacheSlot        /* => Unused */
);
```

Description The `mdlWorkDgn_findEof` function returns the EOF position for the file *fileP*. The *fileP* parameter must be a file opened by the `fopen` library function.

The *cacheSlot* parameter is currently unused.

Returns `mdlWorkDgn_findEof` returns the address of the EOF. If an error occurs, 0 is returned and the reason code is returned in `mdlErrno`.

See Also `mdlWorkDgn_write`.

mdlWorkDgn_delete

```
#include <stdio.h>
#include <mselems.h>

ULong mdlWorkDgn_delete
(
    MSElementDescr *elemDescrP, /* => Element descr to delete */
    ULong           filePos,      /* => File position */
    FILE            *fileP,       /* => File pointer */
    int             cacheSlot,    /* => Unused */
    int             fileType      /* => File type (2d or 3d) */
);
```

Description The `mdlWorkDgn_delete` function deletes the element(s) pointed to by *elemDescrP* at *filePos* in file *fileP*. If an element descriptor is not available when doing the delete, pass NULL for *elemDescrP* to cause MicroStation to read the element(s) from the file before performing the delete. This adds some overhead to `mdlWorkDgn_delete`, so always pass the element descriptor if it is available. The *fileP* parameter must be a file opened by the `fopen` library function.

The *fileType* parameter specifies the dimension type of the file being referenced. A value of 0 means the file contains 2D elements and a value of 1 means the file contains 3D elements.

The *cacheSlot* parameter is currently unused.

Returns `mdlWorkDgn_delete` returns `SUCCESS` if the element is deleted.

See Also `mdlWorkDgn_read`, `mdlElmdscr_undoableDelete`.

mdlWorkDgn_igdsSize

```
#include <mselems.h>

ULong mdlWorkDgn_igdsSize
(
    MSElementDescr    *elemDescrP,    /* <= Element dscr */
    int                 elementType      /* => Element type (2d or 3d) */
);
```

Description The mdlWorkDgn_igdsSize function returns the size of the elements contained in the element descriptor pointed to by *elemDescrP*. The *elementType* parameter specifies the dimension type of the elements in *elemDescrP*. A value of 0 means the elements are 2D and a value of 1 means the elements are 3D.

The *cacheSlot* parameter is currently unused.

Returns mdlWorkDgn_igdsSize returns the total size of the elements in *elemDescrP*.

See Also mdlElmdscr_igdsSize.

mdlWorkDgn_validate

```
#include <mselems.h>

void mdlWorkDgn_validate
(
    MSElementDescr    *elemDescrP,    /* <=> Element dscr */
    int                 elementType      /* => Element type (2d or 3d) */
);
```

Description The `mdlWorkDgn_validate` sets the information in the complex header elements contained in *elemDescrP* based upon the component elements in the descriptor. It sets the following fields:

Element types	fields validated
CMPLX_STRING_ELM CMPLX_SHAPE_ELM SURFACE_ELM SOLID_ELM	Header range, number of elements, total words in description.
TEXT_NODE_ELM	Header range, number of text strings, maximum length of text strings, total words in description.
CELL_HEADER_ELM CELL_LIB_ELM SHAREDCELL_DEF_ELM	Header range, class mask, level mask, total words in description.
BSPLINE_CURVE_ELM	Header range, number of poles, total words in description.
BSPLINE_SURFACE_ELM	Header range, total words in description.
SHARED_CELL_ELM	Header range, level mask.
DIMENSION_ELM MULTILINE_ELM	Header range.



There is an “isValid” flag in the header of the element descriptor structure. `mdlWorkDgn_validate` sets this flag to `TRUE` when it is done and does nothing if it is already `TRUE` before it is called.

The *elementType* parameter specifies the dimension type of the elements in *elemDescrP* being sized. A value of 0 means the elements are 2D and a value of 1 means the elements are 3D.

Returns `mdlWorkDgn_validate` does not return a value.

See Also `mdlElmdscr_validate`.

File List Functions

File list functions assist the programmer in using lists of files, directories and drives. Seven specialized functions let the user choose a file for opening or creating.

The following table lists file list functions:

Function	Used to
mdlFileList_get	fill a file list with data corresponding to the given input.
mdlFileList_edit	edit/create a file list.
mdlDialog_fileOpen	let the user choose a file for opening.
mdlDialog_fileCreate	let the user create a file.
mdlDialog_fileCreateFromSeed	let the user create a file based on a seed file.
mdlDialog_defFileOpen	let the user choose a file for opening. This differs from mdlDialog_fileOpen in that the entered filename and file filter become the defaults.
mdlDialog_defFileCreate	let the user choose a file for creating. This differs from mdlDialog_fileCreate in that the entered filename and file filter become the defaults.
mdlFileList_fromString	searches the specified directories for files matching the criteria.

mdlFileList_get

```
#include <filelist.h>
#include <dlogitem.h>

int mdlFileList_get
(
    StringList *stringListP,      /* <= list to hold output */
    unsigned int attributes,      /* => specify what to get */
    char *defaultDirectory,      /* => dir to start searching */
    char *fileFilter              /* => file filter */
);
```

Description The mdlFileList_get function puts a list of files/directories/drives as specified by *attributes*, *defaultDirectory* and *fileFilter* in *stringListP*.

stringListP should be created before this call. All current members are deleted.

attributes accepts any (or all) of the following values:

FILELISTATTR_FILES, FILELISTATTR_DIRECTORIES and FILELISTATTR_DRIVES. FILELISTATTR_FILES requests a list of files for the given directory. FILELISTATTR_DIRECTORIES requests a list of directories for the given directory. FILELISTATTR_DRIVES requests a list of drives associated with the system. If the programmer specifies more than one

value on the line (by joining them with the logical OR operator), all requested information will be in the list.

defaultDirectory points to the directory where the selection process starts.

fileFilter points to the filter to use for selecting files to include in the file list. It is valid only when the `FILELISTATTR_FILES` attribute is set.

By default, the list is sorted.

Returns The `mdlFileList_get` function returns `SUCCESS` if operation is successful. Otherwise, it returns a string list manager error.

See Also `mdlFile_findFiles`, `mdlFileList_edit`.

mdlFileList_edit

```
#include <filelist.h>
#include <dlogitem.h>

StringList *mdlFileList_edit
(
    FileInfo *lastInfoP,      /* <= state of things on exit */
    StringList *stringListP, /* => string list to edit, NULL if none */
    long      attributes,    /* => file selection */
    char      *dialogTitle,  /* => title of dialog box */
    char      *listLabel,    /* => heading for list box */
    char      *fileFilter,   /* => standard file filter */
    char      *defaultDirectory /* => dir to start search in */
);
```

Description The `mdlFileList_edit` function gives the user a convenient way (through a dialog box) to create and edit a list of files, directories or drives.

lastInfoP is a pointer to a `FileInfo` structure returning the “state” of the file list dialog box upon termination of the box. The structure of `FileInfo` is:

```
typedef struct filelistinfo
{
    int lastAction;
    char lastDirectory[MAXFILELENGTH];
    char lastFilter[MAXEXTENSIONLENGTH];
} FileInfo;
```

lastAction contains the last action taken by the user. It will contain either `ACTIONBUTTON_CANCEL` or `ACTIONBUTTON_OK`.

lastDirectory contains the directory the user was in when the CANCEL or OK buttons were pressed.

lastFilter contains the file filter the user was using when the CANCEL or OK buttons were pressed.



Memory for lastInfoP must be allocated before mdlFileList_edit is called.

stringListP contains the list of files, directories, or drives to be edited. Before the list can be edited, it is validated. If *stringListP* is NULL, a new empty list is created and edited.

Specifies the characteristics desired for the file list.

attributes specifies the characteristics desired for the file list, and can have the following values:

Attribute	Meaning
FILELISTATTR_SORT	The list is always sorted. This attribute is case-sensitive.
FILELISTATTR_UNIQUE	Duplicate members are not allowed.
FILELISTATTR_CASESENSITIVE	Members in the list are case-sensitive. If case is not specified, all members will be converted to lower case.
FILELISTATTR_MULTIPLE	List can contain more than one member.
FILELISTATTR_OPEN	Only existing files are allowed in list.
FILELISTATTR_CREATE	Only non-existing files are allowed in list.
FILELISTATTR_OPENCREATE	Either new or existing files are allowed in list.
FILELISTATTR_FILES	Create/edit a list of files.
FILELISTATTR_DIRECTORIES	Create/edit a list of directories.
FILELISTATTR_DRIVES	Create/edit a list of drives.
FILELISTATTR_DEFAULT	This includes: FILELISTATTR_SORT, FILELISTATTR_UNIQUE, FILELISTATTR_MULTIPLE, FILELISTATTR_OPEN and FILELISTATTR_FILES.

Certain conflicts can arise and are handled as follows:

Only one of the following should be specified, but if more than one is, they take precedence in the following order:

1. FILELISTATTR_FILES
2. FILELISTATTR_DIRECTORIES
3. FILELISTATTR_DRIVES

Only one of the following can be specified: FILELISTATTR_OPEN, FILELISTATTR_CREATE or FILELISTATTR_OPENCREATE. If none or more than one is specified, the default is FILELISTATTR_OPEN.

When a string list is passed as an argument (is not NULL), it must be validated so that it matches the attributes. Each member in the list is modified or deleted, depending on how it matches the attributes. For

example, if `FILELISTATTR_FILES` and `FILELISTATTR_OPEN` are specified, all list members that are not existing files are deleted. If `FILELISTATTR_UNIQUE` is specified, all duplicates in the list are removed. If `FILELISTATTR_CASESENSITIVE` is omitted, all members are changed to lower case.

The list is edited through one of four dialog boxes. The box the user selects depends on whether the user chooses `FILELISTATTR_MULTIPLE` and also on whether the user is selecting files, directories, or drives. The dialog handler ensures that all selected or typed-in data fields conform to the desired attributes. If the user clicks CANCEL, the original, unchanged string list is returned.

dialogTitle points to the dialog box title.

listLabel points to the title of the list box containing the list of files, directories, or drives to be edited if the `FILELISTATTR_MULTIPLE` attribute is set. Otherwise, it points to the label of the text input field.

fileFilter contains the filter used to determine which files to include in the file list. It is useful when trying to limit the files displayed to a particular type. Simple wildcarding is allowed. An asterisk '*' will match any string and a question mark '?' will match any single character. If *fileFilter* is NULL, the filter string defaults to matching all files (*.*). This argument is valid only when the `FILELISTATTR_FILES` attribute is set.

defaultDirectory contains the directory where the selection process starts. It can also be a configuration variable. In this case, the directory associated with that variable is used.

Returns `mdlFileList_edit` returns a valid pointer if it is successful and NULL otherwise. If *stringListP* is NULL, the function returns a new pointer, and *stringListP* otherwise.

See Also `mdlDialog_fileOpenExt`, `mdlDialog_fileCreate`, `mdlDialog_fileOpen`.

mdlDialog_fileOpen, mdlDialog_fileCreate

```
#include <rsdefs.h>

int mdlDialog_fileOpen
(
    char    *fileName,          /* <= fully specified file name */
    RscFileHandle rFileH,      /* => resource file handle */
    int     resourceId,        /* => resource ID */
    char    *suggestedFileName, /* => suggested file name */
    char    *fileFilter,       /* => file filter */
    char    *defaultDirectory, /* => directory to start search */
    char    *dialogTitle      /* => title of dialog box */
);

int mdlDialog_fileCreate
(
```

```

char    *fileName,           /* <= fully specified file name */
RscFileHandle  rFileH,       /* => resource file handle */
int      resourceId,         /* => resource ID */
char    *suggestedFileName, /* => suggested file name */
char    *fileFilter,         /* => file filter */
char    *defaultDirectory,   /* => directory to start search */
char    *dialogTitle         /* => title of dialog box */
);

```

Description The `mdlDialog_fileOpen` and `mdlDialog_fileCreate` functions let the user conveniently choose a file (through a dialog box) for opening or creating. It also lets the user modify the standard open/create dialog box by specifying an alternate dialog box that uses some functionality of the standard one.

fileName returns the name of the file to be opened or created. On Open, *filename* is guaranteed to list a valid existing file. On Create, if the file already exists, an alert will display asking the user if the existing file should be overwritten. If the user chooses CANCEL, the user can choose another file name. If the user chooses OK, the *filename* will be returned.

rFileH and *resourceId* offer users the functionality of standard file open/create dialog boxes in their own dialog boxes. *rFileH* is a handle to the resource file to use for loading a user-specified dialog box. If *rFileH* is NULL, the default resource file will be used.

resourceId is the ID of the dialog box to use within the resource file. If *resourceId* is 0, the default dialog box will be used. The first six items in the dialog box that the user creates should match these same items in the standard file open/create dialog box.

suggestedFileName suggests a filename for creating a file. It displays in the text field of the dialog. If a directory is attached to the filename, this argument serves as the default directory and the *defaultDirectory* argument is ignored. This argument should normally be NULL when it is called for opening files.

fileFilter contains the filter to use for determining which files to include in the file list. It is useful for limiting files displayed to a particular type. Simple wildcarding is allowed. An asterisk '*' will match any string and a question mark '?' will match any single character. If *fileFilter* is NULL, the filter string will match all files (*.*)

defaultDirectory contains the directory where the selection process starts, but can also be an environment variable. In the latter case, the directory associated with the variable is used. This argument can be overwritten with the *suggestedFileName* argument. If *defaultDirectory* is NULL, the current working directory will be used.

dialogTitle contains the title of the dialog box.

Returns `mdlDialog_fileOpen` and `mdlDialog_fileCreate` return `TRUE` if the CANCEL button is pressed, `FALSE` if the OK button is pressed, and `ERROR` if an error occurred while the dialog box was being created.

See Also `mdlDialog_fileOpenExt`, `mdlFileList_edit`, `mdlDialog_fileCreateFromSeed`.

mdlDialog_fileCreateFromSeed

```
#include <rscldefs.h>

int mdlDialog_fileCreateFromSeed
(
    char      *fileName,          /* <= fully specified file name */
    RscFileHandle  rFileH,        /* => resource file handle */
    int      resourceId,         /* => resource ID */
    char      *suggestedFileName, /* => suggested file name */
    char      *fileFilter,        /* => file filter */
    char      *defaultDirectory,  /* => dir. to start search */
    char      *dialogTitle,       /* => title of dialog box */
    char      *seedFile,          /* => dir. to start search */
    char      *seedDirectory,     /* => dir. to start search */
    char      *seedFilter         /* => title of dialog box */
);
```

Description `mdlDialog_fileCreateFromSeed` lets the user conveniently choose an existing seed file and a new file name for creating a design file. It also lets the user modify the create dialog box by specifying a dialog box to use. This dialog box uses some functionality of the standard one.

fileName returns the name of the file to be opened or created. To create a file, the user must check to see if the *fileName* returned currently exists and handle the situation appropriately. (This often involves setting up an alert box asking the user if the desired file should be overwritten).

rFileH and *resourceId* offer users the functionality of standard file open/create dialog boxes in their own dialog boxes. *rFileH* is a handle to the resource file to use for loading a user-specified dialog box. If *rFileH* is `NULL`, the default resource file will be used.

resourceId is the ID of the dialog box to use within the resource file. If *resourceId* is 0, the default dialog box will be used. The first six items in the dialog box that the user creates should match these same items in the standard file open/create dialog box.

suggestedFileName suggests a filename for creating a file. It displays in the text field of the dialog. If a directory is attached to the filename, this argument serves as the default directory and the *defaultDirectory* argument is ignored. This argument should normally be `NULL` when it is called for opening files.

fileFilter contains the filter to use for determining which files to include in the file list. It is useful for limiting files displayed to a particular type. Simple wildcarding is allowed. An asterisk '*' matches any string and a question mark '?' matches any single character. If *fileFilter* is NULL, the filter string will match all files (*.*) .

defaultDirectory contains the directory where the selection process starts. It can also be an environment variable. In this case, the directory associated with the variable is used. This argument can be overwritten with the *suggestedFileName* argument. If *defaultDirectory* is NULL, the current working directory will be used.

dialogTitle contains the title of the dialog box.

seedFile, *seedDirectory* and *seedFilter* are identical to *suggestedFileName*, *defaultDirectory* and *fileFilter* except that these are used for the seed file dialog box. The seed file dialog box can be invoked from the create dialog box and lets the user specify which seed file to use in the creation of the design file.

Returns The mdlDialog_fileCreateFromSeed functions return TRUE if the CANCEL button is pressed, FALSE if the OK button is pressed, and ERROR if an error occurred while the dialog box was being created.

See Also mdlDialog_fileOpenExt, mdlFileList_edit, mdlDialog_fileOpen, mdlDialog_fileCreate.

mdlDialog_defFileOpen, mdlDialog_defFileCreate

```
#include <rsdefs.h>

int mdlDialog_defFileOpen
(
    char          *fileName,          /* <= fully specified file name */
    RscFileHandle rFileH,             /* => dlogbox rsc file handle */
    int           resourceId,          /* => dlogbox resource ID */
    char          *suggestedFileName, /* => suggested file name */
    char          *fileFilter,         /* => file filter */
    char          *defaultDirectory,   /* => directory to start search */
    char          *dialogTitle,        /* => title of dialog box */
    int           defaultFileInfoID,   /* => Rsc ID of default info. */
    RscFileHandle userprefH            /* => File w/ default info */
);

int mdlDialog_defFileCreate
(
    char          *fileName,          /* <= fully specified file name */
    RscFileHandle rFileH,             /* => dlogbox rsc file handle */
    int           resourceId,          /* => dlogbox resource ID */
    char          *suggestedFileName, /* => suggested file name */
    char          *fileFilter,         /* => file filter */

```

```

char      *defaultDirectory,      /* => directory to start search */
char      *dialogTitle,          /* => title of dialog box */
int       defaultFileInfoID,     /* => Rsc ID of default info. */
RscFileHandle userprefH         /* => File w/ default info */
);

```

Description The `mdlDialog_defFileOpen` and `mdlDialog_defFileCreate` functions are extensions to the `mdlDialog_fileOpen` and `mdlDialog_fileCreate` functions. These two functions are also used to let the user conveniently choose a file (through a dialog box) for opening or creating. However, if the *defaultFileInfoID* parameter is provided, these functions will also “remember” the *suggestedFileName* and *fileFilter* parameters used to open or create the last file of the same type. These remembered parameters will then be used in place of the current *suggestedFileName* and *fileFilter* parameters. The *defaultDirectory* parameter will also be set to the directory where *suggestedFileName* was last loaded.

Since users generally group files of a given type in the same directory, using these functions in place of `mdlDialog_fileOpen` and `mdlDialog_fileCreate` will reduce the amount of directory changes required by the user in the file selection dialog box.

fileName, *rFileH*, *resourceId*, *suggestedFileName*, *fileFilter*, *defaultDirectory*, and *dialogTitle* are described in the function documentation for `mdlDialog_fileOpen` and `mdlDialog_fileCreate`.

suggestedFileName and *fileFilter* are overridden if the *defaultFileInfoID* parameter is provided and resource information for these parameters exists in the file specified by *userprefH*.

defaultFileInfoID identifies a resource in a user preferences file. This resource is used in two ways. First, it will be loaded just prior to the display of the file open or file creation dialog box to obtain the *suggestedFileName* and *fileFilter* used during the last execution of this dialog box (using the same *defaultFileInfoID*). If the user successfully chooses the same or some other file in the dialog box (indicated by clicking OK), the new filename and filter are saved back to the resource in the user preference file.

userprefH is the handle of a user preference resource file opened by the calling application or `NULL` if the MicroStation User Preferences file is to be used. This is where the default file information is loaded from and saved after the user makes a new file selection.

To access filenames and filters saved internally by MicroStation, you must provide one of the resourceIDs listed in the MDL system header file `deffiles.h` and set *userprefH* to `NULL` to indicate MicroStation’s User Preferences file.



A “user preference” resource file is a file created by the calling application for the purpose of storing user-defined operating parameters. Such a file is

reused by the application in later sessions. The MicroStation User Preferences file is automatically created by MicroStation.

Returns The mdlDialog_defFileOpen and mdlDialog_defFileCreate functions return TRUE if the CANCEL button is pressed, FALSE if the OK button is pressed, and ERROR if an error occurred while the dialog box was being created.

See Also mdlDialog_fileOpenExt, mdlDialog_fileOpen, mdlDialog_fileCreate. See mdlResource_createFile and mdlResource_openFile regarding the creation and opening of resource files.

mdlFileList_fromString

```
int mdlFileList_fromString
(
    StringList **fileListPP, /* <= StringList /w file names */
    char *fileNameP /* => file name string */
);
```

Description mdlFileList_fromString searches the specified directories for files matching the criteria in *fileNameP*. Directories are specified as paths preceding the filenames. It creates a string list in *fileListPP* that contains a list of files located from the string *fileNameP*. The individual file names can be extracted using mdlStringList_getMember.

fileListPP is a pointer to a pointer to a string list. On successful return, the string list will contain the full file names of all the files that exist that satisfy the file name criteria in *fileNameP*. mdlFileList_fromString allocates the memory necessary to hold the string list.

fileNameP is a list of one or more path names (separated by semicolons), each of which may contain wildcard characters.



MDL programs must free the memory allocated by mdlFileList_fromString by calling mdlStringList_destroy.

Returns mdlFileList_fromString returns the number of file names contained in *fileList*.

See Also mdlFileList_edit.

Text File Functions

The text file functions provide a platform independent method of reading and writing ASCII text files. The functions are similar to the standard C runtime libraries fopen, fclose, fgets and fputs with the exception that the text file functions automatically handle all types of line termination (CR, LF, CR/LF) regardless of the platform or operating system in use.

The following table lists the text file functions:

Function	Used to
mdlTextFile_open	open an ASCII text file.
mdlTextFile_close	close an ASCII text file.
mdlTextFile_getString	read one string from an ASCII text file.
mdlTextFile_putString	write one string to an ASCII text file.

mdlTextFile_open

```
#include <mdl.h>
#include <msfile.fdf>

FILE *mdlTextFile_open
(
char    *fileNameP,    /* => Name of file to open */
int     openMode       /* => File open mode */
);
```

Description The mdlTextFile_open function opens the text file specified by *fileNameP* for use with the mdlTextFile_getString and mdlTextFile_putString functions. The second argument, *openMode*, specifies the type of access requested. The options for *openMode* are defined as follows:

<i>openMode</i> value	Meaning
TEXTFILE_READ	Open the file for read access only. If the file does not exist or cannot be opened the call will fail.
TEXTFILE_WRITE	Open the file for write access. If the specified file does not exist it will be created. If the file does exist it will be truncated.
TEXTFILE_APPEND	Opens the file for writing at the end of the file. If the file does not exist it will be created.



Although this function performs essentially the same operation as the standard C run-time library function fopen, the pointer returned by mdlTextFile_open is not guaranteed to be suitable for use with fputs or fgets.

Returns The mdlTextFile_open function returns a pointer to the newly opened file or NULL if the requested operation was not successful.

See Also mdlTextFile_close, mdlTextFile_getString, mdlTextFile_putString, fopen.

mdlTextFile_close

```
#include <mdl.h>
#include <msfile.fdf>

int mdlTextFile_close
(
    FILE    *fileP    /* => FILE previously opened by mdlTextFile_open */
);
```

Description The mdlTextFile_close function closes a file that was opened by mdlTextFile_open. *fileP* is pointer to the file previously opened by mdlTextFile_open.

Returns The mdlTextFile_close function returns SUCCESS if the operation was successful or EOF to indicate an error.

See Also mdlTextFile_open, mdlTextFile_getString, mdlTextFile_putString, fclose.

mdlTextFile_getString

```
#include <mdl.h>
#include <msfile.fdf>

char *mdlTextFile_getString
(
    char    *stringP,    /* <= Output string */
    int     maxlen,      /* => Size of output buffer */
    FILE    *fileP,      /* => File to read from */
    int     option        /* => Newline option */
);
```

Description The mdlTextFile_getString function operates similar to the standard C run-time library function fgets but it provides more flexible handling of line termination. Characters are read from the input stream, *fileP*, and copied to *stringP* until:

1. Any type of newline is encountered (CR, LF, CR/LF).
2. End-of-file is reached, or:
3. *maxLen*-1 characters have been read without encountering end-of-file or a newline.

A NULL character is then appended to the string.

fileP must be pointer to an ASCII text file open by a previous call to mdlTextFile_open.

If the input is terminated because of a newline and the value of *option* is set to TEXTFILE_KEEP_NEWLINE a newline character '\n' is appended to the character string, *stringP*, directly preceding the terminating NULL character.

If the value of *option* is set to TEXTFILE_DEFAULT (zero) the newline character is not appended to *stringP*.

Returns `mdlTextFile_getString` returns the value of *stringP* upon successful completion. If end-of-file is encountered before any characters have been read then a `NULL` pointer is returned and the content of the buffer pointed to by *stringP* is unchanged.

See Also `mdlTextFile_open`, `mdlTextFile_close`, `mdlTextFile_putString`, `fgets`.

mdlTextFile_putString

```
#include <mdl.h>
#include <msfile.fdf>

int mdlTextFile_putString
(
  char    *stringP,      /* => String to append to file */
  FILE    *fileP,        /* => FILE to write to */
  int     option          /* => Newline option */
);
```

Description The `mdlTextFile_putString` function operates similar to the standard C run-time library function `fputs`. All characters except the terminating `NULL` from the input string *stringP* are written to the output stream *fileP*.

fileP must be pointer to an ASCII text file open by a previous call to `mdlTextFile_open`.

If the value of *option* is set to `TEXTFILE_DEFAULT` (zero) then the appropriate newline character(s) (CR, LF, CR/LF) for the current platform and operating system will also be appended to the output stream.

If the value of *option* is set to `TEXTFILE_NO_NEWLINE`, no terminating newline character(s) will be written to the output stream.

Returns If the requested operation is successful, `mdlTextFile_putString` returns a non-negative value. If an error occurs, `mdlTextFile_putString` returns `EOF`.

See Also `mdlTextFile_open`, `mdlTextFile_close`, `mdlTextFile_getString`, `fputs`.

Low Level I/O Support

Support for the standard low-level file functions such as `open`, `close` and `creat` is introduced in MicroStation 5.5. These functions are not part of the ANSI C specification but they are available in most C runtime libraries, so they have been added to the MDL runtime library. Since these functions are not part of the ANSI C specification, there is some variation from system to system. The MDL implementation of these functions reflects these system dependencies. For maximum portability, use the stream I/O functions such as `fopen` and `fclose`.

Whenever possible, MicroStation just uses the underlying native implementation of these functions. The only exceptions are for `open`, `creat`, `sopen` and `close`. Intercepting these functions allows MicroStation to track the file handles on an application-by-application basis. If an application opens a file and then terminates without closing the file, MicroStation closes the file. This is the only special processing performed by MicroStation. Otherwise, these functions are as provided in the C runtime library that is linked with MicroStation.

These functions are all implemented in the library `mdllib.ml`. There are built-in functions with similar names but with the prefix `_`. For example, the function `read` is implemented in the library `mdllib.ml`. It calls the built-in function `_read`. This implementation scheme allows you to implement `read` as you like, instead of using the `read` provided with MicroStation. This is particularly important for this group of functions, since some programmers implemented their own versions of these functions before they were added to MicroStation.

The following table lists low level I/O support functions:

Function	Used to
<code>chmod</code>	change the permission setting of the specified file.
<code>close</code>	close the specified file.
<code>creat</code>	create the specified file or truncates it if it already exists.
<code>eof</code>	to determine if the specified file is positioned at the end-of-file.
<code>lockf</code>	lock a specified portion of a file, available on all Unix platforms supported by MicroStation.
<code>locking</code>	lock a specified portion of the file, available on all Windows platforms, DOS and OS/2.
<code>lseek</code>	reposition the current file position.
<code>open</code>	open the specified file.
<code>read</code>	read from the specified file.
<code>sopen</code>	open the specified file and set the share mode.
<code>tell</code>	return the current file position.
<code>write</code>	write to the specified file.

Interaction with Dynamic Link Modules

The file handles used by these functions are the actual file handles returned by the C library's `open` and `close`, and the same handles passed as arguments to the C library's `read` and `write`. Consequently, it is possible to share these handles among MDL applications and Dynamic Link Modules.

If a DLM calls `open`, `creat`, `close` or `sopen` it calls the real function rather than the MDL equivalent. This allows a DLM to open files that are not associated with MDL applications. This is essential if the DLM needs to keep the files open when the MDL application terminates. The DLM can also call the MDL equivalents if it needs to force the file to be associated with the application. To call the MDL equivalents, the MDL application uses the names `d1mSystem_md1open`, `d1mSystem_md1Close`, `d1mSystem_md1Sopen` and `d1mSystem_md1Creat`. These are the functions that MicroStation calls to handle MDL's `open`, `close`, `sopen` and `creat`.

Files opened with the `d1mSystem_md1...` functions must also be closed via `d1mSystem_md1Close`. Files opened with the real `open`, `sopen` and `creat` must be closed with the real `close`.

Record Locking

Record locking is extremely platform-specific. MicroStation provides access to the locking function on the Windows platforms, DOS and OS/2. It provides access to `lockf` on the Unix platforms. No record locking is provided on Macintosh platforms.

chmod

```
#include <osio.h>

int chmod
(
    char    *filename,
    int     pmode
);
```

Description The `chmod` function changes the permission setting of the file specified by *filename*.

The *filename* parameter specifies the file to change.

The *pmode* parameter applies only when the file is created. It specifies the file's permissions.

For DOS, Windows platforms and OS/2, the meaning of *pmode* is:

<i>pmode</i>	meaning
<code>S_IREAD</code>	reading is permitted.
<code>S_IWRITE</code>	writing is permitted.
<code>S_IWRITE S_IREAD</code>	reading and writing are permitted.

For Unix platforms, *pmode* is formed by using the bitwise OR operator to combine values from the following list:

<i>pmode</i>	meaning
S_ENFMT	record locking enforced.
S_IRUSR	read by owner.
S_IWUSR	write by owner.
S_IXUSR	execute (search) by owner.
S_IRGRP	read by group.
S_IWGRP	write by group.
S_IXGRP	execute (search) by group.
S_IROTH	read by others (that is, anybody else).
S_IWOTH	write by others.
S_IXOTH	execute (search) by others.

chmod returns 0 if it successfully changed the file permissions, -1 otherwise.

See Also sopen, open, creat.

close

```
#include <osio.h>

int close
(
    int    handle
);
```

Description The close function closes the file specified via *handle*.

The *handle* parameter is a value returned by sopen, open or creat.

Returns close returns 0 if it successfully closes the file. Otherwise, it returns -1.

creat

```
#include <osio.h>

int creat
(
    char    *filename,
    int     pmode    /* Ignored and optional unless O_CREAT used */
);
```

Description The `creat` function creates the file specified by *filename*, or truncates it if it already exists.

The *filename* parameter specifies the name of the file to open.

The *pmode* parameter applies only when the file is created. It specifies the file's permissions. These permissions are saved with the file, or with the file's directory entry.

For DOS, Windows platforms and OS/2, the meaning of *pmode* is:

<i>pmode</i>	meaning
S_IREAD	reading is permitted.
S_IWRITE	writing is permitted.
S_IWRITE S_IREAD	reading and writing are permitted.

For Unix platforms, *pmode* is formed by using the bitwise OR operator to combine values from the following list:

<i>pmode</i>	meaning
S_ENFMT	record locking enforced.
S_IRUSR	read by owner.
S_IWUSR	write by owner.
S_IXUSR	execute (search) by owner.
S_IRGRP	read by group.
S_IWGRP	write by group.
S_IXGRP	execute (search) by group.
S_IROTH	read by others (that is, anybody else).
S_IWOTH	write by others.
S_IXOTH	execute (search) by others.

Returns `creat` returns a file handle if it successfully opened the file. Otherwise, it returns -1.

See Also `open`, `sopen`.

eof

```
#include <osio.h>

int eof
(
    int    handle
);
```


Description The `eof` function can be used to determine if the specified file is positioned at the end-of-file. `eof` is only available on Windows platforms, DOS and OS/2.

The *handle* parameter specifies the file.

Returns `eof` returns 1 if the file is at the end, 0 if it is not, and -1 if an error occurs.

lockf

```
#include <osio.h>

int lockf
(
    int     handle,
    int     function,
    long    size
);
```

Description The `lockf` function locks a portion of the file specified by the parameter *handle*. `lockf` is available on all Unix platforms supported by MicroStation. It is not available on other MicroStation platforms.

The `lockf` function supports both advisory-mode record locking and enforced-mode record locking. Advisory-mode means that one lock prevents other locks but does not prevent reads and writes of the file. An enforced-mode lock prevents other locks, reads, and writes.

Locking a file with `lockf` does not lock out other MDL applications. It only locks out processes other than MicroStation.

Closing a file releases the locks, even if the process (MicroStation) has other opens of the file.

The *handle* parameter specifies an open file. The file must have been opened with write permission. That is, it must have been opened as `O_WRONLY` or `O_RDWR`.

The *function* parameter specifies one of the following values:

<i>function</i> value	meaning
<code>F_ULOCK</code>	removes locks from a region of the file.
<code>F_LOCK</code>	lock a region if it is available. Otherwise, make MicroStation sleep until it is available.
<code>F_TLOCK</code>	lock a region if it is available. Otherwise, return -1.
<code>F_TEST</code>	used to test if another process has the region locked. <code>lockf</code> returns 0 if the region is accessible, -1 otherwise.

The *size* parameter specifies the number of bytes to lock. It may be positive or negative. If it is negative, it includes the number of bytes up to but not including the current file position. It is possible to lock beyond the end of file. Locks may overlap.

Regular files with the mode of `S_ENFMT` will have the enforcement mode enabled. Only advisory locking is available for NFS file systems.

To use this reliably, the file should only be locked for very brief periods. The application should be designed to only rely on advisory-mode.

Returns `lockf` returns 0 upon successful completion, 0 otherwise.

See Also `locking`, `sopen`, `open`, `creat`.

locking

```
#include <osio.h>

int locking
(
    int    handle,
    int    function,
    long   size
);
```

Description The `locking` function locks a portion of the file specified by the parameter *handle*. `locking` is available on all Windows platforms, DOS and OS/2. It is not available on other MicroStation platforms.

Locking bytes in a file prevents access to those bytes by another process.

The *handle* parameter specifies the file to lock. *Handle* must be a value returned by `sopen`, `open` or `creat`. The file must still be open.

The *function* parameter must be one of the following values:

function value	meaning
<code>LK_LOCK</code>	lock <i>size</i> bytes starting at the current file position. A lock may extend beyond the end of the file. If the bytes cannot be locked, <code>locking</code> tries again after 1 second. It retries up to 10 times if necessary, <code>LK_NBLCK</code> - lock <i>size</i> bytes. If unable to lock, <code>locking</code> returns immediately.
<code>LK_NBLCK</code>	same as <code>LK_NBLCK</code> .
<code>LK_RLCK</code>	same as <code>LK_LOCK</code> .
<code>LK_UNLCK</code>	unlocks the specified blocks. It is an error to try to unlock bytes which were not previously locked.

The *size* parameter specifies the number of bytes to lock. It must be a positive value.

Multiple regions can be locked, but the regions cannot overlap. Regions should be locked as briefly as possible, and should be unlocked before closing the file or exiting the application.

Returns `locking` returns 0 if successful, -1 otherwise.

See Also lockf, sopen, open, creat.

lseek

```
#include <osio.h>

long lseek
(
    int    handle,
    long    offset,
    int    origin
);
```

Description The lseek function repositions the current file position. This determines where the next read or write will occur.

The *handle* parameter specifies the file.

The *offset* parameter specifies a number of bytes relative to some origin.

The *origin* parameter determines how *offset* is interpreted. It must be one of the following values:

<i>origin</i> value	meaning
SEEK_SET	interpret <i>offset</i> as an absolute value.
SEEK_CUR	interpret <i>offset</i> as an offset from the current file position.
SEEK_END	interpret <i>offset</i> as an offset from the end of the file.

Returns lseek returns the new absolute file position. If an error occurred, it returns -1.

See Also tell, read, write.

open

```
#include <osio.h>

int open
(
    char    *filename,
    int    oflg,      /* File open information. */
    int    pmode      /* Ignored and optional unless O_CREAT used. */
);
```

Description The open function opens the file specified by *filename*.

The *filename* parameter specifies the name of the file to open.

The *oflg* parameter controls how the file is opened. This value is created using the bitwise OR operator (|) to combine values from the following

list. Since this parameter is passed on to the C runtime's `open`, it is possible to add additional platform-specific flags.

<i>oflg</i> value	meaning
<code>O_RDONLY</code>	open file for read only.
<code>O_WRONLY</code>	open file for write only.
<code>O_RDWR</code>	open file for read and write.
<code>O_APPEND</code>	all writes are appended to the end of the file.
<code>O_CREAT</code>	creates and opens new file for writing. If the file exists, open succeeds regardless of whether <code>O_CREAT</code> is specified. If the file does not exist, the open fails unless <code>O_CREAT</code> is specified. <code>O_CREAT</code> does not cause open to truncate an existing file.
<code>O_TRUNC</code>	opens file and truncates length to zero. <code>O_TRUNC O_CREAT</code> causes open to open a zero-length file regardless of whether it already exists.
<code>O_EXCL</code>	causes open to fail if the file already exists. <code>O_EXCL</code> is meaningless unless it is used with <code>O_CREAT</code> .
<code>O_TEXT</code>	opens the file in text mode.
<code>O_BINARY</code>	opens the binary mode.

The *pmode* parameter applies only when the file is created. It specifies the file's permissions. These permissions are saved with the file, or with the file's directory entry.

For DOS, Windows platforms and OS/2, the meaning of *pmode* is:

<i>oflg</i> value	meaning
<code>S_IREAD</code>	reading is permitted.
<code>S_IWRITE</code>	writing is permitted.
<code>S_IWRITE S_IREAD</code>	reading and writing are permitted.

For Unix platforms, *pmode* is formed by using the bitwise OR operator to combine values from the following list:

<i>pmode</i>	meaning
<code>S_ENFMT</code>	record locking enforced.
<code>S_IRUSR</code>	read by owner.
<code>S_IWUSR</code>	write by owner.
<code>S_IXUSR</code>	execute (search) by owner.
<code>S_IRGRP</code>	read by group.
<code>S_IWGRP</code>	write by group.

<i>pmode</i>	meaning
S_IXGRP	execute (search) by group.
S_IROTH	read by others (that is, anybody else).
S_IWOTH	write by others.
S_IXOTH	execute (search) by others.

Returns open returns a file handle if it successfully opened the file. Otherwise, it returns -1.

See Also creat, sopen repositions the current file position.

read

```
#include <osio.h>

int read
(
    int          handle,    /* File handle */
    void          *buffer,
    unsigned int count
);
```

Description The read function reads from the file specified by *handle*.

The *handle* parameter designates the file. *handle* is a value returned by open, sopen or creat.

The *buffer* parameter points to the location to receive the data.

The *count* parameter specifies the size of the buffer.

Returns read returns the count of bytes moved into the buffer. If the file is opened in text mode on a system where the newline representation is CR LF or LF CR, then this count may be less than the number of bytes advanced in the file. That is true because the newline representation in the buffer occupies just 1 byte. If read tries to read from the end of the file, it returns 0. If an error occurs, read returns -1.

See Also write, open, creat opens the file specified.

sopen

```
#include <osio.h>

int sopen
(
    char          *filename,
    int           oflg,      /* File open information. */
    int           shflg,     /* Sharing information. */
    int           pmode     /* Ignored and optional unless O_CREAT used */
);
```

Description The `sopen` function opens the file specified by *filename* and sets the share mode. `sopen` is only available on Windows platforms, DOS and OS/2.

The *filename* parameter specifies the name of the file to open.

The *oflg* parameter controls how the file is opened. This value is created using the bitwise OR operator (`|`) to combine values from the following list. Since this parameter is passed on to the C runtime's `sopen`, it is possible to add additional platform-specific flags.

<i>pmode</i>	meaning
<code>O_RDONLY</code>	open file for read only.
<code>O_WRONLY</code>	open file for write only.
<code>O_RDWR</code>	open file for read and write.
<code>O_APPEND</code>	all writes are appended to the end of the file.
<code>O_CREAT</code>	creates and opens new file for writing. If the file exists, <code>sopen</code> succeeds regardless of whether <code>O_CREAT</code> is specified. If the file does not exist, the <code>sopen</code> fails unless <code>O_CREAT</code> is specified. <code>O_CREAT</code> does not cause <code>sopen</code> to truncate an existing file.
<code>O_TRUNC</code>	opens file and truncates length to zero. <code>O_TRUNC O_CREAT</code> causes <code>sopen</code> to open a zero-length file regardless of whether it already exists.
<code>O_EXCL</code>	causes <code>sopen</code> to fail if the file already exists. <code>O_EXCL</code> is meaningless unless it is used with <code>O_CREAT</code> .
<code>O_TEXT</code>	opens the file in text mode.
<code>O_BINARY</code>	opens the binary mode.

The *shflg* parameter specifies the share mode to be used when opening the file. It controls whether read or write access is available to other attempts to open the same file. One of these values must be specified. They cannot be combined with the OR operator. The possible values are:

<i>oflg</i> value	meaning
<code>SH_DENYNO</code>	deny nothing. Permit other opens to have read and write access.
<code>SH_DENYRD</code>	deny read access to other opens.
<code>SH_DENYWR</code>	deny write access to other opens.
<code>SH_DENYRW</code>	deny read and write access to other opens.

The *pmode* parameter applies only when the file is created. It specifies the file's permissions. These permissions are saved with the file, or with the file's directory entry.

For DOS, Windows platforms and OS/2, the meaning of *pmode* is:

<i>pmode</i>	meaning
S_IREAD	reading is permitted.
S_IWRITE	writing is permitted.
S_IWRITE S_IREAD	reading and writing are permitted.

For Unix platforms, *pmode* is formed by using the bitwise OR operator to combine values from the following list:

<i>pmode</i>	meaning
S_ENFMT	record locking enforced.
S_IRUSR	read by owner.
S_IWUSR	write by owner.
S_IXUSR	execute (search) by owner.
S_IRGRP	read by group.
S_IWGRP	write by group.
S_IXGRP	execute (search) by group.
S_IROTH	read by others (that is, anybody else).
S_IWOTH	write by others.
S_IXOTH	execute (search) by others.

Returns `sopen` returns a file handle if it successfully opened the file. Otherwise, it returns -1.

See Also `creat`, `open`.

tell

```
#include <osio.h>

long tell
(
    int    handle
);
```

Description The `tell` function returns the current file position of the file associated with *handle*.

The *handle* parameter specifies the file.

Returns `tell` returns -1 if an error occurred, otherwise it returns the current file position.

See Also `lseek`, `read`, `write`.

write

```
#include <osio.h>

int writes
(
    int          handle,    /* File handle */
    void          *buffer,
    unsigned int count
);
```

Description The write function writes to the file specified by *handle*.

The *handle* parameter designates the file. *handle* is a value returned by open, sopen or creat.

The *buffer* parameter points to the data.

The *count* parameter specifies the number of bytes to write.

Returns write returns the count of bytes copied from the buffer. If the file is opened in text mode on a system where the newline representation is CR LF or LF CR, then this count may be less than the number of bytes written to the file. That is true because the newline representation in the buffer occupies just 1 byte. If an error occurs, write returns -1.

See Also read, open, creat.

Constraint Problem Solving

The constraint functions described in this chapter are used to set up and solve constraint problems. They form part of the “dimension-driven design” functionality of MicroStation (currently comprised of “dimension-driven cell creation” and “dimension-driven cell placement”).

This chapter contains the following sections:

- Construction frame functions
- Constraint parameter functions
- Constraint functions
- Equation constraint functions
- Attachment functions
- Constraint model functions
- Constraint object functions
- Solver variable functions

Constraints represent required relationships among geometric entities (and/or algebraic variables). The system solves a constraint problem by modifying the target geometry (or variables) so that all constraints are satisfied.

Constraints are **two-way** relationships, not calculation formulas. There is no concept of ordering or dominance among constraints. All must be satisfied simultaneously, except for weak constraints and post-checks. The task of ordering constraints is left to the solver. Likewise, there is no problem with circular references and re-ordering constraints in the problem will not affect the result.

Constraints do not apply directly to MicroStation elements. That is, constraints and the entities to which they apply are dynamic objects which are created and destroyed by the application in memory. They are not connected in any way to MicroStation elements and cannot be stored in the .dgn file. Of course, the application can use the information which is stored in these entities to create or modify MicroStation elements, as needed.

Components

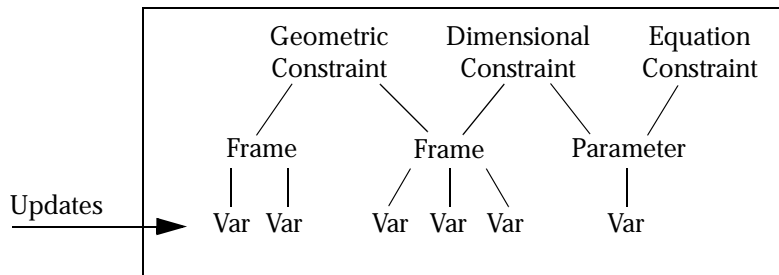
A constraint is represented by a **constraint** data structure, defined in terms of one or more construction frames or parameters. Pure geometric constraints represent geometric relationships, such as parallelism or tangency, and apply strictly to construction frames. Dimensional Constraints represent measurable geometric attributes or relationships, such as drafting dimensions. They apply to geometric entities and may be linked via “assignment” to the value of a parameter.

An **equation constraint** (ConsEquation) represents an algebraic equation and applies strictly to parameters.

The geometric attributes of a frame, the value of a parameter, and the value of a dimensional constraint are all represented by **solver variables** (Var). The application gets access to the attributes of frames, parameters, and constraints by getting and manipulating the solver variables which represent them.

A **constraint model** (ConsModel) data structure represents the constraint problem to be solved. There are functions to create a model, add and drop objects, analyze the problem, and solve it. The model directly modifies the frames' and parameters' Solver Variables with the results of analysis and solving.

Constraints, construction frames, and parameters may be added to and dropped from a Model at any time. The Model should be re-solved whenever it is modified. Many Models may be created and solved independently, and Constraints and their targets may belong to different models at the same time, although it becomes the application's responsibility to synchronize them. A Model may contain many disconnected graphs.



Constraint Model Data Structure and Relationships

Defining and Solving a Constraint Problem

To define and solve a constraint problem:

1. Create a Constraint Model.
2. Create Construction Frames to represent the geometry. Add to the Model.
3. Create Construction Parameters to represent algebraic unknowns and constants. Add to the Model.
4. Create Constraints which capture the required geometric relationships, dimensions, etc. Add to the Model.
5. Create Constraints which link Parameters to dimensional Constraints. Add to the Model.
6. Create Equation Constraints which capture the required algebraic constraints. Add to the Model.
7. Solve the Model and check the return status.
8. Query the `Vars` of the Frames and Parameters for the results, if a solution was reached.
9. Use the results.
10. Destroy the frames, parameters, and constraints. There is a function to free all objects in the model automatically. Destroy the constraint model.

Analyzing the Constraint Problem

To analyze the information content of a constraint problem, you must “validate” the Constraint Model. The Model updates the status flags of the Frames, Parameters and Constraints with the results. You then query these objects for details. For example, to detect under-determined geometry, call the `mdlConsMod_validate` function and then check each Frame with the `mdlConsFrame_isUnderDetermined` function. To check if a constraint is redundant, use the `mdlConstraint_rGroup` function to query its **redundancy group**. If the group number is non-zero, the constraint conflicts with all other constraints assigned to that group.



You must solve the model before validating. Validating a model which contains many redundant constraints can be tricky and is best avoided. It is not always possible to tell which or how many constraints in a redundancy group should be removed in order to reconcile a multi-way conflict. Moreover, if a constraint is redundant with one set of constraints in one way and with another set in another way, validation may not see all of the conflicts, and the results may be difficult to interpret. It is therefore recommended that you build up and analyze a constraint scheme in relatively small steps.

Units and the Current Transform



Construction Frame geometry and dimensional constraint values must be in Master Units. Therefore, if your application is going to be setting up constraint problems using geometric data extracted from MicroStation elements you *must* set up the current transform, as follows:

```
mdlCurrTrans_begin();
mdlCurrTrans_masterUnitsIdentity(TRUE);
```

It is also recommended that you set the origin of the current transform to a point which is somewhere near the elements that you will be working with.



Also, remember that constraint problems are planar. So, when you are working in 3D you can only work with elements which lie in a common plane, and you must set the origin and rotation of the current transform to match that plane.

Constraints Example

The tangcons MDL example illustrates a fundamental use of constraints.

Construction Frame Functions

Construction Frame functions are used to define and query the data structures that represent construction points, lines, circles, ellipses and B-splines in constraint-solving problems.

Construction Frames represent planar geometry, even in 3D, and are defined (implicitly) in the current coordinate system and plane.

A Construction Frame's geometric attributes are represented internally by a set of solver variables. For example, a construction point has two internal variables controlling its x and y-coordinate attributes. Attributes have symbolic names. For example, the attributes of a point are called `CONS_POINT_X` and `CONS_POINT_Y`. The solver variables which represent a frame's attributes may be accessed by these symbolic names. See `mscons.h`.

Geometric constraints apply to frames, and the constraint solver updates the frames' attribute variables in the course of satisfying such constraints.



Construction frames contain dynamically allocated memory. Call `mdlCons_destroy` to free this memory when done with an object, or call `mdlConsMod_destroyNodes` to destroy and free all objects in a constraint model.

The following table lists Construction Frame functions:

Function	Used to
mdlConsFrame_createPoint	define a construction point.
mdlConsFrame_createLine	define a construction line.
mdlConsFrame_createEllipse	define a construction ellipse.
mdlConsFrame_createCircle	define a construction circle.
mdlConsFrame_createBspline	define a B-spline construction frame.
mdlConsFrame_createBsplineCell	define a B-spline cell construction frame.
mdlConsFrame_isUnderDetermined	query if any of the frame's geometric attributes are not fully determined by constraints.

mdlConsFrame_createPoint

```
#include <mscons.h>

ConsFrame *mdlConsFrame_createPoint
(
    ConsFrame    *this,      /* <= construction point */
    DPoint3d     *p          /* => initial location */
);
```

Description mdlConsFrame_createPoint defines a construction frame point whose attributes are CONS_POINT_X and CONS_POINT_Y.

this is the storage to fill out.

p is the point's initial location.

Returns mdlConsFrame_createPoint returns *this*.

See Also mdlConsFrame_createLine.

mdlConsFrame_createLine

```
#include <mscons.h>

ConsFrame *mdlConsFrame_createLine
(
    ConsFrame    *this,      /* <= construction line*/
    DPoint3d     *p1,        /* => initial location of a point on line */
    DPoint3d     *p2         /* => initial location of a point on line */
);
```

Description mdlConsFrame_createLine defines a construction line. A construction frame line represents an infinite line. The two input point parameters simply specify its initial slope and intercept, not its length or extent. A construction line has a direction,

defined as the direction from the first input point parameter toward the second point. Line direction is important for defining constraints.

A construction frame line's attributes are `CONS_LINE_DIRECTION`, it's angle (in radians), and `CONS_LINE_DO`, the perpendicular distance from the current coordinate system origin to the line.

this is the storage to fill out.

p1 and *p2* are two points which define the line's initial geometry.

Returns `mdlConsFrame_createLine` returns *this*.

See Also `mdlConsFrame_createPoint`.

mdlConsFrame_createEllipse

```
#include <mscons.h>

ConsFrame *mdlConsFrame_createEllipse
(
    ConsFrame    *this,      /* <= construction ellipse */
    DPoint3d     *p,         /* => initial location of center */
    double       r,          /* => initial value of primary radius */
    double       r2,         /* => initial value of secondary radius */
    double       zr          /* => initial value of z-axis rotation */
);
```

Description `mdlConsFrame_createEllipse` defines a construction ellipse.

A construction ellipse has attributes `CONS_ELLIPSE_X`, the x-coordinate of its center, `CONS_ELLIPSE_Y`, the y-coordinate of its center, `CONS_ELLIPSE_R`, its major radius, `CONS_ELLIPSE_R2`, its minor radius, and `CONS_ELLIPSE_ANGLE`, the rotation angle (about the current z axis) of its major axis.

this is the storage to fill out.

p is the initial x, y location of the center of the ellipse.

r and *r2* are the initial major and minor radii of the ellipse.

zr is the initial rotation angle (about the z axis) of the ellipse, in radians, measured up from the x-axis.

Returns `mdlConsFrame_createEllipse` returns the construction ellipse in *this*.

See Also `mdlConsFrame_createCircle`.

mdlConsFrame_createCircle

```
#include <mscons.h>

ConsFrame *mdlConsFrame_createCircle
(
    ConsFrame    *this,      /* <= construction circle */
    DPoint3d     *p,         /* => initial location of center */
    double       r,          /* => initial value of radius */
);
```

Description mdlConsFrame_createCircle defines a construction circle.

A construction circle has attributes CONS_ELLIPSE_X, the x-coordinate of it's center, CONS_ELLIPSE_Y, the y-coordinate of it's center, CONS_ELLIPSE_R, its radius. (CONS_ELLIPSE_R2 and CONS_ELLIPSE_ANGLE are also defined, so that a circle is truly a special case of an ellipse).

this is the storage to fill out.

p is the initial x, y location of the center of the circle.

r is the initial value for the radius of the circle.

Returns mdlConsFrame_createCircle returns the construction circle in *this*.

See Also mdlConsFrame_createEllipse.

mdlConsFrame_createBspline

```
#include <mscons.h>
#include <mdlbspln.h>

ConsFrame *mdlConsFrame_createBspline
(
    ConsFrame      *this,      /* <= B-spline construction frame def'n.*/
    MSBsplineCurve *pCurve     /* => B-spline curve geometry */
);
```

Description The mdlConsFrame_createBspline function defines a B-spline Construction Frame object for use in constraint solving. The B-spline's geometry, defined by *pCurve*, is assumed to be invariant, so this frame has no solver attribute variables at all. You may, nevertheless, change the poles of the B-spline curve on the fly.



The *pCurve* pointer is stored in the B-spline frame object and must not be freed.

this is the storage to fill out with the B-spline Construction Frame definition data.

pCurve should point to a fully defined B-spline curve definition data structure, such as would be returned by mdlBspline_convertToCurve.

`mdlCons_destroy` will free the curve when you destroy the B-spline frame object.

Returns `mdlConsFrame_createBspline` returns the construction B-spline in *this*.

See Also `mdlConsFrame_createBsplineCell`.

mdlConsFrame_createBsplineCell

```
#include <mscons.h>

ConsFrame *mdlConsFrame_createBsplineCell
(
    ConsFrame      *this,      /* <= B-spline cell constr. frame def'n.*/
    MSBsplineCurve *pCurve,    /* => B-spline curve geometry */
    Dpoint3d       *pCenter,    /* => initial coords of cell center */
    double         angle       /* => initial rotation angle of cell */
);
```

Description The `mdlConsFrame_createBsplineCell` function defines a B-spline cell Construction Frame object for use in constraint solving. The B-spline's intrinsic geometry is assumed to be fixed (although you may change poles of the B-spline curve on the fly), but the B-spline as a whole can translate and rotate. A B-spline cell would be useful for modeling a cam, for example.

B-spline cell attribute variables are `CONS_BSPLINE_X`, `CONS_BSPLINE_Y`, and `CONS_BSPLINE_ANGLE`.



The `pCurve` pointer is stored in the B-spline cell frame object and must not be freed

this is the storage to fill out with the B-spline cell Construction Frame definition data.

pCurve should point to a fully defined B-spline curve definition data structure, such as would be returned by `mdlBspline_convertToCurve`.

pCenter is the initial location of the B-spline cell construction frame object.

angle is the initial (planar) rotation of the B-spline cell construction frame object, in radians.

Returns `mdlConsFrame_createBsplineCell` returns the construction B-spline Cell in *this*.

See Also `mdlConsFrame_createBspline`.

mdlConsFrame_isUnderDetermined

```
#include <mscons.h>

boolean mdlConsFrame_isUnderDetermined
(
  ConsFrame    *this      /* => the frame object to query */
);
```

Returns mdlConsFrame_isUnderDetermined queries if any of the frame's geometric attributes are not fully determined by constraints. If any attribute is under-determined, the frame as a whole is under-determined.

this is the frame object to query.

Returns mdlConsFrame_isUnderDetermined returns TRUE if any of the frame's attributes is under-determined; FALSE if its geometry well-determined.

See Also mdlCons_getVar, mdlVar_isUnderDetermined.

Constraint Parameter Functions

Constraint Parameter functions are used to define and modify the data structure which represents named variables and constants in constraint-solving problems. A Constraint Parameter has a symbolic name, a value and units.

Algebraic constraints apply to parameters, and the constraint solver updates the parameters' solver variables in the course of satisfying such constraints.

A constraint parameter may also be assigned to a dimensional constraint, becoming a synonym for the dimension value and representing it in algebraic constraints. One parameter may be assigned to any number of dimensional constraints, thereby synchronizing them all.



Constraint parameters contain dynamically allocated memory. Call mdlCons_destroy to free this memory when done with the object, or call mdlConsMod_destroyNodes to destroy and free all objects in a constraint model.

The following table lists Constraint Parameter functions:

Function	Used to
mdlConsParm_create	define a constraint parameter.
mdlConsParm_setUnits	change the units of a constraint parameter.

mdlConsParm_create

```
#include <mscons.h>

ConsParm *mdlConsParm_create
(
  ConsParm    *this,      /* <= Constraint parameter */
  char        *name,      /* => identifier */
  double      value,      /* => initial value or NA */
  int         units,      /* => units of value or 0 if undefined */
);
```

Description The `mdlConsParm_create` defines a constraint parameter.

this is the storage to fill out.

name is the constraint parameter's symbolic identifier. Algebraic constraints refer to parameters by name. A parameter name must begin with a letter and may contain up to 32 letters or numbers in total.

value is the constraint parameter's initial value, stated in the specified units. The special value NA means that the parameter's value is undefined.

units specifies how the constraint parameter's value is to be interpreted. Specify `UNITS_DEGREES` for angles in degrees, `UNITS_MU_SU_PU` for magnitudes in master units, or `UNITS_NONE` for a unit-less (pure) number. A value of zero means that the parameter's units are yet to be defined (e.g., by assignment). Units are used both for interpreting the parameter's value and for consistency checking when a parameter is assigned to a dimensional constraint or used in an algebraic equation constraint.

Returns `mdlConsParm_create` returns *this* if successful or NULL if *name* is invalid.

See Also `mdlConsParm_setUnits`.

mdlConsParm_setUnits

```
#include <mscons.h>
#include <msextok.h>

ConsParm *mdlConsParm_setUnits
(
  ConsParm    *this,      /* <=> Constraint parameter, units updated */
  int         u,          /* => new units specifier; 0 if undefined */
);
```

Description `mdlConsParm_setUnits` changes the units of the constraint parameter.

this is a Constraint parameter data structure.

u is a units specifier. See `mdlConsParm_create` for details.

Returns `mdlConsParm_setUnits` returns *this*.

See Also `mdlConsParm_create`.

Constraint Functions

Constraint functions are used to delete, modify and query the data structures that represent geometric constraints in constraint-solving problems.

Geometric constraints apply to construction frames. Since construction frames represent planar geometry, constraints operate only in the plane.

Constraints apply uniformly to related types of construction frames. Specifically, any constraint that targets a point will also target a circle, ellipse or B-spline cell, because these frames have a center and are therefore *point-like* (see definitions below). Many constraints apply to construction frames that represent curves uniformly, whether line, circle, ellipse, B-spline or B-spline cell. And, many angle constraints apply uniformly to construction frames that have a direction, including lines, ellipses (referring to the major axis) and B-spline cells. In this section, we use the following general terms to identify the classes of construction frames:

Term	Meaning
<i>point-like</i>	Point, circle, ellipse or B-spline cell
<i>line-like</i>	Line, ellipse or B-spline cell
<i>curve</i>	Line, circle, ellipse, B-spline or B-spline cell

Even though their names or descriptions may suggest a one-way dependency of one construction's geometry upon another, all constraints that relate two or more constructions define a two-way or multi-way *relationship* between or among the constructions. Which geometry is calculated relative to which is determined by the solver, considering all constraints together.

Dimensional constraints and PointLocation constraints are controlled by attribute variables, which can be accessed and modified.



Constraints contain dynamically allocated memory. Call `mdlCons_destroy` to free this memory when done with the object, or call `mdlConsMod_destroyNodes` to destroy and free all objects in a constraint model.

The following table lists the constraint functions:

Function	Used to
<code>mdlConstraint_createAssignment</code>	define a constraint which synchronizes the values of a constraint parameter and a dimensional constraint.
<code>mdlConstraint_createPointLocation</code>	define a fixed-point-location constraint.
<code>mdlConstraint_createRadius</code>	define a radius constraint.

Function	Used to
mdlConstraint_createDiameter	define a diameter constraint.
mdlConstraint_createPointOn	define a point-on-curve constraint.
mdlConstraint_createConcentric	define a concentric/coincident constraint.
mdlConstraint_createIntersection	define a point at intersection constraint.
mdlConstraint_createDistance	define a point-point distance constraint.
mdlConstraint_createPCDistance	define a point-curve distance constraint.
mdlConstraint_createLLDistance	define a line-line distance constraint.
mdlConstraint_createOffset	define a difference between values constraint.
mdlConstraint_createAngle	define an angle between lines constraint.
mdlConstraint_createAngleX	define an angle of line constraint.
mdlConstraint_createParallel	define a parallel lines constraint.
mdlConstraint_createPerpendicular	define a perpendicular lines constraint
mdlConstraint_createIOTangent	define a circle-curve tangency constraint.
mdlConstraint_createTangent	define a curve-curve tangency constraint.
mdlConstraint_createPinPoint	define a curve-through-point constraint
mdlConstraint_createLockFrame	define a frame geometry invariant constraint.
mdlConstraint_createLockVar	define a variable invariant constraint.
mdlConstraint_createMatch	define a frame=frame constraint.
mdlConstraint_rGroup	determine if a constraint is redundant.
mdlConstraint_getTarget	retrieve the target of a constraint.
mdlConstraint_flipInsideOutside	invert inside/outside sense of a constraint.
mdlConstraint_chooseSolution	choose solution to track.
mdlConstraint_setWeak	make constraint weak.
mdlConstraint_setPostCheck	make constraint post-check.

mdlConstraint_createAssignment

```
#include <mscons.h>

Constraint *mdlConstraint_createAssignment
(
    Constraint *this,    /* <= assignment constraint */
    Constraint *pTarget, /* <=> constraint to be controlled */
    ConsParm *pParm     /* => constraint param. to be assigned */
);
```

Description mdlConstraint_createAssignment defines a constraint which forces the values of a constraint parameter (ConsParm) and a dimensional constraint to be the same. A constraint parameter may be assigned to any number of constraints, thereby synchronizing them all.

The units of the constraint parameter and of the target constraint must be consistent. For example, it is an error to assign a variable whose units are UNITS_DEGREES to a distance dimension (whose units are UNIT_MU_SU_PU).



Creating this constraint has the immediate side-effect of setting the units of the constraint parameter to match the units of the targeted constraint, if the constraint parameter's units are undefined. Likewise, if the parameter's value is undefined (NA), it is set to the dimension's value.

This constraint stores the assigned parameter's attribute variable and the target dimensional constraint's attribute variable, which can be accessed at index positions 0 and 1.

this is the storage to fill out.

pTarget is the constraint to which the constraint parameter is to be assigned.

pParm is the constraint parameter object to be assigned to the target constraint.

Returns mdlConstraint_createAssignment returns *this* if successful or NULL if either target is invalid or if the target's units are inconsistent.

See Also mdlConsParm_create, mdlConsParm_setUnits.

mdlConstraint_createPointLocation

```
#include <mscons.h>

Constraint *mdlConstraint_createPointLocation
(
    Constraint *this,      /* <= location constraint */
    ConsFrame *pPoint     /* => point-like object */
);
```

Description mdlConstraint_createPointLocation defines a constraint which fixes the x,y location of the target.

A point location constraint is not a dimensional constraint. It has two attributes, which can be accessed as CONS_POINT_X and CONS_POINT_Y, nevertheless.

The dimension's values are initialized to the current x- and y-coordinates of the point.

this is the storage to fill out.

pPoint is the point-like construction frame to be constrained.

Returns mdlConstraint_createPointLocation returns *this* if successful or NULL if the target is invalid.

See Also mdlConstraint_createAssignment.

mdlConstraint_createRadius

```
#include <mscons.h>

Constraint *mdlConstraint_createRadius
(
    Constraint *this,      /* <= radius constraint */
    ConsFrame *pEllipse, /* => ellipse (or simple circle) */
    int ri                /* => CONS_ELLIPSE_R or CONS_ELLIPSE_R2 */
);
```

Description mdlConstraint_createRadius defines a constraint which fixes the radius of a circle or ellipse.

A radius constraint is a dimensional constraint. It has one attribute, which can be accessed as attribute index 0.

The dimension value is initialized to the current magnitude of the radius.

this is the storage to fill out.

pEllipse is the circle or ellipse frame object to be constrained.

ri specifies which radius of an ellipse to constrain. CONS_ELLIPSE_R specifies the major axis and CONS_ELLIPSE_R2 specifies the minor axis. If the target is a circle, *ri* must be CONS_ELLIPSE_R.

Returns mdlConstraint_createRadius returns *this* if successful or NULL if *pEllipse* is not a circle or ellipse or *ri* is invalid.

See Also mdlConstraint_createDiameter.

mdlConstraint_createDiameter

```
#include <mscons.h>

Constraint *mdlConstraint_createDiameter
(
    Constraint *this,      /* <= diameter constraint */
    ConsFrame *pEllipse, /* => ellipse (or simple circle) */
    int ri                /* => CONS_ELLIPSE_R or CONS_ELLIPSE_R2 */
);
```

Description mdlConstraint_createDiameter defines a constraint fixing the diameter of a simple circle or an ellipse. Two such constraints may be applied to an ellipse.

this is the constraint data structure to be filled out.

pEllipse points to the ellipse which is to be constrained.

ri indicates which radius the constraint pertains to. The possible values are CONS_ELLIPSE_R to indicate primary radius, or CONS_ELLIPSE_R2 to indicate the secondary radius. Only CONS_ELLIPSE_R makes sense if the target is a simple circle.

Returns mdlConstraint_createDiameter returns NULL if the specified target is not an ellipse or simple circle; otherwise it returns *this*.

See Also mdlConstraint_createRadius.

mdlConstraint_createPointOn

```
#include <mscons.h>

Constraint *mdlConstraint_createPointOn
(
    Constraint *this,      /* <= point-on constraint */
    ConsFrame *pPoint,    /* => point-like object */
    ConsFrame *pC         /* => curve */
);
```

Description mdlConstraint_createPointOn defines a constraint which forces a point to be “on” a curve (or, equivalently, the constraint forces a curve to run “through” a point). If the target point is in fact a circle or ellipse, the constraint applies to its center. The constraint does not specify where on the second target the point must be.

this is the storage to fill out.

pPoint is the point-like construction frame to constrain.

pC is the curve construction frame to which the point is constrained. *pC* may not be a point.

Returns mdlConstraint_createPointOn returns *this* if successful or NULL if either of the target frames is invalid.

See Also mdlConstraint_createIntersection, mdlConstraint_createConcentric.

mdlConstraint_createConcentric

```
#include <mscons.h>

Constraint *mdlConstraint_createConcentric
(
    Constraint *this,      /* <= concentric constraint */
    ConsFrame *pPoint,    /* => point-like object */
    ConsFrame *pPoint2    /* => point-like object */
);
```

Description `mdlConstraint_createConcentric` defines a constraint which forces the target points to be coincident. The constraint does not specify where the targets are to be located, only that they will be in the same place.

this is the storage to be filled out.

pPoint and *pPoint2* are the construction point-like construction frames that are to be constrained. Lines are invalid.

Returns `mdlConstraint_createConcentric` returns *this* if successful or NULL if either target is invalid.

See Also `mdlConstraint_createPointOn`, `mdlConstraint_createIntersection`.

mdlConstraint_createIntersection

```
#include <mscons.h>

Constraint *mdlConstraint_createIntersection
(
    Constraint *this,      /* <= intersection constraint */
    ConsFrame *pPoint,    /* => point-like object @ intersection */
    ConsFrame *pC,        /* => curve object */
    ConsFrame *pC2        /* => curve object */
);
```

Description `mdlConstraint_createIntersection` defines a constraint forcing a point to lie at the intersection of two curve construction frames (or, equivalently, two curve construction frames to intersect at the given point). If there are two or more possible points of intersection, the point will be constrained to the intersection to which it is initially closest. The solver will then favor the chosen intersection thereafter if possible.

this is the storage to be filled out.

pPoint is the point-like construction frame to be constrained to the curves' intersection.

pC and *pC2* are the curve construction frame objects to be intersected. Points are invalid.

Returns `mdlConstraint_createIntersection` returns *this* if successful or NULL if any target is invalid.

See Also `mdlConstraint_createPointOn`, `mdlConstraint_createConcentric`.

mdlConstraint_createDistance

```
#include <mscons.h>

Constraint *mdlConstraint_createDistance
(
    Constraint *this,      /* <= point-point distance constraint */
    ConsFrame *pPoint,    /* => point-like object */
    ConsFrame *pPoint2    /* => point-like object */
);
```

Description mdlConstraint_createDistance defines a constraint fixing the “true” distance (or Euclidean norm) between two points in the plane. The constraint does not specify the relative (or absolute) locations of the target points or the direction from one to the other, only the distance between them.

A distance constraint is a dimensional constraint. It has one attribute, which can be accessed as attribute index 0.

The dimension value is initialized to the current distance between the points.

this is the storage to be filled out.

pPoint and *pPoint2* are the point-like construction frames to be constrained.

Returns mdlConstraint_createDistance returns *this* if successful or NULL if either target is invalid.

See Also mdlConstraint_createPCDistance, mdlConstraint_createLLDistance, mdlConstraint_createOffset.

mdlConstraint_createPCDistance

```
#include <mscons.h>

Constraint *mdlConstraint_createPCDistance
(
    Constraint *this,      /* <= point-point distance constraint */
    ConsFrame *pPoint,    /* => point-like object */
    ConsFrame *pCurve     /* => curve object */
);
```

Description mdlConstraint_createPCDistance defines a constraint fixing the perpendicular distance from a point to a curve in the plane. The distance measured is along an imaginary line through the point and perpendicular to the curve. The relative positions of the point and curve are significant and are preserved. (See IOTangent for a discussion of “inside” and “outside”).

A point-curve distance constraint is a dimensional constraint. It has one attribute, which can be access as attribute index 0.

The dimension value is initialized to the current distance between the point and the curve.

this is the storage to be filled out.

pPoint is the point-like construction frame from which the distance is to be measured.

pCurve is the curve construction frame to which the distance is measured along its normal.

Returns `mdlConstraint_createPCDistance` returns *this* if successful or `NULL` if either target is invalid.

See Also `mdlConstraint_createDistance`, `mdlConstraint_createLLDistance`, `mdlConstraint_createOffset`, `mdlConstraint_createPointOn`.

mdlConstraint_createLLDistance

```
#include <mscons.h>

Constraint *mdlConstraint_createLLDistance
(
    Constraint *this,      /* <= line-line distance constraint */
    ConsFrame *pLine,     /* => first line */
    ConsFrame *pLine2     /* => second line */
);
```

Description `mdlConstraint_createLLDistance` defines a constraint fixing the parallel distance from the first line to the second. The distance is measured from the first to the second, so that the constraint maintains the lines' relative positions.

The targets must be lines.



This constraint does not also constrain the lines to be parallel, but assumes that they have been so constrained.

this is the storage to be filled out.

pLine and *pLine2* are the construction lines to be constrained.

Returns `mdlConstraint_createLLDistance` returns *this* if successful or `NULL` if either target is not a line.

See Also `mdlConstraint_createParallel`, `mdlConstraint_createDistance`, `mdlConstraint_createPCDistance`, `mdlConstraint_createOffset`, `mdlConstraint_flipInsideOutside`, `mdlConstraint_chooseSolution`.

mdlConstraint_createOffset

```
#include <mscons.h>

Constraint *mdlConstraint_createOffset
(
    Constraint *this,      /* <= offset constraint */
    ConsFrame *pFrame,    /* => frame to be constrained */
    ConsFrame *pFrame2,   /* => frame to be constrained */
    int attr              /* => attribute to be constrained */
);
```

Description mdlConstraint_createOffset defines a constraint which fixes the difference between two frame attributes.

this is the storage to be filled out.

pFrame and *pFrame2* are the construction frame objects to be constrained. The frames are not required to be the same type, but the specified attribute must be defined for each of them.

attr specifies which attribute of the frames is to be constrained. *attr* may be any construction frame attribute index value.

An offset constraint is a dimensional constraint. It has one attribute, which can be access as attribute index 0.

The dimension value is initialized to the current difference between the attribute values.

Returns mdlConstraint_createOffset returns *this* if successful or NULL if either target is not a frame or if the attribute is defined for either target.

See Also mdlConstraint_createDistance, mdlConstraint_createPCDistance, mdlConstraint_createLLDistance.

mdlConstraint_createAngle

```
#include <mscons.h>

Constraint *mdlConstraint_createAngle
(
    Constraint *this,      /* <= angle constraint */
    ConsFrame *pLine,     /* => line-like object */
    ConsFrame *pLine2,    /* => line-like object */
    Dpoint3d arms[3]      /* => picture of angle to measure */
);
```

Description mdlConstraint_createAngle defines a constraint fixing the angle between two line-like construction frames.

The intersection of two infinite lines defines four angles. Fixing any one of them would be adequate to relate the lines, which is the fundamental purpose of this constraint. Nevertheless, if you want to relate the constraint

to an existing dimension or to use the fixed angle value in some particular way, you will want the constraint to apply to one angle in particular.

You identify which angle to measure by creating a “picture” of the angle dimension that would measure it. An angle dimension has two “arms,” one from *arms*[0] -> *arms*[1] and the second from *arms*[0] -> *arms*[2]. The first arm corresponds to *pLine*, and the second arm corresponds to *pLine2*. The dimension measures the angle between the two arms, from the first arm to the second arm.

An angle constraint is a dimensional constraint. It has one attribute, which can be accessed as attribute index 0.

The dimension value is initialized to the current angle between the lines.

The two targets do not have to intersect. The Parallel constraint is a better choice for maintaining parallelism. The Perpendicular constraint is better choice when perpendicularity, rather than a specific angle such as $_ / 2$, is required.

this is the storage to be filled out.

pLine and *pLine2* are the line-like construction frames to be related.

The *arms* array defines two the direction vectors which identify which of the four possible angles is to be measured by this constraint.

Returns `mdlConstraint_createAngle` returns *this* if successful or NULL if the targets are not lines or ellipses.

See Also `mdlConstraint_createAngle`, `mdlConstraint_createParallel`, `mdlConstraint_createPerpendicular`.

mdlConstraint_createAngleX

```
#include <mscons.h>

Constraint *mdlConstraint_createAngleX
(
  Constraint *this,      /* <= angle of line constraint */
  ConsFrame *pLine,     /* => line-like object */
  boolean    fromX       /* => measure from x-axis? else, from y */
);
```

Description `mdlConstraint_createAngleX` defines a constraint fixing the angle of a construction frame line or ellipse.

You may specify whether the fixed-angle value is measured from the x-axis or from the y-axis. The constraint has the same effect of holding the line's slope constant, regardless of how the angle is measured.

An *AngleX* constraint is a dimensional constraint. You may access its value at attribute index 0.

this is the storage to be filled out.

pLine is the line-like construction frame whose angle is to be constrained.

fromX specifies if the angle to be fixed is measured from the x-axis (TRUE) or from the y-axis (FALSE).

Returns mdlConstraint_createAngleX returns *this* if successful or NULL the target is not valid.

See Also mdlConstraint_createAngle, mdlConstraint_createParallel, mdlConstraint_createPerpendicular.

mdlConstraint_createParallel

```
#include <mscons.h>

Constraint *mdlConstraint_createParallel
(
    Constraint *this,      /* <= parallel constraint */
    ConsFrame *pLine,     /* => line-like object */
    ConsFrame *pLine2     /* => line-like object */
);
```

Description mdlConstraint_createParallel defines a constraint requiring two line-like construction frames to be parallel. This constraint only forces the lines to be parallel, and does not restrict the direction of lines or the position of one relative to the other.

this is the storage to be filled out.

pLine and *pLine2* are the line-like construction frames to be constrained.

Returns mdlConstraint_createParallel returns *this* if successful or NULL if either target is invalid.

See Also mdlConstraint_createPerpendicular, mdlConstraint_createAngle, mdlConstraint_createAngle, mdlConstraint_createLLDistance.

mdlConstraint_createPerpendicular

```
#include <mscons.h>

Constraint *mdlConstraint_createPerpendicular
(
    Constraint *this,      /* <= perpendicular constraint */
    ConsFrame *pLine,     /* => line-like object */
    ConsFrame *pLine2     /* => line-like object */
);
```

Description mdlConstraint_createPerpendicular defines a constraint which forces two line-like construction frames to be mutually perpendicular. This constraint only forces

the lines to define a right angle between them and does not restrict the direction of the lines and does not specify the location of the point of intersection.

this is the storage to be filled out.

pLine and *pLine2* are the line-like construction frames to be constrained.

Returns `mdlConstraint_createPerpendicular` returns *this* if successful or NULL if either target is invalid.

See Also `mdlConstraint_createParallel`, `mdlConstraint_createAngle`, `mdlConstraint_createAngleX`.

mdlConstraint_createIOTangent

```
#include <mscons.h>

Constraint *mdlConstraint_createIOTangent
(
    Constraint *this, /* <= inside-outside tangent constraint */
    ConsFrame *pCircle, /* => circle */
    ConsFrame *pCurve /* => curve object */
);
```

Description `mdlConstraint_createIOTangent` forces the target circle to be tangent to the specified curve. That is, the circle must touch the other construction at just one point, and, so, their normals must be parallel at that point.

This constraint also forces the circle is to remain “inside” or “outside” of the curve. If the curve is a circle, ellipse, or closed B-spline, then inside and outside have the obvious meanings. If the other construction is a line or open B-spline then “inside” means that the circle must lie between the curve and the local origin, while “outside” means the reverse.

The inside/outside sense of the constraint is inferred from the geometry of the circle and the curve at the time the constraint is created, but may be modified later.



Despite the sense of the constraint, the circle may move from “inside” to “outside” if its radius becomes negative. This constraint does not force the circle’s radius to be positive.

The constraint does not specify where the circle must touch the curve or how the circle and the curve are to be positioned in relation to each other.

The first target of the constraint must be a circle. The second target can be any curve, except a point.

The `IOTangent` constraint is a more specialized version of the `Tangent` constraint. `IOTangent` applies only if a circle is to be constrained, and `IOTangent` has an inside/outside sense. `IOTangent` can be more robust and more efficient than `Tangent` as a result.

this is the storage to be filled out.

pCircle is the construction frame circle to be constrained. It must be a circle, not an ellipse.

pCurve2 is the curve construction frame which *pCircle* must touch.

Returns mdlConstraint_createIOTangent returns *this* if successful or NULL if *pCircle* is not a circle or *pCurve2* is invalid.

See Also mdlConstraint_createTangent, mdlConstraint_createPCDistance, mdlConstraint_flipInsideOutside, mdlConstraint_chooseSolution.

mdlConstraint_createTangent

```
#include <mscons.h>

Constraint *mdlConstraint_createTangent
(
    Constraint    *this,      /* <= tangent constraint */
    ConsFrame     *pCurve,    /* => curve object */
    ConsFrame     *pCurve2,   /* => curve object */
    Dpoint3d      *pTPoint    /* => optional: guess at tangency point */
);
```

Description mdlConstraint_createTangent defines a constraint which forces two curves to be tangent. That is, the curves must touch at exactly one point, so that their normals are parallel at the point of tangency.

The constraint does not control the relative positioning of the two curves in any other way and so, has no “inside/outside” sense like the IOTangent constraint.

The target curves may not both be lines or B-splines (although either or both may be B-spline cells). Neither target may be a point.

this is the storage to be filled out.

pCurve and *pCurve2* are the curve construction frames to be constrained.

pTPoint is optional. It specifies the approximate location of the point of tangency. This is useful in case there are many possible points of tangency and you want to control which is found.

Returns mdlConstraint_createTangent returns *this* if successful or NULL if either target is invalid.

See Also mdlConstraint_createIOTangent, mdlConstraint_chooseSolution.

mdlConstraint_createPinPoint

```
#include <mscons.h>

Constraint *mdlConstraint_createPinPoint
(
    Constraint  *this,      /* <= pinpoint constraint */
    ConsFrame   *pCurve,    /* => curve object */
    Dpoint3d    *pIPoint    /* => initial location of fixed point */
);
```

Description `mdlConstraint_createPinPoint` forces a curve to run through a fixed point. `PinPoint` is similar in effect to the `PointOn` constraint, except that the point is always fixed and is internal to the constraint.

This constraint has two attributes, it's (x, y) coordinates, which may be accessed as `CONS_POINT_X` and `CONS_POINT_Y`.

this is the storage to be filled out.

pCurve is the curve construction frame to constrain.

pIPoint is the fixed-point's initial location.

Returns `mdlConstraint_createPinPoint` returns *this*.

See Also `mdlConstraint_createPointOn`.

mdlConstraint_createLockFrame

```
#include <mscons.h>

Constraint *mdlConstraint_createLockFrame
(
    Constraint  *this,      /* <= lock-frame constraint */
    ConsFrame   *pFrame     /* => construction frame to be locked */
);
```

Description `mdlConstraint_createLockFrame` defines a constraint which freezes all attributes of the geometry of a construction frame. That is, it makes the geometry of the construction frame invariant.

The `LockFrame` constraint has one attribute for each attribute of the target frame which defines its fixed value. The constraint's attributes may be accessed at the same indices as the frame's attributes would be.

this is the storage to be filled out.

pFrame is the construction frame object to be constrained.

Returns `mdlConstraint_createLockFrame` returns *this*.

See Also `mdlConstraint_createLockVar`.

mdlConstraint_createLockVar

```
#include <mscons.h>

Constraint *mdlConstraint_createLockVar
(
    Constraint *this,      /* <= LockVar constraint */
    Var *pVar             /* => Solver variable to be constrained */
);
```

Description mdlConstraint_createLockVar defines a constraint which fixes the value of a Solver variable. The variable may be the attribute variable of another Cons object, such as a construction frame.

The LockVar constraint has one attribute, the fixed value, which may be accessed at index 0.

this is the storage to be filled out.

pVar is the solver variable to be constrained.

Returns mdlConstraint_createLockVar returns *this*.

See Also mdlConstraint_createLockFrame.

mdlConstraint_createMatch

```
#include <mscons.h>

Constraint *mdlConstraint_createMatch
(
    Constraint *this,      /* <= match constraint */
    ConsFrame *pFrame,    /* => a construction frame */
    ConsFrame *pFrame2    /* => a construction frame */
);
```

Description mdlConstraint_createMatch defines a constraint which forces the geometric attributes of two frames to match. This effectively equates the frames, so that the effects of constraints on the one will be transmitted to the other and vice versa.

this is the storage to be filled out.

pFrame and *pFrame2* are the construction frames to be equated.

Returns mdlConstraint_createMatch returns *this* if successful or NULL if either target is not a construction frame object.

See Also mdlConstraint_createConcentric.

mdlConstraint_rGroup

```
#include <mscons.h>

int mdlConstraint_rGroup
(
```

```
Constraint  *this      /* => constraint to query */
);
```

Description `mdlConstraint_rGroup` queries if the specified constraint is redundant or in conflict with any other constraint in a constraint model. If a constraint is redundant, the constraint model will have assigned it along with the other constraints which are part of the conflict to a common “redundancy group.” The members of the group can be identified by their common `rGroup` number. If a constraint is not redundant, it will have no `rGroup` number.



This query is only meaningful after the constraint has been added to a constraint model and the model has been validated.

this is the constraint to query.

Returns `mdlConstraint_rGroup` returns 0 if the constraint is not redundant or the non-zero number of its redundancy group if it is redundant.

See Also `mdlConsMod_validate`.

mdlConstraint_getTarget

```
#include <mscons.h>

ConsFrame *mdlConstraint_getTarget
(
Constraint  *this,      /* => constraint to query */
int         i           /* => which target to access */
);
```

Description `mdlConstraint_getTarget` retrieves the specified target of the constraint.

this is the constraint to query.

i specifies which target to retrieve and must be a number between 0 and `CONSMAXTARGETS`.

Returns `mdlConstraint_getTarget` returns a pointer to the *i*th target construction frame of the constraint or NULL if *i* is out of range.

mdlConstraint_flipInsideOutside

```
#include <mscons.h>

int mdlConstraint_flipInsideOutside
(
Constraint  this        /* <=> constraint to modify */
);
```

Description `mdlConstraint_flipInsideOutside` inverts the inside/outside sense of an `IOtangent`, `Offset` or `LLDistance` constraint.

this is the constraint to modify.

Returns mdlConstraint_flipInsideOutside returns SUCCESS if the specified constraint has a sense or ERROR if not.

See Also mdlConstraint_chooseSolution.

mdlConstraint_chooseSolution

```
#include <mscons.h>

int mdlConstraint_chooseSolution
(
  Constraint *this,      /* <=> constraint to modify */
  Dpoint3d *pPoint      /* => point close to desired solution */
);
```

Description mdlConstraint_chooseSolution modifies the constraint's sense or other internal control information which determines which solution out of all possibilities the solver will favor when satisfying the constraint. The solution to be chosen is the one which is located geometrically closest to the specified point. For example, if the constraint is an intersection and there are two possible points of intersection, the solver will favor the one closest to the specified point. Or, if the constraint is a tangency, the solver will favor an inside or outside solution, depending on which puts the point of tangency closest to the specified point.

this is the constraint to modify.

pPoint is the x,y location near the desired constraint solution.

Returns mdlConstraint_chooseSolution returns SUCCESS if the constraint's solution can be controlled by location or ERROR if this is not appropriate.

See Also mdlConstraint_flipInsideOutside.

mdlConstraint_setWeak

```
#include <mscons.h>

void mdlConstraint_setWeak
(
  Constraint *this,      /* <=> constraint to modify */
  boolean isWeak        /* => TRUE for weak, FALSE for strong */
);
```

Description The mdlConstraint_setWeak function sets the "weak" attribute of a constraint.

If a constraint is marked as "weak" then it is no longer a hard and fast requirement, and the constraint model is not required to satisfy it. The constraint model does, nevertheless, try to satisfy weak constraints and will attempt to find a solution to the overall constraint problem which minimizes the error in each weak constraint.

Weak constraints are useful for controlling which solution a constraint model will find and favor. For example, if a Concentric constraint between

two points is made weak, then the points will attract each other, but may not actually be made to touch. A weak constraint is therefore sometimes called a “gravity” constraint. If one of the points is fixed in place and the other is constrained to larger model, then the gravity will pull the model toward the solution closest to the fixed point. The fixed point in this example is sometimes called a “track point.”

this is the constraint to modify.

isWeak is TRUE if the constraint is to be weak, or FALSE otherwise.

Returns mdlConstraint_setWeak is of type void.

See Also mdlConstraint_setPostCheck.

mdlConstraint_setPostCheck

```
#include <mscons.h>

void mdlConstraint_setPostCheck
(
  Constraint *this,          /* <=> constraint or ConsEquation */
  boolean    isPostCheck    /* => TRUE = post-check, else FALSE */
);
```

Description The mdlConstraint_setPostCheck function sets the “post-check” attribute of a constraint or ConsEquation.

A “post-check” constraint is used by the constraint model only to verify a solution to the constraint problem, not to derive it. This is a subtle distinction. A post-check constraint is a requirement, just like an ordinary constraint. But a post-check does not add useful information to the constraint model and, so, does not remove degrees of freedom from the problem. Therefore, any number of post-check constraints may be added as requirements to a constraint model, without over-constraining it.

Post-check constraints are useful for representing input validation checks. For example, a post-check constraint equation could specify that the value of a particular parameter must be in a certain range, must equal one of a specific set of values, or must not equal some particular numbers. This requirement does not really help the solver solve the constraint problem, but it does enable the model to reject illegal values for the parameter.

this is the constraint to modify. *this* may also be a ConsEquation object.

isPostCheck defines whether the constraint’s post-check flag is being set (TRUE) or turned off (FALSE).

Returns mdlConstraint_setPostCheck is of type void; it returns no value.

See Also mdlConstraint_setWeak, mdlConsEquation_create.

Equation Constraint Functions

Equation constraint functions are used to define and query the data structures which represent algebraic constraints in constraint-solving problems.

Algebraic equation constraints apply to constraint parameters.

An equation constraint is created from an algebraic expression in terms of arithmetic operators, built-in functions, constants and references to constraint parameters. The syntax of an equation follows the usual rules of mathematical expressions. See *Algebraic Constraints* in the *MicroStation Reference Guide* for details.

An equation constraint can be created only in the context of a constraint model. The model must contain all of the constraint parameters to which the equation refers.

Creating an equation is done in four steps:

1. Initialize the equation data structure.
2. Parse the algebraic expression string. Syntax errors are detected in this step.
3. “Resolve” the expression’s references to constraint parameters. The resolve function looks up the parameters by name in the constraint model. Parameters not found in the constraint model are flagged in this step.
4. Generate the equation constraint. The constraint is ready to be used after this step.

If there is an error in the parsing or resolution step, the equation constraint cannot be created. The creation process can be resumed at any step, e.g., restarting step (3) after adding missing constraint parameters to the constraint model.



An equation containing a logical or inequality operator is automatically flagged as a post-check constraint.



Equation constraints contain dynamically allocated memory. Call `mdlCons_destroy` to free this memory when done with the object, or call `mdlConsMod_destroyNodes` to destroy and free all objects in a constraint model.

The following table lists Equation Constraint functions:

Function	Used to
mdlConsEquation_create	initialize an equation constraint data structure.
mdlConsEquation_parse	parse an algebraic expression and detect syntax errors.
mdlConsEquation_resolve	attempt to resolve a constraint model.
mdlConsEquation_gen	activate an equation constraint.

mdlConsEquation_create

```
#include <mscons.h>

ConsEquation *mdlConsEquation_create
(
    ConsEquation *this,    /* <= empty constraint equation */
    boolean      isCheck   /* => is this to be a post-check? */
);
```

Description mdlConsEquation_create initializes a constraint equation data structure, making it ready for the parsing step.

this is the storage to be filled out.

Returns mdlConsEquation_create returns *this*.

See Also mdlConsEquation_parse, mdlConsEquation_resolve, mdlConsEquation_gen.

mdlConsEquation_parse

```
#include <mscons.h>

int mdlConsEquation_parse
(
    ConsEquation *this,    /* <=> parsed equation */
    char         *str      /* => expression to parse */
);
```

Description mdlConsEquation_parse parses the specified algebraic expression and detects syntax errors. Syntax errors must be fixed before the equation constraint can be resolved and activated.

this is the initialized constraint equation data structure. It is updated to hold the results of the parse. In particular, if there is a parsing error, the *parse.errStr* member will describe the offending token.

The *str* parameter is a NULL-terminated string containing the expression to be parsed.

Returns mdlConsEquation_parse returns SUCCESS if no syntax errors were encountered; ERROR otherwise.

See Also mdlConsEquation_create, mdlConsEquation_resolve, mdlConsEquation_gen.

mdlConsEquation_resolve

```
#include <mscons.h>

int mdlConsEquation_resolve
(
    ConsEquation *this, /* <=> resolved equation */
    ConsModel *pModel /* => model in which to look for parms */
);
```

Description mdlConsEquation_resolve looks in the specified constraint model for the constraint parameter objects to which the equation refers by name. mdlConsEquation_resolve fails if any parameter cannot be found in the model. All references must be resolved before the equation constraint can be activated.

this is the parsed equation constraint to be resolved. It is updated with the results of the resolve step. In particular, if a constraint parameter cannot be found, its name is placed in *parse.errStr*.

pModel is the Constraint Model in which Constraint Parameters are to be found.

Returns mdlConsEquation_resolve returns SUCCESS if all references to Constraint Parameters are resolved; ERROR otherwise.

See Also mdlConsEquation_create, mdlConsEquation_parse, mdlConsEquation_gen.

mdlConsEquation_gen

```
#include <mscons.h>

int mdlConsEquation_gen
(
    ConsEquation *this /* <=> activated equation */
);
```

Description mdlConsEquation_gen activates an equation constraint. An equation cannot be activated if it has not already been successfully parsed and resolved.

this is the parsed and resolved equation constraint to activate.

Returns mdlConsEquation_gen returns ERROR if *this* could not be activated (because of parsing or resolution errors); otherwise, SUCCESS.

See Also mdlConsEquation_create, mdlConsEquation_parse, mdlConsEquation_resolve.

Attachment Functions

Attachment functions are used to define the data structure which represent “attachments.” An attachment is like an observer in a constraint-solving problem. It contributes no constraints to the problem and has no attribute variables of its own to be solved. Instead, an attachment simply points to one or more Construction Frames and/or Constraints as its “targets.” The constraint solver does not “see” attachments and does nothing with or to them. Attachments may be added to a Constraint Model for the convenience of the application; they are completely inactive in the model. Attachments can be used by an application as a grouping mechanism or to relate a MicroStation element to the geometric attributes of one or more Construction Frames and/or Constraints.

The following table lists Attachment functions:

Function	Used to
<code>mdlConsAttachment_create</code>	define an attachment.

`mdlConsAttachment_create`

```
#include <mscons.h>

ConsAttachment *mdlConsAttachment_create
(
    ConsAttachment *this,          /* <= attachment */
    int typ,                      /* => type code to be stored */
    Cons *targets[],              /* => targets */
    int nTargets                  /* => number of targets */
);
```

Description `mdlConsAttachment_create` defines an attachment. The specified targets are stored in the attachment’s targets array.



An attachment’s targets may be retrieved using `mdlConstraint_getTarget`.

this is the storage to be filled out.

typ is the value to be stored as the attachment’s type code. This value has no significance to the constraint solver.

targets is an array of pointers to the construction frames or constraints which the attachment is targeting. The *nTargets* is the number of items in the *targets* array.

Returns `mdlConsAttachment_create` returns *this*.

See Also `mdlConstraint_getTarget`.

Constraint Model Functions

Constraint Model functions define and manage that data structure which represents a constraint model.

A constraint model is the agent which “solves” a constraint problem by modifying the geometric attributes of a set of construction frames and the values of constraint parameters so as to satisfy all of the constraints which apply to them. A model is therefore a group of geometric and equation constraints which are to be satisfied, along with the construction frames and constraint parameters which are to be modified (or which are constants and serve to define the parameters of the problem). The order in which constraints, constructions, and parameters appear in a model is not important, as constraints are solved simultaneously.

A constraint model is created by initializing it and then adding constraints, frames, and parameters to it. Attachments may also be added to a model, but they are inactive.

After all constraints and constructions have been added to it, a constraint model can be solved by calling the model’s solve function. The results can then be checked by querying the attribute variables of the construction frames and constraint parameters which were modified.

To remove a constraint from the problem, call `mdlConsMod_drop`, and then be sure to destroy the constraint

After solving a constraint problem, the application will want to know which frames and parameters have changed. To make this query possible, the application must first save the current or baseline values of all attribute variables before solving. It will then be possible to find out if variables have been changed, in comparison with the saved values. The `mdlConsMod_backupVars` function saves the current state of all variables. This function should be called whenever the baseline is to be reestablished, e.g., just before solving. The `mdlConsMod_restoreVars` function will restore the saved values of all variables if need be.



If the model cannot solve a set of constraints, it will automatically restore the saved values of all attribute variables. The model never calls the `mdlConsMod_backupVars` function, however; when to establish a new baseline is strictly up to the caller.

To determine if a model is incomplete, use the `mdlConsMod_validate` function. This function analyzes the constraint problem, calculates the remaining degrees of freedom, and identifies any under-constrained variables and/or redundant constraints. To query the summary results of the analysis, check the return values of `mdlConsMod_validate` and `mdlConsMod_dof`. To check the detailed results of the analysis, query the under-determined status of each construction frame and constraint parameter and the “rGroup” assignment of each constraint.

An under-constrained or over-constrained problem may be solvable, despite being incomplete or inconsistent. The results may be useful feedback in the modeling process.



It is not necessary to validate a model before solving it.

The following table lists Constraint Model functions:

Function	Used to
mdlConsMod_create	define a constraint model.
mdlConsMod_destroyNodes	drop, destroy and free all items in a model.
mdlConsMod_destroy	free memory associated with a model.
mdlConsMod_add	add constraint, frame or other items to a model.
mdlConsMod_drop	remove constraint, frame, or other items from a model.
mdlConsMod_concat	add items from one model to another.
mdlConsMod_solve	modify variables to satisfy constraints.
mdlConsMod_validate	analyze dependencies to modify flags.
mdlConsMod_dof	return degrees of freedom in a constraint problem.
mdlConsMod_backupVars	save current values of all attribute variables.
mdlConsMod_restoreVars	set current values of attribute variables to a set of saved values.
mdlConsMod_apply	process each item in model.
mdlConsMod_chooseSolution	direct a model to choose solutions closest to current geometry.
mdlConsMod_isFound	look for an item in a model.
mdlConsMod_transform	transform model geometry.
mdlConsMod_offset	translate model geometry.
mdlConsMod_mirror	mirror model geometry across line of symmetry.

mdlConsMod_create

```
#include <mscons.h>

ConsModel *mdlConsMod_create
(
    ConsModel *this      /* <= constraint model */
);
```

Description mdlConsMod_create defines an empty constraint model.



A constraint model contains dynamically allocated memory and must be destroyed when not longer needed in order to release this memory.

this is the storage to be filled out.

Returns mdlConsMod_create returns *this*.

See Also mdlConsMod_destroy.

mdlConsMod_destroyNodes

```
#include <mscons.h>

void mdlConsMod_destroyNodes
(
    ConsModel *this      /* => constraint model */
);
```

Description mdlConsMod_destroyNodes drops, destroys, and frees each constraint, construction frame, constraint parameter, equation and attachment which has been added to the model. The model itself is not destroyed.



Each item is freed. Do not call this function if the model contains items that were not created with dynamic memory.

this is the constraint model

Returns mdlConsMod_destroyNodes is of type void.

See Also mdlConsMod_destroy.

mdlConsMod_destroy

```
#include <mscons.h>

void mdlConsMod_destroy
(
    ConsModel *this      /* <=> constraint model */
);
```

Description `mdlConsMod_destroy` frees the memory allocated by `mdlConsMod_create`. The constraint model is then invalid and should not be used.



The input pointer is not freed.

this is the constraint model. It is not freed.

Returns `mdlConsMod_destroy` is of type `void`.

See Also `mdlConsMod_destroyNodes`.

mdlConsMod_add

```
#include <mscons.h>

void *mdlConsMod_add
(
  ConsModel *this,    /* <=> constraint model */
  void *pNode        /* <=> object to add */
);
```

Description `mdlConsMod_add` adds a constraint, construction, etc. to a constraint model.



Duplicates are not checked, and adding the same object to a model more than once is not advisable. Duplicates would not cause problems in constraint solving, but would cause memory errors in `mdlConsMod_destroyNodes`.

this is the constraint model to update.

pNode points to a constraint, construction frame, constraint parameter, equation constraint, or attachment.

Returns `mdlConsMod_add` returns *this*.

See Also `mdlConsMod_drop`.

mdlConsMod_drop

```
#include <mscons.h>

void mdlConsMod_drop
(
  ConsModel *this,    /* <=> constraint model */
  void *pNode        /* => constraint or other object to remove */
);
```

Description `mdlConsMod_drop` removes an object from a constraint model. The object is not destroyed. If the object is a constraint, its effects on the constraint-solving problem are removed.



Dropping an object that is not in the model has no effect.

this is the constraint model.

pNode is the Cons object to drop from the model.

Returns mdlConsMod_drop is of type void.

See Also mdlConsMod_add, mdlConsMod_destroyNodes, mdlConsMod_destroy.

mdlConsMod_concat

```
#include <mscons.h>

void mdlConsMod_concat
(
  ConsModel    *this,      /* <=> constraint model to augment */
  ConsModel    *pModel2   /* => constraint model to read */
);
```

Description The mdlConsMod_concat adds all of the objects in the second model to the first model. Each object added is assigned a new model ID number in the process.



The objects to be concatenated are not copied and are not dropped from the second model. Therefore, after calling this function, both models will point to these objects. The second model should therefore be destroyed or emptied out, but you should not call mdlConsMod_destroyNodes on the second model or destroy its contents by any other means.

this is the constraint model to which the objects are added.

pModel2 is the constraint model from which the objects are read.

Returns mdlConsMod_concat is of type void.

See Also mdlConsMod_add.

mdlConsMod_solve

```
#include <mscons.h>

int mdlConsMod_solve
(
  ConsModel    *this      /* <=> constraint model to solve */
);
```

Description mdlConsMod_solve solves a constraint problem by modifying the attribute variables of construction frames and constraint parameters as necessary. If all constraints are already satisfied, nothing is done. Solve restores the values of all attribute variables to their saved values if the constraint problem cannot be solved.

this is the constraint model to solve.

Returns mdlConsMod_solve returns SUCCESS if the constraint problem required solving and was solved, ERROR if the problem required solving and could not be solved, or NOP if the constraint problem did not require solving.

See Also mdlConsMod_backupVars, mdlConsMod_restoreVars.

mdlConsMod_validate

```
#include <mscons.h>

int mdlConsMod_validate
(
  ConsModel *this      /* => constraint model to diagnose */
);
```

Description mdlConsMod_validate diagnoses under-determined variables and redundant constraints in a constraint-solving problem. The under-determined status of the attribute variables of construction frames and constraint parameters is updated, and the rGroup value of constraints is reset.

this is the constraint model to diagnose.



You must solve the model before validating.

Returns mdlConsMod_validate returns ERROR if the constraint model contains redundant equations; otherwise, SUCCESS.

See Also mdlConsMod_dof, mdlConstraint_rGroup.

mdlConsMod_dof

```
#include <mscons.h>

int mdlConsMod_dof
(
  ConsModel *this      /* => constraint model to diagnose */
);
```

Description mdlConsMod_dof returns the degrees of freedom left in a constraint problem. If the degrees of freedom is zero and there are no redundant constraints in the model, then the problem is well constrained.

this is the constraint model to diagnose.



You must solve the model before validating.

Returns mdlConsMod_dof returns the number of degrees of freedom left in the model.

See Also mdlConsMod_validate.

mdlConsMod_backupVars

```
#include <mscons.h>

void mdlConsMod_backupVars
(
  ConsModel    *this      /* => constraint model */
);
```

Description mdlConsMod_backupVars commands each object in the constraint model to save the current values of its attribute variables. This then establishes the baseline for comparison and restoring for later calls to the solve function.



All attribute variable values are backed up, including the attributes of dimensions and constants.

this is the constraint model containing the construction frames, etc., which are to be backed up.

Returns mdlConsMod_backupVars is of type void.

See Also mdlConsMod_restoreVars, mdlCons_isChanged.

mdlConsMod_restoreVars

```
#include <mscons.h>

void mdlConsMod_restoreVars
(
  ConsModel    *this      /* => constraint model */
);
```

Description mdlConsMod_restoreVars commands each object in the constraint model to restore the values of its attribute variables. This includes the attribute variables of constraints and constants.

mdlConsMod_solve calls mdlConsMod_restoreVars automatically whenever it cannot solve a constraint problem.

this is the constraint model containing the construction frames, etc., which are to be restored.

Returns mdlConsMod_restoreVars is of type void.

See Also mdlConsMod_backupVars, mdlConsMod_solve.

mdlConsMod_apply

```
#include <mscons.h>

void mdlConsMod_apply
(
  ConsModel *this, /* => constraint model */
  void (*fp)(), /* => MDL function to apply to each object */
  void *arg /* => second argument to MDL function */
);
```

Description `mdlConsMod_apply` applies the supplied MDL function to each object in the constraint model. The function should expect a pointer to the object as its first argument and the supplied *arg* as its second argument. The function's return value, if any, is ignored. All objects in the model are processed.

this is the constraint model.

fp is a pointer to an MDL function, and *arg* is passed to it as its second argument.

Returns `mdlConsMod_apply` is of type `void`.

mdlConsMod_chooseSolution

```
#include <mscons.h>

void mdlConsMod_chooseSolution
(
  ConsModel *this /* <=> Constraint Model */
);
```

Description `mdlConsMod_chooseSolution` directs the constraint model *pModel* to identify and track the solutions to the constraint problem closest to the model's current geometry.

This allows you to “teach” the model by example which solutions you want it to track. If, for example, there were an Intersection constraint applying to a particular point in the model and if there were two or more possible intersections to choose from, then the model would choose and favor the intersection closest to the current location of the point. Or, if there were an IOTangent constraint, the model would reset the inside/outside sense of the tangency to match the current relative locations of the target circle and other curve.

`mdlConsMod_chooseSolution` may be called as often as needed. It has no effect on model validation.



You must call `mdlConsMod_solve` in order to obtain the new solutions.

Returns `mdlConsMod_chooseSolution` is of type `void`.

See Also `mdlConsMod_solve`.

mdlConsMod_isFound

```
boolean mdlConsMod_isFound
(
  ConsModel *this,      /* => constraint model */
  void *p               /* => object to find */
);
```

Description mdlConsMod_isFound looks for an object in the constraint model by its pointer value.

this is the constraint model to be searched.

p is a pointer to the object to find.

Returns mdlConsMod_isFound returns TRUE if the object is found in the model; FALSE otherwise.

See Also mdlConsMod_add, mdlConsMod_drop.

mdlConsMod_transform

```
#include <mscons.h>

void mdlConsMod_transform
(
  ConsModel *this,      /* <=> Constraint Model */
  Transform *pTransform /* => transformation matrix */
);
```

Description mdlConsMod_transform transforms the geometry of all objects in the specified constraint model.



Transforming model geometry will have no effect on whether or not a model is solved if the transformation entails only translation and/or rotation, since it represents only a change in coordinate system. It should not be necessary to re-solve or re-validate the model in this case. The attribute values of the objects in the model will change, nevertheless.



Applying a non-uniform scaling or a non-orthogonal transformation will change relative geometry and will potentially change both the solution conditions and the possibility of solving. In this case, you must call mdlConsMod_solve in order to obtain a new solution and you may have to call mdlConsMod_validate to check the model's viability.

this is the model containing the construction frames, etc. which are to be modified.

pTransform is the transformation matrix to apply to each object's geometry.

Returns mdlConsMod_transform is of type void.

See Also mdlConsMod_offset, mdlConsMod_mirror.

mdlConsMod_offset

```
#include <mscons.h>

void mdlConsMod_offset
(
  ConsModel    *this,      /* <=> Constraint Model */
  Dpoint3d     *pOff       /* => offset to apply */
);
```

Description mdlConsMod_offset translates all objects in the specified constraint model by the specified offsets in the x and y directions in the current plane (the z offset is ignored, since the model is planar).



Translating model geometry will have no effect on whether or not a model is solved, since it represents only a change in coordinate system. It should not be necessary to re-solve or re-validate the model in this case. The attribute values of the objects in the model will change, nevertheless.

this is the model containing the construction frames, etc. which are to be modified.

pOff contains the x and y offsets to add to each object's location.

Returns mdlConsMod_offset is of type void.

See Also mdlConsMod_transform.

mdlConsMod_mirror

```
#include <mscons.h>

void mdlConsMod_mirror
(
  ConsModel    *this,      /* <=> Constraint Model */
  Dpoint3d     *pO,        /* => local coordinate system origin */
  RotMatrix     *pRot       /* => local coordinate system axes */
);
```

Description mdlConsMod_mirror modifies the geometry of the objects in the specified constraint model by mirroring across the specified line of symmetry in the current plane.

The line of symmetry is defined by a point, *pO*, and a rotation matrix, *pRot*. The rotation matrix should be a simple 2D rotation matrix, such as mdlRMATRIX_fromAngle would create. The direction of the line of symmetry is implicitly the x-axis-column of the rotation matrix.



Mirroring model geometry will have no effect on whether or not a model is solved, since it represents only a change in coordinate system (flipping

the imaginary z-axis over). It should not be necessary to re-solve or re-validate the model in this case. The attribute values of the objects in the model will change, nevertheless.

this is the model containing the construction frames, etc. which are to be modified.

Returns mdlConsMod_mirror is of type void.

See Also mdlConsMod_transform.

Constraint Object Functions

Constraint Object functions define the common behavior of all Constraint, Construction Frame, Constraint Parameter, Equation Constraint and Attachment objects, providing uniform access to these objects, their attributes and capabilities. The objects are all types of “Cons” objects.

The Constraint Object functions can be used to determined the operational status of a constraint object, query the results of a constraint problem solution, and get direct access to attribute variables.

The following table lists Constraint Object functions:

Function	Used to
mdlCons_isChanged	compare attribute vars and/or targets to saved values.
mdlCons_getRMatrix	get rotation matrix describing object's current geometry.
mdlCons_getPoint	get (xy) location of an object.
mdlCons_destroy	free memory associated with an object.
mdlCons_isOperational	check that an object is properly defined.
mdlCons_getVar	retrieve an attribute variable.

mdlCons_isChanged

```
#include <mscons.h>

boolean mdlCons_isChanged
(
  Cons    *this      /* => the object to check */
);
```

Description `mdlCons_isChanged` queries if an object's attribute variables or targets have been changed, compared to their saved values.

Attribute variable values are saved by `mdlConsMod_backupVars`.

this is the object to query.

Returns `mdlCons_isChanged` returns `TRUE` if the object's attributes or targets have changed; `FALSE` otherwise.

See Also `mdlConsMod_backupVars`, `mdlConsMod_solve`, `mdlCons_getVar`, `mdlVar_isChanged`.

mdlCons_getRMatrix

```
#include <mscons.h>

RotMatrix *mdlCons_getRMatrix
(
  RotMatrix    *pRot,    /* <= RMatrix describing object's geometry */
  Cons         *this     /* => object to query */
);
```

Description `mdlCons_getRMatrix` returns a rotation matrix describing object's current geometry.

If the object is a construction frame, the rotation matrix describes its orientation about the z-axis. For lines, the rotation is the direction of the line. For ellipses, the rotation is the angle of the major axis. For B-spline cells, the cell's rotation is returned. For points and circles, the rotation matrix is the identity matrix.

If the object is a a constraint which defines one or more points ((IO)Tangent, PointOn, Intersection, or Perpendicular are examples), the rotation matrix describes the normal to the curve at the point. If the object is any other kind of constraint, the matrix is the identity matrix.

If the object is any other type, the matrix is the identity matrix.

In 3D, the object is assumed to lie in the current plane, so the matrix describes only its rotation about the z-axis; the z-axis column of the matrix will always be (0,0,1).

pRot is the output rotation matrix.

this is the object to query.

Returns `mdlCons_getRMatrix` returns *pRot*.

See Also `mdlCons_getPoint`.

mdlCons_getPoint

```
#include <mscons.h>

Dpoint3d *mdlCons_getPoint
(
```

```
Dpoint3d    *pPoint, /* <= location of object */
Cons        *this    /* => object to query */
);
```

Description mdlCons_getPoint returns the (x,y) coordinates of the object's location in the current coordinate system.

In 3D, the object is assumed to lie in the current plane, so that its z-coordinate will always be zero.

If the object is a construction frame point, circle, ellipse, or B-spline cell this function will return its center point. If the object is a construction frame line, mdlCons_getPoint will return a point on the line.

If the object is a constraint which defines a point, this function returns the (x,y) location of the point so defined. If the object is any other kind of constraint, (0,0,0) is returned.

If the object is any other type, (0,0,0) is returned.

pPoint is the returned (x,y,0) location of object in the current coordinate system.

this is the object to query.

Returns mdlCons_getPoint returns *pPoint*.

See Also mdlCons_getRMatrix.

mdlCons_destroy

```
#include <mscons.h>

void mdlCons_destroy
(
Cons    *this    /* <=> object to clean up */
);
```

Description mdlCons_destroy frees any dynamic memory allocated for the Cons object. The Cons object pointer itself is *not* freed.

this is the Cons object to be cleaned up. It is not usable after being destroyed.

Returns mdlCons_destroy is of type void.

See Also mdlConsMod_destroyNodes.

mdlCons_isOperational

```
#include <mscons.h>

boolean mdlCons_isOperational
(
  Cons    *this      /* => object to test */
);
```

Description mdlCons_isOperational determines if the specified `Cons` object is fully defined and has been activated. For example, an equation constraint which has not been activated, say, due to parsing errors, will not be operational. A `NULL` pointer is considered not operational. This function is also useful when re-creating `Cons` objects from data stored in an external format, to test that all references were successfully resolved.

this is the object to test.

Returns mdlCons_isOperational returns `TRUE` if the object is defined and activated; else `FALSE`.

See Also mdlCons_destroy.

mdlCons_getVar

```
#include <mscons.h>
#include <msslv.h>

Var *mdlCons_getVar
(
  Cons    *this,      /* => object to access */
  int     i            /* => attribute to retrieve */
);
```

Description mdlCons_getVar retrieves the *i*th attribute variable of the specified `Cons` object. *this* is the object to be accessed.

i is the index of the attribute to retrieve.

Returns mdlCons_getVar returns a pointer to the *i*th attribute variable or `NULL` if *i* is out of range.

See Also mdlVar_getVal, mdlVar_setVal, mdlVar_isChanged, mdlVar_isUnderDetermined, mdlVar_isConstant, mdlVar_setConstant.

Solver Variable Functions

Solver variable functions create, modify and query internal data structures known as “Vars” which are used to represent variables and constants in constraint-solving problems. (This document refers to these data structures synonymously as “Vars,”

“Solver Variables” and “variables”). The MicroStation Solver operates on `Vars`; and Solver Equations are defined in terms of `Vars`.

A `Var` data structure represents a double precision value and the state of that value (i.e., is it defined?, has it changed?, is it constant?), plus a saved “old” value. A `Var` is an opaque data structure and can be accessed only via the functions given here.

`Vars` are dynamic, special-purpose data structures and do not correspond directly to elements in the design file. It is the application’s responsibility to destroy unused variables using the `mdlVar_destroy` function.

Solver variables are not related to MicroStation’s built-in global variables, nor are they related to environment variables.

The following table lists the solver variable functions:

Function	Used to
<code>mdlVar_create</code>	allocate and initialize a <code>Var</code> .
<code>mdlVar_setConstant</code>	set the “constant” attribute.
<code>mdlVar_isConstant</code>	determine if the value is marked as being constant.
<code>mdlVar_isUnderDetermined</code>	determine if the variable is marked as being “under-determined”.
<code>mdlVar_setUnderDetermined</code>	set or clear the “under-determined” attribute
<code>mdlVar_isChanged</code>	determine the difference between the current and old values.
<code>mdlVar_hasNoValue</code>	determine if the current value has been set to the special value of NA.
<code>mdlVar_setVal</code>	set the value of the variable.
<code>mdlVar_getVal</code>	query the value of the variable.
<code>mdlVar_saveVal</code>	save current values as “old” value.
<code>mdlVar_restoreVal</code>	set current value = “old” value.
<code>mdlVar_use</code>	add a reference to a <code>Var</code> .
<code>mdlVar_destroy</code>	remove reference to a <code>Var</code> (possibly freeing it).

The NA constant is defined in `msslvr.h`.

mdlVar_create

```
#include <msvvr.h>

Var *mdlVar_create
(
    double value      /* => initial value or NA */
);
```

Description The `mdlVar_create` function allocates and initializes a `Var` data structure.

value specifies the variable's initial value. Passing the constant `NA` signifies that the variable has no value (yet).

Returns `mdlVar_create` returns a pointer to the `Var`'s storage, or `NULL` if insufficient memory is available.

See Also `mdlVar_destroy`.

mdlVar_setConstant

```
#include <msvvr.h>

void mdlVar_setConstant
(
    Var *this,          /* <=> variable to modify */
    boolean isConstant  /* => value for 'constant' attribute */
);
```

Description The `mdlVar_setConstant` function sets the “constant” attribute of the `Var`. *this* to the value of *isConstant*.

Returns `mdlVar_setConstant` is of type `void`.

See Also `mdlVar_isConstant`.

mdlVar_isConstant

```
#include <msvvr.h>

boolean mdlVar_isConstant
(
    Var *this          /* => variable to query */
);
```

Description The `mdlVar_isConstant` function queries the “constant” attribute of `Var` *this*.

Returns `mdlVar_isConstant` returns `TRUE` if the `Var` is marked constant; `FALSE`, otherwise.

See Also `mdlVar_setConstant`.

mdlVar_isUnderDetermined

```
#include <msvvr.h>

boolean mdlVar_isUnderDetermined
(
    Var    *this      /* => variable to query */
);
```

Description The mdlVar_isUnderDetermined function queries the “under-determined” attribute of Var *this*.

Returns mdlVar_isUnderDetermined returns TRUE if the Var is marked under-determined; FALSE, otherwise.

See Also mdlVar_setUnderDetermined.

mdlVar_setUnderDetermined

```
#include <msvvr.h>

void mdlVar_setUnderDetermined
(
    Var    *this,          /* <=> variable to modify */
    boolean isUnderDetermined /* => value of under-determined attribute */
);
```

Description The mdlVar_setUnderDetermined function sets the “under-determined” attribute of Var *this* to the value *isUnderDetermined*.

Returns mdlVar_setUnderDetermined is of type void.

See Also mdlVar_isUnderDetermined.

mdlVar_isChanged

```
#include <msvvr.h>

boolean mdlVar_isChanged
(
    Var    *this,          /* => variable to query */
    double tolerance       /* => threshold for change test */
);
```

Description The mdlVar_isChanged function tests if the current value of Var *this* is different from its “old” value by at least *tolerance*.

Returns mdlVar_isChanged returns TRUE if the variable’s current value is significantly different from its old value; FALSE, otherwise.

See Also mdlVar_saveVal, mdlVar_restoreVal.

mdlVar_hasNoValue

```
#include <msvvr.h>

boolean mdlVar_hasNoValue
(
  Var      *this      /* <=> variable to query */
);
```

Description The `mdlVar_hasNoValue` function queries if the value of *this* has not been set yet (or has been set the to special value NA).

Returns `mdlVar_hasNoValue` returns TRUE if the value of the variable is NA; else FALSE.

See Also `mdlVar_setVal`, `mdlVar_getVal`.

mdlVar_setVal

```
#include <msvvr.h>

Var *mdlVar_setVal
(
  Var      *this,      /* <=> variable to update */
  double   value       /* => new value */
);
```

Description The `mdlVar_setVal` function sets the current value of Var *this* to *value*.

Returns `mdlVar_setVal` returns *this*.

See Also `mdlVar_getVal`, `mdlVar_saveVal`, `mdlVar_restoreVal`.

mdlVar_getVal

```
#include <msvvr.h>

double mdlVar_getVal
(
  Var      *this      /* <=> variable to query */
);
```

Description The `mdlVar_getVal` function returns the current value of Var *this*. If *this* has no value, then the special value NA is returned.

Returns `mdlVar_getVal` returns the current value of *this*.

See Also `mdlVar_setVal`, `mdlVar_saveVal`, `mdlVar_restoreVal`, `mdlVar_hasNoValue`.

mdlVar_saveVal

```
#include <msvvr.h>

void mdlVar_saveVal
(
    Var    *this      /* <=> variable to update */
);
```

Description The mdlVar_saveVal function saves the current value of Var *this* as it's “old” value.

Returns mdlVar_saveVal is of type void.

See Also mdlVar_restoreVal.

mdlVar_restoreVal

```
#include <msvvr.h>

void mdlVar_restoreVal
(
    Var    *this      /* <=> variable to link to */
);
```

Description mdlVar_restoreVal sets the current value of *this* to its old value.

Returns mdlVar_restoreVal is of type void.

See Also mdlVar_saveVal.

mdlVar_use

```
#include <msvvr.h>

void mdlVar_use
(
    Var    *this      /* <=> variable to link to */
);
```

Description The mdlVar_use function increments the “use count” for variable *this*.

Returns mdlVar_use is of type void.

See Also mdlVar_destroy.

mdlVar_destroy

```
#include <msvvr.h>

void mdlVar_destroy
(
    Var    *this      /* <=> variable to unlink and/or free */
);
```

Description The `mdlVar_destroy` function decrements the “use count” for variable *this* and, if the use count becomes zero, frees the variable’s storage.

Returns `mdlVar_destroy` is of type `void`.

See Also `mdlVar_use`.

MicroStation for Windows NT, Clipboard and DDE Interface

MicroStation for Windows NT supports clipboard cut/copy/paste operations, Client and Server Dynamic Data Exchange Protocols (DDE), and file drag & drop operations.

Overview of the Windows NT Interface

This section will describe the concepts and functionality related to the MDL functions described later in this chapter.

Dynamic Data Exchange (DDE)

MicroStation's DDE interface is modeled after Microsoft's Dynamic Data Exchange Management Library (DDEML). MicroStation's DDE interface hides many of the complexities of DDE by encapsulating messages, atom management and memory management in a function call mechanism.

What is DDE?

Dynamic data exchange (DDE) is a form of interprocess communication that uses shared memory to exchange data between applications. Applications can use DDE for one-time data transfers and for ongoing exchanges in which the applications send updates to one another as new data becomes available.

Dynamic data exchange differs from the clipboard data-transfer mechanism in that the clipboard is almost always used as a one-time response to a specific action by the user — such as choosing the Paste command from a menu. Although DDE may also be initiated by a user, it typically continues without the user's further involvement. A DDE conversation is the interaction between client and server applications.

MicroStation's DDE interface provides an MDL programming interface that simplifies the task of adding DDE capability to a MDL application. This interface is called the Dynamic Data Exchange Management Library (DDEML).

Instead of sending, posting, and processing DDE messages through an external program or Dynamic Load Module, an MDL application uses the functions provided by MicroStation to manage DDE conversations. MicroStation provides the MDL

programmer a facility for managing the strings and data elements that are shared among DDE applications.

Instead of using atoms and pointers to shared memory objects, DDE applications create and exchange string handles, which identify strings, and data handles, which identify DDE data objects. The DDEML provides a service that makes it possible for a server application to register the service names that it supports. The names are broadcast to other applications in the system, which can then use the names to connect to the server.

Since MicroStation's DDE interface is modeled after Microsoft's DDEML, all programming guidelines and rules apply. With minor changes, most DDEML compliant source code could be incorporated into an MDL application.

Sample Programs

See the supplied MDL applications `ddeclkcn.mc` and `ntcbtext.mc` for suggested programming techniques using the DDEML and the Clipboard. `ddeclkcn.mc` is an MDL-DDEML client program that reports the current time supplied by the Windows NT DDEML clock program (See `\mstools\samples\ddeml\clock`). `ntcbtext.mc` copies and prints text from the clipboard into the debugging window.

The programs `ddeclkcn.mc` and `ntcbtext.mc` are included in this distribution.

The Clipboard

The Clipboard Paste/Copy/Cut operations are invoked by the MicroStation *Edit* pull down menu.

Each of the Clipboard operations have sub-menus which are based on the currently available data formats. The user selects the *Edit->Paste* option to take data from the clipboard into the design file, and the *Edit->Copy* or *Edit->Cut* options to take data from the design file into the clipboard.



Refer to the *Clipboard Functions* section for a more detailed discussion of clipboard strategy functions and format process functions.

MDL Clipboard Interface (Pasting)

Every MDL application that wants to participate in clipboard paste operation uses the `mdlClipboard_setFunction` to set up its clipboard function (i.e., clipboard strategy function). The clipboard strategy function will be called during the following circumstances:

1. `mdlClipboard_setFunction` will immediately call the clipboard strategy function to inform the MDL application of the currently available clipboard formats.
2. When the Windows NT Clipboard changes, MicroStation will call the clipboard strategy function to inform the MDL application of the available Clipboard data formats.

There are 2 types of functions associated with clipboard processing: clipboard strategy function and format process function. The clipboard strategy function is called whenever the clipboard changes. This function determines if it can process any of the available clipboard data formats. For each data format, the clipboard strategy function can nominate another MDL function (i.e., format process function) to actually process the clipboard data if the user picks that format.

Each format process function has a “quality” associated with its ability to process the clipboard data format. When the clipboard strategy function nominates a format process function it also specifies either `CLIPBOARD_FORMAT_CANHANDLE` or `CLIPBOARD_FORMAT_IDEALHANDLER`.

For `CLIPBOARD_FORMAT_CANHANDLE`, the corresponding menu name is put into the sub-menu only if there is no other application that says it can handle this format.

For `CLIPBOARD_FORMAT_IDEALHANDLER`, the corresponding menu name will always appear in the sub-menu. An MDL application can nominate more than one function for a particular clipboard data format as long as it describes them all as `CLIPBOARD_FORMAT_IDEALHANDLERS`.

Processing Example

MicroStation supplies one clipboard text data format handler. This handler can paste text into dialog boxes. The command window *Edit->Paste* pull down menu looks like this:

```

Edit
.
Paste
. To Dialog

```

Other Paste options are added to the command window *Edit->Paste* menu as a consequence of MDL programs registering themselves.

When MicroStation receives a `WM_DRAWCLIPBOARD` message (signifying that new data is in the clipboard), it gets all of the available clipboard data formats, and sets up an internal structure for each clipboard data format. All registered MDL applications clipboard strategy function are called once with all the available formats. The clipboard strategy function responds with an array of structure that tells MicroStation what the

MDL application wants to do with the Paste sub-menu of the command window. At that point the user is presented with a list of paste options. If the user picks a format, that particular format process function will be invoked by MicroStation.

Clipboard programming guide

At initialization time (i.e., main) the MDL application should execute:

```
mdlClipboard_setFunction(CLIPBOARD_SET_PASTE,
newClipboardData);
```

Where `newClipboardData` is called whenever the clipboard changes. This function was referred to as *userNewClipboardDataFunction* earlier in this document. `newClipboardData` should be coded as follows:

```
Private int newClipboardData
(
int *numResponsesP,          /* <= Number of responses */
ClipboardResponse *crP,      /* <= Nominated functions */
ClipboardFormatInfo *cfiP,   /* => Clipboard formats */
int numFormats              /* => Number of formats */
)
{
    int iFormat;
    *numResponsesP=0;
    for (iFormat=0; iFormat<numFormats; iFormat++, cfiP++)
    {
        switch (cfiP->formatNum)
        {
            case CF_TEXT:
                crP->formatNum=cfiP->formatNum;
                crP->handlerQuality=CLIPBOARD_FORMAT_IDEALHANDLER;
                crP->userDataP=NULL;
                crP->handlerFunction=clipboard_pasteTextToDgnFile;
                strcpy(crP->pasteMenuName, 'Text To Design File');
                crP++;
                (*numResponsesP)++;
                break;

            case CF_METAFILEPICT:
                crP->formatNum=cfiP->formatNum;
                crP->handlerQuality=CLIPBOARD_FORMAT_IDEALHANDLER;
                crP->userDataP=NULL;
                crP->handlerFunction=clipboard_pasteMetafileToDgnFile;
                strcpy (crP->pasteMenuName, 'Win 3.1 Metafile (Vectors)');
                crP++;
                (*numResponsesP)++;
                break;
        }
    }
}
```



```

        case CF_DIB:
            crP->formatNum = cfiP->formatNum;
            crP->handlerQuality = CLIPBOARD_FORMAT_IDEALHANDLER;
            crP->userDataP = NULL;
            crP->handlerFunction = clipboard_pasteDIBitmapToDgnFile;
            strcpy(crP->pasteMenuName, 'Raster Bitmap (DIB)');
            crP++;
            (*numResponsesP)++;
            break;

        case CF_BITMAP:
            crP->formatNum = cfiP->formatNum;
            crP->handlerQuality = CLIPBOARD_FORMAT_IDEALHANDLER;
            crP->userDataP = CF_BITMAP;
            crP->handlerFunction = clipboard_pasteBitmapToDgnFile;
            strcpy(crP->pasteMenuName, 'Raster Bitmap');
            crP++;
            (*numResponsesP)++;
            break;

        default:
            break;
    }
}
return 0;
}

```

The following structure is passed to `newClipboardData` as parameter 3. This data structure describes the available clipboard data formats. Parameter 4, *numFormats* specifies the number of entries in this list:

```

typedef struct
{
    ULong formatNum;
    char *formatNameP;
} ClipboardFormatInfo;

```

cfiP->formatNum is the format number for the clipboard data. It can be either a standard windows format `CF_TEXT`, `CF_BITMAP`, etc. or a registered private format. If it is a private format, the *formatNameP* field is filled in.

cfiP->formatNameP is the format name for the clipboard data corresponding to *formatNum*. The string pointed to by this pointer must not be changed by the MDL program. The pointer is `NULL` for standard formats.

The *newClipboardData* function replies to MicroStation by filling in the array of structures that MDL supplies. The response structure looks like this:

```
typedef struct
{
    ULong formatNum;
    int handlerQuality;
    char pasteMenuName[CLIPBOARD_MENU_NAMELEN];
    MdIFunctionP handlerFunction;
} ClipboardResponse;
```

crP->formatNum is the format number referred to by *handlerQuality*.

crP->handlerQuality is the application should fill in one of the following values:

```
CLIPBOARD_FORMAT_CANHANDLE
CLIPBOARD_FORMAT_IDEALHANDLER
```

If the application replies with `CLIPBOARD_FORMAT_CANHANDLE`, the corresponding menu name is put in only if there is no other application that says it can handle this format. If the application replies with `CLIPBOARD_FORMAT_IDEALHANDLER`, The *pasteMenuName* will always appear in the sub-menu. Note that an MDL application can put more than one entry in its *ClipboardResponse* array for a particular format number, as long as it describes them all `CLIPBOARD_FORMAT_IDEALHANDLERS`.

handlerFunction is the data handler function that the MDL application wants to be called when the user selects the *pasteMenuName* from the menu.



An application can handle clipboard data either by setting itself up to be the handler for a particular format using the protocol above, or by putting a command of its own (that can be activated from a private pull down menu selection) that calls its own *handlerFunction*.

An MDL application provided with MicroStation handles the common formats that we know what to do with—`CF_TEXT`, `CF_BITMAP`, `CF_METAFILE`.



The code on the previous page uses hard-coded text for menu names. This is done merely to promote a simple example; an actual application should store its text fields in an external resource to simplify internationalization and localization.

After determining what formats there are handlers for, MicroStation adjusts the pull down menu to show those formats. The Edit menu may look like this:

```

Edit
.
Paste
. To Dialog
  Bitmap
  Text (goes to design file as Nexus paste function does)

```

If the application's intrinsic Paste command is called, it can call its own internal function to do the Paste.

When the MDL application goes to paste the clipboard information, it can call the built-ins MicroStation supplies to get the data from the clipboard. It can handle the data from more than one format in that way (for example, if there is a palette and a bitmap, it will need them both).

Memory Management Functions

The following functions are available to MDL applications that will run under MicroStation for Windows NT. These functions allow access to Windows NT memory management and error status functions.

Function	Used to
mdlWin32_getLastError	report the current error number.
mdlWin32_readDataFromHandle	count members of a dynamic array.
mdlWin32_createDataHandle	create and copy data to a memory handle.
mdlWin32_destroyDataHandle	free resources allocated by.
mdlWin32_writeDataToHandle	copy data to a memory handle.

mdlWin32_getLastError

```

#include <msw32util.fdf> /* NT Util function prototypes */

ULONG mdlWin32_GetLastError(void);

```

Description The mdlWin32_getLastError function returns the most recent error code set by a mdlWin32_... function call.

Returns mdlWin32_getLastError returns the most recent error code set by a Win32 interface function.

mdlWin32_readDataFromHandle

```
#include <msw32util.fdf> /* NT Util function prototypes */

int mdlWin32_readDataFromHandle
(
    HANDLE hMem,          /* => Handle to memory block */
    void **mdlMemPP,      /* <=> Pointer address */
    UInt *mdlMemLen       /* <= Size (in bytes) of data */
);
```

Description The `mdlWin32_readDataFromHandle` function extracts data from the specified memory handle *hMem* and copies the data into a malloced data buffer. The address of this buffer is placed in *mdlMemPP*. The size, in bytes, of this buffer is placed into *mdlMemLenP*.

hMem specifies the Windows memory handle on which to operate.

mdlMemPP specifies the address of a buffer pointer. A memory block large enough to hold the contents of the data portion of the memory handle is allocated and that address is placed into this parameter.

mdlMemLenP points to the integer to receive the size of the allocated memory block.

Returns `mdlWin32_readDataFromHandle` returns `SUCCESS` if the function is successful.

See Also `mdlWin32_writeDataToHandle`, `mdlWin32_createDataHandle`.

mdlWin32_createDataHandle

```
#include <msw32util.h> /* NT Util constants */
#include <msw32util.fdf> /* NT Util function prototypes */

HANDLE mdlWin32_createDataHandle
(
    UInt flags,           /* => WinNT mem alloc flags */
    void *mdlMemP,       /* => pointer to data. */
    UInt mdlMemLen       /* => length (bytes) of *mdlMemP */
);
```

Description The `mdlWin32_createDataHandle` function allocates the specified number of bytes from the heap and copies *mdlMemLen* bytes from *mdlMemP* into the memory handle.

flags specifies how to allocate memory and can be a combination of the following flags:

Flag	Description
GMEM_MOVEABLE	Allocates moveable memory.
GHND	Combines the <code>GMEM_MOVEABLE</code> and <code>GMEM_ZEROINIT</code> flags.

Flag	Description
GMEM_DDESHARE	Allocates memory to be used in the dynamic data exchange (DDE) functions for a DDE conversation and should be set if the memory is to be used for DDE. Only processes that use DDE or the clipboard for interprocess communication should specify this flag. Most applications will use this value.
GMEM_SHARE	Same as GMEM_DDESHARE
GMEM_ZEROINIT	Initialize memory contents to zero.

mdlMemP specifies the source of the data to copy into the memory handle.

mdlMemLen specifies the number of bytes to allocate for the memory handle and to copy from *mdlMemP*.



It is the responsibility of the programmer to insure that the memory allocated by this function is eventually freed.

Returns *mdlWin32_createDataHandle* returns the handle of the newly allocated memory object; otherwise, it return NULL. To get extended error information, use the *mdlWin32_getLastError* function.

See Also *mdlWin32_getLastError*.

mdlWin32_destroyDataHandle

```
#include <msw32util.fdf> /* NT Util function prototypes */
HANDLE mdlWin32_createDataHandle
(
HANDLE hMem      /* => WinNT mem handle */
);
```

Description The *mdlWin32_destroyDataHandle* function frees to resources allocated by *mdlWin32_createDataHandle*.

hMem specifies the memory handle to return to the system.

Returns *mdlWin32_destroyDataHandle* returns 0 for success. To get extended error information, use the *mdlWin32_getLastError* function.

See Also *mdlWin32_getLastError*, *mdlWin32_createDataHandle*.

mdlWin32_writeDataToHandle

```
#include <msw32util.fdf> /* NT Util function prototypes */

int mdlWin32_writeDataToHandle
(
    HANDLE hMem,          /* => Handle to memory block */
    void *mdlMemP,        /* => Buffer pointer */
    UInt mdlMemLen        /* => Size (in bytes) of data */
);
```

Description The `mdlWin32_writeDataToHandle` function copies data from the specified memory buffer pointed to by *mdlMemP*, for *mdlMemLen* bytes into the memory handle *hMem*.

hMem specifies the Windows memory handle on which to operate.

mdlMemP specifies the address of the source data buffer.

mdlMemLen specifies the size, in bytes, of the source buffer.

Returns `mdlWin32_writeDataToHandle` returns *hMem* on success or NULL on failure.

See Also `mdlWin32_readDataFromHandle`, `mdlWin32_createDataHandle`.

Windows NT DDE Functions

The `mdlWind32_Dde...` functions are used to exchange data between applications. Applications can use DDE for one-time data transfers and for on-going exchanges in which the applications send updates to one another as new data becomes available.

Dynamic data exchange differs from the clipboard data transfer mechanism in that the clipboard is almost always used as a one-time response to a specific action by the user — such as choosing the Paste command from a menu. Although DDE may also be initiated by a user, it typically continues without the user's further involvement.

MicroStation's DDE interface is modeled after Microsoft's Dynamic Data Exchange Management Library (DDEML). MicroStation's DDEML interface hides many of the complexities of DDE by encapsulating messages, atom management, and memory management in a function call mechanism.

You are encouraged to review Microsoft's on-line SDK documentation for further details. The following documentation is based on that information.

The BSI MDL coding standard has is not followed for these functions. These functions are modeled on the similarly named Windows NT functions and follow that standard.

Please review the sample MDL application `ddeclcn.mc` for examples of using these built-in functions.

The following table lists the MDL-DDEML functions:

Function	Used to
mdlWin32_DdeAbandonTransaction	abandon an asynchronous transaction.
mdlWin32_DdeAccessData	access a data object.
mdlWin32_DdeAddData	add data to a data object.
mdlWin32_DdeClientTransaction	begin a DDE data transaction.
mdlWin32_DdeCmpStringHandles	compare two DDE string handles.
mdlWin32_DdeConnect	establish a conversation with a server.
mdlWin32_DdeConnectList	establish multiple conversations.
mdlWin32_DdeCreateDataHandle	create a data handle.
mdlWin32_DdeCreateStringHandle	create a string handle.
mdlWin32_DdeDisconnect	terminate a conversation
mdlWin32_DdeDisconnectList	destroy a conversation list
mdlWin32_DdeEnableCallback	enable or disables one or more conversations.
mdlWin32_DdeFreeDataHandle	free a DDE data object.
mdlWin32_DdeFreeStringHandle	free a DDE string handle.
mdlWin32_DdeGetData	copy data from a DDE data object to a buffer.
mdlWin32_DdeGetLastError	return an error code set by the most recent DDEML function.
mdlWin32_DdeInitialize	register an application.
mdlWin32_DdeKeepStringHandle	increment the usage count for a string handle.
mdlWin32_DdeNameService	register or unregisters a service name.
mdlWin32_DdePostAdvise	prompt a server to send advise data to a client.
mdlWin32_DdeQueryConvInfo	get information about a conversation.
mdlWin32_DdeQueryNextServer	obtain the next handle in a conversation list.
mdlWin32_DdeQueryString	copy string-handle text to a buffer.
mdlWin32_DdeReconnect	reestablish a DDE conversation.
mdlWin32_DdeSetUserHandle	associate a user-defined handle with a transaction
mdlWin32_DdeUnaccessData	free a DDE data object.
mdlWin32_DdeUninitialize	free an application's DDEML resources.

The following table lists DDE user functions. The programmer determines the user functions names. These functions are all designated to MDL through function pointers. MDL does not use the names. The following function names are used merely as illustrations:

Function	MicroStation calls when
<i>userWin32_DdeUserCallback</i>	a DDE message is sent from another application.

All of the functions return error status through the `mdlWin32_DdeGetLastError` function. The symbolic names are defined in the file `w32dde.h` and are listed below:

DMLERR_NO_ERROR	DMLERR_ADVACKTIMEOUT
DMLERR_BUSY	DMLERR_DATAACKTIMEOUT
DMLERR_DLL_NOT_INITIALIZED	DMLERR_DLL_USAGE
DMLERR_EXEACKTIMEOUT	DMLERR_INVALIDPARAMETER
DMLERR_LOW_MEMORY	DMLERR_MEMORY_ERROR
DMLERR_NOTPROCESSED	DMLERR_NO_CONV_ESTABLISHED
DMLERR_POKEACKTIMEOUT	DMLERR_POSTMSG_FAILED
DMLERR_REENTRANCY	DMLERR_SERVER_DIED
DMLERR_SYS_ERROR	DMLERR_UNADVACKTIMEOUT
DMLERR_UNFOUND_QUEUE_ID	

mdlWin32_DdeAbandonTransaction

```
#include <w32dde.h> /* For MDL-DDEML constants. */
#include <msw32dde.fdf> /* For MDL-DDEML prototypes. */

BOOL mdlWin32_DdeAbandonTransaction
(
    DWORD    idInst,          /* => Instance id */
    HCONV    hConv,          /* => Conversation handle */
    DWORD    idTransaction /* => XTYP_ or NULL for all */
);
```

Description The `mdlWin32_DdeAbandonTransaction` function abandons the asynchronous transaction identified by the *idTransaction* parameter and releases all resources associated with the transaction.

dInst specifies the application-instance identifier obtained by a previous call to the `mdlWin32_DdeInitialize` function.

hConv identifies the conversation in which the transaction was initiated. If NULL, all transactions are abandoned (the *idTransaction* parameter is ignored).

idTransaction identifies the transaction to terminate. If this parameter is NULL, all active transactions in the specified conversation are abandoned.

Returns mdlWin32_DdeAbandonTransaction returns TRUE if the function is successful and FALSE otherwise. Use the mdlWin32_DdeGetLastError function to retrieve the error value, which may be one of the following: DMLERR_DLL_NOT_INITIALIZED, DMLERR_INVALIDPARAMETER, DMLERR_NO_ERROR, DMLERR_UNFOUND_QUEUE_ID.

See Also mdlWin32_DdeClientTransaction, mdlWin32_DdeGetLastError, mdlWin32_DdeInitialize, mdlWin32_DdeQueryConvInfo.

mdlWin32_DdeAccessData

```
#include <w32dde.h> /* For MDL-DDEML constants. */
#include <msw32dde.fdf> /* For MDL-DDEML prototypes. */

LPBYTE mdlWin32_DdeAccessData
(
    HDEADATA    hData,          /* => DDE object handle */
    LPDWORD     pcbDataSize     /* <= Size of DDE object */
);
```

Description mdlWin32_DdeAccessData returns a pointer to the data in the DDE object identified by the *hData* parameter. An application must call mdlWin32_DdeUnaccessData when it is finished accessing the data in the object.

hData identifies the DDE object to access.

pcbDataSize points to a variable that receives the size, in bytes, of the DDE object identified by *hData*. If this parameter is NULL, no size information is returned.

If the *hData* parameter has not been passed to a DDEML function, an application can use the pointer returned by mdlWin32_DdeAccessData for read-write access to the DDE object's data area. If *hData* has already been passed to a DDEML function, the pointer can only be used for read-only access to the memory object.

Returns mdlWin32_DdeAccessData returns a pointer to the first byte of data in the DDE object associated with the *hData* parameter. Otherwise, the return value is NULL. Use the mdlWin32_DdeGetLastError function to retrieve the error value, which may be one of the following: DMLERR_DLL_NOT_INITIALIZED, DMLERR_INVALIDPARAMETER, DMLERR_NO_ERROR.

See Also mdlWin32_DdeAddData, mdlWin32_DdeCreateDataHandle, mdlWin32_DdeFreeDataHandle, mdlWin32_DdeGetLastError, mdlWin32_DdeUnaccessData.

mdlWin32_DdeAddData

```
#include <w32dde.h> /* For MDL-DDEML constants */
#include <msw32dde.fdf> /* For MDL-DDEML prototypes */

HDEDATA mdlWin32_DdeAddData
(
    HDEDATA    hData,          /* => DDE object handle */
    LPBYTE     pSrc,           /* => Source data buffer */
    DWORD      cb,             /* => Source data size */
    DWORD      cbOff           /* => Offset into source buffer */
);
```

Description *mdlWin32_DdeAddData* adds data to the given DDE data object. An application can add data beginning at any offset from the beginning of the object. If new data overlaps data already in the object, the new data overwrites the old data in the bytes where the overlap occurs. The contents of locations in the object that have not been written to are undefined. After a data handle has been used as a parameter in another DDEML function or returned by a DDE callback function, the handle may only be used for read access to the data object identified by the handle.

If the amount of memory originally allocated is not large enough to hold the added data, *mdlWin32_DdeAddData* will reallocate a DDE data object of the appropriate size.

hData identifies the DDE data object that receives additional data.

pSrc points to a buffer containing the data to add to the DDE data object.

cb specifies the length, in bytes, of the data to be added to the DDE data object.

cbOff specifies an offset, in bytes, from the beginning of the DDE data object. The additional data is copied to the object beginning at this offset.

Returns If the *mdlWin32_DdeAddData* function succeeds, the return value is a new handle of the DDE data object. The new handle should be used in all references to the object. If an error occurs, the return value is zero. Use the *mdlWin32_DdeGetLastError* function to retrieve the error value, which may be one of the following:

```
DMLERR_DLL_NOT_INITIALIZED, DMLERR_INVALIDPARAMETER, DMLERR_MEMORY_ERROR,
DMLERR_NO_ERROR.
```

See Also *mdlWin32_DdeAccessData*, *mdlWin32_DdeCreateDataHandle*,
mdlWin32_DdeGetLastError, *mdlWin32_DdeUnaccessData*.

mdlWin32_DdeClientTransaction

```
#include <wnc1pext.h> /* For CF_xx clipboard formats. */
#include <w32dde.h> /* For MDL-DDEML constants. */
#include <msw32dde.fdf> /* For MDL-DDEML prototypes. */

HDEDATA mdlWin32_DdeClientTransaction
(
```

```
LPBYTE pData,          /* => Data to pass to server */
DWORD cbData,          /* => Data size */
HCONV hConv,          /* => Conversation handle */
HSZ hszItem,          /* => Item string handle */
UINT wFmt,            /* => Clipboard format */
UINT wType,           /* => XTYP_ */
DWORD dwTimeout,      /* => Timeout */
LPDWORD pdwResult     /* <= Transaction result */
);
```

Description The `mdlWin32_DdeClientTransaction` function begins a data transaction between a client and a server. Only a client application can call this function, and only after establishing a conversation with a server. When the application is finished using the data handle returned by the `mdlWin32_DdeClientTransaction` function, the application should free the handle by calling the `mdlWin32_DdeFreeDataHandle` function.

Transactions can be synchronous or asynchronous. During a synchronous transaction, the `mdlWin32_DdeClientTransaction` function does not return until the transaction completes successfully or fails. Synchronous transactions cause the client to enter a modal DDE loop while waiting for various asynchronous events. Because of this, the client application can still respond to user input while waiting on a synchronous transaction but cannot begin a second synchronous transaction because of the activity associated with the first.

During an asynchronous transaction, this function returns after the transaction is begun, passing a transaction identifier for reference. When the server's DDE callback function finishes processing an asynchronous transaction, the system sends an `XTYP_XACT_COMPLETE` transaction to the client. This transaction provides the client with the results of the asynchronous transaction that it initiated by calling the `mdlWin32_DdeClientTransaction` function. A client application can choose to abandon an asynchronous transaction by calling the `mdlWin32_DdeAbandonTransaction` function.

pData points to the beginning of the data that the client needs to pass to the server. Optionally, an application can specify a data handle (`HDEEDATA`) to pass to the server, in which case the *cbData* parameter should be set to `0xFFFFFFFF`. This parameter is required only if the *wType* parameter is `XTYP_EXECUTE` or `XTYP_POKE`. Otherwise, this parameter should be `NULL`.

cbData specifies the length, in bytes, of the data pointed to by the *pData* parameter. A value of `0xFFFFFFFF` indicates that *pData* is a data handle that identifies the data being sent.

hConv identifies the conversation in which the transaction is to take place.

hszItem identifies the data item for which data is being exchanged during the transaction. This handle must have been created by a previous call to

the `mdlWin32_DdeCreateStringHandle` function. This parameter is ignored and should be set to `NULL` if the *uType* parameter is `XTYP_EXECUTE`.

wFmt specifies a standard clipboard format in which the data item is being submitted or requested. If the transaction specified by the *uType* parameter does not pass data or is `XTYP_EXECUTE`, this parameter should be set to zero.

wType specifies the transaction type and can have of the following values:

wType value	meaning
<code>XTYP_ADVSTART</code>	<p>Begins an advise loop. Any number of distinct advise loops can exist within a conversation. An application can alter the advise loop type by combining the <code>XTYP_ADVSTART</code> transaction type with one or more of the following flags:</p> <p><code>XTYPF_NODATA</code> Instructs the server to notify the client of any data changes without actually sending the data. This flag gives the client the option of ignoring the notification or requesting the changed data from the server.</p> <p><code>XTYPF_ACKREQ</code> Instructs the server to wait until the client acknowledges that it received the previous data item before sending the next data item. This flag prevents a fast server from sending data faster than the client can process it.</p>
<code>XTYP_ADVSTOP</code>	Ends an advise loop.
<code>XTYP_EXECUTE</code>	Begins an execute transaction.
<code>XTYP_POKE</code>	Begins a poke transaction.
<code>XTYP_REQUEST</code>	Begins a request transaction.

dwTimeout specifies the maximum length of time (milliseconds) that the client will wait for a response from the server application in a synchronous transaction. This parameter should be set to `TIMEOUT_ASYNC` for asynchronous transactions.

pdwResult points to a variable that receives the result of the transaction. An application that does not check the result can set this value to `NULL`. For synchronous transactions, the low-order word of this variable will contain any applicable DDE_ flags resulting from the transaction. This provides support for existing applications dependent on `DDE_APPSTATUS` bits. (It is recommended that applications no longer use these bits because they may not be supported in future versions). For asynchronous transactions, this variable is filled with a unique transaction identifier for use with the `mdlWin32_DdeAbandonTransaction` function and the `XTYP_XACT_COMPLETE` transaction.

Returns mdlWin32_DdeClientTransaction return value is a data handle that identifies the data for successful synchronous transactions in which the client expects data from the server. The return value is TRUE for successful asynchronous transactions and for synchronous transactions in which the client does not expect data. The return value is FALSE for all unsuccessful transactions.

Use the mdlWin32_DdeGetLastError function to retrieve the error value, which may be one of the following: DMLERR_ADVACKTIMEOUT, DMLERR_BUSY, DMLERR_DATAACKTIMEOUT, DMLERR_DLL_NOT_INITIALIZED, DMLERR_EXECACKTIMEOUT, DMLERR_INVALIDPARAMETER, DMLERR_MEMORY_ERROR, DMLERR_NO_CONV_ESTABLISHED, DMLERR_NO_ERROR, DMLERR_NOTPROCESSED, DMLERR_POKEACKTIMEOUT, DMLERR_POSTMSG_FAILED, DMLERR_REENTRANCY, DMLERR_SERVER_DIED, DMLERR_UNADVACKTIMEOUT.

See Also mdlWin32_DdeAbandonTransaction, mdlWin32_DdeAccessData, mdlWin32_DdeConnect, mdlWin32_DdeConnectList, mdlWin32_DdeCreateStringHandle.

mdlWin32_DdeCmpStringHandles

```
#include <w32dde.h> /* For MDL-DDEML constants. */
#include <msw32dde.fdf> /* For MDL-DDEML prototypes. */

int mdlWin32_DdeCmpStringHandles /* <= -1,0,1 for <=,=,> */
(
    HSZ    hsz1,      /* => String handle to compare */
    HSZ    hsz2       /* => String handle to compare */
);
```

Description The mdlWin32_DdeCmpStringHandles function compares the values of two string handles. The value of a string handle is not related to the case of the associated string. An application that needs to do a case-sensitive comparison of two string handles should compare the string handles directly; however, this is strongly discouraged. An application should use mdlWin32_DdeCmpStringHandles for all other comparisons to preserve the case-insensitive nature of dynamic data exchange (DDE).

hsz1 and *hsz2* specify the second string handles to compare.

Returns mdlWin32_DdeCmpStringHandles returns one of the following values:

value	meaning
-1	The value of <i>hsz1</i> is either 0 or less than the value of <i>hsz2</i> .
0	The values of <i>hsz1</i> and <i>hsz2</i> are equal (both can be 0).
1	The value of <i>hsz2</i> is either 0 or less than the value of <i>hsz1</i> .

See Also mdlWin32_DdeAccessData, mdlWin32_DdeCreateStringHandle, mdlWin32_DdeFreeStringHandle.

mdlWin32_DdeConnect

```
#include <w32dde.h> /* For MDL-DDEML constants. */
#include <msw32dde.fdf> /* For MDL-DDEML prototypes. */

HCONV mdlWin32_DdeConnect /* <= Conversation handle */
(
    DWORD          idInst,          /* => Process instance */
    HSZ            hszService,      /* => Service string handle */
    HSZ            hszTopic,        /* => Topic string handle */
    PCONVCONTEXT pCC                /* => Restore conversation context */
);
```

Description The `mdlWin32_DdeConnect` function establishes a conversation with a server application that supports the specified service name and topic name pair. If more than one such server exists, the system selects only one.

The client application should not make assumptions regarding which server will be selected. If an instance-specific service name is specified in *hszService*, a conversation will be established only with the specified instance. (Instance-specific service names are passed to an application's DDE callback function (i.e., `userWin32_DdeUserCallback`) during the `XTYP_REGISTER` and `XTYP_UNREGISTER` transactions).

All members of the default `CONVCONTEXT` data structure are set to zero except `cb`, which specifies the size of the structure, and *iCodePage*, which specifies `CP_WINANSI` (the default code page).

idInst specifies the application-instance identifier obtained by a previous call to the `mdlWin32_DdeInitialize` function.

hszService specifies the string handle service name of the server application to which a conversation is to be established. This handle must have been created by a previous call to the `mdlWin32_DdeCreateStringHandle` function. If this parameter is `NULL`, a conversation will be established with any available server.

hszTopic specifies the string handle topic name of the server to which a conversation is to be established. This handle must have been created by a previous call to the `mdlWin32_DdeCreateStringHandle` function. If this parameter is `NULL`, a conversation on any topic supported by the selected server will be established.

pCC points to the `CONVCONTEXT` structure that contains conversation context information. If this parameter is `NULL`, the server receives the default `CONVCONTEXT` structure during the `XTYP_CONNECT` or `XTYP_WILDCONNECT` transaction.

Returns `mdlWin32_DdeConnect` returns the value of the handle of the established conversation; otherwise, it is `NULL`. Use the `mdlWin32_DdeGetLastError` function to retrieve the error value, which may be one of the following:

```
DMLERR_DLL_NOT_INITIALIZED, DMLERR_INVALIDPARAMETER,
DMLERR_NO_CONV_ESTABLISHED, DMLERR_NO_ERROR.
```

See Also mdlWin32_DdeConnectList, mdlWin32_DdeCreateStringHandle, mdlWin32_DdeDisconnect, mdlWin32_DdeDisconnectList, mdlWin32_DdeInitialize, CONVCONTEXT, XTYP_CONNECT, XTYP_REGISTER, XTYP_UNREGISTER.

mdlWin32_DdeConnectList

```
#include <w32dde.h> /* For MDL-DDEML constants */
#include <msw32dde.fdf> /* For MDL-DDEML prototypes */

HCONVLIST mdlWin32_DdeConnectList
(
    DWORD          idInst,          /* => Instance id */
    HSZ            hszService,      /* => Service string handle */
    HSZ            hszTopic,        /* => Topic string handle */
    HCONVLIST      hConvList,      /* => conversation list handle */
    PCONVCONTEXT pCC               /* => context data structure */
);
```

Description mdlWin32_DdeConnectList establishes a conversation with all server applications that support the specified service and topic name pair. An application can also use this function to enumerate a list of conversation handles by passing in an existing conversation handle. During enumeration, the DDEML removes the handles of any terminated conversations from the conversation list. The resulting conversation list contains the handles of all conversations currently established that support the specified service and topic name pair. An application must free the conversation-list handle returned by this function, regardless of whether any conversation handles within the list are active. To free the handle, an application can call mdlWin32_DdeDisconnectList. All members of the default CONVCONTEXT data structure are set to zero except cb, which specifies the size of the structure, and iCodePage, which specifies CP_WINANSI (the default code page).

idInst specifies the application-instance identifier obtained by a previous call to mdlWin32_DdeInitialize.

hszService identifies the string that specifies the service name of the server application with which a conversation is to be established. If this parameter is NULL, the system will attempt to establish conversations with all available servers that support the specified topic name.

hszTopic identifies the string specifying the name of the topic on which a conversation is to be established. This handle must have been created by a previous call to mdlWin32_DdeCreateStringHandle. If this parameter is NULL, the system will attempt to establish conversations on all topics supported by the selected server (or servers).

hConvList identifies the conversation list to be enumerated. This parameter should be set to NULL if a new conversation list is to be established.

pCC points to the CONVCONTEXT structure containing conversation-context information. If this parameter is NULL, the server receives the default

CONVCONTEXT structure during the XTP_CONNECT or XTP_WILDCONNECT transaction.

Returns mdlWin32_DdeConnectList returns the handle of a new conversation list if successful, and NULL otherwise. The handle of the old conversation list is no longer valid. Use mdlWin32_DdeGetLastError to retrieve the error value, which may be one of the following: DMLERR_DLL_NOT_INITIALIZED, DMLERR_INVALID_PARAMETER, DMLERR_NO_CONV_ESTABLISHED, DMLERR_NO_ERROR, DMLERR_SYS_ERROR.

See Also mdlWin32_DdeConnect, mdlWin32_DdeCreateStringHandle, mdlWin32_DdeDisconnect, mdlWin32_DdeDisconnectList, mdlWin32_DdeInitialize, mdlWin32_DdeQueryNextServer, CONVCONTEXT, XTP_CONNECT.

mdlWin32_DdeCreateDataHandle

```
#include <w32dde.h> /* For MDL-DDEML constants */
#include <msw32dde.fdf> /* For MDL-DDEML prototypes */

HDDADATA mdlWin32_DdeCreateDataHandle
(
    DWORD    idInst,          /* => Instance id */
    LPBYTE   pSrcBuf,         /* => Source data buffer */
    DWORD    cbInitData,      /* => Source data size */
    DWORD    cbOffSrcBuf,     /* => Offset into source buffer */
    HSZ      hszItem,         /* => Item name string handle */
    UINT     wFmt,            /* => Clipboard format */
    UINT     afCmd            /* => Creation flag */
);
```

Description mdlWin32_DdeCreateDataHandle creates a DDE data object and fills the object with the data pointed to by the *pSrcBuf* parameter. An application uses this function during transactions that involve passing data to the partner application. Any locations in the DDE data object that are not filled are undefined. After a data handle has been used as a parameter in another DDEML function or has been returned by a DDE callback function, the handle may be used only for read access to the DDE data object identified by the handle.

idInst specifies the application-instance identifier obtained by a previous call to mdlWin32_DdeInitialize.

pSrcBuf points to a buffer that contains data to be copied to the DDE data object. If this parameter is NULL, no data is copied to the object.

cbInitData specifies the amount, in bytes, of memory to allocate for the DDE data object. If this parameter is zero, the *pSrcBuf* parameter is ignored.

cbOffSrcBuf specifies an offset, in bytes, from the beginning of the buffer pointed to by the *pSrcBuf* parameter. The data beginning at this offset is copied from the buffer to the DDE data object.

hszItem identifies the string that specifies the data item corresponding to the DDE data object. This handle must have been created by a previous call to `mdlWin32_DdeCreateStringHandle`. If the data handle is to be used in an `XTYP_EXECUTE` transaction, this parameter must be set to `NULL`.

wFmt specifies the standard clipboard format number of the data.

afCmd specifies the creation flags. This parameter can be `HDATA_APPOWNED`, which specifies that the application that calls `mdlWin32_DdeCreateDataHandle` will own the data handle that this function creates. This makes it possible for the application to share the data handle with other DDEML applications instead of creating a separate handle to pass to each application. If this flag is set, the application must eventually free the shared memory object associated with this handle by using `mdlWin32_DdeFreeDataHandle`. If this flag is not set, after the data handle is returned by the application's DDE callback function (i.e., `userWin32_DdeUserCallback`) or used as a parameter in another DDE Management Library function, the handle becomes invalid in the application that creates the handle.

Returns `mdlWin32_DdeCreateDataHandle` returns a data handle; otherwise, it is `NULL`. Use the `mdlWin32_DdeGetLastError` function to retrieve the error value, which may be one of the following: `DMLERR_DLL_NOT_INITIALIZED`, `DMLERR_INVALIDPARAMETER`, `DMLERR_MEMORY_ERROR`, `DMLERR_NO_ERROR`.

See Also `mdlWin32_DdeAccessData`, `mdlWin32_DdeFreeDataHandle`, `mdlWin32_DdeGetData`, `mdlWin32_DdeInitialize`, `XTYP_EXECUTE`.

mdlWin32_DdeCreateStringHandle

```
#include <w32dde.h> /* For MDL-DDEML constants */
#include <msw32dde.fdf> /* For MDL-DDEML prototypes */

HSZ mdlWin32_DdeCreateStringHandle
(
    DWORD    idInst,          /* => Instance id */
    LPSTR    psz,             /* => Source char string */
    int      iCodePage        /* => CP_WINANSI or CP_UNICODE */
);
```

Description The `mdlWin32_DdeCreateStringHandle` function creates a handle that identifies the string pointed to by the *psz* parameter. A client or server application can pass the string handle as a parameter to other DDEML functions. When an application has either created a string handle or received one in the callback function (i.e., `userWin32_DdeUserCallback`) and has used the `mdlWin32_DdeKeepStringHandle` function to keep it, the application must free that string handle when it is no longer needed. An instance-specific string handle is not mappable from string handle to

string to string handle again. There can be two unique string handles for the same underlining string.

idInst specifies the application-instance identifier obtained by a previous call to the `mdlWin32_DdeInitialize` function.

psz points to a buffer that contains the NULL terminated string for which a handle is to be created. This string may be any length.

codepage specifies the code page used to render the string. This value should be either `CP_WINANSI` (the default code page) or `CP_WINUNICODE`, depending on whether the ANSI or Unicode version of the `mdlWin32_DdeInitialize` function was called by the client application.

Returns `mdlWin32_DdeCreateStringHandle` returns a string handle; otherwise, it is NULL. Use the `mdlWin32_DdeGetLastError` function to retrieve the error value, which may be one of the following: `DMLERR_INVALIDPARAMETER`, `DMLERR_NO_ERROR`, `DMLERR_SYS_ERROR`.

See Also `mdlWin32_DdeAccessData`, `mdlWin32_DdeCmpStringHandles`, `mdlWin32_DdeFreeStringHandle`, `mdlWin32_DdeInitialize`, `mdlWin32_DdeKeepStringHandle`, `mdlWin32_DdeQueryString`.

mdlWin32_DdeDisconnect

```
#include <w32dde.h> /* For MDL-DDEML constants */
#include <msw32dde.fdf> /* For MDL-DDEML prototypes */

BOOL mdlWin32_DdeDisconnect
(
    HCONV    hConv      /* => Conversation handle */
);
```

Description The `mdlWin32_DdeDisconnect` function terminates a conversation started by either the `mdlWin32_DdeConnect` or `mdlWin32_DdeConnectList` function and invalidates the given conversation handle. Any incomplete transactions started before calling `mdlWin32_DdeDisconnect` are immediately abandoned. The `XTYP_DISCONNECT` transaction type is sent to the callback function (i.e., `userWin32_DdeUserCallback`) of the partner in the conversation. Generally, only client applications need to terminate conversations.

hConv identifies the active conversation to be terminated.

Returns `mdlWin32_DdeDisconnect` returns If the function succeeds, the return value is `TRUE`, otherwise, it is `FALSE`. Use the `mdlWin32_DdeGetLastError` function to retrieve the error value, which may be one of the following: `DMLERR_DLL_NOT_INITIALIZED`, `DMLERR_NO_CONV_ESTABLISHED`, `DMLERR_NO_ERROR`.

See Also `mdlWin32_DdeConnect`, `mdlWin32_DdeConnectList`, `mdlWin32_DdeDisconnectList`, `XTYP_DISCONNECT`.

mdlWin32_DdeDisconnectList

```
#include <w32dde.h> /* For MDL-DDEML constants */
#include <msw32dde.fdf> /* For MDL-DDEML prototypes */

BOOL mdlWin32_DdeDisconnectList
(
    HCONVLIST    hConvList
);
```

Description The `mdlWin32_DdeDisconnectList` function destroys the given conversation list and terminates all conversations associated with the list. An application can use the `mdlWin32_DdeDisconnect` function to terminate individual conversations in the list.

hConvList identifies the conversation list. This handle must have been created by a previous call to the `mdlWin32_DdeConnectList` function.

Returns `mdlWin32_DdeDisconnectList` returns `TRUE` if the function succeeds; otherwise, it is `FALSE`. Use the `mdlWin32_DdeGetLastError` function to retrieve the error value, which may be one of the following: `DMLERR_DLL_NOT_INITIALIZED`, `DMLERR_INVALIDPARAMETER`, `DMLERR_NO_ERROR`.

See Also `mdlWin32_DdeConnect`, `mdlWin32_DdeConnectList`, `mdlWin32_DdeDisconnect`, `XTYP_DISCONNECT`.

mdlWin32_DdeEnableCallback

```
#include <w32dde.h> /* For MDL-DDEML constants */
#include <msw32dde.fdf> /* For MDL-DDEML prototypes */

BOOL mdlWin32_DdeEnableCallback
(
    DWORD    idInst,          /* => Instance id */
    HCONV    hConv,          /* => Conversation handle */
    UINT     wCmd             /* => EC_ flag */
);
```

Description `mdlWin32_DdeEnableCallback` enables or disables transactions for a specific conversation or for all conversations that the calling application currently has established. After disabling transactions for a conversation, the system places the transactions for that conversation in a transaction queue associated with the application. The application should reenab the conversation as soon as possible to avoid losing queued transactions.

An application can disable transactions for a specific conversation by returning `CBR_BLOCK` from its DDE callback function (i.e., `userWin32_DdeUserCallback`). When the conversation is reenabled by `mdlWin32_DdeEnableCallback`, the system generates the same transaction as was in process when the conversation was disabled.

Using the `EC_QUERYWAITING` flag does not change the enable state of the conversation and does not cause transactions to be issued within the context of the call to `mdlWin32_DdeEnableCallback`.

idInst specifies the application-instance identifier obtained by a previous call to the `mdlWin32_DdeInitialize` function.

hConv identifies the conversation to enable or disable. If this parameter is `NULL`, the function affects all conversations.

wCmd specifies the function code. This parameter can be one of the following values:

value	description
<code>EC_ENABLEALL</code>	Enables all transactions for the specified conversation.
<code>EC_ENABLEONE</code>	Enables one transaction for the specified conversation.
<code>EC_DISABLE</code>	Disables all blockable transactions for the specified conversation. A server application can disable the following transactions: <code>XTYP_ADVSTART</code> , <code>XTYP_ADVSTOP</code> , <code>XTYP_EXECUTE</code> , <code>XTYP_POKE</code> , <code>XTYP_REQUEST</code> . A client application can disable the following transactions: <code>XTYP_ADVDATA</code> , <code>XTYP_XACT_COMPLETE</code> .
<code>EC_QUERYWAITING</code>	Determines whether any transactions are in the queue for the specified conversation.

Returns `mdlWin32_DdeEnableCallback` returns `TRUE` if the function succeeds, the return value is `TRUE`; otherwise, it is `FALSE`. If the *uCmd* parameter is `EC_QUERYWAITING` and the queue contains one or more transactions, the return value is `TRUE`; otherwise, it is `FALSE`. Use `mdlWin32_DdeGetLastError` to retrieve the error value, which may be `DMLERR_DLL_NOT_INITIALIZED`, `DMLERR_NO_ERROR` or `DMLERR_INVALIDPARAMETER`.

See Also `mdlWin32_DdeConnect`, `mdlWin32_DdeConnectList`, `mdlWin32_DdeDisconnect`, `mdlWin32_DdeInitialize`.

mdlWin32_DdeFreeDataHandle

```
#include <w32dde.h> /* For MDL-DDEML constants. */
#include <msw32dde.fdf> /* For MDL-DDEML prototypes. */

BOOL mdlWin32_DdeFreeDataHandle
(
    HDDEDATA    hData          /* => DDE object handle */
);
```

Description The `mdlWin32_DdeFreeDataHandle` function frees a DDE data object and deletes the data handle associated with the object. An application must call `mdlWin32_DdeFreeDataHandle` under the following circumstances:

1. To free a DDE data object that the application allocated by calling the `mdlWin32_DdeCreateDataHandle` function if the object's data handle was never passed by the application to another MDL-DDEML function.
2. To free a DDE data object that the application allocated by specifying the `HDATA_APPOWNED` flag in a call to the `mdlWin32_DdeCreateDataHandle` function.
3. To free a DDE data object whose handle the application received from the `mdlWin32_DdeClientTransaction` function.

The system automatically frees an unowned object when its handle is returned by a DDE callback function (i.e., `userWin32_DdeUserCallback`) or used as a parameter in a DDEML function.

hdata identifies the DDE data object to be freed. This handle must have been created by a previous call to the `mdlWin32_DdeCreateDataHandle` function or returned by the `mdlWin32_DdeClientTransaction` function.

Returns `mdlWin32_DdeFreeDataHandle` returns `TRUE` if the function succeeds; otherwise, it is `FALSE`. Use the `mdlWin32_DdeGetLastError` function to retrieve the error value, which may be one of the following: `DMLERR_INVALIDPARAMETER`, `DMLERR_NO_ERROR`.

See Also `mdlWin32_DdeAccessData`, `mdlWin32_DdeCreateDataHandle`.

mdlWin32_DdeFreeStringHandle

```
#include <w32dde.h> /* For MDL-DDEML constants */
#include <msw32dde.fdf> /* For MDL-DDEML prototypes */

BOOL mdlWin32_DdeFreeStringHandle /* <= TRUE = success */
(
    DWORD    idInst,    /* => Instance id */
    HSZ      hsz        /* => String handle to save */
);
```

Description The `mdlWin32_DdeFreeStringHandle` function frees a string handle in the calling application. An application can free string handles that it creates with the `mdlWin32_DdeCreateStringHandle` function but should not free those that the system passed to the application's DDE callback function (i.e., `userWin32_DdeUserCallback`) or those returned in the `CONVINFO` structure by the `mdlWin32_DdeQueryConvInfo` function.

idInst specifies the application-instance identifier obtained by a previous call to the `mdlWin32_DdeInitialize` function.

hsz identifies the string handle to be freed. This handle must have been created by a previous call to the `mdlWin32_DdeCreateStringHandle` function.

Returns `mdlWin32_DdeFreeStringHandle` returns TRUE for success; FALSE otherwise.

See Also `mdlWin32_DdeCmpStringHandles`, `mdlWin32_DdeCreateStringHandle`, `mdlWin32_DdeInitialize`, `mdlWin32_DdeKeepStringHandle`, `mdlWin32_DdeQueryString`.

mdlWin32_DdeGetData

```
#include <w32dde.h> /* For MDL-DDEML constants */
#include <msw32dde.fdf> /* For MDL-DDEML prototypes */

DWORD mdlWin32_DdeGetData
(
    HDEDEDATAhData,      /* => DDE object handle */
    LPBYTE pDst,          /* <=> Destination buffer */
    DWORD cbMax,          /* => Destination buffer size */
    DWORD cbOff           /* => Offset into buffer */
);
```

Description The `mdlWin32_DdeGetData` function copies data from the given DDE data object to the specified local buffer.

hData identifies the DDE data object that contains the data to copy.

pDst points to the buffer that receives the data. If this parameter is NULL, the `mdlWin32_DdeGetData` function returns the amount, in bytes, of data that would be copied to the buffer.

cbMax specifies the maximum amount, in bytes, of data to copy to the buffer pointed to by the *pDst* parameter. Typically, this parameter specifies the length of the buffer pointed to by *pDst*.

cbOff specifies an offset within the DDE data object. Data is copied from the object beginning at this offset.

Returns If *pDst* parameter points to a buffer, `mdlWin32_DdeGetData` returns the lessor of the *cbMax* parameter or the size, in bytes, of the memory object associated with the data handle. If the *pDst* parameter is NULL, the return value is the size, in bytes, of the memory object associated with the data handle. Use `mdlWin32_DdeGetLastError` to retrieve the error value, which may be one of the following: `DMLERR_DLL_NOT_INITIALIZED`, `DMLERR_INVALID_HDEDEDATA`, `DMLERR_INVALIDPARAMETER`, `DMLERR_NO_ERROR`.

See Also `mdlWin32_DdeAccessData`, `mdlWin32_DdeCreateDataHandle`, `mdlWin32_DdeFreeDataHandle`.

mdlWin32_DdeGetLastError

```
#include <w32dde.h> /* For MDL-DDEML constants */
#include <msw32dde.fdf> /* For MDL-DDEML prototypes */

UINT mdlWin32_DdeGetLastError
(
    DWORD    idInst;          /* instance identifier */
);
```

Description The mdlWin32_DdeGetLastError function returns the most recent error value set by a MDL-DDEML function and resets the error value to DMLERR_NO_ERROR.

idInst specifies the application-instance identifier obtained by a previous call to the mdlWin32_DdeInitialize function.

mdlWin32_DdeGetLastError returns one of the following values:

flag	description
DMLERR_ADVACKTIMEOUT	A request for a synchronous advise transaction has timed out.
DMLERR_BUSY	The response to the transaction caused the DDE_FBUSY bit to be set.
DMLERR_DATAACKTIMEOUT	A request for a synchronous data transaction has timed out.
DMLERR_DLL_NOT_INITIALIZED	A DDEML function was called without first calling the mdlWin32_DdeInitialize function, or an invalid instance identifier was passed to a DDEML function.
DMLERR_DLL_USAGE	An application initialized as APPCLASS_MONITOR has attempted to perform a DDE transaction, or an application initialized as APPCMD_CLIENTONLY has attempted to perform server transactions.
MLERR_EXEACKTIMEOUT	A request for a synchronous execute transaction has timed out.

flag	description
DMLERR_INVALIDPARAMETER	A parameter failed to be validated by the DDEML. Some of the possible causes are: <ul style="list-style-type: none"> • The application used a data handle initialized with a different item-name handle than that required by the transaction. • The application used a data handle that was initialized with a different clipboard data format than that required by the transaction. • The application used a client-side conversation handle with a server-side function or vice versa. • The application used a freed data handle or string handle. • More than one instance of the application used the same object.
DMLERR_LOW_MEMORY	A DDEML application has created a prolonged race condition (where the server application outruns the client), causing large amounts of memory to be consumed.
DMLERR_MEMORY_ERROR	A memory allocation failed.
DMLERR_NO_CONV_ESTABLISHED	A client's attempt to establish a conversation has failed.
DMLERR_NOTPROCESSED	A transaction failed.
DMLERR_POKEACKTIMEOUT	A request for a synchronous poke transaction has timed out.
DMLERR_POSTMSG_FAILED	An internal call to the Windows NT PostMessage function has failed.
DMLERR_REENTRANCY	An application instance with a synchronous transaction already in progress attempted to initiate another synchronous transaction, or <code>mdlWin32_DdeEnableCallback</code> was called from within a DDEML callback function, i.e., <code>userWin32_DdeUserCallback</code> .
DMLERR_SERVER_DIED	A server-side transaction was attempted on a conversation that was terminated by the client, or the server terminated before completing a transaction.
DMLERR_SYS_ERROR	An internal error has occurred in the DDEML.
DMLERR_UNADVACKTIMEOUT	A request to end an advise transaction has timed out.
DMLERR_UNFOUND_QUEUE_ID	An invalid transaction identifier was passed to a function. Once the application has returned from an <code>XTYP_XACT_COMPLETE</code> callback, the transaction identifier for that callback is no longer valid.

See Also `mdlWin32_DdeInitialize`.

mdlWin32_DdeInitialize

```
#include <w32dde.h> /* For MDL-DDEML constants. */
#include <msw32dde.fdf> /* For MDL-DDEML prototypes. */

UINT mdlWin32_DdeInitialize
(
    LPDWORD      lpIdInst,      /* <=> instance identifier */
    PFNCALLBACK  pfnCallback,   /* => callback function */
    DWORD        afCmd,         /* => command and filter flags */
    DWORD        uRes           /* => reserved */
);
```

Description `mdlWin32_DdeInitialize` registers an asynchronous DDE function with MicroStation. An application must call this function before calling any other MDL-DDEML functions.

A DDE monitoring application should not attempt to perform DDE (establish conversations, issue transactions, and so on) within the context of the same application instance.

A synchronous transaction will fail with a `DMLERR_REENTRANCY` error if any instance of the same task has a synchronous transaction already in progress.

The `CBF_FAIL_ALLSVRXACTIONS` flag causes the DDEML to filter all server transactions and may be changed by a subsequent call to `mdlWin32_DdeInitialize`.

The `APPCMD_CLIENONLY` flag prevents the DDEML from creating key resources for the server and cannot be changed by a subsequent call to `mdlWin32_DdeInitialize`.

There is an ANSI version and a Unicode version of this function. The version called determines the type of the window procedures used to control DDE conversations (ANSI or Unicode), and the default value for the *iCodePage* member of the `CONVCONTEXT` structure (`CP_WINANSI` or `CP_WINUNICODE`).

pidInst returns a pointer to the application-instance identifier. At initialization, this parameter should point to 0. However, this parameter must not be `NULL`. If the function succeeds, this parameter points to the instance identifier for the MDL application. This value should be passed as the *idInst* parameter in all other DDEML functions that require it. If *pidInst* points to a nonzero value, this implies a reinitialization of the DDEML. In this case, *pidInst* must point to a valid application-instance identifier.

pfnCallback points to the application-defined DDE callback function. This function processes DDE transactions sent by other applications. For more

information, see the description of the `userWin32_DdeUserCallback` callback function.

afCmd specifies a logical expression of `APPCMD_`, `CBF_` and `MF_` flags. The `APPCMD_` flags provide special instructions to `mdlWin32_DdeInitialize`. The `CBF_` flags set filters that prevent specific types of transactions from reaching the callback function. The `MF_` flags specify the types of DDE activity that a DDE monitoring application will monitor. Using these flags enhances the performance of a DDE application by eliminating unnecessary calls to the callback function. This parameter can be a combination of the following flags:

flag	description
<code>APPCCLASS_MONITOR</code>	Makes it possible for the application to monitor DDE activity in the system. This flag is for use by DDE monitoring applications. The application specifies the types of DDE activity to monitor by combining one or more monitor flags with the <code>APPCCLASS_MONITOR</code> flag.
<code>APPCCLASS_STANDARD</code>	Registers the application as a standard (nonmonitoring) DDEML application. Most applications will use this flag.
<code>APPCMD_CLIENTONLY</code>	Prevents the application from becoming a server in a DDE conversation. The application can be only a client. This flag reduces resource consumption by the DDEML. It includes the functionality of the <code>CBF_FAIL_ALLSVRXACTIONS</code> flag.
<code>APPCMD_FILTERINITS</code>	Prevents the DDEML from sending <code>XYP_CONNECT</code> and <code>XYP_WILDCONNECT</code> transactions to the application until the application has created its string handles and registered its service names or has turned off filtering by a subsequent call to the <code>mdlWin32_DdeNameService</code> or <code>mdlWin32_DdeInitialize</code> functions. This flag is always in effect when an application calls <code>mdlWin32_DdeInitialize</code> for the first time, regardless of whether the application specifies this flag. On subsequent calls to <code>mdlWin32_DdeInitialize</code> , not specifying this flag turns off the application's service-name filters; specifying this flag turns on the application's service-name filters.
<code>CBF_FAIL_ALLSVRXACTIONS</code>	Prevents the callback function from receiving server transactions. The system will return <code>DDE_FNOTPROCESSED</code> to each client that sends a transaction to this application. This flag is equivalent to combining all <code>CBF_FAIL_</code> flags.
<code>CBF_FAIL_ADVISES</code>	Prevents the callback function from receiving <code>XYP_ADVSTART</code> and <code>XYP_ADVSTOP</code> transactions. The system will return <code>DDE_FNOTPROCESSED</code> to each client that sends an <code>XYP_ADVSTART</code> or <code>XYP_ADVSTOP</code> transaction to the server.

flag	description
CBF_FAIL_CONNECTIONS	Prevents the callback function from receiving XTYP_CONNECT and XTYP_WILDCONNECT transactions.
CBF_FAIL_EXECUTES	Prevents the callback function from receiving XTYP_EXECUTE transactions. The system will return DDE_FNOTPROCESSED to a client that sends an XTYP_EXECUTE transaction to the server.
CBF_FAIL_POKEs	Prevents the callback function from receiving XTYP_POKE transactions. The system will return DDE_FNOTPROCESSED to a client that sends an XTYP_POKE transaction to the server.
CBF_FAIL_REQUESTS	Prevents the callback function from receiving XTYP_REQUEST transactions. The system will return DDE_FNOTPROCESSED to a client that sends an XTYP_REQUEST transaction to the server.
CBF_FAIL_SELFCONNECTIONS	Prevents the callback function from receiving XTYP_CONNECT transactions from the application's own instance. This prevents an application from establishing a DDE conversation with its self. An application should use this flag if it needs to communicate with other instances of itself but not with itself.
CBF_SKIP_ALLNOTIFICATIONS	Prevents the callback function from receiving any notifications. This flag is equivalent combining all CBF_SKIP_ flags.
CBF_SKIP_CONNECT_CONFIRMS	Prevents the callback function from receiving XTYP_CONNECT_CONFIRM notifications.
CBF_SKIP_DISCONNECTS	Prevents the callback function from receiving XTYP_DISCONNECT notifications.
CBF_SKIP_REGISTRATIONS	Prevents the callback function from receiving XTYP_REGISTER notifications.
CBF_SKIP_UNREGISTRATIONS	Prevents the callback function from receiving XTYP_UNREGISTER notifications
MF_CALLBACKS	Notifies the callback function whenever a transaction is sent to any DDE callback function in the system
MF_CONV	Notifies the callback function whenever a conversation is established or terminated.
MF_ERRORS	Notifies the callback function whenever a DDE error occurs.
MF_HSZ_INFO	Notifies the callback function whenever a DDE application creates, frees or increments the use count of a string handle or whenever a string handle is freed as a result of a call to mdlWin32_DdeUninitialize.

flag	description
MF_LINKS	Notifies the callback function whenever an advise loop is started or ended.
MF_POSTMSGs	Notifies the callback function whenever the system or an application posts a DDE message.
MF_SENDMSGs	Notifies the callback function whenever the system or an application sends a DDE message.

uRes is reserved and must be set to 0.

Returns `mdlWin32_DdeInitialize` returns one of the following: `DMLERR_DLL_USAGE`, `DMLERR_INVALIDPARAMETER`, `DMLERR_NO_ERROR`, `DMLERR_SYS_ERROR`.

See Also `mdlWin32_DdeClientTransaction`, `mdlWin32_DdeConnect`, `mdlWin32_DdeCreateDataHandle`, `mdlWin32_DdeEnableCallback`, `mdlWin32_DdeNameService`, `mdlWin32_DdePostAdvise`, `mdlWin32_DdeUninitialize`.

mdlWin32_DdeKeepStringHandle

```
#include <w32dde.h> /* For MDL-DDEML constants. */
#include <msw32dde.fdf> /* For MDL-DDEML prototypes. */

BOOL mdlWin32_DdeKeepStringHandle /* <= TRUE = success */
(
    DWORD    idInst,          /* => Instance id */
    HSZ      hsz              /* => String handle to save */
);
```

Description `mdlWin32_DdeKeepStringHandle` increments the usage count (by one) associated with the given handle. This function makes it possible for an application to save a string handle that was passed to the application's callback function. Otherwise, a string handle passed to the callback function is deleted when the callback function returns. This function should also be used to keep a copy of a string handle referenced by the `CONVINFO` structure returned by `mdlWin32_DdeQueryConvInfo`.

idInst specifies the application-instance identifier obtained by a previous call to `mdlWin32_DdeInitialize`.

hsz identifies the string handle to be saved.

Returns `mdlWin32_DdeKeepStringHandle` returns `TRUE` for success; `FALSE` otherwise.

See Also `mdlWin32_DdeCreateStringHandle`, `mdlWin32_DdeFreeStringHandle`, `mdlWin32_DdeInitialize`, `mdlWin32_DdeQueryConvInfo`, `mdlWin32_DdeQueryString`, and the `CONVINFO` structure in the include files.

mdlWin32_DdeNameService

```
#include <w32dde.h> /* For MDL-DDEML constants */
#include <msw32dde.fdf> /* For MDL-DDEML prototypes */

HDDADATA mdlWin32_DdeNameService
(
    DWORD    idInst,          /* => Instance id    */
    HSZ      hsz1,           /* => Service name string handle */
    HSZ      hszRes,         /* => Reserved, pass NULL */
    UINT     afCmd           /* => Service flags */
);
```

Description The `mdlWin32_DdeNameService` function registers or unregisters the service names that a DDE server supports. This function causes the system to send `XTYP_REGISTER` or `XTYP_UNREGISTER` transactions to other running client applications. A server application should call this function to register each service name that it supports and to unregister names that it previously registered but no longer supports. A server should also call this function to unregister its service names just before terminating. The service name identified by the *hsz1* parameter should be a base name (that is, the name should contain no instance-specific information). The system generates an instance-specific name and sends it along with the base name during the `XTYP_REGISTER` and `XTYP_UNREGISTER` transactions. The receiving applications can then connect to the specific application instance.

idInst specifies the application-instance identifier obtained by a previous call to `mdlWin32_DdeInitialize`.

hsz1 identifies the string that specifies the service name that the server is registering or unregistering. An application that is unregistering all of its service names should set this parameter to `NULL`.

hszRes is reserved and should be set to `NULL`.

afCmd specifies the service-name flags. This parameter can be one of the following values:

value	description
<code>DNS_REGISTER</code>	Registers the given service name.
<code>DNS_UNREGISTER</code>	Unregisters the given service name. If the <i>hsz1</i> parameter is <code>NULL</code> , all service names registered by the server will be unregistered.

value	description
DNS_FILTERON	Turns on service-name initiation filtering. This filter prevents a server from receiving XTYP_CONNECT transactions for service names that it has not registered. This is the default setting for this filter. If a server application does not register any service names, the application cannot receive XTYP_WILDCONNECT transactions.
DNS_FILTEROFF	Turns off service-name initiation filtering. If this flag is set, the server will receive an XTYP_CONNECT transaction whenever another DDE application calls the mdlWin32_DdeConnect function, regardless of the service name.

Returns mdlWin32_DdeNameService returns TRUE if the function succeeds; FALSE otherwise. Use mdlWin32_DdeGetLastError to retrieve the error value, which may be one of the following: DMLERR_DLL_NOT_INITIALIZED, DMLERR_DLL_USAGE, DMLERR_INVALIDPARAMETER, DMLERR_NO_ERROR.

See Also mdlWin32_DdeConnect, mdlWin32_DdeConnectList, mdlWin32_DdeInitialize, XTYP_REGISTER, XTYP_UNREGISTER.

mdlWin32_DdePostAdvise

```
#include <w32dde.h> /* For MDL-DDEML constants */
#include <msw32dde.fdf> /* For MDL-DDEML prototypes */

BOOL mdlWin32_DdePostAdvise
(
    DWORD    idInst,          /* => Instance id */
    HSZ      hszTopic,        /* => Topic name string handle */
    HSZ      hszItem          /* => Item name string handle */
);
```

Description mdlWin32_DdePostAdvise causes the system to send an XTYP_ADVREQ transaction to the calling (server) application's DDE callback function for each client that has an advise loop active on the specified topic or item name pair. A server application should call this function whenever the data associated with the topic or item name pair changes. A server that has nonenumerable topics or items should set the *hszTopic* and *hszItem* parameters to NULL so that the system will generate transactions for all active advise loops. The server's DDE callback function returns NULL for any advise loops that do not need to be updated. If a server calls mdlWin32_DdePostAdvise with a topic/item/format name set that includes the set currently being handled in a XTYP_ADVREQ callback, a stack overflow may result.

idInst specifies the application-instance identifier obtained by a previous call to mdlWin32_DdeInitialize.

hszTopic identifies a string that specifies the topic name. To send notifications for all topics with active advise loops, an application can set this parameter to NULL.

hszItem identifies a string that specifies the item name. To send notifications for all items with active advise loops, an application can set this parameter to NULL.

Returns `mdlWin32_DdePostAdvise` returns TRUE if function succeeds, otherwise, it is FALSE. Use `mdlWin32_DdeGetLastError` to retrieve the error value, which may be one of the following: `DMLERR_DLL_NOT_INITIALIZED`, `DMLERR_DLL_USAGE`, `DMLERR_NO_ERROR`.

See Also `mdlWin32_DdeInitialize`, `XTYP_ADVREQ`.

mdlWin32_DdeQueryConvInfo

```
#include <w32dde.h> /* For MDL-DDEML constants */
#include <msw32dde.fdf> /* For MDL-DDEML prototypes */

UINT mdlWin32_DdeQueryConvInfo
(
    HCONV          hConv,          /* => Conversation handle */
    DWORD          idTransaction, /* => XTYP_ or NULL for all */
    PCONVINFO      pConvInfo      /* <= CONVINFO with .cb loaded */
);
```

Description `mdlWin32_DdeQueryConvInfo` retrieves information about a DDE transaction and about the conversation in which the transaction takes place. An application should not free a string handle referenced by the `CONVINFO` structure. If an application needs to use one of these string handles, it should call the `mdlWin32_DdeKeepStringHandle` to create a copy of the handle for itself.

If *idTransaction* is set to `QID_SYNC`, the *hUser* member of the `CONVINFO` structure is associated with the conversation and can be used to hold data associated with the conversation. If *idTransaction* is the identifier of an asynchronous transaction, the *hUser* member is associated only with the current transaction and is valid only for the duration of the transaction.

hconv identifies the conversation.

idTransaction specifies the transaction. For asynchronous transactions, this parameter should be a transaction identifier returned by `mdlWin32_DdeClientTransaction`. For synchronous transactions, this parameter should be `QID_SYNC`.

pConvInfo points to the `CONVINFO` structure that will receive information about the transaction and conversation. The *cb* member of the `CONVINFO` structure must specify the length of the buffer allocated for the structure.

Returns `mdlWin32_DdeQueryConvInfo` returns number of bytes copied into the `CONVINFO` structure or zero for an error. Use `mdlWin32_DdeGetLastError` to retrieve the error value, which may be one of the following: `DMLERR_DLL_NOT_INITIALIZED`, `DMLERR_NO_CONV_ESTABLISHED`, `DMLERR_NO_ERROR`, `DMLERR_UNFOUND_QUEUE_ID`.

See Also mdlWin32_DdeConnect, mdlWin32_DdeConnectList, mdlWin32_DdeKeepStringHandle, mdlWin32_DdeQueryNextServer, CONVINFO.

mdlWin32_DdeQueryNextServer

```
HCONV mdlWin32_DdeQueryNextServer
(
HCONVLIST    hConvList,
HCONV        hConvPrev    /* => Conversation handle */
);
```

Description mdlWin32_DdeQueryNextServer obtains the next conversation handle in the given conversation list.

hConvList identifies the conversation list. This handle must have been created by a previous call to mdlWin32_DdeConnectList.

hConvPrev identifies the conversation handle previously returned by this function. If this parameter is NULL, this function returns the first conversation handle in the list.

Returns mdlWin32_DdeQueryNextServer returns the next conversation handle in the list. A NULL is returned when there are no more conversations.

See Also mdlWin32_DdeConnectList, mdlWin32_DdeDisconnectList.

mdlWin32_DdeQueryString

```
#include <w32dde.h> /* For MDL-DDEML constants */
#include <msw32dde.fdf> /* For MDL-DDEML prototypes */

DWORD mdlWin32_DdeQueryString
(
DWORD    idInst,        /* => Instance id    */
HSZ      hsz,           /* => String handle source */
LPSTR    psz,           /* => Char buffer destination */
DWORD    cchMax,        /* => destination size */
int      iCodePage      /* => CP_WINANSI or CP_UNICODE */
);
```

Description mdlWin32_DdeQueryString copies text associated with a string handle into a buffer. The string returned in the buffer is always NULL-terminated. If the string is longer than (*cchMax* - 1), only the first (*cchMax* - 1) characters of the string are copied. If the *lpz* parameter is NULL, this function obtains the length, in bytes, of the string associated with the string handle. The length does not include the terminating NULL character.

idInst specifies the application-instance identifier obtained by a previous call to mdlWin32_DdeInitialize.

hsz identifies the string to copy. This handle must have been created by a previous call to mdlWin32_DdeCreateStringHandle.

psz points to a buffer that receives the string. To obtain the length of the string, this parameter should be set to `NULL`.

cchMax specifies the length, in characters, of the buffer pointed to by the *lpz* parameter. If the string is longer than (*cchMax* - 1), it will be truncated. If the *lpz* parameter is set to `NULL`, this parameter is ignored.

codepage specifies the code page used to render the string. This value should be either `CP_WINANSI` or `CP_WINUNICODE`.

Returns `mdlWin32_DdeQueryString` returns 0 (meaning an error occurred) or a length. If *psz* specified a valid pointer, the return value is the length (in characters) of the returned text (not including the terminating `NULL` character). If the *psz* parameter specified a `NULL` pointer, the return value is the length of the text associated with the *hsz* parameter (not including the terminating `NULL` character).

See Also `mdlWin32_DdeCmpStringHandles`, `mdlWin32_DdeCreateStringHandle`, `mdlWin32_DdeFreeStringHandle`, `mdlWin32_DdeInitialize`.

mdlWin32_DdeReconnect

```
#include <w32dde.h> /* For MDL-DDEML constants */
#include <msw32dde.fdf> /* For MDL-DDEML prototypes */

HCONV mdlWin32_DdeReconnect
(
    HCONV hConv /* => Conversation handle */
);
```

Description `mdlWin32_DdeReconnect` allows a client application to attempt to reestablish a conversation with a service that has terminated. When the conversation is reestablished, the MDL-DDEML attempts to reestablish any preexisting advise loops.

hConv identifies the conversation to be reestablished. A client must have obtained the conversation handle by a previous call to `mdlWin32_DdeConnect` or from an `XTYP_DISCONNECT` transaction.

Returns `mdlWin32_DdeReconnect` returns the value of the handle of the reestablished conversation; otherwise, it is `NULL`. Use `mdlWin32_DdeGetLastError` to retrieve the error value, which may be one of the following: `DMLERR_DLL_NOT_INITIALIZED`, `DMLERR_INVALIDPARAMETER`, `DMLERR_NO_CONV_ESTABLISHED`, `DMLERR_NO_ERROR`.

See Also `mdlWin32_DdeConnect`, `mdlWin32_DdeDisconnect`.

mdlWin32_DdeSetUserHandle

```
#include <w32dde.h> /* For MDL-DDEML constants. */
#include <msw32dde.fdf> /* For MDL-DDEML prototypes. */

BOOL mdlWin32_DdeSetUserHandle
(
    HCONV hConv, /* => Conversation handle */
    ...
);
```

```

DWORD id,          /* => QID_ value */
DWORD hUser        /* => Value to attach to message */
);

```

Description `mdlWin32_DdeSetUserHandle` associates an application-defined 32-bit value with a conversation handle or a transaction identifier. This is useful for simplifying the processing of asynchronous transactions. `mdlWin32_DdeQueryConvInfo` can be used to retrieve this value.

hConv identifies the conversation.

id specifies the transaction identifier to associate with the value specified by the *hUser* parameter. An application should set this parameter to `QID_SYNC` to associate *hUser* with the conversation identified by the *hConv* parameter.

hUser identifies the value to associate with the conversation handle.

Returns `mdlWin32_DdeSetUserHandle` returns `TRUE` if the function succeeds; otherwise, it is `FALSE`. Use `mdlWin32_DdeGetLastError` to retrieve the error value, which may be one of the following: `DMLERR_DLL_NOT_INITIALIZED`, `DMLERR_INVALIDPARAMETER`, `DMLERR_NO_ERROR`, `DMLERR_UNFOUN_QUEUE_ID`.

See Also `mdlWin32_DdeQueryConvInfo`.

mdlWin32_DdeUnaccessData

```

#include <w32dde.h> /* For MDL-DDEML constants. */
#include <msw32dde.fdf> /* For MDL-DDEML prototypes. */

BOOL mdlWin32_DdeUnaccessData
(
  HDEDATA hData          /* => DDE object handle */
);

```

Description `mdlWin32_DdeUnaccessData` unaccesses a DDE data object. An application must call this function when it is finished accessing the object. This function is used with `mdlWin32_DdeAccessData`.

hData identifies the DDE data object.

Returns `mdlWin32_DdeUnaccessData` returns `TRUE` if the function succeeds, otherwise, `FALSE`. Use the `mdlWin32_DdeGetLastError` function to retrieve the error value, which may be one of the following: `DMLERR_DLL_NOT_INITIALIZED`, `DMLERR_INVALIDPARAMETER`, `DMLERR_NO_ERROR`.

See Also `mdlWin32_DdeAccessData`, `mdlWin32_DdeAddData`, `mdlWin32_DdeCreateDataHandle`, `mdlWin32_DdeFreeDataHandle`.

mdlWin32_DdeUninitialize

```
#include <w32dde.h> /* For MDL-DDEML constants. */
#include <msw32dde.fdf> /* For MDL-DDEML prototypes. */

BOOL mdlWin32_DdeUninitialize
(
    DWORD    idInst          /* => Instance id */
);
```

Description The `mdlWin32_DdeUninitialize` function terminates any conversations currently open and frees all of the DDE resources associated with the calling application.

idInst specifies the application-instance identifier obtained by a previous call to the `mdlWin32_DdeInitialize` function.

Returns `mdlWin32_DdeUninitialize` returns `TRUE` if the function succeeds, `FALSE` otherwise.

See Also `mdlWin32_DdeDisconnect`, `mdlWin32_DdeDisconnectList`, `mdlWin32_DdeInitialize`.

userWin32_DdeUserCallback

```
#include <wnclpext.h> /* For CF_xx clipboard formats. */
#include <w32dde.h> /* For MDL-DDEML constants. */
#include <msw32dde.fdf> /* For MDL-DDEML prototypes. */

HDDDEDATA CALLBACK userWin32_DdeUserCallback
(
    WORD    wType,          /* => transaction type */
    WORD    wFmt,           /* => clipboard data format */
    HCONV    hConv,         /* => handle of the conversation */
    HSZ      hsz1,          /* => handle of a string */
    HSZ      hsz2,          /* => handle of a string */
    HDDDEDATAhData,        /* => handle of a DDE object */
    DWORD    dwData1,       /* => transaction-specific data */
    DWORD    dwData2        /* => transaction-specific data */
);
```

Description `userWin32_DdeUserCallback` is an application defined function that processes transactions sent to the application as a result of DDEML calls by other applications. An MDL application designates an callback function by calling `mdlWin32_DdeInitialize`. The application programmer determines the function name; `userWin32_DdeUserCallback` is used merely as an example.

The callback function is called synchronously for transactions that do not involve creating or terminating conversations. The DDEML notifies an application of DDE activity that affects the application by sending transactions to the application's DDE callback function. A DDE transaction is similar to a message—it is a named constant accompanied by other parameters that contain additional information about the transaction. The DDEML passes a transaction to an application-defined DDE callback

function, which carries out the appropriate action depending on the type of the transaction. For example, when a client application attempts to establish a conversation with a server application, the client calls `mdlWin32_DdeConnect`. This causes the DDEML to send an `XTYP_CONNECT` transaction to the server's callback function. The callback function can allow the conversation by returning `TRUE`, or it can deny the conversation by returning `FALSE`. After a client has established a conversation with a server, the client can send transactions to obtain data and services from the server.

wType specifies the type of the current transaction. This parameter consists of a combination of transaction-*class* flags and transaction-*type* flags. Most applications should be unconcerned with the particular value of *wType* and should instead compare its value with the `XTY_` constants provided.

wFmt specifies the format in which data is sent or received.

hConv identifies the conversation associated with the current transaction.

hsz1 and *hsz2* identifies a string. The meaning of this parameter depends on the type of the current transaction. See the description of the transaction type for the meaning of this parameter.

hData identifies DDE data. The meaning of this parameter depends on the type of the current transaction. See the description of the transaction type for the meaning of this parameter.

dwData1 and *dwData2* specifies transaction-specific data. See the description of the transaction type for the meaning of this parameter.

Returns The return value of *userWin32_DdeUserCallback* depends on the transaction class.

See Also `mdlWin32_DdeEnableCallback`, `mdlWin32_DdeInitialize`.

wType Details

The following list describes the transaction classes and types that are passed in *wType* parameter. The class of a transaction is not typically used when writing an

asynchronous hook function but are presented here to document the type of data that is expected from each of the DDE transactions:

<i>wType</i>	transaction class and type
XCLASS_BOOL	A DDE callback function should return TRUE or FALSE when it finishes processing a transaction that belongs to this class. The XCLASS_BOOL transaction types are the following: XTYP_ADVSTART or XTYP_CONNECT.
XCLASS_DATA	A DDE callback function should return a DDE data handle, CBR_BLOCK, or NULL when it finishes processing a transaction that belongs to this class. The XCLASS_DATA transaction types are the following: XTYP_ADVREQ, XTYP_REQUEST, XTYP_WILDCONNECT.
XCLASS_FLAGS	A DDE callback function should return DDE_FACK, DDE_FBUSY or DDE_FNOTPROCESSED when it finishes processing a transaction that belongs to this class. The XCLASS_FLAGS transaction types are the following: XTYP_ADVDATA, XTYP_EXECUTE, XTYP_POKE.
XCLASS_NOTIFICATION	The transaction types that belong to this class are for notification purposes only. The return value from the callback function is ignored. The XCLASS_NOTIFICATION transaction types are the following: XTYP_ADVSTOP, XTYP_CONNECT_CONFIRM, XTYP_DISCONNECT, XTYP_ERROR, XTYP_MONITOR, XTYP_REGISTER, XTYP_XACT_COMPLETE, XTYP_UNREGISTER.

The following is a list of DDE transactions that can be passed to your callback function:

DDE Transaction	Action
XTYP_ADVDATA	Passes advise data to a client.
XTYP_ADVREQ	Prompts a server to send advise data to a client.
XTYP_ADVSTOP	Ends an advise loop.
XTYP_ADVSTART	Requests an advise loop.
XTYP_CONNECT	Requests a DDE conversation.
XTYP_CONNECT_CONFIRM	Confirms a DDE conversation.
XTYP_DISCONNECT	Terminates a DDE conversation.
XTYP_ERROR	Notifies a DDEML application of a critical error.
XTYP_EXECUTE	Executes a server command.
XTYP_MONITOR	Informs a DDE monitor application of a DDE event.
XTYP_POKE	Sends unsolicited data to a server.
XTYP_REGISTER	Registers a service name.
XTYP_REQUEST	Requests data from a server.

DDE Transaction	Action
XTYP_UNREGISTER	Unregisters a service name.
XTYP_WILDCONNECT	Requests multiple DDE conversation.
XTYP_XACT_COMPLETE	Confirms completion of asynchronous transaction.

Transaction (uType)	Send. ¹	Recv.	wFmt	hConv	hsz1	hsz2	hData	dwData1	dwData2	Returns
XTYP_ADVDATA	S	C	†	†	topic	item	data or NULL	N/A	N/A	DDE_FACK, DDE_FBUSY, DDE_FNOTPROCESSED
XTYP_ADVREQ	W	S	†	†	topic	item	N/A	# of requests	N/A	ignored
XTYP_ADVSTOP	C	S	†	†	topic	item	N/A	N/A	N/A	ignored
XTYP_ADVSTART	C	S	†	†	topic	item	N/A	N/A	N/A	TRUE or FALSE
XTYP_CONNECT	C	S	N/A	N/A	topic	service	N/A	0 or CONVCONTEXT	instance ²	TRUE or FALSE
XTYP_CONNECT_CONFIRM	W	S	N/A	†	topic	service	N/A	N/A	²	ignored
XTYP_DISCONNECT	C	S	N/A	†	N/A	N/A	N/A	N/A	²	ignored
XTYP_ERROR	W	B	N/A	or NULL	N/A	N/A	N/A	error code	N/A	ignored
XTYP_EXECUTE	C	S	N/A	†	topic	N/A	†	N/A	N/A	DDE_FACK, DDE_FBUSY, DDE_FNOTPROCESSED
XTYP_MONITOR	W	B	N/A	N/A	N/A	N/A	DDE object	N/A	MF_value	0 or 1
XTYP_POKE	C	S	†	†	topic	item	†	N/A	N/A	DDE_FACK, DDE_FBUSY, DDE_FNOTPROCESSED
XTYP_REGISTER	S	B	N/A	N/A	service	instance service	N/A	N/A	N/A	ignored
XTYP_REQUEST	C	S	N/A	†	topic	item	N/A	N/A	N/A	Data Handle or NULL
XTYP_UNREGISTER	W	B	N/A	N/A	service	instance service	N/A	N/A	N/A	ignored
XTYP_WILDCONNECT	W	S	N/A	N/A	topic or NULL	service or NULL	N/A	0 or CONVCONTEXT	instance ²	HSZPAIR or NULL2
XTYP_XACT_COMPLETE	W	B	or NULL	†	topic	item	data, TRUE or NULL	transaction id	N/A not supported	ignored

† means the parameter contains a meaningful value.

¹“C” means client application, “S” means server application, “B” means both client and server applications, and “W” means Windows Operating System.

²If this parameter is 1 then the client is another MDL application running within this instance of MicroStation.

³The data handle is returned from `mdlWin32_DdeCreateDataHandle`.

File Drag and Drop Utility Functions

The following functions are available to MDL applications that will run under MicroStation for Windows NT. These functions allow MDL application to be notified when a user drags files from Windows' File Manager to MicroStation.

Function	Used to
mdlFileDragAndDrop_setFunction	registers a program's asynchronous drag and drop function.

The following table lists file drag and drop user functions. The programmer determines the user functions names. These functions are all designated to MDL through function pointers. MDL does not use the names. The following function names are used merely as illustrations:

Function	MicroStation calls when
userFileDragAndDrop_handleDrop	files are dropped onto any MicroStation window.

mdlFileDragAndDrop_setFunction

```
#include <dragdrop.h> /* For File Drag and Drop structures. */
#include <dragdrop.fdf> /* For function prototypes */

MdlFunctionP mdlFileDragAndDrop_setFunction
(
    int          type,          /* => Type of function */
    MdlFunctionP function      /* => MDL application function */
);
```

Description The mdlFileDragAndDrop_setFunction registers a user function to be called when a particular the operator drops files on any MicroStation window. The specified function will be called asynchronously.

type indicates how MicroStation registers the specified user function. This parameter must be set to zero.

function points to an MDL function to be registered or NULL. If this parameter is NULL, the function's previous registration will be discarded.

Returns mdlFileDragAndDrop_setFunction is of type void and returns no value.

See Also userFileDragAndDrop_handleDrop.

userFileDragAndDrop_handleDrop

```
#include <dragdrop.h> /* For File Drag and Drop structures */
#include <dragdrop.fdf> /* For function prototypes */

void userFileDragAndDrop_handleDrop
(
  DragDropFileEvent      *ddfeP      /* <=> Drop context structure */
);
```

Description An MDL application designates an MDL file drag and drop function by calling `mdlFileDragAndDrop_setFunction`. The application programmer determines the function name; *userFileDragAndDrop_handleDrop* is used merely as an example.

The *userFileDragAndDrop_handleDrop* registers a user function to be called when the operator drags files from the Windows NT File Manager and drops files on any MicroStation window. The specified function will be called asynchronously.

**ddfeP* points to a `DragDropFileEvent` structure. The members of this structure are presented below:

data type	field	description
MSWindow	<i>*gwP</i>	Identifies to the MicroStation window that the drop occurred. This parameter may be NULL if the target window is undetermined.
Point2d	<i>p2dDropPoint</i>	Contains the coordinates of the drop point. This structure may not be valid if <i>gwP</i> is NULL.
ULong	<i>bIsOnFrame</i>	A flag indicating that the drop occurred on the frame of a window.
ULong	<i>nFiles</i>	Contains the number of files in this drop.
DragDropFileInfo	<i>ddfipFileInfo[]</i>	Contains <i>nFiles</i> elements with the following meanings: ULong <i>bIsProcessed</i> Is a flag specifying that this file has been processed. When a MDL application processes a file, it should set this variable to TRUE. char <i>szFileName[]</i> A fully qualified filename.

Returns *userFileDragAndDrop_handleDrop* should return `SUCCESS` under most conditions. If this function returns `ERROR` then MicroStation will abort its processing and not call any other drop handlers in any other MDL application. The value `ERROR` should be used only in the most severe error states.

See Also `mdlFileDragAndDrop_setFunction`.

Clipboard Functions

MDL applications running in MicroStation Version 5.0 or later for Windows NT can participate in clipboard operations by using the `mdlClipboard_...` set of functions. Throughout this section, the term clipboard refers to both the Windows clipboard and MicroStation's internal clipboard.



`ntcbtext.mc` is a sample MDL clipboard application that demonstrates use of the clipboard interface functions.

User's View

The user selects the *Edit->Paste* option to take data from the clipboard into the design file, and the *Edit->Copy* or *Edit->Cut* options to take data from the design file and places it into the clipboard. The paste, copy and cut choices display options to the user on how the data is to be manipulated. For example, a user may copy a view from MicroStation as a bitmap image or as a Windows Metafile. The choices displayed in these sub-menus are determined by responses from MDL applications that have registered themselves as clipboard participants.

Programmer's View

In general, the `mdlClipboard_...` protocol is a two step process. A programmer registers a function (a.k.a. the “strategy” function) that is responsible for notifying MicroStation which clipboard formats the MDL application is able to process, what message to place into the particular sub-menu, and which application function (a.k.a. the “process” function) to invoke for processing that format.

Each of the clipboard operations have sub-menus. The contents of these sub-menus are based on the data returned from each *strategy* functions from all of the currently registered MDL applications.

When the operator selects a clipboard operation (copy, cut or paste) from the edit menu, all of the registered MDL functions for that operation are called and based on the returned values, a menu is created and presented to the user. When the operator selects a choice from that sub-menu, MicroStation will call the specified MDL application's *process* function to process the particular request.

The contents of the clipboard sub-menus is not static. The contents may change based on events both internal and external to MicroStation. For example, an operator will not see an option to paste a bitmap or raster image if there is no bitmap data in the clipboard. The contents of the sub-menus is specified by the registered MDL strategy functions for each clipboard operation.

For each of the clipboard operations (i.e., paste, copy or cut) that the MDL application participates, the programmer must register a unique strategy function. The function is

registered with the `mdlClipboard_setFunction` call. A MDL application may have only one strategy function registered for each clipboard operation. Therefore, a MDL program may have at most three registered strategy functions—one each for the clipboard operations copy, cut and paste.

For this discussion, the registered functions will be referred to as *userClipboard_copyStrategy*, *userClipboard_cutStrategy* or *userClipboard_pasteStrategy*. The actual function names are determined by the programmer. The strategy functions have the following properties and responsibilities:

1. This function is called synchronously.
2. Optionally examine the current runtime environment.
3. Determine what clipboard data formats to process.
4. For each pertinent clipboard format, provide:
 - the clipboard format number.
 - the “quality” of this format handler.
 - optional programmer defined context data.
 - a function to process the format.
 - a character string to display in the sub-menu.
5. Return the number of formats that the function can process.

In step 4 bullet 4, the strategy function must provide MicroStation with the address of a function process the particular clipboard format. For this discussion, the registered functions will be referred to as *userClipboard_copyProcess*, *userClipboard_cutProcess* or *userClipboard_pasteProcess*. The actual function names are determined by the programmer. This function is called synchronously. This function processes the clipboard and design file as appropriate and consistent with the operator requested action.

The following table lists the Windows NT clipboard interface functions:

Function	Used to
<code>mdlClipboard_openClipboard</code>	open the clipboard and locks it for other Windows applications.
<code>mdlClipboard_closeClipboard</code>	close the clipboard and free it for use by other Windows applications.
<code>mdlClipboard_emptyClipboard</code>	clear the clipboard.
<code>mdlClipboard_getClipboardData</code>	retrieve data from the clipboard.
<code>mdlClipboard_getClipboardFormatName</code>	retrieve the name associated with this clipboard format number.

Function	Used to
mdlClipboard_setClipboardData	place data into the clipboard.
mdlClipboard_registerClipboardFormat	register a clipboard format.
mdlClipboard_setFunction	designate one of the asynchronous clipboard functions listed below.

The following table lists clipboard user functions. The programmer determines the user functions names. These functions are all designated to MDL through function pointers. MDL does not use the names. The following function names are used merely as illustrations:

Function	MicroStation calls when
<i>userClipboard_pasteStrategy</i>	the clipboard changes state and during mdlClipboard_setFunction (CLIPBOARD_SET_PASTE, ...).
<i>userClipboard_copyStrategy</i>	the <i>Edit->Copy</i> sub-menu must be built, or during mdlClipboard_setFunction (CLIPBOARD_SET_PASTE, ...).
<i>userClipboard_cutStrategy</i>	need to build the <i>Edit->Cut</i> sub-menu and during mdlClipboard_setFunction (CLIPBOARD_SET_CUT, ...).
<i>userClipboard_pasteProcess</i>	to process a particular clipboard data format from the <i>Edit->Paste</i> menu.
<i>userClipboard_copyProcess</i>	a particular clipboard data format must be processed from the <i>Edit->Copy</i> menu.
<i>userClipboard_cutProcess</i>	to process a particular clipboard data format from the <i>Edit->Cut</i> menu.

mdlClipboard_openClipboard, mdlClipboard_closeClipboard

```
#include <mclipbd.fdf> /* For function prototypes */
UInt mdlClipboard_openClipboard(void);
UInt mdlClipboard_closeClipboard(void);
```

Description mdlClipboard_openClipboard opens the Windows clipboard for examination by your application and prevents other applications from changing the clipboard contents. Your application should call mdlClipboard_closeClipboard after it is finished processing the clipboard.

mdlClipboard_closeClipboard releases your MDL application's exclusive lock on the Windows Clipboard. You must call this function after each call to mdlClipboard_openClipboard.

Returns `mdlClipboard_openClipboard` and `mdlClipboard_closeClipboard` return `SUCCESS` if they are successful or `ERROR` if an error occurs.

See Also `mdlClipboard_emptyClipboard`, `mdlClipboard_setClipboardData`, `userClipboard_...process`.

mdlClipboard_emptyClipboard

```
#include <mclipbd.fdf> /* For function prototypes */
UInt mdlClipboard_emptyClipboard(void);
```

Description The `mdlClipboard_emptyClipboard` function empties the clipboard. All data objects in the clipboard are discarded. This function is typically called before one or more `mdlClipboard_setClipboardData` functions are called. It is bad practice to add data to the clipboard without first clearing the existing contents of the clipboard.

Returns `mdlClipboard_emptyClipboard` returns `SUCCESS` if the functions is successful or `ERROR` if an error occurs.

See Also `mdlClipboard_openClipboard`, `mdlClipboard_closeClipboard`, `mdlClipboard_setClipboardData`, `userClipboard_...process`.

mdlClipboard_getClipboardData

```
#include <clipbrd.h> /* For clipboard structures. */
#include <wnclpext.h> /* For clipboard format IDs */
#include <mclipbd.fdf> /* For function prototypes */

void *mdlClipboard_getClipboardData
(
    ULONG    cFmt      /* => Clipboard format number */
);
```

Description The `mdlClipboard_getClipboardData` function retrieves a data handle or buffer to the current clipboard data having a specified format. The clipboard must have been opened previously.



It is the responsibility of the programmer to ensure that the memory allocated by this function is eventually freed.

Returns `mdlClipboard_getClipboardData` returns a pointer to a data buffer or `NULL` if an errors occurs of the specified data format is unavailable.

See Also `userClipboard_copyProcess`, `userClipboard_cutProcess`, `userClipboard_pasteProcess`.

mdlClipboard_getClipboardFormatName

```
#include <clipbrd.h> /* For clipboard structures */
```

```
#include <wnclpext.h> /* For clipboard format IDs */
#include <mclipbd.fdf> /* For function prototypes */

int mdlClipboard_getClipboardFormatName
(
    UInt      wFormat,          /* => Clipboard format number */
    char      *clipboardFormatName, /* <= Output character string */
    int       formatNameLen     /* => Max size of <clipboardFormatName> */
);
```

Description The `mdlClipboard_getClipboardFormatName` function retrieves from the clipboard the name of the specified registered format. The function copies the name to the specified buffer.

wFormat specifies the type of format to be retrieved. This parameter must not specify any of the predefined clipboard formats.

clipboardFormatName points to the buffer that is to receive the format name.

formatNameLen specifies the maximum length, in characters, of the string to be copied to the buffer. If the name exceeds this limit, it is truncated.

Returns If the function succeeds, the return value is the length, in characters, of the string copied to the buffer. Otherwise, the return value is zero, indicating the requested format does not exist or is predefined.

mdlClipboard_setClipboardData

```
#include <clipbrd.h> /* For clipboard structures */
#include <wnclpext.h> /* For clipboard format IDs */
#include <mclipbd.fdf> /* For function prototypes */

void *mdlClipboard_setClipboardData
(
    ULong      cfFmt,          /* => Clipboard format number */
    void      *pClipboardData, /* => Clipboard data buffer */
    int       nDataLength     /* => Length in bytes of data buffer */
);
```

Description The `mdlClipboard_setClipboardData` function adds or replaces the specified data format to the clipboard. The clipboard must have been opened previously.

cfFmt specifies the format of the data. It can be any one of the system-defined clipboard formats, or a format registered by the `mdlClipboard_registerClipboardFormat` function.

pClipboardData specifies the data to be placed into the clipboard. This parameter must not be NULL and will be rejected if it is. If your application manages its own handle memory, you may place a handle to the data in this parameter.

nDataLength specifies the length in bytes of the above data buffer. If this parameter is -1 then the value of *pClipboardData* is treated as a handle and not an address. When this value is positive, the data buffer is copied into an allocated data handle.

Private data formats in the range of CF_PRIVATEFIRST to CF_PRIVATELAST are not automatically freed when the data is deleted from the clipboard.

If the Windows clipboard application is running, it will not update its window to show the data placed in the clipboard by the SetClipboardData until after the mdlClipboard_closeClipboard function is called.

Returns mdlClipboard_setClipboardData returns a pointer to a data buffer or NULL if an errors occurs of the specified data format is unavailable.

See Also userClipboard_copyProcess, userClipboard_cutProcess, userClipboard_pasteProcess, mdlClipboard_closeClipboard, mdlClipboard_openClipboard, mdlClipboard_getClipboardData, mdlClipboard_registerClipboardFormat.

mdlClipboard_registerClipboardFormat

```
#include <mshclipb.fdf> /* For function prototypes */

UInt mdlClipboard_registerClipboardFormat
(
    char    *pszFormatName    /* => Format name string */
);
```

Description The mdlClipboard_registerClipboardFormat function registers a new clipboard format name. The registered format can be used in subsequent clipboard functions as a valid format in which to render or retrieve data. It is this name that other applications can identify the various data formats within the clipboard.

pszFormatName points to a NULL-terminated string that names the new format.

The following table lists the standard Windows NT clipboard formats. These formats are predefined need not be registered.

Symbolic Name	Value	Meaning
CF_TEXT	1	Text format. Each line ends with a carriage return/linefeed (CR-LF) combination. A NULL character signals the end of the data.
CF_BITMAP	2	A handle of a bitmap.
CF_METAFILEPICT	3	Handle of a metafile picture format as defined by the METAFILEPICT structure.
CF_SYLK	4	Microsoft Symbolic Link (SYLK) format.
CF_DIF	5	The data is in Software Arts' Data Interchange Format.

Symbolic Name	Value	Meaning
CF_TIFF	6	Tag image file format (TIFF).
CF_OEMTEXT	7	Text format containing characters in the OEM character set. Each line ends with a carriage return/linefeed (CR-LF) combination. A NULL character signals the end of the data.
CF_DIB	8	The data is a memory block containing a BITMAPINFO structure followed by the bitmap data.
CF_PALETTE	9	Handle of a color palette. Whenever an application places data in the clipboard that depends on or assumes a color palette, it should place the palette in the clipboard as well. If the clipboard contains data in the CF_PALETTE (logical color palette) format, the application should select and realize any other data in the clipboard against that logical palette. When displaying clipboard data, Windows Clipboard always uses as its current palette any object on the clipboard that is in the CF_PALETTE format.
CF_PENDATA	10	Data for the pen extensions to the Windows operating system.
CF_RIFF	11	Represents audio data more complex than can be represented in a CF_WAVE standard wave format.
CF_WAVE	12	Represents audio data in one of the standard wave formats, such as 11 kHz or 22 kHz PCM.
CF_UNICODETEXT	13	Unicode text format. Each line ends with a carriage return/linefeed (CR-LF) combination. A NULL character signals the end of the data.
CF_ENHMETAFILE	14	A handle of an enhanced metafile (HENHMETAFILE).

The following format numbers are defined for completeness but are not used for clipboard aware MDL applications:.

Symbolic Name	Value	Meaning
CF_OWNERDISPLAY	0x008 0	Owner display format. The clipboard owner must display and update the clipboard viewer window, and receive the WM_ASKCBFORMATNAME, WM_HSCROLLCLIPBOARD, WM_PAINTCLIPBOARD, WM_SIZECLIPBOARD and WM_VSCROLLCLIPBOARD messages. The <i>data</i> parameter must be NULL. This is not valid for MDL application.
CF_DSPTEXT	0x008 1	Text display format associated with a private format. The data is a handle of data that can be displayed in text format in lieu of the privately formatted data.

Symbolic Name	Value	Meaning
CF_DSPBITMAP	0x008 2	Bitmap display format associated with a private format. The data parameter must be a handle of data that can be displayed in bitmap format in lieu of the privately formatted data.
CF_DSPMETAFILEPICT	0x008 3	Metafile-picture display format associated with a private format. The value is a handle of data that can be displayed in metafile-picture format in lieu of the privately formatted data.
CF_DSPENHMETAFILE	0x008 E	Enhanced metafile display format associated with a private format. The data must be a handle of data that can be displayed in enhanced metafile format in lieu of the privately formatted data.
CF_PRIVATELAST CF_PRIVATEFIRST	0x02F F 0x020 0	Range of integer values for private clipboard formats. Handles associated with private clipboard formats are not freed automatically; the clipboard owner must free such handles. This format is not valid for MDL application.

Windows supports three clipboard formats for text: CF_UNICODETEXT, CF_TEXT and CF_OEMTEXT. Specify CF_TEXT for ANSI text, CF_UNICODETEXT for Unicode text, and CF_OEMTEXT for text in the OEM character set. If any of the three text formats is placed on the clipboard, all three can be retrieved by using the `mdlClipboard_getClipboardData` function. If necessary, Windows automatically converts data in an available text format to the requested format. An application can place more than one text format on the clipboard if it does so at one time (between the same calls to `mdlClipboard_openClipboard` and `mdlClipboard_closeClipboard`).

Windows supports two clipboard formats for metafiles: CF_ENHMETAFILE and CF_METAFILEPICT. Specify CF_ENHMETAFILE for enhanced metafiles and CF_METAFILEPICT for Windows metafiles. If either of the two metafile formats is placed on the clipboard, both can be retrieved by `mdlGetClipboardData`. If necessary, Windows automatically converts data in the available metafile format to the requested format. An application can place both metafile formats on the clipboard if it does so at one time (between the same calls to `mdlClipboard_openClipboard` and `mdlClipboard_closeClipboard`).

Returns If the function succeeds, the return value identifies the registered clipboard format; otherwise, it is zero. If a registered format with the specified name already exists, a new format is not registered and the return value identifies the existing format. This enables more than one application to copy and paste data using the same registered clipboard format.

Registered clipboard formats are identified by values in the range 0xC000 through 0xFFFF.

See Also mdlClipboard_setClipboardData, mdlClipboard_getClipboardData.

mdlClipboard_setFunction

```
#include <clipbrd.h> /* For clipboard structures. */
#include <wnclpext.h> /* For clipboard format IDs */
#include <mclipbd.fdf> /* For function prototypes */

MdlFunctionP mdlClipboard_setFunction
(
    int          type,          /* => Type of function */
    MdlFunctionP newDataFunc    /* => MDL application function */
);
```

Description The mdlClipboard_setFunction registers a user function to be called when a particular clipboard event occurs. The specified function will be called synchronously.

type indicates how MicroStation registers the specified user function. The valid values are: CLIPBOARD_SET_PASTE, CLIPBOARD_SET_COPY, CLIPBOARD_SET_CUT.

newDataFunc points to an MDL function to be registered or NULL. If this parameter is NULL, the function's previous registration will be discarded. The following table lists the circumstances for which MicroStation calls *newDataFunc*:

CLIPBOARD_SET_COPY:

1. When any MDL application registers a clipboard strategy routine.
2. When any MDL application unregistered a clipboard strategy routine.
3. When any MDL application that has a registered clipboard strategy routine is unloaded.
4. When a MicroStation user clicks on the *Edit* menu option.

CLIPBOARD_SET_CUT:

1. When any MDL application registers a clipboard strategy routine.
2. When any MDL application unregistered a clipboard strategy routine.
3. When any MDL application that has a registered clipboard strategy routine is unloaded.
4. When a MicroStation user clicks on the *Edit* menu option.

CLIPBOARD_SET_PASTE:

1. When any MDL application registers a clipboard strategy routine.
2. When any MDL application unregistered a clipboard strategy routine.
3. When any MDL application that has a registered clipboard strategy routine is unloaded.
4. When the Windows NT clipboard changes (i.e., when a user copies or cuts something into the clipboard).

Returns `mdlClipboard_setFunction` returns a pointer to the user function (of the same type) that was previously set using `mdlClipboard_setFunction`.

See Also `userClipboard_copyStrategy`, `userClipboard_cutStrategy`, `userClipboard_pasteStrategy`.

***userClipboard_pasteStrategy, userClipboard_copyStrategy,
userClipboard_cutStrategy***

```
int userClipboard_...strategy /* <= SUCCESS or ERROR */
(
    int                *numResponsesP,    /* <= Number of choices */
    ClipboardResponse *crP,                /* <= Sub-menu choices, etc */
    ClipboardFormatInfo *cfiP,            /* => Paste only: available formats */
    int                numFormats         /* => Paste only: num. avail formats */
);
```

Description The `userClipboard_copyStrategy`, `userClipboard_cutStrategy` and `userClipboard_pasteStrategy` functions are called to alert MDL functions of clipboard activity.

numResponsesP specifies the number of options copied into the `ClipboardResponse` structure.

crP specifies the information needed to setup a sub-menu on the *Edit->Copy*, *Edit->Cut*, or *Edit->Paste* menu. The members of this structure are typically populated as follows:

```
crP->formatNum = cfiP->formatNum;
crP->handlerQuality = CLIPBOARD_FORMAT_IDEALHANDLER;
crP->userDataP = NULL;
crP->handlerFunction = userClipboard_pasteProcess;
strcpy(crP->menuName, 'Displayed sub-menu choice');
crP++;
(*numResponsesP)++;
```

Where:

Field	Description
<i>crP->formatNum</i>	is assigned a predefined or registered clipboard format number.
<i>crP->handlerQuality</i>	is either <code>CLIPBOARD_FORMAT_IDEALHANDLER</code> or <code>CLIPBOARD_FORMAT_CANHANDLE</code> . <code>CLIPBOARD_FORMAT_IDEALHANDLER</code> tells MicroStation that this option <i>must</i> appear in the sub-menu list. <code>CLIPBOARD_FORMAT_CANHANDLE</code> tells MicroStation that this option <i>may</i> appear in the sub-menu list if no other option appears with this format number.
<i>crP->userDataP</i>	is optional data to be passed to the <i>userClipboard_...process</i> routine.
<i>crP->handlerFunction</i>	is an address of the <i>userClipboard_...process</i> function for this option.
<i>crP->menuName</i>	is a string to copy into this array. This string will be displayed in the sub-menu. This string should be loaded from a string list resource to allow for internationalization of your application.
<i>crP++</i>	is required to advance the response structure pointer.
<i>(*numResponsesP)++</i>	is required to advance the count of responses.

cfiP specifies the current list of available formats in the clipboard. This array is only used for the *paste* strategy function. This variable consists of a structure of clipboard format name and format number. MDL routines should use this array to determine which options are to be listed in the response array.

numFormats specify the number of valid entries in the *cfiP* array.

Each *process function* has a “quality” associated with its ability to process the clipboard data format. When the *strategy function* nominates a *process function* it also specifies a quality as either `CLIPBOARD_FORMAT_CANHANDLE` or `CLIPBOARD_FORMAT_IDEALHANDLER`.

For `CLIPBOARD_FORMAT_CANHANDLE`, the corresponding menu name is put into the sub-menu only if there is no other application that says it can handle this format.

For `CLIPBOARD_FORMAT_IDEALHANDLER`, the corresponding menu name will always appear in the sub-menu. An MDL application can nominate more than one function for a particular clipboard data format as long as it describes them all as `CLIPBOARD_FORMAT_IDEALHANDLERS`.

Returns *userClipboard_copyStrategy*, *userClipboard_cutStrategy*, *userClipboard_pasteStrategy*, must return `SUCCESS` for its responses to be included in the Edit sub-menus. If this function returns any other value, the response structure will be ignored.

See Also *mdlClipboard_setFunction*, *userClipboard_copyProcess*, *userClipboard_cutProcess*, *userClipboard_pasteProcess*.

userClipboard_pasteProcess*, *userClipboard_copyProcess*, *userClipboard_cutProcess

```
#include <clipbrd.h> /* For clipboard structures. */
#include <wnclpext.h> /* For clipboard format IDs */
#include <msclipbd.fdf> /* For function prototypes */

void userClipboard_...process
(
    void      *userDataP      /* <=> User data from strategy routine */
);
```

Description The *userClipboard_...process* function processes the clipboard data in the specific format. An MDL applications designates an MDL clipboard process function by calling *mdlClipboard_setFunction*. The application programmer determines the function name and *userClipboard_...process* is used merely as an example.

userDataP is passed in as specified in the corresponding *strategy* function.

Returns *userClipboard_...process* is of type `void`; it returns nothing.

See Also *mdlClipboard_setFunction*, *mdlClipboard_...strategy*.

This chapter contains the following sections:

- String list functions
- String functions
- License management functions
- Dynamic array functions
- Help functions
- Function key functions
- Undo functions
- Tutorial functions
- Plotting functions
- Wide text functions
- BASIC interface functions
- User preferences functions
- Miscellaneous functions

String List Manager

The string list manager aids in the use of strings and any associated data that is stored with these strings. It is a general, multi-purpose manager that makes manipulating and storing strings easier.

A string list consists of a group of string members. Each member contains a string (possibly empty), and can optionally include a simulated array of long integers, called

information fields. Members are accessed through an index, and the indices do not need to be consecutive. (Think of it as an array).

Information fields can be set to whatever the programmer desires by `mdlStringList_setMember` or `mdlStringList_setInfoField`. The current value for a particular member within the `stringList` can be determined using `mdlStringList_getInfoField` or `mdlStringList_getMember`. The number of information fields associated with each string in a string list is specified when the string list is created, and cannot be changed later. `mdlStringList_numInfoFields` will return the number of information fields.

String lists as resources

A `StringList` lists strings and any associated information fields used by the application. Each string in the list has any number of corresponding information fields attached. An example of a `StringList` follows:

```
StringList COMMON_USE_LIST_ID=
{
    3,    /* Number of information fields per string */
    {
        {{1, 2, 3}, 'This is the first string.'},
        {{12, -23, 72}, 'This is the second string.'},
        {{-1, -45, 12}, 'This is the third string.'};
    }
};
```

Each string has exactly three information fields. A less common, but still valid use follows:

```
StringList RARE_USE_LIST_ID=
{
    3,    /* Number of information fields per string */
    {
        {{1, 2, 3}, 'Same as before.'},
        {{1, 0, 7, 8}, 'Note: 4 information fields.'},
        {{}, 'Note: No information fields.'},
        {{2, 4, 8}, ''};
    }
};
```

In this example, a string list with 3 information fields per string is created. The first string will be as seen above. The second string will have the last information field (8) discarded. The next string will have defaults of 0 for its 3 information fields. The last string implies that this member will have no string associated with it.



This is different from a member having a string of length 0.

An application can obtain the data contained in a resource by calling the appropriate String List Manager functions.

The following table lists string list functions:

Function	Used to
mdlStringList_create	allocate storage for the string list.
mdlStringList_destroy	free storage for the string list.
mdlStringList_setMember	set a string list member to the given data.
mdlStringList_getMember	get the data for the given member.
mdlStringList_setInfoField	set a specific information field to a given value.
mdlStringList_getInfoField	get data for a specific information field within the given member.
mdlStringList_insertMember	insert new member(s) in the string list.
mdlStringList_deleteMember	delete the member(s) from the string list.
mdlStringList_copy	copy the data (string and information fields) from one string list to another.
mdlStringList_size	return the number of members in the string list.
mdlStringList_numInfoFields	return the number of information fields associated with the string list.
mdlStringList_moveMembers	move members from one index to another.
mdlStringList_sortByIndex mdlStringList_sort mdlStringList_sortByColumn	sort a string list in ascending or descending order, or by a user-supplied sort function.
mdlStringList_search	perform a linear search of the members.
mdlStringList_searchByColumn	perform a linear search, by column, of the members.
mdlStringList_binarySearch	perform a binary search of the members.
mdlStringList_binarySearchByColumn	perform a binary search, by column, of the members.
mdlStringList_addResource mdlStringList_addResourceWithType	add a string list to a resource file.
mdlStringList_writeResource mdlStringList_writeResourceWithType	add a string list to a resource file, overwriting any existing resource with the same resource ID.
mdlStringList_loadResource mdlStringList_loadResourceWithType	load a string list from a resource file.

mdlStringList_create

```
#include <dlogitem.h>
#include <msstrngl.fdf>

StringList *mdlStringList_create
(
    int      initialSize,          /* => requested number of strings */
    int      nInfoFields          /* => number of info fields desired */
);
```

Description The `mdlStringList_create` function creates a string list with *initialSize* members and *nInfoFields* information fields. If *nInfoFields* is less than or equal to 0, no information fields will be associated with the string list. If *initialSize* is less than or equal to 0, no members will be allocated and the user should add additional members to the string list as needed. (See `mdlStringList_insertMember`.)

Returns The `mdlStringList_create` function returns a valid pointer if it is successful. Otherwise, it returns NULL.

See Also `mdlStringList_insertMember`, `mdlStringList_copy`, `mdlStringList_destroy`.

mdlStringList_destroy

```
#include <mdlerrs.h>
#include <dlogitem.h>
#include <msstrngl.fdf>

int mdlStringList_destroy
(
    StringList *stringListP      /* => list to delete */
);
```

Description The `mdlStringList_destroy` function deallocates memory associated with the string list *stringListP*.

Returns The `mdlStringList_destroy` function returns `SUCCESS` if the operation is successful. Otherwise, it returns the following error:

`MDLERR_ADDRNOTVALID` *stringListP* is an invalid string list pointer.

See Also `mdlStringList_create`.

mdlStringList_setMember

```
#include <mdlerrs.h>
#include <dlogitem.h>
#include <msstrngl.fdf>

int mdlStringList_setMember
(
    StringList *stringListP, /* => string list to work on */
    int      memberIndex,    /* => index where member should be set */
    ...
);
```



```
char    *stringP,        /* => string, NULL=none */
long    *infoFieldsP    /* => infofield(s), NULL=none */
);
```

Description The mdlStringList_setMember function sets the member (at *memberIndex*) to the given data (*stringP* and *infoFieldsP*).

If *stringP* or *infoFieldsP* is NULL, the corresponding field of that member will be unaffected. To delete the string, pass an empty string in *stringP*. Data in *infoFieldsP* cannot be deleted.

The given data is copied into the string list so the user does not need to handle memory management.

Returns mdlStringList_setMember returns SUCCESS if the operation is successful. Otherwise, it returns one of the following errors:

Return Value	Description
MDLERR_ADDRNOTVALID	<i>stringListP</i> is an invalid string list pointer.
MDLERR_BADINDEX	<i>memberIndex</i> is out of range.
MDLERR_INFOMEMORY	Memory is insufficient to add items.

See Also mdlStringList_setInfoField, mdlStringList_getMember, mdlStringList_insertMember, mdlStringList_moveMembers.

mdlStringList_getMember

```
#include <mdlerrs.h>
#include <dlogitem.h>
#include <msstrngl.fdf>

int mdlStringList_getMember
(
char        **stringP,    /* <= string of member, NULL=none */
long        **infoFieldsP, /* <= infofield(s) of member, NULL=none */
StringList *stringListP, /* => stringlist to work on */
int         memberIndex   /* => index of member to get data from */
);
```

Description The mdlStringList_getMember function returns information for the *memberIndex* member of the string list.

It returns a pointer to the associated string in *stringP* and to the associated information field in *infoFieldsP*. It does not copy the information.

stringP or *infoFieldsP* can be NULL. If either is NULL, no information will be returned for that item.

Returns The `mdlStringList_getMember` function returns `SUCCESS` if the operation is successful. Otherwise, it returns one of the following errors:

Return Value	Description
<code>MDLERR_ADDRNOTVALID</code>	<i>stringListP</i> is an invalid string list pointer.
<code>MDLERR_BADINDEX</code>	<i>memberIndex</i> is out of range.

See Also `mdlStringList_getInfoField`, `mdlStringList_setMember`, `mdlStringList_size`, `mdlStringList_numInfoFields`.

mdlStringList_setInfoField

```
#include <mdlerrs.h>
#include <dlogitem.h>
#include <msstrngl.fdf>

int mdlStringList_setInfoField
(
StringList *stringListP,    /* => stringlist to work on */
int      memberIndex,      /* => index of member to set */
int      infoFieldIndex,   /* => index of particular infofield */
long     infoField         /* => data for infofield */
);
```

Description The `mdlStringList_setInfoField` function sets a member's information field to the given data.

memberIndex is the index of the member to be operated on.

infoFieldIndex specifies the information field to be changed.

infoField is the data to be placed in the information field.

Returns The `mdlStringList_setInfoField` function returns `SUCCESS` if the operation is successful. Otherwise, it returns one of the following errors:

Return Value	Description
<code>MDLERR_ADDRNOTVALID</code>	<i>stringListP</i> is an invalid string list pointer.
<code>MDLERR_BADINDEX</code>	<i>memberIndex</i> is out of range.

See Also `mdlStringList_getInfoField`, `mdlStringList_setMember`, `mdlStringList_getMember`.

mdlStringList_getInfoField

```
#include <mdlerrs.h>
#include <dlogitem.h>
#include <msstrngl.fdf>

int mdlStringList_getInfoField
(
    long          *infoFieldP,    /* <= infofield data */
    StringList    *stringListP,   /* => stringlist to work on */
    int           memberIndex,    /* => index of member to get data from */
    int           infoFieldIndex /* => index of particular infofield */
);
```

Description mdlStringList_getInfoField returns the value associated with the information field specified by *infoFieldIndex* from the member specified by *memberIndex* in *infoFieldP*.

Returns The mdlStringList_getInfoField function returns SUCCESS if the operation is successful. Otherwise, it returns one of the following errors:

Return Value	Description
MDLERR_ADDRNOTVALID	<i>stringListP</i> is an invalid string list pointer.
MDLERR_BADINDEX	<i>memberIndex</i> is out of range.

See Also mdlStringList_setInfoField, mdlStringList_getMember, mdlStringList_setMember.

mdlStringList_insertMember

```
#include <mdlerrs.h>
#include <dlogitem.h>
#include <msstrngl.fdf>

int mdlStringList_insertMember
(
    int          *indexP,        /* <= member added here, or -1 */
    StringList    *stringListP,   /* => stringlist to work on */
    int          memberIndex,    /* => index member at index, -1-end */
    int          numToInsert     /* => number of members to insert */
);
```

Description mdlStringList_insertMember creates *numToInsert* new members and inserts them at *memberIndex* in the string list. If *memberIndex* is out of range, the function appends the members to the end.

indexP returns the actual index of the member inserted. It also returns -1 if an error occurs. This return value is useful when *memberIndex* is invalid.

Returns `mdlStringList_insertMember` returns `SUCCESS` if the operation is successful. Otherwise, it returns one of the following errors:

Return Value	Description
<code>MDLERR_ADDRRNOTVALID</code>	<i>stringListP</i> is an invalid string list pointer.
<code>MDLERR_BADARG</code>	<i>numToInsert</i> is invalid.
<code>MDLERR_INFOMEMORY</code>	Memory is insufficient to add items.

See Also `mdlStringList_create`, `mdlStringList_moveMembers`.

mdlStringList_deleteMember

```
#include <mdlerrs.h>
#include <dlogitem.h>
#include <msstrngl.fdf>

int mdlStringList_deleteMember
(
    StringList *stringListP,      /* => stringlist to work on */
    int memberIndex,             /* => index of member to delete */
    int numToDelete              /* => number of members to delete */
);
```

Description `mdlStringList_deleteMember` deletes *numToDelete* members, starting at *memberIndex*, from the string list.

If *numToDelete* is out of range (<0 or >number of elements in *stringListP*), all members from *memberIndex* to the end of the string list are deleted.

Returns `mdlStringList_deleteMember` returns `SUCCESS` if the operation is successful. Otherwise, it returns one of the following errors:

Return Value	Description
<code>MDLERR_ADDRRNOTVALID</code>	<i>stringListP</i> is an invalid string list pointer.
<code>MDLERR_BADINDEX</code>	<i>memberIndex</i> is out of range.
<code>MDLERR_INFOMEMORY</code>	Memory is insufficient to reallocate string list.

See Also `mdlStringList_insertMember`.

mdlStringList_copy

```
#include <mdlerrs.h>
#include <dlogitem.h>
#include <msstrngl.fdf>

int mdlStringList_copy
(
    StringList *toStringListP,    /* => destination stringlist */
    ...
```

```
StringList *fromStringListP          /* => stringlist to copy */
);
```

Description The mdlStringList_copy function copies the string members from one string list to another.

toStringListP must have been created by a call to mdlStringList_create. All information currently in *toStringListP* will be lost. *toStringListP* (including the information fields) will be sized to the same size as *fromStringListP*.

Returns The mdlStringList_copy function returns SUCCESS if the operation is successful. Otherwise, it returns one of the following errors:

Return Value	Description
MDLERR_ADDRNOTVALID	<i>stringListP</i> is an invalid string list pointer.
MDLERR_INFSMEMORY	Memory is insufficient to copy items.

See Also mdlStringList_moveMembers.

mdlStringList_size

```
#include <dlogitem.h>
#include <msstrngl.fdf>

int mdlStringList_size
(
StringList *stringListP          /* => stringlist to size */
);
```

Description mdlStringList_size returns the number of members in the string list.

Returns mdlStringList_size returns -1 if an invalid address was passed. Otherwise, it returns the number of members in the string list.

See Also mdlStringList_numInfoFields, mdlStringList_getMember.

mdlStringList_numInfoFields

```
#include <dlogitem.h>
#include <msstrngl.fdf>

int mdlStringList_numInfoFields
(
StringList *stringListP          /* stringlist to work on */
);
```

Description mdlStringList_numInfoFields returns the number of information fields expected on each member of the string list.

Returns mdlStringList_numInfoFields returns -1 if an invalid address is passed. Otherwise, it returns the number of information fields associated with the string list.

See Also mdlStringList_size, mdlStringList_getMember.

mdlStringList_moveMembers

```
#include <mdlerrs.h>
#include <dlogitem.h>
#include <msstrngl.fdf>

int mdlStringList_moveMembers
(
StringList  *stringListP, /* => stringlist to work on */
int         toIndex,      /* => move members to here */
int         fromIndex,    /* => from here */
int         numToMove     /* => number of members to move */
);
```

Description mdlStringList_moveMembers moves *numToMove* members from *fromIndex* to *toIndex*. Data for any members already occupying the destination locations will be lost. The previous locations of moved members are set to empty.

Overlapping is handled properly; if the destination overlaps the origin, all of the data being moved is retained and moved to the proper destination locations.

Attempting to move members before the beginning or after the end of the list is illegal. In this case, mdlStringList_moveMembers will return MDLERR_BADARG.

Returns The mdlStringList_moveMembers function returns SUCCESS if the operation is successful. Otherwise, it returns one of the following errors:

Return Value	Description
MDLERR_ADDRNOTVALID	<i>stringListP</i> is an invalid string list pointer.
MDLERR_BADINDEX	<i>toIndex</i> and/or <i>fromIndex</i> are out of range.
MDLERR_BADARG	Members are referenced past the end of the string list.

See Also mdlStringList_insertMember, mdlStringList_deleteMember.

mdlStringList_sortByIndex

```
#include <mdlerrs.h>
#include <dlogitem.h>
#include <msstrngl.fdf>

int mdlStringList_sortByIndex /* <= SUCCESS or error code */
(
StringList  *stringListP,      /* => list to sort */
int         numToSort,         /* => # of rows to sort */
boolean     ascending,         /* => FALSE means descending */
);
```

```
MdlFunctionP sortFunc,          /* => NULL = use strcmp */
int          numColumns,        /* => number of columns in list */
int          startMember        /* => 0-based row number */
);
```

Description The mdlStringList_sortByIndex function sorts the members in a string list. It is the most general purpose of the functions that sort string lists.

stringListP is a pointer to the string list to be sorted. If this is `NULL` or points to something other than a string list, mdlStringList_sortByIndex returns the error indicator MDLERR_ADDRNOTVALID.

numToSort specifies the number of members (rows) to sort.
(*numToSort* \nless *numColumns*) gives the number of string list entries affected by the sort. If *numToSort* is invalid (i.e., is less than one or greater than the number of items), all items will be sorted.

If *ascending* is `TRUE`, the sort is ascending based on the values returned by *sortFunc*.

sortFunc specifies a function that will be used to compare members in the sort. The format of the function is as follows:

```
int sortFunc
(
int index1,
int index2
);
```

Both *index1* and *index2* are indices into the string list. The function must return -1 if it considers the entry indexed by *index1* to be less than the entry indexed by *index2*, 0 if it considers then equal, or 1 if it considers the entry indexed by *index1* to be greater than the entry indexed by *index2*. Remember that the string list can have `NULL` members, and that the string list functions turn 0-length strings into `NULL` members.

numColumns specifies the number of columns the string list represents.

startMember specifies the first member (row) included in the sort.

mdlStringList_sortByIndex starts the sort using the index (*startIndex* \nless *numColumns*).

Returns mdlStringList_sortByIndex returns `SUCCESS` if it does not detect any errors. MDLERR_ADDRNOTVALID is returned if *stringListP* does not point to a string list. MDLERR_BADINDEX is returned if *startMember* is an index that is less than 0 or greater than or equal to the number of elements in the string list. MDLERR_BADARG is returned if number of entries in the string list is not a multiple of *numColumns*.

See Also mdlStringList_sort, mdlStringList_sortByColumn, mdlStringList_search, mdlStringList_binarySearch, mdlStringList_searchByColumn, mdlStringList_binarySearchByColumn.

mdlStringList_sort, mdlStringList_sortByColumn

```

#include <mdlerrs.h>
#include <dlogitem.h>
#include <msstrngl.fdf>

int mdlStringList_sort
(
StringList  *stringListP, /* => stringlist to sort */
int         numToSort,    /* => # of members to sort, -1=all */
boolean     ascending,    /* => FALSE=descending */
MdlFunctionP sortFunc     /* => NULL=use strcmp w/strings */
);

int mdlStringList_sortByColumn
(
StringList  *stringListP, /* => stringlist to sort */
int         numToSort,    /* => # of members to sort, -1=all */
boolean     ascending,    /* => FALSE=descending */
MdlFunctionP sortFunc,    /* => NULL=use strcmp w/strings */
int         numColumns,   /* => # of cols in stringlist */
int         columnIndex   /* => Column to sort */
);

```

Description *mdlStringList_sort* and *mdlStringList_sortByColumn* sort the members in the string list. When used in conjunction with the *ListBox* manager, it is more appropriate to sort by column.

numToSort specifies the number of members to sort, starting from member 0. This argument is most useful when the number of allocated members exceeds the members that are actually set and all set members are continuous from member 0. An invalid value specifies all members will be sorted.

If *ascending* is *TRUE*, the sort is ascending based on the strings within members. Otherwise, it is descending. This option is overridden by *sortFunc*.

If *sortFunc* is not *NULL*, the function it points to will be used to compare members in the sort. The format of the function is as follows:

```

int sortFunc
(
char *member1strP,
long *member1infoFieldsP,
char *member2strP,
long *member2infoFieldsP
);

```

The valid return values from *sortFunc* are: -1, 0, and 1 depending on whether the first string and infofield pair are less than, equal to or greater than the second pair.



You are responsible for handling `NULL` members. If the string list contains `NULL` elements, you cannot pass `NULL` for the *sortfunc* parameter because `strcmp` will not accept `NULL`s. You must write your own sort function.

numColumns specifies the number of columns the string list represents.
columnIndex specifies which column in the string list to sort.

Returns `mdlStringList_sort` and `mdlStringList_sortByColumn` return `SUCCESS` if the operation is successful. Otherwise, they return one of the following errors:

`MDLERR_ADDRNOTVALID` - *stringListP* is an invalid string list pointer.

`MDLERR_BADINDEX` - *columnIndex* is out of range.

`MDLERR_BADARG` - Number of entries in the string list is not a multiple of *numColumns*.

See Also `mdlUtil_quickSort`, `mdlStringList_search`, `mdlStringList_binarySearch`, `mdlStringList_searchByColumn`, `mdlStringList_binarySearchByColumn`.

mdlStringList_search, mdlStringList_searchByColumn

```
#include <mdlerrs.h>
#include <dlogitem.h>
#include <msstrngl.fdf>

int mdlStringList_search
(
    int          *indexP,          /* <= loc of found elm or where to add it */
    StringList   *stringListP,     /* => stringlist to search */
    char         *stringP,         /* => string to find or NULL */
    long         *infoFieldsP,     /* => infofield(s) to find or NULL */
    int          startIndex,       /* => where to start search */
    MdlFunctionP searchFun        /* => NULL=use strcmp w/ strings */
);

int mdlStringList_searchByColumn
(
    int          *indexP,          /* <= loc of found elm or where to add it */
    StringList   *stringListP,     /* => stringlist to search */
    char         *stringP,         /* => string to find or NULL */
    long         *infoFieldsP,     /* => infofield(s) to find or NULL */
    int          startIndex,       /* => where to start search */
    MdlFunctionP searchFunc,       /* => NULL=use strcmp w/strings */
    int          numColumns,       /* => # of cols in stringlist */
    int          columnIndex      /* => column to search */
);
```

Description `mdlStringList_search` and `mdlStringList_searchByColumn` search the string list for the given data, starting from the member at *startIndex*. When used in conjunction with the `ListBox` manager, it is more appropriate to search by column.

If the function finds the member, it returns `SUCCESS` and *indexP* returns this member. If the function does not find the member, it returns `MDLERR_NOMATCH` and *indexP* returns the location where the member should be inserted. Otherwise, a value of -1 will be returned for *indexP*.

If *searchFunc* is `NULL`, the search compares member string elements and ignores *infoFieldsP*.

If *searchFunc* is not `NULL`, it will be used to compare members in the search. The format of the function is as follows:

```
int searchFunc
(
char *searchStringP,
long *searchInfoFieldsP,
char *memberStringP,
long *memberInfoFieldsP
);
```

searchFunc should return 0 if the first string and infofield pair are not equal to the second pair and 1 if they are equal.



You are responsible for handling `NULL` members. If the string list contains `NULL` elements, you cannot pass `NULL` for the *sortfunc* parameter because `strcmp` will not accept `NULL`s. You must write your own sort function.

numColumns specifies the number of columns the string list represents.

columnIndex specifies which column in the string list to search.

Returns `mdlStringList_search` and `mdlStringList_searchByColumn` return `SUCCESS` if the operation is successful. Otherwise, it returns one of the following errors:

Return Value	Description
<code>MDLERR_ADDRNOTVALID</code>	<i>stringListP</i> is an invalid string list pointer.
<code>MDLERR_BADINDEX</code>	<i>startIndex</i> or <i>columnIndex</i> is out of range.
<code>MDLERR_BADARG</code>	<i>stringP</i> and <i>searchFunc</i> are <code>NULL</code> or number of entries in the string list is not a multiple of <i>numColumns</i> .
<code>MDLERR_NOMATCH</code>	The item was not found in the string list.

See Also `mdlStringList_binarySearch`, `mdlStringList_binarySearchByColumn`, `mdlStringList_sort`, `mdlStringList_sortByColumn`.

mdlStringList_binarySearch, mdlStringList_binarySearchByColumn

```

#include <mdlerrs.h>
#include <dlogitem.h>
#include <msstrngl.fdf>

int mdlStringList_binarySearch
(
    int          *indexP,          /* <= loc of found elm or where to add it */
    StringList   *stringListP,     /* => stringlist to search */
    char         *stringP,         /* => string to find or NULL */
    long         *infoFieldsP,     /* => infofield(s) to find or NULL */
    MdlFunctionP bSearchFunc       /* => NULL=use strcmp w/strings */
);

int mdlStringList_binarySearchByColumn
(
    int          *indexP,          /* <= loc of found elm or where to add it */
    StringList   *stringListP,     /* => stringlist to search */
    char         *stringP,         /* => string to find or NULL */
    long         *infoFieldsP,     /* => infofield(s) to find or NULL */
    MdlFunctionP bSearchFunc,      /* => NULL=use strcmp w/strings */
    int          numColumns,       /* => # of cols in stringlist */
    int          columnIndex      /* => column to search */
);

```

Description mdlStringList_binarySearch and mdlStringList_binarySearchByColumn perform a binary search of the string list for the given data. The user must ensure that the string list was previously sorted. When used in conjunction with the ListBox manager, it is more appropriate to search by column.

If the function finds the member, it returns **SUCCESS** and *indexP* returns this member. If the function does not find the member, it returns **MDLERR_NOMATCH** and *indexP* returns the location where the member should be inserted. Otherwise, the function returns -1.

If *bSearchFunc* is **NULL**, the search compares member string elements as specified by *stringP* and ignores *infoFieldsP*.

If *bSearchFunc* is not **NULL**, it will be used to compare members in the search. *stringP* and *infoFieldsP* will be passed as arguments to *bSearchFunc*. The format of the function is as follows:

```

int bSearchFunc
(
    char *searchStringP,
    long *searchInfoFieldsP,
    char *memberStringP,

```

```
long *memberInfoFieldsP
);
```

The valid return values from `bSearchFunc` are: -1, 0 and 1 depending on whether the first string and infofield pair are less than, equal to, or greater than the second pair.



You are responsible for handling `NULL` members. If the string list contains `NULL` elements, you cannot pass `NULL` for the *sortfunc* parameter because `strcmp` will not accept `NULL`s. You must write your own sort function.

numColumns specifies the number of columns the string list represents.

columnIndex specifies which column in the string list to search.

Returns The `mdlStringList_binarySearch` and `mdlStringList_binarySearchByColumn` functions return `SUCCESS` if the operation is successful. Otherwise, it returns one of the following errors:

Return Value	Description
<code>MDLERR_ADDRNOTVALID</code>	<i>stringListP</i> is an invalid string list pointer.
<code>MDLERR_BADINDEX</code>	<i>columnIndex</i> is out of range.
<code>MDLERR_BADARG</code>	<i>stringP</i> and <i>searchFunc</i> are <code>NULL</code> or number of entries in the string list is not a multiple of <i>numColumns</i> .
<code>MDLERR_NOMATCH</code>	The item was not found in the string list.

See Also `mdlStringList_search`, `mdlStringList_searchByColumn`, `mdlStringList_sort`, `mdlStringList_sortByColumn`.

mdlStringList_addResource, mdlStringList_addResourceWithType

```
#include <mdlerrs.h>
#include <rtypes.h>
#include <rsdefs.h>
#include <dlogitem.h>
#include <msstrngl.fdf>

int mdlStringList_addResource
(
    RscFileHandle    rfHandle,        /* => resource file handle */
    unsigned long    resourceID,      /* => id of resource to save */
    StringList *stringListP /* => stringlist to be saved */
);

int mdlStringList_addResourceWithType
(
    RscFileHandle    rfHandle,        /* => resource file handle */
    unsigned long    resourceclass, /* => user defined rsc type */
```

```

unsigned long    resourceID,    /* => id of resource to save */
StringList *stringListP    /* => stringlist to be saved */
);

```

Description The `mdlStringList_addResource` function adds the string list to the specified resource file with the given ID. If a string list resource with the same resource ID already exists, the string list will not be added.

`mdlStringList_addResourceWithType` is used to add a string list with a user-defined resourceclass. Calling `mdlStringList_addResource` is equivalent to calling `mdlStringList_addResourceWithType` with a resourceclass of `RTYPE_STRINGLIST`.

rfHandle is the resource file handle of the file to receive the string list.

resourceclass (only necessary for `mdlStringList_addResourceWithType`) specifies the resourceclass of the string list. A resourceclass is sometimes referred to as a type ID.

resourceID will be the resource identifier of the string list.

stringListP is a pointer to the string list to be added.

Returns `mdlStringList_addResource` and `mdlStringList_addResourceWithType` return `SUCCESS` if the operation is successful. Otherwise, they return the following string list error or a resource manager error:

Return Value	Description
<code>MDLERR_ADDRNOTVALID</code>	<i>stringListP</i> is an invalid string list pointer.

See Also `mdlStringList_writeResource`, `mdlStringList_loadResource`.

mdlStringList_writeResource, mdlStringList_writeResourceWithType

```

#include <mdlerrs.h>
#include <rtypes.h>
#include <dlogitem.h>
#include <rsdefs.h>
#include <msstrngl.fdf>

int mdlStringList_writeResource
(
StringList *stringListP, /* => stringlist to be saved */
RscFileHandle rfHandle, /* => resource file handle */
unsigned long resourceID /* => id of resource to save */
);

int mdlStringList_writeResourceWithType
(
StringList *stringListP, /* => stringlist to be saved */
RscFileHandle rfHandle, /* => resource file handle */

```

```

unsigned long    resourceclass, /* => user defined rsc type */
unsigned long    resourceID    /* => id of resource to save */
);

```

Description The `mdlStringList_writeResource` function adds the string list to the specified resource file with the given ID. A string list resource existing with the same resource ID will be overwritten. `mdlStringList_writeResourceWithType` is used to create a string list with a user-defined resourceclass. Calling `mdlStringList_writeResource` is equivalent to calling `mdlStringList_writeResourceWithType` with a resourceclass of `RTYPE_STRINGLIST`.

stringListP is a pointer to the string list to be saved.

rfHandle is the resource file handle of the file to receive the string list.

resourceclass (necessary for `mdlStringList_writeResourceWithType` only) specifies the resourceclass of the string list. A resourceclass is sometimes referred to as a type ID.

resourceID will be the resource identifier of the string list.

Returns `mdlStringList_writeResource` and `mdlStringList_writeResourceWithType` return `SUCCESS` if the operation is successful. Otherwise, they return the following string list error or a resource manager error:

Return Value	Description
<code>MDLERR_ADDRNOTVALID</code>	<i>stringListP</i> is an invalid string list pointer.

See Also `mdlStringList_addResource`, `mdlStringList_loadResource`.

mdlStringList_loadResource, mdlStringList_loadResourceWithType

```

#include <rtypes.h>
#include <rscdefs.h>
#include <dlogitem.h>
#include <msstrngl.fdf>

StringList *mdlStringList_loadResource
(
RscFileHandle    rfHandle,      /* => resource file handle */
unsigned long    resourceID     /* => id of resource to read */
);

StringList *mdlStringList_loadResourceWithType
(
RscFileHandle    rfHandle,      /* => resource file handle */
unsigned long    resourceclass, /* => user defined rsc type */
unsigned long    resourceID     /* => id of resource to read */
);

```

Description The `mdlStringList_loadResource` function loads a string list with the given ID from the specified file. `mdlStringList_loadResourceWithType` is used when the string list to be loaded has a user-defined resourceclass. Calling `mdlStringList_loadResource` is equivalent to calling `mdlStringList_loadResourceWithType` with a resourceclass of `RTYPE_STRINGLIST`. When the string list is no longer needed, it should be freed using the `mdlStringList_destroy` function.

rfHandle specifies the resource file from which the string list is to be loaded.

resourceclass (only necessary for `mdlStringList_loadResourceWithType`) specifies the resourceclass of the string list. A resourceclass is sometimes referred to as a type ID.

resourceID specifies the resource ID of the string list.

Returns `mdlStringList_loadResource` and `mdlStringList_loadResourceWithType` return a pointer to a string list if they are successful. Otherwise, they return `NULL`.

See Also `mdlStringList_addResource`, `mdlStringList_writeResource`, `mdlStringList_destroy`.

String Functions

The string functions are used to interpret and manipulate user-entered strings.

Function	Used to
<code>mdlString_toPoint</code>	convert a string in MU:SU:PU, MU:SU:PU, MU:SU:PU format to a point in a <code>Dpoint3d</code> structure.
<code>mdlString_fromPoint</code>	convert a point in a <code>Dpoint3d</code> structure to a string in MU:SU:PU, MU:SU:PU, MU:SU:PU format.
<code>mdlString_toAngle</code>	convert a string in DD°MM'SS" or decimal degrees format to a double.
<code>mdlString_fromAngle</code>	convert a double to a string in DD°MM'SS" or decimal degrees format.
<code>mdlString_fromDirection</code>	convert a double to a string specifying direction.
<code>mdlString_toUors</code>	convert a string in MU:SU:PU format to a double.
<code>mdlString_fromUors</code>	convert a double to a string in MU:SU:PU format.

Function	Used to
<code>mdlString_matchRE</code>	search the string for a match with the regular expression.
<code>mdlString_matchREExtended</code>	provide a way to stop matching the symbol '^' (beginning of line) and symbol '\$' (end of line) recursively.
<code>mdlString_setFunction</code>	designate a user-supplied function called by MicroStation when it needs the application to manipulate entered string(s).

Function	MicroStation call when
<code>userString_toUors</code>	the user enters a string that must be converted to UORs.
<code>userString_fromUors</code>	formatting a string representing a coordinate for presentation to the user.
<code>userString_fromDirection</code>	formatting a string representing a direction for presentation to the user.
<code>userString_fromAngle</code>	formatting a string representing an angle for presentation to the user.
<code>userString_toAngle</code>	the user enters a string that must be converted to an angle.

mdlString_toPoint

```
#include <mdl.h>

int mdlString_toPoint
(
    Dpoint3d    *pointP,          /* <=> receives the point */
    char        **leftoverP,      /* <=> portion of string not used. */
    char        *stringP,        /* => string to convert */
    int         threeD            /* => should be a 3D string */
);
```

Description The `mdlString_toPoint` function converts the string *stringP* in working units format (MU:SU:PU, MU:SU:PU, MU:SU:PU) to a point in *pointP*. It does not adjust for global origin. The coordinates stored in *pointP* are UORs.

The point *leftoverP* points to the first character of the string not included in the conversion. *leftoverP* can be NULL.

If *threeD* is TRUE, mdlString_toPoint tries to convert three coordinate specifications to supply values for *pointP*->*x*, *pointP*->*y* and *pointP*->*z*. Otherwise, it tries to convert two specifications to supply values for *pointP*->*x* and *pointP*->*y*.

Returns mdlString_toPoint returns SUCCESS if the coordinates in *stringP* are in valid working units format. Otherwise, it returns ERROR.

mdlString_fromPoint

```
#include <mdl.h>

void mdlString_fromPoint
(
    char    *stringP,      /* <=> where string is stored */
    Dpoint3d *pointP,      /* => point specification. */
    int     origin        /* => TRUE means subtract global origin */
);
```

Description The mdlString_fromPoint function creates a string in working units format (MU:SU:PU, MU:SU:PU, MU:SU:PU) based on the point in *pointP*. The string is stored in the buffer that *stringP* points to. The coordinates in *pointP* are UORs.

If *origin* is TRUE, mdlString_fromPoint subtracts the global origin before creating the string.

If MicroStation is operating on a 3D file, mdlString_fromPoint generates coordinates for three dimensions. Otherwise, it generates coordinates for two dimensions.

Returns mdlString_fromPoint is of type void; it returns no value.

mdlString_toAngle

```
#include <mdl.h>

int mdlString_toAngle
(
    double *angleP, /* <=> receives the angle */
    char    *stringP /* => point specification. */
);
```

Description The mdlString_toAngle function converts the string in *stringP* to an angle. mdlString_toAngle returns the angle in *angleP*. The value in *angleP* is in degrees.

The angle in *stringP* can have decimal degrees format (DDD.DDD) or degrees-minutes-seconds format (DD°MM'SS").

Returns mdlString_toAngle returns ERROR if a syntax error occurs in parsing the string. Otherwise, it returns SUCCESS.

mdlString_fromAngle

```
#include <mdl.h>

void mdlString_fromAngle
(
    char    *stringP,          /* <=> point specification */
    double  angle,             /* => the angle */
    int     dms,               /* => TRUE means display in DD°MM'SS" */
    int     igdsChars,         /* => TRUE means use IGDS degrees char */
    int     decimalPlaces,     /* => decimal places to display */
    int     trailingZeros      /* => TRUE means allow trailing zeros */
);
```

Description The `mdlString_fromAngle` function converts the angle in *angle* to a string in the buffer pointed to by *stringP*.

angle is an angle specified in degrees.

If *dms* is non-zero, the string's format is degrees-minutes-seconds (DD°MM'SS"). Otherwise, it is specified in degrees.

Certain hardware platforms (currently only the Macintosh) have a special character for degrees. If *igdsChars* is non-zero, this character is substituted for the standard IGDS degrees symbol, '^'. On MS-DOS machines and Intergraph workstations, this argument is not used.

decimalPlaces specifies the number of decimal places to use in the display. If *decimalPlaces* is -1, `mdlString_fromAngle` uses the active setting specified in the Accuracy field of the Coordinate Readout settings box.

If *trailingZeros* is FALSE, all trailing zeros are removed from the string.

Returns The `mdlString_fromAngle` function is of type `void`. It returns no value.

mdlString_fromDirection

```
#include <mdl.h>

void mdlString_fromDirection
(
    char    *stringP,          /* <=> point specification. */
    double  *angleP,           /* => receives the angle */
    int     dms,               /* => TRUE means display in DD°MM'SS" */
    int     igdsChars,         /* => TRUE means use ^ for degrees */
    int     trueNorth,         /* => TRUE means adjust for true north */
    int     angleMode,         /* => specifies string format */
    int     decimalPlaces,     /* => decimal places to display */
    int     trailingZeros      /* => TRUE means allow trailing zeros */
);
```

Description The mdlString_fromDirection function converts the angle in *angleP* to a string in the buffer pointed to by *stringP*.

The value pointed to by *angleP* is an angle specified in seconds.

If *dms* is non-zero, string format is degrees-minutes-seconds (DD°MM'SS"). Otherwise, it is specified in degrees.

Certain hardware platforms (currently only the Macintosh) have a special character for degrees. If *igdsChars* is non-zero, this character is substituted for the standard IGDS degrees symbol, '^'. On MS-DOS machines and Intergraph workstations, this argument is not used.

If *trueNorth* is TRUE, the angle is adjusted for true north.

If *angleMode* is 0, the angle displays in conventional mode. If *angleMode* is 1, the string is an azimuth angle. If *angleMode* is 2, the string is a bearing angle. If *angleMode* is -1, the format is determined by the value of the Mode field in the Coordinate Readout settings box.

decimalPlaces specifies the number of decimal places to use in the display. If *decimalPlaces* is -1, mdlString_fromDirection uses the active setting specified in the Accuracy field of the Coordinate Readout settings box.

If *trailingZeros* is FALSE, all trailing zeros are removed from the string.

Returns The mdlString_fromDirection function is of type void. It returns no value.

mdlString_toUors

```
#include <mdl.h>

int mdlString_toUors
(
    double *uors,          /* <=> receives generated value */
    char *string           /* => string in MU:SU:PU format */
);
```

Description The mdlString_toUors function converts the string in *string* to a double in *uors*. The string must have working units format (MU:SU:PU).

The leftmost number in the string is always interpreted as master units. "MU" can be used to compute master units. " : : PU" can be used for primary units.

Returns The mdlString_toUors function returns SUCCESS if *string* points to a string in valid working units format. Otherwise, it returns ERROR.

See Also mdlString_fromUors, sscanf format strings.

mdlString_fromUors

```
#include <mdl.h>

void mdlString_fromUors
(
    char    *string, /* <=> buffer to receive string */
    double  uors     /* => value to convert */
);
```

Description The `mdlString_fromUors` function creates a string in working units format (MU:SU:PU) using the value provided in *uors*.

The string is stored in the buffer that *string* points to. Units are not stored in the string.

Returns The `mdlString_fromUors` function is of type `void`. It returns no value.

See Also `mdlString_toUors`, `printf` format strings.

mdlString_matchRE

```
#include <mdl.h>

int mdlString_matchRE
(
    char    *string,           /* => string to search */
    char    *regularExpression, /* => expression to search for */
    char    **start,          /* <=> substring start satisfying reg expresn */
    char    **end              /* <=> substring end satisfying reg expresn */
);
```

Description `mdlString_matchRE` searches the string pointed to by *string* for a match with the regular expression pointed to by *regularExpression*. When a match is found, *start* points to the first character of the match, and *end* points to the last character in the match.

Regular expressions are a powerful string matching mechanism and are defined by the following rules:

Character	Meaning
c	any non-special character c matches itself.
\c	turn off special meaning of character c .
^	beginning of line.
\$	end of line.
.	any single character.
:a	any alphabetic character [a-z A-Z] .
:d	any digit [0-9] .
n	any alphanumeric character [a-z A-Z 0-9] .

Character	Meaning
	A colon followed by a space also matches any punctuation character.
[...]	any one of characters in ...; ranges like a-z are legal.
[^...]	any single character not in ...; ranges are legal.
s*	zero or more occurrences of string s .
s+	one or more occurrences of string s .
st	string s followed by string t .

Returns The mdlString_matchRE function returns **SUCCESS** if a match is found and if *start* and *end* are valid. It returns 1 if no match is found, and 2 if *regularExpression* is an invalid regular expression.

mdlString_matchREExtended

```
#include <mdl.h>

int mdlString_matchREExtended
(
    char    *string,                /* => string to search */
    char    *regularExpression,     /* => expression to search for */
    char    **start,               /* <=> substring start satisfying reg. expr. */
    char    **end,                /* <=> substring end satisfying reg. expr. */
    int     *stopState             /* <=> stop match "^" or "$" recursively */
);
```

Description mdlString_matchREExtended is similar to mdlString_matchRE except that it provides a way to stop matching the symbol '^' (beginning of line) and symbol '\$' (end of line) recursively.

stopState must be initialized to zero before mdlString_matchREExtended is called. When *stopState* is zero, mdlString_matchREExtended will set the first bit of the content of *stopState* to 1.

If *regularExpression* contains '^', and set the second bit of the content of *stopState* to 1.

If *regularExpression* contains '\$', and set both the first bit and the second bit of the content of *stopState* to 1.

If *regularExpression* contains both '^' and '\$'.

When *stopState* is non-zero, mdlString_matchREExtended will not match anything, but return **SUCCESS**.

Example Assume *textStringP* points to a multiline text. If you want to prepend a string to the beginning of each line, the beginning of line should be matched only once.

Otherwise, the beginning of the new string (after prepending) will be matched again.

```
char *start, *end;
int stopState = 0;

while (!stopState && mdlString_matchREExtended(textStringP, '^',
    &start, &end, &stopState)
{
    ... do prepending
}
```

See Also For description of other arguments and return values, see `mdlString_matchRE`.

mdlString_setFunction

```
#include <userfnc.h>

void mdlString_setFunction
(
    int          type,      /* => type of string operation */
    MdlFunctionP function /* => function to perform that operation */
);
```

Description MicroStation provides a method for an application to affect the interpretation of user-entered strings representing input coordinates and angles, and to affect the formatting of coordinates and angles on output. The `mdlString_setFunction` function is used to designate functions in the application program that MicroStation calls to perform those operations. The *type* parameter must be one of the values in the table below, and the function that MicroStation calls is designated in the *functionP* parameter. MicroStation maintains a separate function for each of the

possible string operations. The possible *type* values and the circumstances under which the corresponding function is called are tabulated below:

type	<i>function called</i>
STRING_TO_UORS	when a user-entered coordinate string needs to be interpreted as a UOR value (see <code>userString_toUors</code>).
STRING_FROM_UORS	when design file coordinates are being formatted for presentation to the user (see <code>userString_fromUors</code>).
STRING_TO_ANGLE	when a user-entered angle string needs interpretation (see <code>userString_toAngle</code>).
STRING_FROM_ANGLE	when angle is formatted for presentation to user (see <code>userString_fromAngle</code>).
STRING_FROM_DIRECTION	when a direction is being formatted for presentation to the user (see <code>userString_fromDirection</code>).

To stop MicroStation from calling an application function for any of the string operations described above, call `mdlString_setFunction` with the appropriate value of *type* and `NULL` for the *function* parameter.

Returns `mdlString_setFunction` is of type `void`. It does not return a value.

See Also `userString_toUors`, `userString_fromUors`, `userString_toAngle`, `userString_fromAngle`, `userString_fromDirection`.

userString_toUors

```
#include <userfnc.h>
#include <mdl.h>

int userString_toUors
(
    double *uorsP,          /* <= output value, uors */
    char *uorString         /* => user-entered UOR string */
);
```

Returns If an MDL application designates it as a `STRING_TO_UORS` function, `userString_toUors` is called whenever the user enters a string that must be converted to UORs. The application programmer determines the function name; `userString_toUors` is used merely as an example.

The function is passed a pointer to the string that the user entered in *uorString*. The function should return the UORs represented by that string in the address passed to in the *uorsP* parameter. The output should not be corrected for the global origin nor for the current transformation.

Returns `userString_toUors` should return zero if it successfully parses the input string to UORs. If it returns any other value, MicroStation will attempt to parse the input string to UORs using its standard algorithm.

See Also `mdlString_setFunction`.

userString_fromUors

```
#include <userfnc.h>
#include <mdl.h>

int userString_fromUors
(
    char    *uorStringP,          /* <= formatted UOR string */
    double  uors,                 /* => input value, uors */
    ULong   iMasterUnits,         /* => integer master units */
    ULong   iSubUnits,           /* => integer sub units */
    ULong   iPositionalUnits,     /* => integer positional units */
    double  masterUnits,          /* => double precision master units */
    double  subUnits,             /* => double precision sub units */
    boolean negativeFlag,         /* => TRUE=negative. */
    boolean *unitFlag,           /* => TRUE=unit labels in output */
    char    masterUnitsLabel,     /* => master unit name */
    char    subUnitsLabel,        /* => subunit name */
    int     *formatP,             /* <=> format desired */
    boolean *useFractionsP,       /* <=> TRUE=fractions, FALSE=decimal */
    int     *precisionP           /* <=> output precision */
);
```

Description If an MDL application designates it as a `STRING_FROM_UORS` function, *userString_fromUors* is called whenever MicroStation is formatting a string representing a coordinate for presentation to the user. The application programmer determines the function name; *userString_fromUors* is used merely as an example.

MicroStation passes to this routine the user settings for format, whether fractions are desired, and the desired precision. These parameters are passed by reference so that the *userString_fromUors* function can either use them in its formatting algorithm, or change them and return a nonzero result. In the latter case, the nonzero result indicates that *userString_fromUors* has not formatted the string, and thus MicroStation proceeds to format the output with the changed formatting parameters.

The possible values for **formatP* are 0 for reports in master units/sub units, 1 for master units only, and 2 for working units (master:sub:positional).

If **useFractions* is FALSE, then **precisionP* is the number of digits to the right of the decimal place that should be displayed. If **useFractions* is TRUE, then **precisionP* is the denominator of the fraction that should be displayed.

MicroStation does the work of determining the sign of the input UOR value, and rationalizing the UOR value into unsigned long integer values

of master units, sub units and positional units. It passes these values to the *userString_fromUors* function in *iMasterUnits*, *iSubUnits* and *iPositionalUnits*, respectively. If the input *uors* is negative, *negativeFlag* is TRUE, otherwise it is FALSE.

If *formatP* is 1 (calling for master units), you will probably find it convenient to use the double precision value of master units passed to *userString_fromUors* in the *masterUnits* parameter. If *formatP* is 0 (calling for master / subunits), the integer master units value, *iMasterUnits*, together with the double precision subunits remainder, which is passed in as *subUnits* will be helpful. Finally, if working units are requested (*formatP* is 2), the three integer values will be the most helpful.

If the units should be put into the output string, the *unitFlag* parameter will be TRUE. In this case, the master and sub unit labels are passed in to the routine as the *masterUnitsLabel* and *subUnitsLabel*, respectively.

Returns *userString_fromUors* should return zero if it successfully formats the inputs to a string. If it returns any other value, MicroStation formats the string according to its internal algorithm, using the changes made to the formatting parameters as discussed above.

See Also *mdlString_setFunction*.

userString_fromDirection

```
#include <userfnc.h>
#include <mdl.h>

int userString_fromDirection
(
    char    *dirStringP,          /* <= formatted direction string */
    double  angle,                /* => direction angle in degrees */
    int     degreeChar,          /* => character to use for degree */
    boolean *trueNorthP,         /* <=> correct for True North */
    int     *angleModeP,         /* <=> output mode */
    boolean *dmsFlagP,          /* <=> TRUE=degrees, min., sec. */
    int     *precisionP,        /* <=> output precision */
    boolean *trailingZerosP     /* <=> TRUE=trailing zeros desired */
);
```

Description If an MDL application designates it as a *STRING_FROM_DIRECTION* function, *userString_fromDirection* is called whenever MicroStation is formatting a string representing a direction for presentation to the user. The application programmer determines the function name; *userString_fromDirection* is used merely as an example.

MicroStation passes to this routine the user settings for format, the desired precision, and whether or not trailing zeros are desired. These parameters are passed by reference so that the *userString_fromDirection* function can either use them in its formatting algorithm, or change them and return

a nonzero result. In the latter case, the nonzero result indicates that *userString_fromDirection* has not formatted the string, and thus MicroStation proceeds to format the output with the changed formatting parameters. If MicroStation proceeds with its formatting of the string, the application's `STRING_FROM_ANGLE` function will be called (if it has one) to do some of the actual formatting.

If **trueNorthP* is `TRUE`, the angle should be corrected for the design file's True North value. This is accomplished by subtracting `tcb->azimuth` from the input angle. Note that MicroStation never corrects for True North if **angleModeP* is zero.

The **angleModeP* parameter is 0 for Conventional readout, 1 for Azimuth readout, and 2 for bearing angle readout.

If **dmsFlagP* is `TRUE`, output is in degrees, minutes, and seconds format. Otherwise, output is in decimal degrees.

The **precisionP* parameter gives the number of digits to the right of the decimal point that should be presented. If **trailingZerosP* is `TRUE`, trailing zeros are included.

Returns *userString_fromDirection* should return zero if it successfully formats the inputs to a string. If it returns any other value, MicroStation formats the string according to its internal algorithm, using the changes made to the formatting parameters as discussed above.

See Also `mdlString_setFunction`.

userString_fromAngle

```
#include <userfnc.h>
#include <mdl.h>

int userString_fromAngle
(
    char    *angleStringP,      /* <= formatted angle string */
    double  angle,              /* => in value, angle in degrees */
    int     degreeChar,         /* => character to use for degree */
    boolean *dmsFlagP,          /* <=> TRUE = degrees, minutes, seconds */
    int     *precisionP,        /* <=> output precision */
    boolean *leadingZerosP,     /* <=> TRUE = leading zeros desired */
    boolean *trailingZerosP     /* <=> TRUE = trailing zeros desired */
);
```

Description If an MDL application designates it as a `STRING_FROM_ANGLE` function, *userString_fromAngle* is called whenever MicroStation is formatting a string representing an angle for presentation to the user. The application programmer

determines the function name; *userString_fromAngle* is used merely as an example.

MicroStation passes to this routine the user settings for format, the desired precision, and whether or not leading and trailing zeros are desired. These parameters are passed by reference so that the *userString_fromAngle* function can either use them in its formatting algorithm, or change them and return a nonzero result. In the latter case, the nonzero result indicates that *userString_fromAngle* has not formatted the string, and thus MicroStation proceeds to format the output with the changed formatting parameters.

If **dmsFlagP* is TRUE, output is in degrees, minutes and seconds format. Otherwise, output is in decimal degrees.

The **precisionP* parameter gives the number of digits to the right of the decimal point that should be presented. If **leadingZerosP* is TRUE, leading zeros are included, while if **trailingZerosP* is TRUE, trailing zeros are included.

Returns *userString_fromAngle* should return zero if it successfully formats the inputs to a string. If it returns any other value, MicroStation formats the string according to its internal algorithm, using the changes made to the formatting parameters as discussed above.

See Also *userString_toAngle*, *mdlString_setFunction*.

userString_toAngle

```
#include <userfnc.h>
#include <mdl.h>

int userString_toAngle
(
    double *angleP,          /* <= output value, degrees */
    char *angleString        /* => user-entered angle string */
);
```

Description If an MDL application designates it as a STRING_TO_ANGLE function, *userString_toAngle* is called whenever the user enters a string that must be converted to an angle. The application programmer determines the function name; *userString_toAngle* is used merely as an example.

The function is passed a pointer to the string that the user entered in *angleString*. The function should return the angle, in degrees represented by that string in the address passed to in the *angleP* parameter.

Returns *userString_toAngle* should return zero if it successfully parses the input string to an angle. If it returns any other value, MicroStation will attempt to parse the input string to an angle using its standard algorithm.

See Also *userString_fromAngle*, *mdlString_setFunction*.

License Management Functions

The following table lists the license management functions:

Function	Used to
<code>mdlLicense_getCurrentHardwareKeySerial</code>	get the serial number of the hardware key.
<code>mdlLicense_getHardwareKeyPermission</code>	allow evaluation of a package before a license is available if a hardware key is required.
<code>mdlLicense_getInitialHardwareKeySerial</code>	get the serial number of the hardware key present when the base product was launched.
<code>mdlLicense_getStrings</code>	read the current license information.
<code>mdlLicense_getVarietyName</code>	return the variety name if the product is a special variety.
<code>mdlLicense_isAcademicProduct</code>	determine if an academic version is in use.
<code>mdlLicense_isHardwareKeyRequired</code>	determine if this base product requires a hardware key.
<code>mdlLicense_isHardwareKeySupported</code>	determine if this base product supports hardware keys.
<code>mdlLicense_isRegisteredProduct</code>	determine if the product has completed registration.
<code>mdlLicense_isSerializedProduct</code>	determine if a serial number has been entered.

mdlLicense_getCurrentHardwareKeySerial

```
#include <mslicens.h>
#include <mslicens.fdf>

unsigned long mdlLicense_getCurrentHardwareKeySerial(void);
```

Description The `mdlLicense_getCurrentHardwareKeySerial` function returns the serial number of an attached hardware key. There must have been a positive result when the initial value was obtained in order to setup for hardware key processing. This call is mainly intended for determining if that key is still attached.



This function was implemented in MicroStation 95.

Returns `mdlLicense_getCurrentHardwareKeySerial` returns a serial number. The value 0 signifies that no hardware key was found.

mdlLicense_getHardwareKeyPermission

```
#include <mslicens.h>
#include <mslicens.fdf>

unsigned long mdlLicense_getHardwareKeyPermission
(
    unsigned long password, /* => a password */
    unsigned long maxdemo /* => maximum demo launches permitted */
);
```

Description The mdlLicense_getHardwareKeyPermission function is used to supply a product evaluation facility to programs that use hardware keys. This function should not be used once the product is licensed.

password is a value assigned to the application. It indexes evaluation counters in the hardware key.

maxdemo is a limit on the number of demonstration runs permitted.



This function was implemented in MicroStation 95.

Returns mdlLicense_getHardwareKeyPermission returns the number of program launches remaining including this one.

mdlLicense_getInitialHardwareKeySerial

```
#include <mslicens.h>
#include <mslicens.fdf>

unsigned long mdlLicense_getInitialHardwareKeySerial(void);
```

Description The mdlLicense_getInitialHardwareKeySerial function returns the serial number of the hardware key found when the base product was started.



This function was implemented in MicroStation 95.

Returns mdlLicense_getInitialHardwareKeySerial returns a serial number. The value 0 signifies that no hardware key was found.

mdlLicense_getStrings

```
#include <mslicens.h>
#include <mslicens.fdf>

void mdlLicense_getStrings
(
    BrandInfo *theInfoStrings /* <= the structure of strings */
);
```

Description The `mdlLicense_getStrings` function returns the five strings associated with licenses for Bentley products (Serial number, Name, Organization, Registration number, License number).

the *InfoStrings* is a pointer to a `BrandInfo` structure that the strings will be copied to.



This function was implemented in MicroStation 95.

Returns `mdlLicense_getStrings` is of type `void`. It returns no value.

mdlLicense_getVarietyName

```
#include <mslicens.h>
#include <mslicens.fdf>

int mdlLicense_getVarietyName /* <= count of chars copied */
(
    char    *dest      /* => place to copy (may be NULL) */
);
```

Description The `mdlLicense_getVarietyName` function retrieves a text string that describes any specialization of the base product. Standard MicroStation returns the empty string. English Academic MicroStation returns the string “ACADEMIC”.

dest is a pointer to memory which will receive the string.



This function was implemented in MicroStation 95.

Returns `mdlLicense_getVarietyName` returns the number of characters copied as defined by `strlen(string)`.

mdlLicense_isAcademicProduct

```
#include <mslicens.h>
#include <mslicens.fdf>

int mdlLicense_isAcademicProduct(void); /* <= TRUE or FALSE */
```

Description The `mdlLicense_isAcademicProduct` function reports if an academic version of the base product is in use.



This function was implemented in MicroStation 95.

Returns `mdlLicense_isAcademicProduct` returns `TRUE` if the base product is an academic version, otherwise `FALSE` is returned.

mdlLicense_isHardwareKeyRequired

```
#include <mslicens.h>
#include <mslicens.fdf>

int mdlLicense_isHardwareKeyRequired(void);/* <= TRUE / FALSE */
```

Description The mdlLicense_isHardwareKeyRequired function reports if the base product requires a hardware key.



This function was implemented in MicroStation 95.

Returns mdlLicense_isHardwareKeyRequired returns TRUE if the base product requires a hardware key, otherwise FALSE.

mdlLicense_isHardwareKeySupported

```
#include <mslicens.h>
#include <mslicens.fdf>

int mdlLicense_isHardwareKeySupported(void);/* <= TRUE/FALSE */
```

Description The mdlLicense_isHardwareKeySupported function reports if the base product contains support for a hardware key.



This function was implemented in MicroStation 95.

Returns mdlLicense_isHardwareKeySupported returns TRUE if hardware key support is present, otherwise FALSE.

mdlLicense_isRegisteredProduct

```
#include <mslicens.h>
#include <mslicens.fdf>

int mdlLicense_isRegisteredProduct(void);/* <= TRUE or FALSE */
```

Description The mdlLicense_isRegisteredProduct function reports if the base product has completed registration and obtained a license number.



This function was implemented in MicroStation 95.

Returns mdlLicense_isRegisteredProduct returns TRUE if the registration dialog has been completed by entering a valid license number, otherwise it returns FALSE.

mdlLicense_isSerializedProduct

```
#include <mslicens.h>
#include <mslicens.fdf>

int mdlLicense_isSerializedProduct(void);/* <= TRUE or FALSE */
```

Description The `mdlLicense_isSerializedProduct` function reports if the base product has obtained a serial number.



This function was implemented in MicroStation 95.

Returns `mdlLicense_isSerializedProduct` returns `TRUE` if the base product has been serialized, otherwise it returns `FALSE`.

Dynamic Array Functions

The Dynamic Array Object Data Structure (DArray) functions are used to create and manipulate variable-sized arrays of user data. Memory management is handled transparently by the DArray functions, allowing simpler application code. The following table lists the `mdlDArray_...` functions:

Function	Used to
<code>mdlDArray_clear</code>	remove all members from a dynamic array.
<code>mdlDArray_copy</code>	make a copy of a dynamic array.
<code>mdlDArray_create</code>	create a dynamic array.
<code>mdlDArray_destroy</code>	destroy a dynamic array.
<code>mdlDArray_findMember</code>	find the location of a member of a dynamic array.
<code>mdlDArray_getIndex</code>	get the index of a member given a pointer to that member.
<code>mdlDArray_getMemberP</code>	get a pointer to a member given its index.
<code>mdlDArray_insertMembers</code>	insert new members into a dynamic array.
<code>mdlDArray_moveMembers</code>	move members within a dynamic array.
<code>mdlDArray_nMembers</code>	count members of a dynamic array.
<code>mdlDArray_processAll</code>	call user function with members of a dynamic array.
<code>mdlDArray_removeMembers</code>	remove members from a dynamic array.
<code>mdlDArray_setMember</code>	store data in a member of a dynamic array.

mdlDArray_clear

```
boolean mdlDArray_clear    /* <= TRUE if error */
(
  ArrayObjectHdr   *arrayP      /* => dynamic array to remove mbrs */
);
```

Description mdlDArray_clear removes all members from a dynamic array.

arrayP points to the dynamic array to clear.

Returns mdlDArray_clear returns FALSE if successful. TRUE is returned if *arrayP* does not point to a dynamic array header.

See Also mdlDArray_removeMembers, mdlDArray_destroy.

mdlDArray_copy

```
ArrayObjectHdr *mdlDArray_copy /* <= copy of array or NULL */
(
  ArrayObjectHdr   *arrayP      /* => dynamic array to copy */
);
```

Description mdlDArray_copy makes a copy of a dynamic array, including all members.

arrayP points to the header of the dynamic array to copy.

Returns mdlDArray_copy returns a pointer to the header of the new dynamic array or NULL if out of memory.

mdlDArray_create

```
ArrayObjectHdr *mdlDArray_create /* <= NULL means error */
(
  int      hdrSize,      /* => bytes (including ArrayObjectHdr) */
  long     memberSize,   /* => size of an array member */
  int      nMembers,     /* => # of initial members */
  long     type,         /* => type of list */
  long     id            /* => id of list */
);
```

Description mdlDArray_create creates the header and any initial members of a new dynamic array data structure.

hdrSize indicates the number of bytes to be allocated for the dynamic array header. If *hdrSize* is less than the size of *ArrayObjectHdr*, it is set equal to that size.

memberSize indicates the size of each member in bytes.

nMembers specifies the number of initial members to create.

type may be used to assign a type to the array, analogous to resource types.

id may be used to assign an ID to the array, analogous to resource IDs.

Returns `mdlArray_create` returns a pointer to a new dynamic array header or `NULL` if there is an error.

See Also `mdlArray_destroy`.

mdlArray_destroy

```
boolean mdlArray_destroy    /* <= TRUE means error */
(
  ArrayObjectHdr    *arrayP    /* => node to free & remove */
);
```

Description `mdlArray_destroy` destroys the specified dynamic array and all members, freeing all memory allocated to the structure.

arrayP points to the dynamic array object to destroy.

Returns `mdlArray_destroy` returns `FALSE` if it is successful or `TRUE` on error.

See Also `mdlArray_create`.

mdlArray_findMember

```
int mdlArray_findMember    /* <= member index or -1 */
(
  ArrayObjectHdr    *arrayP,          /* => array to find member in */
  void              *testMemberP    /* => contents of member to find */
);
```

Description `mdlArray_findMember` finds the index of a member of a dynamic array given the contents of that member.

arrayP points to the header of the dynamic array in which to search.

testMemberP points to the member data to match.

Returns `mdlArray_findMember` returns the found member index or -1.

mdlArray_getIndex

```
int mdlArray_getIndex /* <= -1 if error */
(
  ArrayObjectHdr    *arrayP, /* => array to find member index of */
  char              *memberP /* => member whose index to get */
);
```

Description `mdlArray_getIndex` returns the index of a member of a dynamic array given a pointer to that member.

arrayP points to a dynamic array header.

memberP points to a member of the dynamic array.

Returns mdlDArray_getIndex returns the index of the member within the array, or -1 on error.

See Also mdlDArray_getMemberP.

mdlDArray_getMemberP

```
void *mdlDArray_getMemberP
(
  ArrayObjectHdr *arrayP,      /* => array to get member from */
  int memberIndex /* => index of array member (0 based) */
);
```

Description mdlDArray_getMemberP gives a pointer to a member of a dynamic array given the member's index.

arrayP points to the dynamic array containing the member to retrieve.

memberIndex indicates which member of *arrayP* to get.

Returns mdlDArray_getMemberP returns a pointer to the requested member of the dynamic array, or -1 on error.

See Also mdlDArray_getIndex.

mdlDArray_insertMembers

```
void *mdlDArray_insertMembers/* <= ptr to 1st inserted member */
(
  ArrayObjectHdr *arrayP,      /* => dynamic array to insert into */
  void *membersP, /* => mbrs to insert (copy made), NULL ok */
  int nToInsert, /* => number of members */
  int memberIndex /* => index to insert before */
);
```

Description mdlDArray_insertMembers inserts new members into a dynamic array.

arrayP points to the header of the dynamic array into which members are to be inserted.

membersP points to a block of data to copy into the newly created members of *arrayP*. If NULL, the new members of the array are initialized to 0.

nToInsert indicates the number of members pointed to by *membersP*.

memberIndex indicates at which index to begin inserting members. If *memberIndex* is set to -1, the new members are appended to the end of the dynamic array.

Returns mdlDArray_insertMembers returns a pointer to the first inserted member of the dynamic array, or NULL on error.

See Also mdlDArray_setMember.

mdlDArray_moveMembers

```
boolean mdlDArray_moveMembers /* <= TRUE if error */
(
  ArrayObjectHdr *arrayP,      /* => array to process */
  int            toIndex,      /* => move to here */
  int            fromIndex,    /* => from here */
  int            numToMove     /* => # of elements to move; -1=end */
);
```

Description mdlDArray_moveMembers moves members within a dynamic array.

arrayP points to the header of the dynamic array whose members are to be moved.

toIndex is the index of the destination for the moved members.

fromIndex is the index of the first of the members to move.

numToMove is the number of members to move.

Returns mdlDArray_moveMembers returns SUCCESS or TRUE if there is an error.

mdlDArray_nMembers

```
int mdlDArray_nMembers /* <= # of members in array */
(
  ArrayObjectHdr *arrayP /* => array to determine # of members */
);
```

Description mdlDArray_nMembers returns the number of members of a dynamic array.

arrayP points to the header of the dynamic array in question.

Returns mdlDArray_nMembers returns the number of members of the dynamic array. A return value of 0 can either mean there are no members in the array or that *arrayP* was NULL.

mdlDArray_processAll

```
void mdlDArray_processAll
(
  ArrayObjectHdr *arrayP,      /* => array to process */
  void *argP,                /* => arg to call processFunc with */
  void *processMD,           /* => NULL means current process */
  MdlFunctionP processFunc    /* => TRUE aborts processing of later mbrs */
);
```

Description mdlDArray_processAll sequentially passes a pointer to each member of a dynamic array to a user-defined function in order of index.

arrayP points to the header of the dynamic array to process.

argP points to a second parameter for the user function, if needed.

processMD is the MDL descriptor of the application containing the user function, or NULL if it is the current application.

processFunc is a pointer to the user function. Returning TRUE, aborts processing of later members.

Returns mdlDArray_processAll is of type void. It returns no value.

mdlDArray_removeMembers

```
boolean mdlDArray_removeMembers /* <= TRUE means error */
(
  ArrayObjectHdr *arrayP,      /* => dynamic array */
  int             nToRemove,    /* => # of members to remove */
  int             memberIndex   /* => start removing from here */
);
```

Description mdlDArray_removeMembers deletes members from a dynamic array and frees the memory they occupied.

arrayP points to the header of the dynamic array from which members are to be removed.

nToRemove indicates how many members to remove.

memberIndex indicates at which member to begin deletion.

Returns mdlDArray_removeMembers returns FALSE on success or TRUE if there is an error.

See Also mdlDArray_destroy, mdlDArray_clear.

mdlDArray_setMember

```
boolean mdlDArray_setMember
(
  ArrayObjectHdr *arrayP,      /* => array to set member of */
  int             memberIndex, /* => index of member to set */
  void           *dataP        /* => data to set member to */
);
```

Description mdlDArray_setMember copies data into a member of a dynamic array.

arrayP points to the header of the dynamic array into which data is to be copied.

memberIndex. indicates which member of *arrayP* is to receive the data.

dataP points to the data to copy into the specified array member.

Returns mdlDArray_setMember returns FALSE on success or TRUE if there is an error.

See Also mdlDArray_insertMembers.

Help Functions

The help system for MicroStation 5 uses a binary file format called I/Help as its main source of help information. Resource-based help files from earlier versions of MicroStation are still supported, but not encouraged, as resource files are unable to take advantage of some of the new features of the help system. See the “Documentation” chapter of the MDL Programmer’s Guide for information on creating I/Help files.

The following table describes help functions:

Function	Used to
<code>mdlHelp_getIHelpTopic</code>	open the help window with a specified file and article.
<code>mdlHelp_setShowMeFunction</code>	designate a function to be called when the “Show Me” button is pressed.

The following table describes the user functions associated with help functions:

User function	MicroStation calls when
<code>userHelp_showMeFunction</code>	the user presses the “Show Me” button or moves to a new help file or article.

mdlHelp_getIHelpTopic

```
void mdlHelp_getIHelpTopic
(
char    *taskIdP,      /* => Task ID used to find help file, or NULL */
char    *filenameP,    /* => Full pathname of help file, or NULL */
char    *logicalP      /* => Logical name of topic to display */
);
```

Description `mdlHelp_getIHelpTopic` opens the MicroStation help window and displays the topic whose logical name matches *logicalP*. Either *taskIdP* or *filenameP* must be specified; the other may be NULL. If *taskIdP* is specified, the help system will search the paths pointed to by `MS_HELPPATH` for an I/Help or resource help file whose base filename matches *taskIdP*. If *filenameP* is specified, that filename will be used. *filenameP* should specify a complete path.

Consider placing a help command in the MicroStation command queue instead of using `mdlHelp_getIHelpTopic`. This improved method allows MicroStation to use the native help system on some platforms, (i.e., PCs running Windows 3.1/Workgroups/NT or DEC Alpha running Windows ANT), while `mdlHelp_getIHelpTopic` always associates MicroStation’s built-in help system.

Returns The mdlHelp_getIHelpTopic function is of type void; it returns no value.

mdlHelp_setShowMeFunction

```
void mdlHelp_setShowMeFunction
(
    char          *labelP,          /* => label of Show Me button or NULL */
    MdlFunctionP functionP         /* => address of user function */
);
```

Description mdlHelp_setShowMeFunction designates an mdl function to be called when the user presses the “Show Me” button in the help window. The “Show Me” button is only displayed when a function is assigned to it. The button label will read “Show Me” unless another label is specified by *labelP*. Note that mdlHelp_setShowMeFunction does not work when using Windows help viewer.

functionP must be NULL or a valid MDL function pointer.

Returns The mdlHelp_setShowMeFunction function is of type void; it returns no value.

See Also userHelp_showMeFunction.

userHelp_showMeFunction

```
int userHelp_showMeFunction
(
    int      buttonPressed, /* => TRUE if Show Me button pressed */
    DialogBox *dbP,         /* => points to the help window */
    char     *fileNameP,    /* => current filename in help window */
    char     *topicNameP    /* => current topic in help window */
);
```

Description An MDL program designates a function associated with the “Show Me” button by calling mdlHelp_setShowMeFunction. The application programmer determines the function name; *userHelp_showMeFunction* is used merely as an example. Note *userHelp_showMeFunction* does not work when using Windows help viewer.

The MicroStation help system calls the function associated with the Show Me button whenever the user presses the “Show Me” button or moves to a new help file or article. *buttonPressed* is TRUE if the function is called as a result of the Show Me button being pressed, or FALSE if the function is called as a result of changing articles.

dbP points to the help window.

fileNameP is a pointer to a string containing the name of the current help file, omitting path and extension. It usually matches the task ID of a loaded application. The help system will search for an I/Help file or resource in the directories specified by the MS_HELPPATH configuration variable.

topicNameP is a pointer to a string containing the logical name of the current help article. If resource help is being used, the string consists of the letters “Cmnd” for Command help or “Topc” for Topic help, followed by an 8-character hexadecimal number representing the current article.

Returns The *userHelp_showMeFunction* function should return **TRUE** if the Show Me button is to be displayed enabled or **FALSE** if it is to be grayed out.

See Also *mdlHelp_setShowMeFunction*.

Function Key Functions

Functions keys provide a convenient way to enter frequently used key-ins with a single keystroke. Function key menus are simple ASCII files that save a complete set of function key definitions.

The following table lists functions for manipulating function key definitions and menus:

Function	Used to
<i>mdlFuncKey_attachMenu</i>	attach a function key menu.
<i>mdlFuncKey_saveMenu</i>	save a function key menu.
<i>mdlFuncKey_numKeys</i>	get the number of function keys available.
<i>mdlFuncKey_currentMenu</i>	get the full file specification of the currently attached function key menu.
<i>mdlFuncKey_getKeyByIndex</i> <i>mdlFuncKey_getKeyByVirtualKey</i>	get the button name & definition, and virtual key code of a function key definition.
<i>mdlFuncKey_getKeyName</i>	get the button name of a function key.
<i>mdlFuncKey_define</i>	add a function key definition.
<i>mdlFuncKey_remove</i>	remove a function key definition.



The functions *mdlFuncKey_getKey* and *mdlFuncKey_setKey* have been removed from MDL. Their functionality is represented in the newer function key functions documented in this section.

mdlFuncKey_attachMenu, mdlFuncKey_saveMenu

```
#include <mdl.h>

void mdlFuncKey_attachMenu
(
    char    *completeFilename, /* <= full menu filename */
```



```
char    *filename          /* => file name to attach */
);

void mdlFuncKey_saveMenu
(
char    *completeFilename, /* <= full menu file name */
char    *filename          /* => file to save */
);
```

Description The mdlFuncKey_attachMenu function attaches function key menus, and the mdlFuncKey_saveMenu function saves function key menus. The menu name is passed in *filename*. If this file specification is not complete, the environment variables MS_FKEYMNU and MS_DATA complete the specification. The complete file specification is returned in *completeFilename*. If a *filename* contains a full file specification, *filename* and *completeFilename* will be identical.

Returns The mdlFuncKey_attachMenu and mdlFuncKey_saveMenu functions return MDLERR_CANTOPENFILE if the menu file cannot be opened and SUCCESS if the operation is successful.

mdlFuncKey_numKeys

```
#include <mdl.h>

int mdlFuncKey_numKeys(void);
```

Description The mdlFuncKey_numKeys function returns the number of function keys available. The number of available function keys is different for each hardware platform supported by MicroStation.

Returns The mdlFuncKey_numKeys function returns the number of available function keys.

mdlFuncKey_currentMenu

```
void mdlFuncKey_currentMenu
(
char    *filename          /* <= full menu name */
);
```

Description The mdlFuncKey_currentMenu function returns, in *filename*, the full file specification of the currently attached function key menu.

See Also mdlFuncKey_attachMenu, mdlFuncKey_saveMenu.

mdlFuncKey_getKeyByIndex

```
int mdlFuncKey_getKeyByIndex
(
char    *buttonName,      /* <= button name */
char    **string,         /* <= button definition */
int     *virtualKey,      /* <= virtual key */
```

```
int      index      /* => index */
);
```

Description mdlFuncKey_getKeyByIndex returns the button name, button definition and virtual key code for a function key definition. *index* specifies the position of the definition in the internal list of function key definitions. The values are returned in *buttonName*, *string* and *virtualKey* respectively.

Returns mdlFuncKey_getKeyByIndex returns SUCCESS if a function key definition was found at *index* position in the list, otherwise ERROR is returned.

See Also mdlFuncKey_getKeyByVirtualKey.

mdlFuncKey_getKeyByVirtualKey

```
int mdlFuncKey_getKeyByVirtualKey
(
char   *buttonName, /* <= button name */
char   **string,    /* <= button definition */
int    *index,      /* <= index */
int    virtualKey   /* => virtual key */
);
```

Description mdlFuncKey_getKeyByVirtualKey returns the button name, button definition and index of a function key. *virtualKey* specifies the function key's virtual key code. The button name and definition are returned in *buttonName* and *string*. The index in the linked list of function key definitions is returned in *index*.

Returns mdlFuncKey_getKeyByVirtualKey returns SUCCESS if a function key definition exists for *virtualKey*, otherwise ERROR is returned.

See Also mdlFuncKey_getKeyByIndex.

mdlFuncKey_getKeyName

```
void mdlFuncKey_getKeyName
(
char   *funcKeyName, /* <= button name */
int    virtualKey    /* => virtual key */
);
```

Description The mdlFuncKey_getKeyName function returns, in *funcKeyName*, the button name of the function key with the virtual key code *virtualKey*.

See Also mdlFuncKey_define.

mdlFuncKey_define

```
void mdlFuncKey_define
(
int    virtualKey, /* => virtual key */
```

```
char    *definition    /* => button definition */  
);
```

Description The mdlFuncKey_define function adds a definition to the internal list of function key definitions. If a definition exists for the virtual key *virtualKey*, the definition passed by *definition* overwrites it.

See Also mdlFuncKey_remove.

mdlFuncKey_remove

```
int mdlFuncKey_remove  
(  
    int    virtualKey    /* => virtual key */  
);
```

Description The mdlFuncKey_remove function removes the definition for virtual key *virtualKey* from the internal list of function key definitions.

Returns mdlFuncKey_remove returns SUCCESS if the virtual key code *virtualKey* was removed, otherwise ERROR is returned.

See Also mdlFuncKey_define.

Undo Functions

MicroStation maintains an undo buffer that saves all changes to elements in the design file. A user who makes a mistake during an editing session and wishes to restore the file to it's condition before the session can execute the UNDO command. When the UNDO command is used, MicroStation copies the original elements from the undo buffer and writes them back to the design file. Undo functions let the user access the undo buffer.

The following table lists undo functions:

Function	Used to
mdlUndo_isActive mdlUndo_redoActive	return indication of an entry in the undo or redo buffer.
mdlUndo_mark	mark the current position in the undo buffer.
mdlUndo_reset	clear all data in the undo buffer.

Function	Used to
mdlUndo_startGroup mdlUndo_endGroup	turn on command grouping so commands are undone as a single unit.
mdlUndo_setFunction	designate an MDL function to be called when a significant event happens in the undo buffer.

The following table lists undo user functions. MicroStation calls these user-supplied functions when certain events occur within MicroStation. The programmer determines the user function name. (The name given below is for illustration purposes.) This function is designated to MDL through function pointer arguments to MDL routines.

Function	MicroStation calls when
userUndo_addToBuffer	an element is being added to the undo buffer.
userUndo_command	the user selects the UNDO command.

mdlUndo_isActive, mdlUndo_redoActive

```
#include <mdl.h>

boolean mdlUndo_isActive
(
  char    *commandString      /* <= command to be undone */
);

boolean mdlUndo_redoActive
(
  char    *commandString      /* <= command to be redone */
);
```

Description mdlUndo_isActive returns an indication of whether an element is in the undo buffer (whether it is possible to execute an UNDO command).
mdlUndo_redoActive returns an indication of whether there is an element in the redo buffer (whether it is possible to execute a REDO command).

Both functions return a character string in *commandString* that describes the command to be undone or redone.

Returns The mdlUndo_isActive and mdlUndo_redoActive functions return TRUE if an entry exists in their respective buffers and FALSE otherwise.

mdlUndo_mark

```
void mdlUndo_mark(void);
```

Description The `mdlUndo_mark` function sets a mark in the undo buffer.

The UNDO command undoes all changes to the design file since the last mark in the undo buffer. MicroStation ordinarily sets this mark every time the user enters a data point or starts a new primitive command. MDL applications that create or manipulate elements can break the operations into multiple undoable portions by calling `mdlUndo_mark`.



With versions 4.0.0 and 4.0.1 of MicroStation, `mdlUndo_mark` required that an `int` parameter be passed to it (though it ignored this parameter.) MicroStation versions 4.0.2 and newer no longer require this parameter.

Returns The `mdlUndo_mark` function is of type `void`. It returns no value.

See Also `mdlUndo_startGroup`, `mdlUndo_endGroup`.

mdlUndo_reset

```
void mdlUndo_reset(void);
```

Description `mdlUndo_reset` clears all information in the undo buffer. This function should be used with discretion, since users generally need the ability to undo their changes to files. However, it is necessary when applications make changes that invalidate the undo buffer.

Returns The `mdlUndo_reset` function is of type `void`. It returns no value.

mdlUndo_endGroup, mdlUndo_startGroup

```
void mdlUndo_startGroup(void);
```

```
void mdlUndo_endGroup(void);
```

Description The `mdlUndo_startGroup` and `mdlUndo_endGroup` functions let applications group changes to the design file so the groups are undone as a unit.

The UNDO command undoes all changes to the design file since the last mark in the undo buffer. MicroStation ordinarily sets this mark every time the user enters a data point or starts a new primitive command. Sometimes applications group several changes to the design file. These changes require multiple data points, so that the groups are always undone as a single unit.



Calls to `mdlUndo_startGroup` can be nested, but should always be matched with a corresponding call to `mdlUndo_endGroup`.

Returns The `mdlUndo_startGroup` and `mdlUndo_endGroup` functions are of type `void`. They return no values.

See Also `mdlUndo_mark`.

mdlUndo_setFunction

```
#include <userfnc.h>

MdlFunctionP mdlUndo_setFunction
(
    int          type,          /* => one of the UNDOFUNC_ constants */
    MdlFunctionP function      /* => address of MDL function */
);
```

Description mdlUndo_setFunction sets a function to process events related to the MicroStation undo buffer. These events include adding an element to the undo buffer and activating the UNDO command.

type can be UNDOFUNC_ADD_ENTRIES or UNDOFUNC_UNDO_COMMAND.

function must be a valid pointer to an MDL function or NULL. If this argument is NULL, no more events of type *type* are sent to the MDL application.

Returns mdlUndo_setFunction returns a pointer to the user function (of the same type) that was previously set using mdlUndo_setFunction. If *type* is invalid, mdlUndo_setFunction returns -1.

See Also userUndo_addToBuffer, userUndo_command.

userUndo_addToBuffer

```
#include <userfnc.h>

void userUndo_addToBuffer
(
    UndoInfo      *undoInfo,    /* => info about action */
    MSElementUnion *element    /* => original elem before change */
);
```

Description When MicroStation is ready to change an element or add a new element to the design file, it first copies the original element into the undo buffer. The UNDOFUNC_ADD_ENTRIES function that an MDL application designates in mdlUndo_setFunction is called as these elements are added to the undo buffer. When MDL calls the user function, the arguments are set as follows.

undoInfo points to a structure that contains information about the change being made to *element*. *undoInfo* contains the following fields:

Field in <i>undoInfo</i>	Meaning
<i>filePos</i>	file position of the element being changed.
<i>processNumber</i>	number of the current process. The process number uniquely identifies every process (such as an MDL application) in MicroStation.
<i>funcName</i>	command number of the active primitive.

Field in <i>undoInfo</i>	Meaning
<i>idNumber</i>	entry number in the undo buffer. All entries in a group have the same number.
<i>undoAction</i>	Type of action being performed on <i>element</i> . Possible values are UNDO_DELETE, UNDO_ADD, UNDO_MODIFY, UNDO_MARK and UNDO_MODIFYFENCE.



The *userUndo_addToBuffer* function modifies elements passed to it, but it must not change element sizes.

Returns MicroStation ignores the return value of the *userUndo_addToBuffer* function.

See Also mdlUndo_setFunction, userUndo_command, userSystem_elmDscrToFile, userSystem_writeToFile, mdlSystem_getProcessNumberFromMdlDesc, mdlSystem_getMdlDescFromProcessNumber.

userUndo_command

```
#include <userfnc.h>

int userUndo_command
(
    UndoInfo      *undoInfo,      /* => info about action t */
    MSElementUnion *element      /* => original elem before change */
);
```

Description The UNDOFUNC_UNDO_COMMAND function that an MDL application designates in mdlUndo_setFunction is called when the user activates the UNDO command. When MDL calls the user function, the arguments are set as follows:

undoInfo points to a structure that contains information about the change that was made to the element *element*. See *userUndo_addToBuffer* for a description of the information in *undoInfo*.

Returns The main utility of the *userUndo_command* function is to prevent changes to the design file from being undone if undoing them would cause problems with external databases. If the *userUndo_command* function returns a status other than 0, MicroStation reports "Undo blocked by application" and does not perform the undo.

See Also mdlUndo_setFunction, mdlSystem_getProcessNumberFromMdlDesc, mdlSystem_getMdlDescFromProcessNumber, userUndo_addToBuffer.

Tutorial Functions

Tutorial functions load tutorial menus, display messages in tutorial fields and position the cursor in specific tutorial fields.

Dialog boxes are replacing tutorials in MicroStation. New applications should use dialog boxes instead of tutorials. The most severe restriction of tutorials is that only one tutorial can be active at a time. This not only severely restricts the application, but it limits the user. If applications use tutorials, the user must choose just one application to be active.

All input generated from a tutorial is placed in the input queue. An application can get the queue elements either by becoming the active application (using `mdlInput_startCommand`) or by monitoring the input queue (using `mdlInput_setMonitorFunction` and `userInput_monitor`).

In queue elements for tutorial key-ins, the value for the source member is `FROM_TUTORIAL`, the value for the `cmdtype` member is `TUTKEYIN`, and the field number is contained in the `uc_fno_value` member. In queue elements for tutorial menu fields, the `source` member is `TUTORIAL`.

When a tutorial is created for an MDL application to use, “fb” should be entered as the servicing software name. This entry tells MicroStation’s tutorial processing that MicroStation, not a user command, will service the tutorial. This method works because the MDL application appears to be an extension of MicroStation. The MDL application will not be notified if the tutorial is closed. To prevent users from removing the tutorial, create a command filter. (See `mdlInput_setFunction` and `userInput_commandFilter`.) The command filter should reject all tutorial commands (such as `AT=`).

The tutorial is not automatically removed when the MDL task is removed. To guarantee the tutorial’s removal when an MDL task terminates, use an unload system user function. (See `mdlSystem_setFunction` and `userSystem_unloadProgram`).

Function	Used to
<code>mdlTutorial_output</code>	display a message in a tutorial field.
<code>mdlTutorial_positionInputField</code>	position the cursor at a specific tutorial field.
<code>mdlTutorial_load</code>	load a tutorial menu.
<code>mdlTutorial_windowGet</code>	get a pointer to the tutorial window.

Example

See `tutorial.mc`.

mdlTutorial_output

```
void mdlTutorial_output
(
    int      fieldNumber,    /* => field number */
    char     *stringP,       /* => string to be displayed */
    int      fieldSize       /* => length of the field */
);
```

Description The mdlTutorial_output function displays the message designated by *stringP* in the tutorial field designated by *fieldNumber*. It can be used to send messages to either output fields or key entry fields.

The entire field is always cleared before the message is displayed. Therefore, if the message is shorter than the field size, the remainder of the field is blank. If the message is longer than the field size, the message is truncated.

Returns The mdlTutorial_output function is of type `void`. It returns no value.

See Also mdlTutorial_windowGet, mdlTutorial_positionInputField.

mdlTutorial_positionInputField

```
void mdlTutorial_positionInputField
(
    int      fieldNumber,    /* => field number */
    int      drawFlag        /* => TRUE means draw box */
);
```

Description The mdlTutorial_positionInputField function establishes *fieldNumber* as the input field. If *fieldNumber* is -1, mdlTutorial_positionInputField advances the tutorial to the next field. If *fieldNumber* is 0, the MicroStation Command Window key-in field becomes the input field.

If *drawFlag* is non-zero, mdlTutorial_positionInputField draws the box when it selects the input field.

mdlTutorial_positionInputField is often used only to select the first field for a tutorial because the user can move freely through the tutorial fields.

Returns The mdlTutorial_positionInputField function is of type `void`. It returns no value.

See Also mdlTutorial_windowGet, mdlTutorial_output.

mdlTutorial_load

```
int mdlTutorial_load
(
    char     *stringP        /* => name of the tutorial */
);
```

Description The `mdlTutorial_load` function loads the tutorial specified by *stringP*. If a tutorial is active when `mdlTutorial_load` is called, `mdlTutorial_load` unloads it before loading the requested tutorial. If *stringP* points to an empty string, the `mdlTutorial_load` function terminates only the active tutorial.

Returns `mdlTutorial_load` returns `SUCCESS` if no errors occur. Otherwise, it returns a non-zero value and displays an error message in the MicroStation Command Window.

mdlTutorial_windowGet

```
MSWindow *mdlTutorial_windowGet(void);
```

Description The `mdlTutorial_windowGet` function returns a pointer to the tutorial window.

Returns `mdlTutorial_windowGet` returns a pointer to the tutorial window if a tutorial is active, otherwise it returns `NULL`.

See Also `mdlTutorial_output`, `mdlTutorial_positionInputField`, `mdlTutorial_load`.

Plotting Functions

Plotting functions are used to generate and manipulate plot files and plotter configuration files.

Function	Used to
<code>mdlPlot_getInfo</code>	get info from a plotter configuration file.
<code>mdlPlot_execute</code>	create a plotfile.
<code>mdlPlot_writeCommand</code>	write binary data to a plotfile.
<code>mdlPlot_writeString</code>	write ASCII data to a plotfile.
<code>mdlPlot_flushBuffer</code>	flush plotfile buffer to disk.

mdlPlot_getInfo

```
#include <pltstrct.h>

int mdlPlot_getInfo
(
    PlotStatic *plotStaticP, /* => must be allocated by user */
    char      configNameP   /* => must be allocated by user */
);
```

Description `mdlPlot_getInfo` reads a plotter configuration file and fills in the information in the structure pointed to by *plotStaticP* using the information contained therein. Memory for the structure must be allocated (using `sizeof(PlotStatic)`) by the MDL application prior to calling `mdlPlot_getInfo`. This memory must be freed by the

application after the plot is complete. *plotStaticP* usually points to the built-in variable *plotStat*.

configNameP is a non-NULL character array of size MAXFILELENGTH. If *configNameP* points to a NULL string, mdlPlot_getInfo uses the plotter configuration file defined by the environment variable MS_PLTR. The full file specification of the opened plotter configuration file is returned in *configNameP*.

Returns mdlPlot_getInfo returns SUCCESS if the plotter configuration file is opened and the *plotStaticP* structure is valid. It returns ERROR if it cannot find the plotter configuration file or if that file is invalid.

See Also mdlPlot_execute, autoplots.mc.

mdlPlot_execute

```
int mdlPlot_execute(void);
```

Description The mdlPlot_execute function causes MicroStation to create a plotfile. mdlPlot_execute should be called after the built-in variable plotStat has been initialized with a call to mdlPlot_getInfo.

Returns mdlPlot_execute returns SUCCESS after the plotfile has been successfully generated. It returns ERROR if an MDL plotter driver is configured but cannot be loaded.

See Also mdlPlot_getInfo.

mdlPlot_writeCommand, mdlPlot_writeString, mdlPlot_flushBuffer



(MDL plotter drivers only)

```
#include <pltstrct.h>

void mdlPlot_writeCommand
(
    char    *plotDataP,    /* => data to write to plotfile */
    int     numChars       /* => number of bytes in plotDataP */
);

void mdlPlot_writeString
(
    char    *plotDataP     /* => NULL-terminated plot data */
);

void mdlPlot_flushBuffer(void);
```

Description The functions `mdlPlot_writeCommand`, `mdlPlot_writeString` and `mdlPlot_flushBuffer` are used by MDL plotter drivers to write plotter commands to the current plotfile. They should only be called from an MDL plotter driver.

`mdlPlot_writeCommand` is used to write binary data to the plotfile.

plotDatP is a pointer to a buffer containing the information to be written to the plotfile. *numChars* is the number of bytes in *plotDataP*.

`mdlPlot_writeString` is used to write ASCII data to the plotfile. *plotDatP* is a pointer to a buffer containing a NULL terminated string to be written.

The plotfile is buffered internally to MicroStation. `mdlPlot_flushBuffer` causes MicroStation to flush this buffer to disk. This function is rarely necessary.

Returns The functions `mdlPlot_writeCommand`, `mdlPlot_writeString` and `mdlPlot_flushBuffer` are of type `void`. They do not return a status.

See Also `adiplot.mc`.

BASIC Interface Functions

The following table list the BASIC interface functions:

Function	Used to
<code>mdlBasic_getMacroInfo</code>	get information about a particular macro program.
<code>mdlBasic_getPublicVariable</code>	get the value of an existing BASIC Public variable.
<code>mdlBasic_setPublicVariable</code>	set the value of an existing BASIC Public variable.

`mdlBasic_getMacroInfo`

```
#include <msbasic.fdf>
#include <msdefs.h>
#include <mdlerrs.h>

int mdlBasic_getMacroInfo
(
    char    *macroTaskIdP,      /* <= TaskId if desired */
    char    *fullBasicFileNameP, /* <= macro file name if desired */
    int     *isLoadingP,        /* <= is it loaded? */
    char    *macroNameP         /* => Name of macro to get info on. */
);
```

Description The mdlBasic_getMacroInfo function is used to obtain information about a particular macro. This is a completely passive operation; it does not cause or affect the execution of a macro.

macroTaskIdP points to a character array to receive the task ID of the macro as it would appear when the macro is executing. This array must be at least TASK_ID_SIZE characters in size. This argument may be NULL if the task ID is not required.

fullBasicFileNameP points to a character array to receive the full file name of the macro as resolved by MicroStation's BASIC subsystem. This array must be at least MAXFILELENGTH characters in size. This argument may be NULL if the file name is not required.

isLoadingP is used to determine if the macro is currently executing. Either TRUE (meaning "the macro is running") or FALSE is returned in the integer pointed to by *isLoadingP*.

macroNameP is the name of the macro to be queried. It can be a simple macro name, or it can be a fully qualified file path (.ba or .bas extension).



This function was implemented in MicroStation 95.

Returns mdlBasic_getMacroInfo returns SUCCESS or MDLERR_CANNOTFINDMACRO if a macro file corresponding to *macroNameP* could not be found.

See Also mdlBasic_getPublicVariable, mdlBasic_setPublicVariable.

mdlBasic_getPublicVariable

```
#include <msbasic.fdf>
#include <mdlerrs.h>

int mdlBasic_getPublicVariable
(
    void    *dataP,          /* <= copy data to here*/
    int     dataSize,        /* => size of buffer pointed to by dataP */
    char    *variableName    /* => the name of the BASIC variable to get */
);
```

Description The mdlBasic_getPublicVariable function is used to copy the value of a BASIC public variable to an MDL variable. The public variable must have been previously defined by a BASIC macro, but the macro that defined it would not necessarily have to be loaded at the time this function call is made.

dataP is a pointer to an MDL variable to receive the value of the specified BASIC variable. It must be a pointer to one of these C data types: short,

`long(int)`, `double` or an array of characters. The following table shows the correlation of BASIC variables to MDL (C) variables:

BASIC	MDL (C)
Integer(%)	short
Long(&)	long int
Single(!)	promoted to double
Double(#)	double
String(\$)	array of char

dataSize is the size of the MDL variable to receive the value.

variableName is a BASIC expression that resolves down to one of these BASIC data types: `Integer(%)`, `Long(&)`, `Single(!)`, `Double(#)` or `String($)`. BASIC User Defined Types (UDTs), objects and arrays cannot be specified. However, a member of a UDT can be obtained if it is one of the valid data types listed above. For example, if a BASIC program defined a point as:

```
Public myPoint as MbePoint
```

The x, y and z components of the point could be obtained in an MDL program as follows:

```
double x,y,z;
mdlBasic_getPublicVariable(&x, sizeof(double), "myPoint.x");
mdlBasic_getPublicVariable(&y, sizeof(double), "myPoint.y");
mdlBasic_getPublicVariable(&z, sizeof(double), "myPoint.z");
```



This function was implemented in MicroStation 95.

Returns `mdlBasic_getPublicVariable` returns `SUCCESS` or one of the following errors defined in `<mdlerrs.h>`:

Error	Description
<code>MDLERR_MACROVARNOTDEFINED</code>	The BASIC variable is not defined or is not a public variable.
<code>MDLERR_VARWRONGSIZE</code>	There is a type mismatch between the BASIC and MDL variables.
<code>MDLERR_NOTATOMICDATATYPE</code>	The BASIC variable name provided specifies a User Defined Type, an object or an array instead of one of the simple data types (<code>%</code> , <code>&</code> , <code>!</code> , <code>#</code> or <code>\$</code>).

See Also `mdlBasic_setPublicVariable`, `mdlBasic_getMacroInfo`.

mdlBasic_setPublicVariable

```
#include <msbasic.fdf>
#include <mdlerrs.h>

int mdlBasic_setPublicVariable
(
void    *dataP,          /* => copy data from here */
int     dataSize,        /* => size of buffer pointed to by dataP */
char    *variableName    /* => the name of the BASIC variable to set */
);
```

Description The `mdlBasic_setPublicVariable` function is used to set the value of a BASIC public variable from an MDL variable. The public variable must have been previously defined by a BASIC macro, but the macro that defined it would not necessarily have to be loaded at the time this function call is made.

dataP is a pointer to an MDL variable to be used to set the value of the specified BASIC variable. It must be a pointer to one of these C data types: `short`, `long (int)`, `double` or an array of characters. The following table shows the correlation of BASIC variables to MDL (C) variables:

BASIC	MDL (C)
Integer(%)	short
Long(&)	long int
Single(!)	promoted to double
Double(#)	double
String(\$)	array of char

dataSize is the size of the MDL variable pointed to by *dataP*.

variableName is a BASIC expression that resolves down to one of these BASIC data types: `Integer(%)`, `Long(&)`, `Single(!)`, `Double(#)` or `String($)`. BASIC User Defined Types (UDT's), objects and arrays cannot be specified. However, a member of a UDT can be set if it is one of the simple data types listed above. For example, if a BASIC program defined a point as:

```
Public myPoint as MbePoint
```

The x, y and z components of the point could be set in an MDL program as follows:

```
double x,y,z;
mdlBasic_setPublicVariable(&x, sizeof(double), "myPoint.x");
mdlBasic_setPublicVariable(&y, sizeof(double), "myPoint.y");
mdlBasic_setPublicVariable(&z, sizeof(double), "myPoint.z");
```



This function was implemented in MicroStation 95.

Returns `mdlBasic_setPublicVariable` returns `SUCCESS` or one of the following errors defined in `<mdlerrs.h>`:

Error	Description
<code>MDLERR_MACROVARNOTDEFINED</code>	The BASIC variable is not defined or is not a public variable.
<code>MDLERR_VARWRONGSIZE</code>	There is a type mismatch between the BASIC and MDL variables.
<code>MDLERR_NOTATOMICDATATYPE</code>	The BASIC variable name provided specifies a User Defined Type, an object or an array instead of one of the simple data types (<code>%</code> , <code>&</code> , <code>!</code> , <code>#</code> or <code>\$</code>).

See Also `mdlBasic_getPublicVariable`, `mdlBasic_getMacroInfo`.

Wide Character String Functions

The following wide character string functions are extended versions of existing text placement functions. These functions accept strings in wide-character format and also new text attributes such as slant, inter character spacing, underline and vertical direction.

A wide character string is an array of `MSWideChar` defined in `widechar.h`. Each character in a wide character string is represented by two bytes (or four bytes on some platforms) to accommodate various languages in the world. An ASCII string needs to be converted to a wide character string by the `mbstowcs` function before calling text creation functions. For example, an ASCII string, "ASCII Text," is converted as:

```
#include <stdlib.h>
#include <widechar.h>
#define MAX_STRING_SIZE 128
char string[MAX_STRING_SIZE];
MSWideChar wString[MAX_STRING_SIZE];
strcpy(string, "ASCII Text");
mbstowcs(wString, string, MAX_STRING_SIZE);
mdlText_createWide(....., wString, .....);
```

Similarly, a wide character string returned by text extraction functions can be converted back to ASCII string by the `wcstombs` function.


```
mdlText_extractWide(wString, .....);
wcstombs(string, wString, MAX_STRING_SIZE);
```

The third argument of `wcstombs` and `mbstowcs` means the length of the output string in number of (wide) characters. For more information regarding the processing of wide character strings, please refer to: *Internationalization, MDL Supplement Guide*.

The following table lists wide character functions:

Function	Used to
<code>mdlText_createWide</code>	create text element with wide-character string and a <code>TextParamWide</code> structure.
<code>mdlText_extractWide</code>	extract information including wide-character string from a text element using a <code>TextParamWide</code> structure.
<code>mdlText_extractStringWide</code>	returns a wide-character string contained in a text element.
<code>mdlTextNode_createWide</code>	create an empty text node element with a <code>TextParamWide</code> structure.
<code>mdlTextNode_createWithStringsWide</code>	create a text node element with wide-character text strings and a <code>TextParamWide</code> structure.
<code>mdlTextNode_extractWide</code>	extract information with <code>TextParamWide</code> structures.
<code>mdlText_extractStringsFromDscrWide</code>	get wide-character strings from text node element descriptor.
<code>mdlText_addStringsToNodeDscrWide</code>	add wide-character strings to text node element descriptor.
<code>mdlText_expandStringWide</code>	expand special characters in wide-character text string.
<code>mdlText_compressStringWide</code>	compress special characters in wide-character text string.

Examples

See `create.mc` and `extract.mc`.

mdlText_createWide

```
#include <mdl.h>
#include <mselems.h>
#include <wchar.h>

int mdlText_createWide
```

```
(
MSElementUnion    *out,           /* <= text element created */
MSElementUnion    *in,           /* => template element */
MSWideChar         *wString,      /* => wide character string */
Dpoint3d           *userOrigin,   /* => origin (or NULL) */
RotMatrix          *rMatrix,      /* => rotation matrix (or NULL) */
TextSizeParam      *textSize,     /* => size (or NULL) */
TextParamWide      *txtParamWide, /* => parameters (or NULL) */
TextEDParam        *edParam       /* => enter data info (or NULL) */
);
```

Description The `mdlText_createWide` function is similar to `mdlText_create` except for *wString* and *txtParamWide*.

wString points to a wide character string.

txtParamWide points to a `TextParamWide` structure, defined in `mdl.h`. This structure defines the created text element's inter character spacing, direction, slant and underline spacing in addition to that defined by the `TextParam` structure. The values for slant, inter character spacing and underline spacing are valid only if corresponding flags in *txtParamWide*->*flags* are set.

Returns `mdlText_createWide` returns `SUCCESS` if a valid MicroStation text element is created. If *in* and *wString* are `NULL`, `mdlText_createWide` returns `MDLERR_INSFINFO`. It returns `MDLERR_BADELEMENT` if the text is beyond the design plane.

See Also `mdlText_create`, `mdlTextNode_createWide`, `mdlText_extractWide`, `mdlText_extractShape`.

mdlText_extractWide

```
#include <mdl.h>
#include <mselems.h>
#include <wchar.h>

int mdlText_extractWide
(
MSWideChar         *wString,      /* <= wide character string */
Dpoint3d           *origin,       /* <= origin (lower-left corner) */
Dpoint3d           *userOrigin,   /* <= snap point */
RotMatrix          *rMatrix,      /* <= rotation matrix */
TextSizeParam      *textSize,     /* <=> tile size or total size */
TextParamWide      *txtParamWide, /* <= font, just, slant, etc. */
TextEDParam        *edParam,      /* <= ed fields info */
MSElementUnion    *in           /* => text element */
);
```

Description The mdlText_extractWide function is similar to the mdlText_extract function except for *wString*, *txtParamWide*, *textSize* and *edParam*.

wString points to a wide character string.

txtParamWide points to a TextParamWide structure, defined in mdl.h. This structure defines the extracted text element's font number, justification, slant, intercharacter spacing, direction and underline parameters. The values for slant, intercharacter spacing and underline spacing are valid only if corresponding flags in *txtParamWide->flags* are set.

textSize points to a TextSizeParam structure, defined in mdl.h. The *textSize->mode* field determines the format of returned text size. If *textSize->mode* is TXT_BY_SIZE, *textSize->size* returns the overall text element size. The size of a single character cell is returned if the value is TXT_BY_TILE_SIZE.

edParam points to a TextEdParam structure which specifies the number and locations of enter-data fields within the text string. To obtain the location of each enter-data field, MDL applications should allocate memory before calling this function. Otherwise, *edParam->edFields* will point to NULL. For example:

```
#include <msdefs.h>
TextEDParam edParam;
TextEDField edFields[MAX_EDFIELDS];
...

edParam.edField=edFields;
textSizeParam.mode=TXT_BY_TILE_SIZE;
mdlText_extractWide(wString, &origin, &userOrigin, NULL,
                    &textSizeParam, &textParam, &edParam, &element);
```

Returns The mdlText_extractWide returns SUCCESS if *in* is a valid MicroStation text element of type TEXT_ELM. Otherwise, it returns MDLERR_BADELEMENT.

See Also mdlText_extract, mdlText_createWide, mdlText_extractStringWide, mdlText_extractShape.

mdlText_extractStringWide

```
#include <mselems.h>
#include <wchar.h>

int mdlText_extractStringWide
(
MSWideChar *wString,          /* <= wide character string */
MSElementUnion *el           /* => text element */
);
```

Description The `mdlText_extractStringWide` function is identical to `mdlText_extractString` except the string is returned as a wide-character string.

Returns The `mdlText_extractStringWide` returns `SUCCESS` if *in* is a valid MicroStation text element of type `TEXT_ELM`. Otherwise, it returns `MDLERR_BADELEMENT`.

See Also `mdlText_extractString`, `mdlText_createWide`, `mdlText_extractWide`, `mdlText_extractShape`.

mdlTextNode_createWide

```
#include <mdl.h>
#include <mselems.h>
#include <wchar.h>

int mdlTextNode_createWide
(
  MSElementUnion    *out,          /* <= text node created */
  MSElementUnion    *in,          /* => template element */
  Dpoint3d           *origin,      /* => origin (or NULL) */
  RotMatrix           *rMatrix,    /* => rotation matrix (or NULL) */
  TextSizeParam      *textSize,    /* => default text size (or NULL) */
  TextParamWide      *txtParamWide /* <=> text paramters (or NULL) */
);
```

Description The `mdlTextNode_createWide` function is similar to `mdlTextNode_create` except for the *textSize* and *txtParamWide* parameters.

textSize points to a `TextSizeParam` structure, defined in `mdl.h`, that determines the resulting text element's size. The text element's size depends on the value of *textSize->mode*. If *txtSize->mode* is `TXT_BY_SIZE`, *textSize->size* is the overall text element size. *textSize->size* is the size of a single character cell if the value is `TXT_BY_TILE_SIZE`.

txtParamWide points to a `TextParamWide` structure, defined in `mdl.h`. This structure defines the created text node element's line spacing, intercharacter spacing, direction, slant, and underline spacing in addition to that defined by the `TextParam` structure. The created text node element's node number is returned in *txtParamWide->nodeNumber*.

Returns `mdlTextNode_createWide` returns `SUCCESS` if a valid MicroStation element is created. It returns `MDLERR_BADELEMENT` if the resulting text node is beyond the design plane.

See Also `mdlTextNode_create`, `mdlText_createWide`, `mdlTextNode_extractWide`, `mdlTextNode_extractShape`.

mdlTextNode_createWithStringsWide

```
#include <mdl.h>
#include <mselems.h>
#include <wchar.h>

int mdlTextNode_createWithStringsWide
(
  MSElementDescr    **out,           /* <= text node created */
  MSElementUnion     *in,           /* => template element */
  MSWideChar          *wStrings[],   /* => array of wide string pnters */
  int                 totallines,    /* => number of lines in strings */
  Dpoint3d            *userOrigin,   /* => origin (or NULL) */
  RotMatrix            *rMatrix,      /* => rotation matrix (or NULL) */
  TextSizeParam        *sizeParam,    /* => default text size (or NULL) */
  TextParamWide        *txtParams,    /* => text parameters (or NULL) */
  TextEDParam          *textEdParam   /* => earray of ED field structs */
);
```

Description The mdlTextNode_createWithStringsWide function is similar to mdlTextNode_createWithStrings except for *wStrings*, *txtParams*, and *txtEdParam*.

wStrings is an array of wide-character string pointers.

sizeParam points to a TextSizeParam structure, defined in mdl.h, that determines the resulting text element's size. The text element's size depends on the value of the *sizeParam->mode*. If *sizeParam->mode* is TXT_BY_SIZE, *textSize->size* is the overall text element size. *sizeParam->size* is the size of a single character cell if the value is TXT_BY_TILE_SIZE.

txtParams points to a TextParamWide structure, defined in mdl.h. This structure defines the created text element's line spacing, intercharacter spacing, direction, slant and underline spacing in addition to what is defined by the TextParam structure. The created text node element's node number is returned in *txtParams->nodeNumber*.

Returns The mdlTextNode_createWithStringsWide function returns SUCCESS if a valid MicroStation element is created. It returns MDLERR_BADELEMENT if the resulting text node is beyond the design plane.

See Also mdlTextNode_createWithStrings, mdlText_createWide, mdlTextNode_extractWide.

mdlTextNode_extractWide

```
#include <mdl.h>
#include <mselems.h>
#include <wchar.h>

int mdlTextNode_extractWide
(
```

```

Dpoint3d      *origin,           /* <= origin (lower-left corner) */
Dpoint3d      *userOrigin,       /* <= snap point */
RotMatrix     *rotMatrix,        /* <= rotation matrix */
TextSizeParam *textSize,         /* <= tile size */
TextParamWide *txtParamWide,     /* <= font, just, etc. */
MSElement    *node              /* => text node element */
);

```

Description `mdlTextNode_extractWide` is similar to `mdlTextNode_extract` except for *origin*, *userOrigin*, *textSize* and *txtParamWide*.

The treatment of text elements and text node elements have been different in the `mdlText_extract` and `mdlTextNode_extract` functions.

The origin of a text node element returned by `mdlTextNode_extract` was traditionally the text node's snap point. The `mdlText_extract` function returned both lower-left corner and snap point. To be consistent with `mdlText_extractWide`, `mdlTextNode_extractWide` returns both lower-left corner (*origin*) and snap point (*userOrigin*). The position of *userOrigin* relative to *origin* depends on the text node element's justification.

textSize points to a `TextSizeParam` structure, defined in `mdl.h`. The *textSize->mode* field determines the format of returned text size. If *txtSize->mode* is `TXT_BY_SIZE`, *textSize->size* returns the overall text element size. The size of a single character cell is returned if the value is `TXT_BY_TILE_SIZE`.

txtParamWide points to a `TextParamWide` structure, defined in `mdl.h`. This structure defines the extracted text node element's node number, font number, justification, line spacing, intercharacter spacing, slant, direction and underline parameters. The values for slant, intercharacter spacing and underline spacing are valid only if corresponding flags in the *TextDrawFlags* structure within *txtParamWide* (otherwise known as *txtParamWide->flags*) are set.

Returns `mdlTextNode_extractWide` returns `SUCCESS` if *node* is a valid MicroStation element of type `TEXT_NODE_ELM`. Otherwise, it returns `MDLERR_BADELEMENT`.

See Also `mdlText_extract`, `mdlTextNode_createWithStringsWide`, `mdlText_createWide`, `mdlTextNode_extractWide`.

mdlText_extractStringsFromDscrWide

```

#include <mdl.h>
#include <wchar.h>

int mdlText_extractStringsFromDscrWide
(
MSWideChar    *wBuffer,         /* <= buffer to receive strings */
int            bufferSize,       /* => maximum characters in buffer */

```

```

MSElementDscr    *elementDscrP /* => text node */
);

```

Description The mdlText_extractStringsFromDscrWide function is similar to mdlText_extractStringsFromDscr.

The strings are returned as wide-character strings in the string buffer, *wBuffer*.

bufferSize contains the number of wide-characters which the string buffer, *wBuffer* can hold.

Returns mdlText_extractStringsFromDscrWide returns the total number of wide-characters placed in the user string buffer *wBuffer*.

See Also mdlText_extractStringsFromDscr, mdlText_addStringsToNodeDscrWide.

mdlText_addStringsToNodeDscrWide

```

#include <mdl.h>
#include <wchar.h>

int mdlText_addStringsToNodeDscrWide
(
MSElementDscr    *elementDscrP, /* <= text node */
MSWideChar        *wBuffer      /* => buffer with wide strings */
);

```

Description mdlText_addStringsToNodeDscrWide is identical to mdlText_addStringsToNodeDscr except that wide-character strings are passed in the string buffer *wBuffer*.

Returns The mdlText_addStringsToNodeDscrWide function returns either SUCCESS or ERROR if the element descriptor *elementDscrP* does not point to an element of type TEXT_NODE_ELM.

See Also mdlText_addStringsToNodeDscr, mdlText_extractStringsFromDscrWide.

mdlText_expandStringWide

```

#include <mdl.h>
#include <wchar.h>

int mdlText_expandStringWide
(
MSWideChar    *outBuffer, /* <= buffer for converted string */
MSWideChar    *inBuffer,  /* => text string to be converted */
int           bufferMax,   /* => size of outBuffer */
int           conversionType /* => type of conversion */
);

```

Description `mdlText_expandStringWide` is identical to the `mdlText_expandString` function except that *inBuffer* and *outBuffer* are wide-character strings.

Returns `mdlText_expandStringWide` returns the number of characters in the expanded string *outBuffer*.

See Also `mdlText_expandString`, `mdlText_compressString`, `mdlText_compressStringWide`.

mdlText_compressStringWide

```
#include <mdl.h>
#include <wchar.h>

int mdlText_compressStringWide
(
    MSWideChar *outBuffer,    /* <= buffer for converted string */
    MSWideChar *inBuffer,    /* => text string to be converted */
    int         bufferSize,   /* => size of outBuffer */
    int         conversionType /* => type of conversion */
);
```

Description `mdlText_compressStringWide` is identical to `mdlText_compressString` except that *inBuffer* and *outBuffer* are wide-character strings.

Returns The `mdlText_compressStringWide` function returns the number of characters in the compressed string *outBuffer*.

See Also `mdlText_compressString`, `mdlText_expandString`, `mdlText_expandStringWide`.

User Preference Functions

The following table lists user preference MDL functions:

Function	Used to
<code>mdlUserPrefs_save</code>	save user preferences to disk and update internal MicroStation variables to reflect the changes.
<code>mdlUserPrefs_saveFileName</code>	save file open dialog default field values for a specific file type.
<code>mdlUserPrefs_getFileName</code>	get the default field values for the file open dialog that will be used for a specific file type.
<code>mdlUserPrefs_deleteStartupInfo</code>	deletes a startup information resource from the user preferences file.

Function	Used to
mdlUserPrefs_loadStartupInfo	load an application's startup information resource.
mdlUserPrefs_saveStartupInfo	save a startup information resource in the user preferences file for later use by MicroStation.

mdlUserPrefs_save

```
#include <mdl.h>
#include <userpref.h>

void mdlUserPrefs_save
(
    UserPrefs    *userPrefs
);
```

Returns The mdlUserPrefs_save function saves the user's preferences to disk. The function also updates internal MicroStation variables to reflect the changes.



The mdlUserPrefs_get function should not be used. MicroStation loads all user preferences during startup, and they are available to programmers in the MicroStation built-in variable *userPrefsP*. MDL programmers can modify what *userPrefsP* points to and then save the information using mdlUserPrefs_save.

Returns The mdlUserPrefs_save function is of type void. It returns no value.

mdlUserPrefs_saveFileName

```
#include <deffiles.h>
#include <rscdefs.h>

void mdlUserPrefs_saveFileName
(
    DefaultFileInfo    *defFileInfoP, /* => save info for file type */
    RscFileHandle      userPrefFileH, /* => userpref file handle */
    ULONG              fileInfoId     /* => file info type to save */
);
```

Description mdlUserPrefs_saveFileName is used to save file search parameters for use in the MicroStation File Open and File Create dialogs. Filling in a default file name and/or file filter helps the user to find files using these dialogs. MicroStation automatically saves this information each time the user uses one of these two dialogs. This function is provided to allow an application to override the automatically saved

information. Each file type (design file, reference file, etc.) gets its own set of file search criteria.

defFileInfoP points to a structure filled in by the calling application with the search criteria. The structure, `DefaultFileInfo`, is defined in `deffiles.h`. The calling application must allocate a variable of this type, initialize it to zeroes, and then set the `fileName` and `fileFilter` members prior to calling this function.

userPrefFileH is a handle to the user preferences file where the file search information will be saved. Obtain this handle using the `mdlDialog_userPrefFileOpen` function.

fileInfoId indicates the type of file for which you are saving the information. All standard MicroStation file types are defined in the file `deffiles.h`. These identifiers will be negative numbers. An application can define its own file types simply by using positive numbers for the ids. Each resource with a positive file ID is qualified with the calling application's name so you don't need to worry about your positive file ID numbers conflicting with any other application's IDs.

Returns The `mdlUserPrefs_saveFileName` function is of type `void`.

See Also `mdlUserPrefs_getFileName`, `mdlDialog_userPrefFileOpen`.

mdlUserPrefs_getFileName

```
#include <deffiles.h>
#include <rsdefs.h>

void mdlUserPrefs_getFileName
(
    DefaultFileInfo  *defFileInfoP, /* <= info for file type */
    RscFileHandle    userPrefFileH, /* => userpref file handle */
    ULONG            fileInfoId     /* => file info type to read */
);
```

Description `mdlUserPrefs_getFileName` is used to obtain the file search parameters last used in the MicroStation File Open or File Create dialog. MicroStation automatically saves this information each time the user uses one of these two dialogs. Each file type (design file, reference file, etc.) has its own set of file search criteria.

defFileInfoP points to a structure to be filled in with the file location information last used by the File Open or File Create dialogs. The structure, `DefaultFileInfo`, is defined in `deffiles.h`. The `fileName` and `fileFilter` structure members are usually of most interest.

userPrefFileH is a handle to the user preferences file where the file search information will be stored. Obtain this handle using the `mdlDialog_userPrefFileOpen` function.

fileInfold indicates the type of file for which you wish to obtain information. See `mdlUserPrefs_saveFileName` for more information on this parameter.

Returns The `mdlUserPrefs_getFileName` function is of type `void`.

See Also `mdlUserPrefs_saveFileName`, `mdlDialog_userPrefFileOpen`.

mdlUserPrefs_deleteStartupInfo

```
int mdlUserPrefs_deleteStartupInfo
(
    ULong   rscType,          /* => Normally RTYPE_DgnAppStartup */
    char    *taskNameP       /* => NULL means task ID of current app */
);
```

Description The `mdlUserPrefs_deleteStartupInfo` function deletes the startup information resource defined by the input fields from the user preferences file.

rscType defines the type of startup information resource to be deleted from user preferences. `RTYPE_DgnAppStartup` is currently the only MicroStation recognized resource.

taskNameP defines the application to which the resource applies and is used as an alias for locating the specific resource information.

Returns `mdlUserPrefs_deleteStartupInfo` returns `SUCCESS` upon successful completion or a non-zero value indicating the reason for failure.

See Also `mdlUserPrefs_saveStartupInfo`, `mdlUserPrefs_loadStartupInfo`.

mdlUserPrefs_loadStartupInfo

```
int mdlUserPrefs_loadStartupInfo
(
    char    *fileNameP,       /* <= receives name of file */
    int     *argcP,           /* <= where to save num args */
    char    **argvP[],        /* <= where to save args and their ptrs */
    ULong   rscType,
    char    *taskNameP       /* => task ID/resource alias */
);
```

Description `mdlUserPrefs_loadStartupInfo` is used to load an application's startup resource information. This information can be used to load the application into memory.

fileNameP is the name of the application file. This name can be used to load the application into memory.

argcP and *argvP* are the command line parameters to be passed to the application at startup.

rscType defines the type of startup information resource to be loaded from user preferences. `RTYPE_DgnAppStartup` is currently the only MicroStation recognized resource.

taskNameP defines the application to which the resource applies and is used as an alias for locating the specific resource information.

Returns `mdlUserPrefs_loadStartupInfo` returns `SUCCESS` if the startup information could be loaded into memory or a non-zero value indicating the reason for failure.

See Also `mdlUserPrefs_saveStartupInfo`, `mdlUserPrefs_deleteStartupInfo`.

mdlUserPrefs_saveStartupInfo

```
int mdlUserPrefs_saveStartupInfo
(
    char    *fileNameP,    /* => file contain. app; NULL=no save filename */
    int     argc,          /* => num args to save for next startup */
    char    *argv[],       /* => the args */
    int     rscType,       /* => normally RTYPE_DgnAppStartup */
    char    *taskNameP     /* => task ID or NULL for current */
);
```

Description `mdlUserPrefs_saveStartupInfo` saves a startup information resource in the user preferences file for later use by MicroStation. If startup resources of type `RTYPE_DgnAppStartup` are found in the user preferences file upon entering design file graphics, the applications indicated are loaded as design file applications (DGNAPPS) by MicroStation.

fileNameP is the name of the application file.

argcP and *argvP* are the command line parameters to be passed to the application at startup.

rscType defines the type of startup information resource to be loaded from user preferences. `RTYPE_DgnAppStartup` is currently the only MicroStation recognized resource.

taskNameP defines the application to which the resource applies and is used as an alias for the specific resource information.

Returns `mdlUserPrefs_saveStartupInfo` returns `SUCCESS` the information was saved successfully or a non-zero value indicating the reason for failure.

See Also `mdlUserPrefs_saveStartupInfo`, `mdlUserPrefs_deleteStartupInfo`.

Miscellaneous Functions

The following table lists miscellaneous MDL functions:

Function	Used to
<code>mdlUtil_quickSort</code>	sort an array of elements of a certain size.
<code>mdlUtil_sortDoubles</code>	sort an array of doubles.
<code>mdlUtil_sortLongs</code>	sort an array of long integers.
<code>mdlUtil_sortStrings</code>	sort an array of pointers to strings based on the values of the strings.
<code>mdlUtil_beep</code>	beep the speaker a specified number of times.
<code>mdlMemory_showHeap</code>	show information on MicroStation's heap. This is a diagnostic tool.
<code>mdlVersion_getPlatform</code>	get platform information for the current session of MicroStation.
<code>mdlVersion_getVersionNumbers</code>	get detailed version information for the currently executing MicroStation.

Example

See `misc.mc`.

`mdlUtil_quickSort`, `mdlUtil_sortDoubles`, `mdlUtil_sortLongs`, `mdlUtil_sortStrings`

```
#include <mdl.h>

void mdlUtil_quickSort
(
    void          *base,
    int           numEntries,
    int           entrySize,
    MdlFunctionP  compareFunc
);

void mdlUtil_sortDoubles
(
    double *base,
    int     numEntries,
    int     ascend
);

void mdlUtil_sortLongs
(
    long *base,
```

```

int      numEntries,
int      ascend
);

void mdlUtil_sortStrings
(
long     *base,
int      numEntries,
int      ascend
);

```

Description The `mdlUtil_quickSort` function is an implementation of the standard quick-sort algorithm that sorts an array of *numEntries* elements, each of size *entrySize*. *base* gives the address of the beginning of the array. The array is sorted in place.

compareFunc points to a function that `mdlUtil_quickSort` calls one or more times to compare two array elements and return a value specifying their relationship. *compareFunc* is called with pointers to the two array entries as its arguments. This argument should return the following:

Return value	Meaning
-1	<i>entry1</i> less than <i>entry2</i> .
0	<i>entry1</i> equal to <i>entry2</i> .
1	<i>entry1</i> greater than <i>entry2</i> .

To reverse the sense of the sort (ascending or descending), reverse the sense of *compareFunc*'s return value.

The `mdlUtil_sortLongs` and `mdlUtil_sortDoubles` functions are optimized implementations of `mdlUtil_quickSort` for sorting an array of *numEntries* long integers or double-precision values. *base* is the start of the array. If *ascend* is `TRUE`, the array is sorted in ascending order. Otherwise, it is sorted in descending order.

The `mdlUtil_sortStrings` function is an optimized implementation of `mdlUtil_quickSort` for sorting an array of *numEntries* pointers. The pointers are sorted based on the result of supplying the pointers to `strcmp`. *base* is the start of the array. If *ascend* is `TRUE`, the array is sorted in ascending order. Otherwise, it is sorted in descending order.

Returns `mdlUtil_quickSort`, `mdlUtil_sortDoubles`, `mdlUtil_sortLongs` and `mdlUtil_sortStrings` are of type `void`; they return no values.

mdlUtil_beep

```

#include <mdl.h>

void mdlUtil_beep
(

```

```
int    numBeeps
);
```

Description The mdlUtil_beep function beeps the speaker *numBeeps* times.

Returns mdlUtil_beep is of type void; it returns no value.

mdlMemory_showHeap

```
void mdlMemory_showHeap
(
char    *headingP,    /* => display at start of dump */
char    *sourceP      /* => the source that allocated memory */
);
```

Description The mdlMemory_showHeap function displays information on all memory allocated by MicroStation. The information is printed to stdout.

headingP points to a string to display at the start of the display.

sourceP points to a string that contains a source-specifier. A source-specifier restricts the display to memory allocated by a source. *sourceP* can be NULL if the dump will not be restricted to a specific source.

All source-specifiers maintained by the heap manager are four-character codes. The source-specifier passed in *sourceP* can have up to four characters. If every character in *sourceP* matches the corresponding characters in the heap manager's code for a block of memory, mdlMemory_showHeap displays information on that block of memory.

Returns The mdlMemory_showHeap function is of type void. It returns no value.

See Also The MDL debugger's MEMORY commands.

mdlVersion_getPlatform

```
#include <mdl.h>

void mdlVersion_getPlatform
(
int    *platformIdP, /* <= save platform id number here */
char    *platformName, /* <= platform name */
int    maxNameSize /* <= maximum size of name */
);
```

Description mdlVersion_getPlatform is used to obtain the ID and name of the current platform where MicroStation is running.

platformIdP points to an int value where the platform id number corresponding to the current platform will be saved. The platform ids for each platform are defined in basedefs.h.

platformName points to a character buffer where the ASCII name of the platform will be returned.

maxNameSize is the size of the buffer *platformName*.

Returns The `mdlVersion_getPlatform` function is of type `void`.

See Also `mdlVersion_getVersionNumbers`.

mdlVersion_getVersionNumbers

```
#include <mdl.h>

void mdlVersion_getVersionNumbers
(
  VersionNumber    *versionNumberP,    /* <= x.x.x.x, e.g., 5.0.0.0 */
  int              *developmentP       /* <= BSI dev number or NULL */
);
```

Description `mdlVersion_getVersionNumbers` is used to obtain the version number of the currently executing MicroStation executable.

versionNumberP points to a `VersionNumber` variable to receive the version information.

developmentP points to an integer variable where the special development version number, if any, is saved. This parameter is usually set to `NULL`.

Returns The `mdlVersion_getVersionNumbers` function is of type `void`.

See Also `mdlVersion_getPlatform`.

HTML Authoring Tool Library

The MDL HTML authoring functions provide a standard method for MDL programmers to generate HTML pages while isolating the HTML tags into an MDL shared library, `htmllib.msl`. Some of the functions create HTML content from standard MDL data structures, such as creating a table from an MDL string list. The more complicated functions are built from simple functions within the library, so you can choose the level of control as you like.

HTML Authoring Functions

Supported HTML tags include tables, frames, lists and image maps along with a set of standard tags.

The following table lists the HTML authoring tool library functions:

Function	Used to
<code>mdlHTMllib_addCaption</code>	add a table caption tag to an HTML file.
<code>mdlHTMllib_addCitationText</code>	add citation text to an HTML file.
<code>mdlHTMllib_addDefinitionListItem</code>	add item tag in a definition list.
<code>mdlHTMllib_addEmphasizedText</code>	add emphasized text to an HTML file.
<code>mdlHTMllib_addFormButtonItem</code>	add a submit or reset button to a form.
<code>mdlHTMllib_addFormCheckBoxRadioButtonItem</code>	add a form check box or radio button to a form.
<code>mdlHTMllib_addFormImageItem</code>	add an image item to a form.
<code>mdlHTMllib_addFormSelectOptionItem</code>	add a new selection to a form selection item.
<code>mdlHTMllib_addFormTextItem</code>	add a text entry field to a form.
<code>mdlHTMllib_addFormHiddenItem</code>	add a hidden text item to a form.
<code>mdlHTMllib_addFrame</code>	add a complete frame tag to an HTML file.
<code>mdlHTMllib_addFrameset</code>	add an opening frameset to an HTML file.
<code>mdlHTMllib_addHeading</code>	add a specified level heading tag to an HTML file.
<code>mdlHTMllib_addHorizontalRuledLine</code>	add a horizontal line to an HTML file.
<code>mdlHTMllib_addHREFAnchor</code>	add an anchor string with a link to the specified level.

Function	Used to
mdlHTMllib_addImage	add an image tag to an HTML file.
mdlHTMllib_addKeyboardSampleText	add keyboard sample text to an HTML file.
mdlHTMllib_addLineBreak	add a line break to an HTML file.
mdlHTMllib_addListItem	add a list item tag to an HTML file.
mdlHTMllib_addListItemOpt	add a list item tag to an HTML file including the text within the list item.
mdlHTMllib_addMapArea	add a clickable area to an image map.
mdlHTMllib_addNameAnchor	add a name anchor tag to an HTML file.
mdlHTMllib_addNewLine	add a newline character in an HTML file.
mdlHTMllib_addNoFrameMessage	add a NOFRAME message to an HTML file.
mdlHTMllib_addPlainText	add plain text to an HTML file.
mdlHTMllib_addProtocolToString	add a protocol (http, mailto, etc.) to a hostname.
mdlHTMllib_addProtocolToStringPath	add a protocol (http, mailto, etc.) to a pathname.
mdlHTMllib_addSourceCodeText	add source code text to to an HTML file.
mdlHTMllib_addStartFormSelectItem	write an opening tag to a form.
mdlHTMllib_addStrongText	add strong text to an HTML file.
mdlHTMllib_addTableCell	write a table cell, with opening and closing tags and text content, to an HTML file.
mdlHTMllib_addTableHeaderRow	add a header row for a table.
mdlHTMllib_addTextAreaItem	add a text area item to a form.
mdlHTMllib_addTextString	add a text string to an HTML file.
mdlHTMllib_addVariableName	add a variable name to an HTML file.
mdlHTMllib_beginFrameDocument	create an HTML file with MicroStation as generator.
mdlHTMllib_beginHTMLDocument	create an HTML file.
mdlHTMllib_closeFile	close an HTML file.
mdlHTMllib_closeFrame	write the closing tag for a frame.
mdlHTMllib_createFile	create an HTML file.
mdlHTMllib_endAnchor	write the closing anchor tag to an HTML file.
mdlHTMllib_endBody	write the closing body tag to an HTML file.

Function	Used to
mdlHTMllib_endCenter	add the closing tag to an HTML file to end centering.
mdlHTMllib_endDefinitionList	write the closing tag for a definition list to an HTML file.
mdlHTMllib_endDirectoryList	write the closing tag for a directory list to an HTML file.
mdlHTMllib_endFileHeader	write the closing file header tag to an HTML file.
mdlHTMllib_endForm	write the ending tag for a form to an HTML file.
mdlHTMllib_endFormSelectItem	write the closing tag for form select to an HTML file.
mdlHTMllib_endFrameDocument	write the closing HTML and frameset tags to an HTML file.
mdlHTMllib_endFrameset	write the closing frameset tag to an HTML file.
mdlHTMllib_endHTML	write the closing tag to an HTML file.
mdlHTMllib_endHTMLDocument	write the closing tag to an HTML file and close the file.
mdlHTMllib_endMap	write the closing tag to an image map.
mdlHTMllib_endMenuList	add the closing tag to a menu list.
mdlHTMllib_endNoFrame	write the closing tag to the NOFRAMES portion of a frame document.
mdlHTMllib_endOrderedList	add the closing tag for a numbered list to an HTML file.
mdlHTMllib_endParagraph	write the closing paragraph tag to an HTML file.
mdlHTMllib_endTable	write the closing tag for a table to an HTML file.
mdlHTMllib_endTableCell	write the closing tag for a table cell to an HTML file.
mdlHTMllib_endTableRow	write the closing tag for a table row to an HTML file.
mdlHTMllib_endUnorderedList	write the closing tag for a bulleted list to an HTML file.
mdlHTMllib_hrefListFromStringList	create a list of the specified type in an HTML file, the first member of the specification must be HREF.

Function	Used to
mdlHTMllib_htmlListFromStringList	create a list of the specified type in an HTML file.
mdlHTMllib_imageTableFromList	create a table of text entered with HREF links in an HTML file.
mdlHTMllib_openFile	open the HTML file specified.
mdlHTMllib_setDocTypeToHTML	write the comment tag that indicates HTML content.
mdlHTMllib_setMetaNameGenerator	set the generator meta name tag with version number.
mdlHTMllib_setPageTitle	write a title tag to an HTML file.
mdlHTMllib_startBody	write the opening body tag to an HTML file.
mdlHTMllib_startCenter	add the opening tag to an HTML file that begins centering.
mdlHTMllib_startDirectoryList	add the header tag for a directory list to an HTML file.
mdlHTMllib_startDefinitionList	write the header tag for a definition list to an HTML file.
mdlHTMllib_startFileHeader	add an opening file header tag to an HTML file.
mdlHTMllib_startForm	write the ending tag for a form to an HTML file.
mdlHTMllib_startHREFAnchor	write an opening tag for an anchor to an HTML file.
mdlHTMllib_startHTML	write the opening HTML tag to an HTML file.
mdlHTMllib_startMap	write the opening tag for an image map to an HTML file.
mdlHTMllib_startMenuList	add a menu list header tag to an HTML file.
mdlHTMllib_startNameAnchor	write the opening tag for a name anchor to an HTML file.
mdlHTMllib_startNoFrame	write the opening tag for the NOFRAMES portion of a frame document.
mdlHTMllib_startOrderedList	add the header tag for a number list to an HTML file.
mdlHTMllib_startParagraph	write the opening paragraph tag to an HTML file.

Function	Used to
mdlHTMMLib_startStandardTable	write the opening tag for a table with no alignment information to an HTML file.
mdlHTMMLib_startTable	write the opening tag for a table to an HTML file.
mdlHTMMLib_startTableCell	write the opening tag for an HTML table cell.
mdlHTMMLib_startTableRow	write the opening tag for an HTML table row.
mdlHTMMLib_startTableRowAlign	write the opening tag for a table row.
mdlHTMMLib_startUnorderedList	add the header tag for a bulleted list.
mdlHTMMLib_textHREFTableFromList	create a table of text entries with HREF links.
mdlHTMMLib_textTableFromList	create a table of text entries.
mdlTag_extractURL	return URL and description from a tagged element.
mdlTag_hasInternetTagSet	check for Internet tags in design file.
mdlTag_createInternetTagSet	create the Internet tags in design file.
mdlTag_attachURL	attach URL and description tags to specified element.
mdlWeb_getUrlToFileBegin	send a request.
mdlWeb_getUrlProgress	check status of current URL request.
mdlWeb_getUrlFinish	clean up data associated with URL request.
mdlWeb_stopBrowser	stop existing URL request.
mdlWeb_getLastModified	return last modified date of a remote file.

mdlHTMMLib_addCaption

```

Public StatusInt
(
    FILE          *pHTMLFile,    /* => handle to HTML text file */
    char          *psCaption,    /* => table caption */
    int           iCaptionAlign, /* => where caption appears the table */
    BoolInt       bNewLine      /* => newline char ? (file formatting) */
);

```

Description mdlHTMMLib_addCaption writes a table caption tag to the HTML text file specified by *pHTMLFile*.

psCaption specifies the text of the table caption.

iCaptionAlign specifies alignment information using the values in `htmlib.h`: `ALIGN_CAPTION_LEFTTOP`, `ALIGN_CAPTION_CENTERTOP`, `ALIGN_CAPTION_RIGHTTOP`, `ALIGN_CAPTION_LEFTBOTTOM`, `ALIGN_CAPTION_CENTERBOTTOM` and `ALIGN_CAPTION_RIGHTBOTTOM`.

Returns `mdlHTMMLib_addCaption` returns `SUCCESS` if the tag was written to the file successfully, `ERROR` otherwise.

mdlHTMMLib_addCitationText

```
Public StatusInt mdlHTMMLib_addCitationText
(
    FILE          *pHTMLFile,      /* => handle to HTML text file */
    char          *psText,         /* => text string to add */
    BoolInt       bNewLine         /* => newline char ? (file formatting) */
);
```

Description `mdlHTMMLib_addCitationText` adds citation text to the HTML text file specified by *pHTMLFile*.

psText specifies the text of the citation.

Returns `mdlHTMMLib_addCitationText` returns `SUCCESS` if the text was written to the file successfully, `ERROR` otherwise.

mdlHTMMLib_addDefinitionListItem

```
Public StatusInt mdlHTMMLib_addDefinitionListItem
(
    FILE          *pHTMLFile,      /* => handle to HTML text file */
    char          *psTitle,        /* => definition term */
    char          *psDefinition,   /* => definition of term */
    char          *psHref,         /* => link to */
    BoolInt       bNewLine         /* => newline char ? (file formatting) */
);
```

Description `mdlHTMMLib_addDefinitionListItem` adds an item tag, in a definition list, to the HTML text file specified by *pHTMLFile*.

psTitle specifies the definition term.

psDefinition specifies the text of the definition term, *psTitle*.

psHref (optional) specifies an HREF link.

Returns `mdlHTMMLib_addDefinitionListItem` returns `SUCCESS` if the tag was written to the file successfully, `ERROR` otherwise.

mdlHTMllib_addEmphasizedText

```
Public StatusInt mdlHTMllib_addEmphasizedText
(
    FILE          *pHTMLFile,      /* => handle to HTML text file */
    char          *psText,         /* => text string to add */
    BoolInt       bNewLine         /* => newline char ? (file formatting) */
);
```

Description *mdlHTMllib_addEmphasizedText* adds emphasized text to the HTML text file specified in *pHTMLFile*.

psText specifies the emphasized text.

Returns *mdlHTMllib_addEmphasizedText* returns SUCCESS if the text was written to the file successfully, ERROR otherwise.

mdlHTMllib_addFormButtonItem

```
Public StatusInt mdlHTMllib_addFormButtonItem
(
    FILE          *pHTMLFile,      /* => handle to HTML text file */
    char          *psName,         /* => item name, opt */
    char          *psValue,        /* => item value, opt */
    int           iButtonType,     /* => FORMTYPEBUTTON_SUBMIT, etc. */
    BoolInt       bNewLine         /* => newline char ? (file formatting) */
);
```

Description *mdlHTMllib_addFormButtonItem* adds a submit or reset button to the form.

psName specifies the name of the button item.

psValue specifies the value, if non-NULL, of the button item.

iButtonType specifies the button type using values found in *htmlib.h*:
 FORMTYPEBUTTON_SUBMIT or FORMTYPEBUTTON_RESET.

Returns *mdlHTMllib_addFormButtonItem* returns SUCCESS if the item was added to the file successfully, ERROR otherwise.

mdlHTMllib_addFormCheckBoxRadioButtonItem

```
Public StatusInt mdlHTMllib_addFormCheckBoxRadioButtonItem
(
    FILE          *pHTMLFile,      /* => handle to HTML text file */
    char          *psName,         /* => item name */
    char          *psValue,        /* => item value, opt */
    int           iButton,         /* => FORMTYPE_RADIO, etc. */
    int           iChecked,        /* => FORMTYPE_NONCHECKED, etc. */
    BoolInt       bNewLine         /* => newline char ? (file formatting) */
);
```

Description mdlHTMMLib_addFormCheckBoxRadioButtonItem adds a form check box or radio button item to the form in the HTML text file specified by *pHTMLFile*.

psName specifies the name of the form check box or radio button item.

psValue specifies the initial value of the form check box or radio button item. *psValue* is optional and may be NULL.

iButton specifies whether a form check box or radio button item is added using values found in `htmllib.h`: `FORMTYPE_RADIO` or `FORMTYPE_CHECKBOX`.

iChecked specifies if the item is checked in the form, see values in `htmllib.h`: `FORMTYPE_NONCHECKED` or `FORMTYPE_CHECKED`. For radio button items, only one button can be checked.

Returns mdlHTMMLib_addFormCheckBoxRadioButtonItem returns `SUCCESS` if the item was written to the file successfully, `ERROR` otherwise.

mdlHTMMLib_addFormImageItem

```
Public StatusInt mdlHTMMLib_addFormImageItem
(
FILE          *pHTMLFile,    /* => handle to HTML text file */
char          *psName,       /* => item name */
char          *psSource,     /* => image source, URL/path */
BoolInt      bNewLine       /* => newline char ? for file formatting */
);
```

Description mdlHTMMLib_addFormImageItem adds an image item to the form in the HTML text file specified by *pHTMLFile*.

psName specifies the name of the image item.

psSource specifies the source URL.

Returns mdlHTMMLib_addFormImageItem returns `SUCCESS` if the image item was written to the file successfully, `ERROR` otherwise.

mdlHTMMLib_addFormSelectOptionItem

```
Public StatusInt mdlHTMMLib_addFormSelectOptionItem
(
FILE          *pHTMLFile,    /* => handle to HTML text file */
char          *psValue,      /* => item value, opt */
int           iSelected,     /* => FORMSELECT_NOSELECTED, etc. */
BoolInt      bNewLine       /* => newline char ? (file formatting) */
);
```

Description mdlHTMMLib_addFormSelectOptionItem adds a new selection to the form selection item in the HTML text file specified by *pHTMLFile*.

psValue specifies the text of the selection.

iSelected specifies whether the entry is initially selected (default) using one of the values found in `htmlib.h`: `FORMSELECT_NOSELECTED` or `FORMSELECT_SELECTED`.

Returns `mdlHTMMLib_addFormSelectOptionItem` returns `SUCCESS` if the selection was added successfully, `ERROR` otherwise.

mdlHTMMLib_addFormTextItem

```
Public StatusInt mdlHTMMLib_addFormTextItem
(
FILE          *pHTMLFile,      /* => handle to HTML text file */
char          *psName,         /* => item name */
char          *psValue,        /* => item value, default for item */
int           iSize,           /* => text item size, opt */
int           iMaxLength,      /* => text item maxlength, opt */
int           iPassword,       /* => FORMTYPETEXT_NONPASSWORD, _PASSWORD */
BoolInt       bNewLine        /* => newline char ? (file formatting) */
);
```

Description `mdlHTMMLib_addFormTextItem` adds a text entry field item to the form in the HTML text file specified by *pHTMLFile*.

psName specifies the name of the text entry field item.

psValue specifies the default value of the text entry field item. *psValue* can be NULL.

iSize specifies size of the text item.

iMaxLength specifies the maximum text string length.

Passing zero for either *iSize* or *iMaxLength* will use HTML defaults for these parameters.

iPassword specifies whether the text item will be used to accept password text, (masked display), by setting to one of the two values in `htmlib.h`: `FORMTYPETEXT_NONPASSWORD` or `FORMTYPETEXT_PASSWORD`.

Returns `mdlHTMMLib_addFormTextItem` returns `SUCCESS` if the text entry field was written to the form successfully, `ERROR` otherwise.

mdlHTMMLib_addFormHiddenItem

```
Public StatusInt mdlHTMMLib_addFormHiddenItem
(
FILE          *pHTMLFile,      /* => handle to HTML text file */
char          *psName,         /* => item name */
char          *psValue,        /* => item value */
BoolInt       bNewLine        /* => newline char ? (file formatting) */
);
```

Description mdlHTMMLib_addFormHiddenItem adds a hidden text item to the form in the HTML text file specified by *pHTMLFile*.

psName specifies the item name.

psValue specifies the default value. *psValue* can be NULL.

Returns mdlHTMMLib_addFormHiddenItem returns SUCCESS if the item was added to the form successfully, ERROR otherwise.

mdlHTMMLib_addFrame

```
Public StatusInt mdlHTMMLib_addFrame
(
    FILE          *pHTMLFile,      /* => handle to HTML text file */
    char          *psSource,        /* => source document (required) */
    char          *psName,          /* => target name or NULL for none */
    BoolInt       bResize,          /* => false adds the NORESIZE attribute */
    int           iScrolling,        /* => permanent, never, default */
    int           iMarginHeight,     /* => pixels above and below frame */
    int           iMarginWidth,      /* => pixels beside the frame */
    BoolInt       bNewLine          /* => newline char ? (file formatting) */
);
```

Description mdlHTMMLib_addFrame adds a complete frame tag, including the source file, to the HTML text file specified by *pHTMLFile*.

psSource specifies the source document.

psName specifies a target name (optional). *psName* can be NULL.

If *bResize* is FALSE, the NORESIZE attribute is added.

iScrolling specifies permanent, never, or default scrolling. *iScrolling* equals 1 for permanent scrolling, -1 for never scrolling and 0 for default.

iMarginHeight and *iMarginWidth* specify the margin around the frame. When either *iMarginHeight* or *iMarginWidth* are 0, the corresponding margin is omitted.

Returns mdlHTMMLib_addFrame returns SUCCESS if the tag was written to the file successfully, ERROR otherwise.

mdlHTMMLib_addFrameset

```
Public StatusInt mdlHTMMLib_addFrameset
(
    FILE          *pHTMLFile,      /* => handle to HTML text file */
    int           iNumRows,         /* => number of rows in frameset */
    int           *piRowSizes,      /* => row size (array of iNumRows ints) */
    BoolInt       bRelRows,         /* => relative or absolute row size */
    int           iNumCols,         /* => number of columns in frameset */
    int           *piColSizes,      /* => col. size (array of iNumCols ints) */
);
```

```

BoolInt      bRelCols,      /* => relative or absolute column size */
BoolInt      bNewLine      /* => newline char ? (file formatting) */
);

```

Description mdlHTMMLib_addFrameset adds an opening frameset tag to the HTML text file specified by *pHTMLFile*.

iNumRows specifies the number of rows.

piRowSizes specifies the row sizes, -1 represents the asterisk. For example, a value of -1 translates to '*', while a value of -3 translates to 3*.

bRelRows determines how the values of *piRowSizes* will be interpreted. If *bRelRows* is TRUE, values in *piRowSizes* represent percentages of the frame area (relative row size), else if FALSE, values represent absolute pixels (absolute row size).

iNumCols specifies the number of columns.

piColSizes specifies the column sizes, -1 represents the asterisk. For example, a value of -1 translates to '*', while a value of -3 translates to 3*.

bRelCols determines how the values of *piColSizes* will be interpreted. If *bRelCols* is TRUE, values in *piColSizes* represent percentages of the frame area (relative column size), else if FALSE, values represent absolute pixels (absolute column size).

Returns mdlHTMMLib_addFrameset returns SUCCESS if the tag was written to the file successfully, ERROR otherwise.

mdlHTMMLib_addHeading

```

Public StatusInt mdlHTMMLib_addHeading
(
FILE          *pHTMLFile,      /* => handle to HTML text file */
char          *psHeading,      /* => text for heading */
int           iHeadingLevel,   /* => 1-6 */
int           iAlignValue,     /* => alignment for heading */
BoolInt       bNewLine        /* => newline char ? (file formatting) */
);

```

Description mdlHTMMLib_addHeading adds a heading to the HTML text file specified by *pHTMLFile*.

psHeading specifies the text of the heading.

iHeadingLevel specifies the level of heading (1-6) added.

iAlignValue specifies the alignment from the values in `htmlLib.h`: ALIGN_ITEM_LEFT, ALIGN_ITEM_CENTER and ALIGN_ITEM_RIGHT.

Returns mdlHTMMLib_addHeading returns SUCCESS if the heading was added to the file successfully, ERROR otherwise.

mdlHTMMLib_addHorizontalRuledLine

```
Public StatusInt mdlHTMMLib_addHorizontalRuledLine
(
FILE          *pHTMLFile,      /* => handle to HTML text file */
int           iSize,           /* => vertical width in pixels */
int           iWidth,          /* => horizontal width in percent */
int           iAlign,          /* => ALIGN_ITEM_LEFT, etc. */
BoolInt       bNoShade,        /* => Draw dividing line as solid bar */
BoolInt       bNewLine         /* => newline char ? (file formatting) */
);
```

Description mdlHTMMLib_addHorizontalRuledLine adds a horizontal line to the HTML text file specified by *pHTMLFile*.

iSize specifies vertical height of the line (thickness) in pixels.

iWidth specifies width of line in percent.

iAlign specifies alignment of the line with respect to the page using values found in `htmlib.h`: ALIGN_ITEM_LEFT, ALIGN_ITEM_CENTER and ALIGN_ITEM_RIGHT.

bNoShade determines 3D shading. 3D shading is turned off when *bNoShade* is TRUE.

Returns mdlHTMMLib_addHorizontalRuledLine returns SUCCESS if the line was added to the file successfully, ERROR otherwise.

mdlHTMMLib_addHREFAnchor

```
Public StatusInt mdlHTMMLib_addHREFAnchor
(
FILE          *pHTMLFile,      /* => handle to HTML text file */
char          *psRefFileName,   /* => string (filename) for HREF */
char          *psText,          /* => text displayed for link */
BoolInt       bNewLine         /* => newline char ? (file formatting) */
);
```

Description mdlHTMMLib_addHREFAnchor adds an anchor string with a link to the HTML text file specified by *pHTMLFile*.

psRefFileName specifies the filename used in the link.

psText specifies the text displayed by the link.

Returns mdlHTMMLib_addHREFAnchor returns SUCCESS if the anchor string was written to the file successfully, ERROR otherwise.

mdlHTMMLib_addImage

```
Public StatusInt mdlHTMMLib_addImage
(
```

```
FILE      *pHTMLFile,          /* => handle to HTML text file */
char      *psImageFileName,    /* => image file (absolute or relative) */
char      *psAltText,          /* => alternate text (optional) */
int       iAlignValue,         /* => image alignment (from htmllib.h) */
int       iImageHeight,        /* => image height (0=no specify) */
int       iImageWidth,         /* => image width (0=no specify) */
char      *psUseMap,           /* => map name (optional) */
BoolInt bNewLine               /* => newline char ? (file formatting) */
);
```

Description mdlHTMllib_addImage adds an image tag to the HTML text file specified by *pHTMLFile*.

psImageFileName specifies the image to display.

psAltText specifies alternate text which is displayed if the image file is not available.

iAlignValue specifies image alignment using values from htmllib.h:

ALIGN_IMAGE_ABSBOTTOM, ALIGN_IMAGE_ABSMIDDLE, ALIGN_IMAGE_BOTTOM,
ALIGN_IMAGE_LEFT, ALIGN_IMAGE_MIDDLE, ALIGN_IMAGE_RIGHT and
ALIGN_IMAGE_TOP.

iImageHeight specifies image height in pixels.

iImageWidth specifies image width in pixels.

psUseMap specifies an optional map name, i.e., if the image is clickable.

psUseMap can be NULL when no map is required.

Returns mdlHTMllib_addImage returns SUCCESS if the tag was written to the file successfully, ERROR otherwise.

mdlHTMllib_addKeyboardSampleText

```
Public StatusInt mdlHTMllib_addKeyboardSampleText
(
FILE      *pHTMLFile,          /* => handle to HTML text file */
char      *psText,             /* => text string to add */
BoolInt bNewLine               /* => newline char ? (for file formatting) */
);
```

Description mdlHTMllib_addKeyboardSampleText adds keyboard sample text to the HTML text file specified by *pHTMLFile*.

psText specifies the keyboard sample text.

Returns mdlHTMllib_addKeyboardSampleText returns SUCCESS if the text was added to the file successfully, ERROR otherwise.

mdlHTMMLib_addLineBreak

```
Public StatusInt mdlHTMMLib_addLineBreak
(
FILE          *pHTMLFile,    /* => handle to HTML text file */
BoolInt       bNewLine      /* => newline char ? (file formatting) */
);
```

Description mdlHTMMLib_addLineBreak adds a line break tag to the HTML text file specified by *pHTMLFile*.

Returns mdlHTMMLib_addLineBreak returns SUCCESS if the tag was written to the file successfully, ERROR otherwise.

mdlHTMMLib_addListItem

```
Public StatusInt mdlHTMMLib_addListItem
(
FILE          *pHTMLFile,    /* => handle to HTML text file */
char          *psListItem,   /* => text to include in the list item */
char          *psHref,       /* => link to */
BoolInt       bNewLine      /* => newline char ? (file formatting) */
);
```

Description mdlHTMMLib_addListItem adds a list item tag to the HTML text file specified by *pHTMLFile*.

psListItem specifies the text of the list item.

psHref specifies an optional HREF link.

Returns mdlHTMMLib_addListItem returns SUCCESS if the tag was written to the file successfully, ERROR otherwise.

mdlHTMMLib_addListItemOpt

```
Public StatusInt mdlHTMMLib_addListItemOpt
(
FILE          *pHTMLFile,    /* => handle to HTML text file */
char          *psListItem,   /* => text to include in list item */
char          *psHref,       /* => link to */
BoolInt       bNewLine      /* => newline char ? (file formatting) */
);
```

Description mdlHTMMLib_addListItemOpt writes the list item tag to the HTML text file specified by *pHTMLFile*.

psListItem specifies the text to include in the list item, can be NULL.

psHref specifies an HREF link, can be NULL.

psListItem can contain the text to be included in the list item or *psHref* can contain an HREF string. Each can be NULL, in which case, only the list item

tag is written to the HTML file. If both *psListItem* and *psHref* are non-NULL, *psListItem* will be ignored.

Returns mdlHTMMLib_addListItemOpt returns SUCCESS if the tag was written to the file successfully, ERROR otherwise.

mdlHTMMLib_addMapArea

```
Public StatusInt mdlHTMMLib_addMapArea
(
FILE          *pHTMLFile,      /* => handle to HTML text file */
int           iShape,          /* => IMAGESHAPE_RECTANGLE, etc. */
int           iNumCoords,      /* => number of coords */
Point2d       *pPtShapeCoords, /* => coords for shape */
char          *psHref,         /* => reference -- NULL for NOHREF */
BoolInt       bNewLine        /* => newline char ? (file formatting) */
);
```

Description mdlHTMMLib_addMapArea adds a clickable area to an image map to the HTML text file specified by *pHTMLFile*.

iShape specifies the shape of the area using values found in *htmlLib.h*, *IMAGESHAPE_RECTANGLE*, *IMAGESHAPE_CIRCLE* and *IMAGESHAPE_POLYGON*.

iNumCoords specifies the number of coordinates that define the shape of the clickable area.

pPtShapeCoords specifies the coordinates of the shape.

psHref (optional) specifies a link.

Returns mdlHTMMLib_addMapArea returns SUCCESS if the image map was added to the file successfully, ERROR otherwise.

mdlHTMMLib_addNameAnchor

```
Public StatusInt mdlHTMMLib_addNameAnchor
(
FILE          *pHTMLFile,      /* => handle to HTML text file */
char          *psRefLabel,     /* => string (label name) for link */
char          *psText,         /* => text displayed for link */
BoolInt       bNewLine        /* => newline char ? (file formatting) */
);
```

Description mdlHTMMLib_addNameAnchor adds a name anchor tag to the HTML text file specified by *pHTMLFile*.

psRefLabel specifies the label name of the link.

psText specifies the text which is displayed as the link.

Returns mdlHTMMLib_addNameAnchor returns SUCCESS if the tag was written to the file successfully, ERROR otherwise.

mdlHTMMLib_addNewLine

```
Public StatusInt mdlHTMMLib_addNewLine
(
FILE      *pHTMLFile      /* => handle to HTML text file */
);
```

Description mdlHTMMLib_addNewLine adds a newline character in the HTML text file specified by *pHTMLFile*, (used to format the file for viewing outside a browser).

Returns mdlHTMMLib_addNewLine returns SUCCESS if the tag was written to the file successfully, ERROR otherwise.

mdlHTMMLib_addNoFrameMessage

```
Public StatusInt mdlHTMMLib_addNoFrameMessage
(
FILE      *pHTMLFile,      /* => handle to HTML text file */
char      *psMessage,      /* => msg displayed in NOFRAMES case */
BoolInt bNewLine          /* => newline char ? (file formatting) */
);
```

Description mdlHTMMLib_addNoFrameMessage adds a NOFRAME message to the HTML text file specified by *pHTMLFile*.

psMessage specifies the displayed message. This message will be displayed if the frame document is accessed with a browser that does not support frames.

Returns mdlHTMMLib_addNoFrameMessage returns SUCCESS if the tag was written to the file successfully, ERROR otherwise.

mdlHTMMLib_addPlainText

```
Public StatusInt mdlHTMMLib_addPlainText
(
FILE      *pHTMLFile,      /* => handle to HTML text file */
char      *psText,          /* => text string to add */
BoolInt bNewLine          /* => newline char ? (file formatting) */
);
```

Description mdlHTMMLib_addPlainText adds plain text to the HTML text file specified by *pHTMLFile*.

psText specifies the text to be added.

Returns mdlHTMMLib_addPlainText returns SUCCESS if the tag was written to the file successfully, ERROR otherwise.

mdlHTMMLib_addProtocolToString

```
Public StatusInt mdlHTMMLib_addProtocolToString
(
char    *psOutString, /* <= where to put protocol//hostName string */
char    *psHostName,  /* => host name */
int     iProtocolType /* => protocol identifier */
);
```

Description mdlHTMMLib_addProtocolToString adds a protocol (http, mailto, etc.) to a hostname.

psOutString specifies the character string that will contain the combination of the host name and protocol strings.



psOutString must be large enough to hold the hostname and protocol string.

psHostName specifies the host name string.

iProtocolType specifies the protocol and uses one of the values defined in `htmlib.h`: `HTTP_PROTOCOL`, `FILE_PROTOCOL` and `FTP_PROTOCOL`.

Returns mdlHTMMLib_addProtocolToString returns `SUCCESS` if the tag was written to the file successfully, `ERROR` otherwise.

mdlHTMMLib_addProtocolToStringPath

```
Public StatusInt mdlHTMMLib_addProtocolToStringPath
(
char    *psOutString, /* <= where to put combined string */
char    *psHostName,  /* => hostname */
char    *psPath,       /* => pathname */
int     iProtocolType /* => protocol identifier */
);
```

Description mdlHTMMLib_addProtocolToStringPath combines the specified protocol, hostname and pathname strings.

psOutString specifies the character string that will contain the combination of the host name and protocol strings.



psOutString must be large enough to hold the hostname, pathname and protocol string.

psHostName specifies the hostname string.

psPath specifies the pathname string. If *psPath* is `NULL`, *psHostName* is assumed to contain the hostname and pathname.

iProtocolType specifies the protocol and uses one of the values defined in `htmlib.h`, `HTTP_PROTOCOL`, `FILE_PROTOCOL` and `FTP_PROTOCOL`.

Returns mdlHTMMLib_addProtocolToStringPath returns SUCCESS if the tag was written to the file successfully, ERROR otherwise.

mdlHTMMLib_addSourceCodeText

```
Public StatusInt mdlHTMMLib_addSourceCodeText
(
    FILE      *pHTMLFile,      /* => handle to HTML text file */
    char      *psText,         /* => text string to add */
    BoolInt   bNewLine         /* => newline char ? (file formatting) */
);
```

Description mdlHTMMLib_addSourceCodeText adds source code text to the HTML text file specified by *pHTMLFile*.

psText specifies the text string.

Returns mdlHTMMLib_addSourceCodeText returns SUCCESS if the text was written to the file successfully, ERROR otherwise.

mdlHTMMLib_addStartFormSelectItem

```
Public StatusInt mdlHTMMLib_addStartFormSelectItem
(
    FILE      *pHTMLFile,      /* => handle to HTML text file */
    char      *psName,         /* => item name */
    int       iMultiple,       /* => FORMSELECT_MULTIPLE, etc. */
    int       iSize,           /* => scroll list length */
    BoolInt   bNewLine         /* => newline char ? (file formatting) */
);
```

Description mdlHTMMLib_addStartFormSelectItem writes the opening tag for form select items to the HTML text file specified by *pHTMLFile*.

psName specifies the name of the item.

iMultiple specifies whether multiple selection is permitted by using one of the values found in *htmlib.h*: FORMSELECT_NOMULTIPLE or FORMSELECT_MULTIPLE.

iSize specifies the length of the list.

Returns mdlHTMMLib_addStartFormSelectItem returns SUCCESS if the tag was written to the file successfully, ERROR otherwise.

mdlHTMMLib_addStrongText

```
Public StatusInt mdlHTMMLib_addStrongText
(
    FILE      *pHTMLFile,      /* => handle to HTML text file */
    char      *psText,         /* => text string to add */
);
```

```

BoolInt      bNewLine      /* => newline char ? (file formatting) */
);

```

Description mdlHTMMLib_addStrongText adds strong text to the HTML text file specified by *pHTMLFile*.

psText specifies the text string to be tagged as strong.

Returns mdlHTMMLib_addStrongText returns SUCCESS if the text was written to the file successfully, ERROR otherwise.

mdlHTMMLib_addTableCell

```

Public StatusInt mdlHTMMLib_addTableCell
(
FILE          *pHTMLFile,      /* => handle to HTML text file */
char          *psCellData,     /* => data specification for table cell */
int           iCellAlign,      /* => alignment of data within cell */
BoolInt       bWrapData,       /* => allow data within cell to wrap? */
int           iRowSpan,        /* => how many rows should cell span?
                                (0=leave tag out) */
int           iColSpan,        /* => how many columns should cell span?
                                (0=leave tag out) */
BoolInt       bNewLine        /* => newline char ? (file formatting) */
);

```

Description mdlHTMMLib_addTableCell writes a table cell, with opening and closing tags and text content to the HTML text file specified by *pHTMLFile*.

Optional arguments are documented under mdlHTMMLib_startTableCell.

Returns mdlHTMMLib_addTableCell returns SUCCESS if the tag was written to the file successfully, ERROR otherwise.

mdlHTMMLib_addTableHeaderRow

```

Public StatusInt mdlHTMMLib_addTableHeaderRow
(
FILE          *pHTMLFile,      /* => handle to HTML text file */
StringList    *pHeaderList,    /* => list of table header strings */
BoolInt       bNewLine        /* => newline char ? (file formatting) */
);

```

Description mdlHTMMLib_addTableHeaderRow adds a header row for a table to the HTML text file specified by *pHTMLFile*.

pHeaderList is a string list which contains a member for each column header.

Returns mdlHTMMLib_addTableHeaderRow returns SUCCESS if the header row was written to the file successfully, ERROR otherwise.

mdlHTMMLib_addTextAreaItem

```
Public StatusInt mdlHTMMLib_addTextAreaItem
(
FILE          *pHTMLFile,    /* => handle to HTML text file */
char          *psName,       /* => item name */
int           iRows,         /* => number of rows */
int           iColumns,      /* => number of columns */
char          *psDefText,    /* => default text in box, opt */
int           iWrap,         /* => FORMTEXTAREA_OFF, etc. */
BoolInt       bNewLine      /* => newline char ? (file formatting) */
);
```

Description mdlHTMMLib_addTextAreaItem adds a text area item to the form in the HTML text file specified by *pHTMLFile*.

psName specifies the item name.

iRows specifies the number of rows.

iColumns specifies the number of columns.

psDefText specifies the text that will be initially displayed if non-NULL.

iWrap determines the type of wrapping performed in this item. The optimal values can be found in *htmllib.h*: FORMTEXTAREA_OFF, FORMTEXTAREA_SOFT and FORMTEXTAREA_HARD.

Returns mdlHTMMLib_addTextAreaItem returns SUCCESS if the item was written to the file successfully, ERROR otherwise.

mdlHTMMLib_addTextString

```
Public StatusInt mdlHTMMLib_addTextString
(
FILE          *pHTMLFile,    /* => handle to HTML text file */
char          *psText,       /* => text string to add */
long          nCharFormatMask, /* => bold, underline, italic */
BoolInt       bNewLine      /* => newline char ? (file formatting) */
);
```

Description mdlHTMMLib_addTextString adds a text string tag to the HTML text file specified by *pHTMLFile*.

psText specifies the text string.

nCharFormatMask specifies formatting using the values in *htmllib.h*: TEXTATTR_BOLD, TEXTATTR_ITALIC and TEXTATTR_UNDERLINE.

Returns mdlHTMMLib_addTextString returns SUCCESS if the tag was written to the file successfully, ERROR otherwise.

mdlHTMMLib_addVariableName

```
Public StatusInt mdlHTMMLib_addVariableName
(
FILE          *pHTMLFile,      /* => handle to HTML text file */
char          *psText,         /* => text string to add */
BoolInt      bNewLine         /* => newline char ? (file formatting) */
);
```

Description mdlHTMMLib_addVariableName adds a variable name to the HTML text file specified by *pHTMLFile*.

psText specifies the variable name to be added to the HTML file.

Returns mdlHTMMLib_addVariableName returns SUCCESS if the tag was written to the file successfully, ERROR otherwise.

mdlHTMMLib_beginFrameDocument

```
Public FILE *mdlHTMMLib_beginFrameDocument
(
char          *psHTMLFileName, /* => at least MAXFILELENGTH */
char          *psDocTitle,     /* => document title (NULL-default) */
int           iNumRows,        /* => number of rows in frameset */
int           *piRowSizes,      /* => size of rows */
BoolInt      bRelRows,         /* => relative or absolute row size */
int           iNumCols,        /* => number of columns in frameset */
int           *piColSizes,     /* => size of columns */
BoolInt      bRelCols         /* => relative or absolute col. size */
);
```

Description mdlHTMMLib_beginFrameDocument creates a frame file, sets the document type to HTML and the generator to MicroStation, writes the opening HTML tag, and writes the opening frameset tag from the information supplied.

psHTMLFileName specifies the filename of the newly create frame file.

psDocTitle (optional) specifies the document title.

iNumRows specifies the number of rows.

piRowSizes specifies an integer array whose members are the size of each *iNumRows* row.

bRelRows determines how the values of *piRowSizes* will be interpreted. If *bRelRows* is TRUE, values in *piRowSizes* represent percentages of the frame area (relative row size), else if FALSE, values represent absolute pixels (absolute row size).

iNumCols specifies the number of columns.

piColSizes specifies an integer array whose members are the size of each *iNumCols* column.

bRelCols determines how the values of *piColSizes* will be interpreted. If *bRelCols* is TRUE, values in *piColSizes* represent percentages of the frame area (relative column size), else if FALSE, values represent absolute pixels (absolute column size).

Returns mdlHTMMLib_beginFrameDocument returns a pointer to the file if created successfully, NULL otherwise.

mdlHTMMLib_beginHTMLDocument

```
Public FILE *mdlHTMMLib_beginHTMLDocument
(
    char    *psHTMLFileName,    /* <=> at least MAXFILELENGTH */
    char    *psDocTitle,        /* => document title (NULL for default) */
    char    *psHeader,          /* => level one document header */
    int     iHeaderAlign        /* => header alignment (ALIGN_ITEM_*) */
);
```

Description mdlHTMMLib_beginHTMLDocument creates an HTML text file, sets the document type to HTML, sets the generator meta name, and writes the opening HTML and body tags.

psHTMLFileName specifies the filename of the HTML text file to create.

psDocTitle (optional) specifies the title.

psHeader (optional) specifies a string to be used as the newly created file's level 1 header. Pass in NULL, for none.

iHeaderAlign specifies the alignment of the header using values from `htmlib.h`: ALIGN_ITEM_LEFT, ALIGN_ITEM_CENTER and ALIGN_ITEM_RIGHT.

Returns mdlHTMMLib_beginHTMLDocument returns a pointer to the file if created successfully, NULL otherwise.

mdlHTMMLib_closeFile

```
Public StatusInt mdlHTMMLib_closeFile
(
    FILE    *pHTMLFile    /* => handle to HTML text file */
);
```

Description mdlHTMMLib_closeFile closes the HTML file specified by *pHTMLFile*.

Returns mdlHTMMLib_closeFile returns SUCCESS if the file was closed successfully, ERROR otherwise.

mdlHTMMLib_closeFrame

```
Public StatusInt mdlHTMMLib_closeFrame
(
    FILE          *pHTMLFile,      /* => handle to HTML text file */
    BoolInt       bNewLine         /* => newline char ? (file formatting) */
);
```

Description mdlHTMMLib_closeFrame writes the closing tag for a frame to the HTML text file specified by *pHTMLFile*.

Returns mdlHTMMLib_closeFrame returns SUCCESS if the tag was written to the file successfully, ERROR otherwise.

mdlHTMMLib_createFile

```
Public StatusInt mdlHTMMLib_createFile
(
    char          *psFullFileName   /* <=> filename to create/created file */
);
```

Description mdlHTMMLib_createFile creates the HTML file specified by *psFullFileName*. If the filename was successfully created, its full file specification will be written into *psFullFileName*.

Returns mdlHTMMLib_createFile returns SUCCESS if the file was created successfully, ERROR otherwise.

mdlHTMMLib_endAnchor

```
Public StatusInt mdlHTMMLib_endAnchor
(
    FILE          *pHTMLFile,      /* => handle to HTML text file */
    BoolInt       bNewLine         /* => newline char ? (file formatting) */
);
```

Description mdlHTMMLib_endAnchor writes the closing anchor tag to the HTML text file specified by *pHTMLFile*.

Returns mdlHTMMLib_endAnchor returns SUCCESS if the tag was written to the file successfully, ERROR otherwise.

mdlHTMMLib_endBody

```
Public StatusInt mdlHTMMLib_endBody
(
    FILE          *pHTMLFile,      /* => handle to HTML text file */
    BoolInt       bNewLine         /* => newline char ? (file formatting) */
);
```

Description mdlHTMMLib_endBody writes the closing body tag to the HTML text file specified by *pHTMLFile*.

Returns mdlHTMMLib_endBody returns SUCCESS if the tag was written to the file successfully, ERROR otherwise.

mdlHTMMLib_endCenter

```
Public StatusInt mdlHTMMLib_endCenter
(
    FILE          *pHTMLFile,    /* => handle to HTML text file */
    BoolInt       bNewLine       /* => newline char ? (file formatting) */
);
```

Description mdlHTMMLib_endCenter adds the closing tag to the file to end centering HTML content to the HTML text file specified by *pHTMLFile*.

Returns mdlHTMMLib_endCenter returns SUCCESS if the tag was written to the file successfully, ERROR otherwise.

mdlHTMMLib_endDefinitionList

```
Public StatusInt mdlHTMMLib_endDefinitionList
(
    FILE          *pHTMLFile,    /* => handle to HTML text file */
    BoolInt       bNewLine       /* => newline char ? (file formatting) */
);
```

Description mdlHTMMLib_endDefinitionList writes the closing tag for a definition list to the HTML text file specified by *pHTMLFile*.

Returns mdlHTMMLib_endDefinitionList returns SUCCESS if the tag was written to the file successfully, ERROR otherwise.

mdlHTMMLib_endDirectoryList

```
Public StatusInt mdlHTMMLib_endDirectoryList
(
    FILE          *pHTMLFile,    /* => handle to HTML text file */
    BoolInt       bNewLine       /* => newline char ? (file formatting) */
);
```

Description mdlHTMMLib_endDirectoryList adds the closing tag for a directory list to the HTML text file specified by *pHTMLFile*.

Returns mdlHTMMLib_endDirectoryList returns SUCCESS if the tag was written to the file successfully, ERROR otherwise.

mdlHTMMLib_endFileHeader

```
Public StatusInt mdlHTMMLib_endFileHeader
(
FILE          *pHTMLFile,    /* => handle to HTML text file */
BoolInt       bNewLine       /* => newline char ? (file formatting) */
);
```

Description mdlHTMMLib_endFileHeader writes the closing file header tag to the HTML text file specified by *pHTMLFile*.

Returns mdlHTMMLib_endFileHeader returns SUCCESS if the tag was written to the file successfully, ERROR otherwise.

mdlHTMMLib_endForm

```
Public StatusInt mdlHTMMLib_endForm
(
FILE          *pHTMLFile,    /* => handle to HTML text file */
BoolInt       bNewLine       /* => newline char ? (file formatting) */
);
```

Description mdlHTMMLib_endForm writes the ending tag for a form to the HTML text file specified by *pHTMLFile*.

Returns mdlHTMMLib_endForm returns SUCCESS if the tag was written to the file successfully, ERROR otherwise.

mdlHTMMLib_endFormSelectItem

```
Public StatusInt mdlHTMMLib_endFormSelectItem
(
FILE          *pHTMLFile,    /* => handle to HTML text file */
BoolInt       bNewLine       /* => newline char ? (file formatting) */
);
```

Description mdlHTMMLib_endFormSelectItem writes the closing tag for form select items to the HTML file.

Returns mdlHTMMLib_endFormSelectItem returns SUCCESS if the tag was written to the file successfully, ERROR otherwise.

mdlHTMMLib_endFrameDocument

```
Public StatusInt mdlHTMMLib_endFrameDocument
(
FILE          *pHTMLFile,    /* => handle to HTML text file */
char          *psNoFrameMsg   /* => noframes message, NULL to leave out tag */
);
```

Description mdlHTMMLib_endFrameDocument writes a message in the NOFRAMES tag (optional), writes the closing HTML and frameset tags, and closes the file specified by *pHTMLFile*.

Returns mdlHTMMLib_endFrameDocument returns SUCCESS if the tags were written to the file and the file was closed successfully, ERROR otherwise.

mdlHTMMLib_endFrameset

```
Public StatusInt mdlHTMMLib_endFrameset
(
    FILE          *pHTMLFile,      /* => handle to HTML text file */
    BoolInt       bNewLine         /* => newline char ? (file formatting) */
);
```

Description mdlHTMMLib_endFrameset writes a closing frameset tag to the HTML text file specified by *pHTMLFile*.

Returns mdlHTMMLib_endFrameset returns SUCCESS if the tag was written to the file successfully, ERROR otherwise.

mdlHTMMLib_endHTML

```
Public StatusInt mdlHTMMLib_endHTML
(
    FILE          *pHTMLFile,      /* => handle to HTML text file */
    BoolInt       bNewLine         /* => newline char ? (file formatting) */
);
```

Description mdlHTMMLib_endHTML writes the closing HTML tag to the HTML text file specified by *pHTMLFile*.

Returns mdlHTMMLib_endHTML returns SUCCESS if the tag was written to the file successfully, ERROR otherwise.

mdlHTMMLib_endHTMLDocument

```
Public StatusInt mdlHTMMLib_endHTMLDocument
(
    FILE          *pHTMLFile      /* => handle to HTML text file */
);
```

Description mdlHTMMLib_endHTMLDocument writes the closing body and HTML tags and closes the file specified by *pHTMLFile*.

Returns mdlHTMMLib_endHTMLDocument returns SUCCESS if the tags were written to the file and the file was closed successfully, ERROR otherwise.

mdlHTMMLib_endMap

```
Public StatusInt mdlHTMMLib_endMap
(
    FILE          *pHTMLFile,      /* => handle to HTML text file */
    BoolInt       bNewLine         /* => newline char ? (file formatting) */
);
```

Description mdlHTMMLib_endMap writes the closing tag for an image map to the HTML text file specified by *pHTMLFile*.

Returns mdlHTMMLib_endMap returns SUCCESS if the tag was written to the file successfully, ERROR otherwise.

mdlHTMMLib_endMenuList

```
Public StatusInt mdlHTMMLib_endMenuList
(
    FILE          *pHTMLFile,      /* => handle to HTML text file */
    BoolInt       bNewLine         /* => newline char ? (file formatting) */
);
```

Description mdlHTMMLib_endMenuList adds the closing tag for a menu list to the HTML text file specified by *pHTMLFile*.

Returns mdlHTMMLib_endMenuList returns SUCCESS if the tag was written to the file successfully, ERROR otherwise.

mdlHTMMLib_endNoFrame

```
Public StatusInt mdlHTMMLib_endNoFrame
(
    FILE          *pHTMLFile,      /* => handle to HTML text file */
    BoolInt       bNewLine         /* => newline char ? (file formatting) */
);
```

Description mdlHTMMLib_endNoFrame writes the closing tag for the NOFRAMES portion of a frame document to the HTML text file specified by *pHTMLFile*.

Returns mdlHTMMLib_endNoFrame returns SUCCESS if the tag was written to the file successfully, ERROR otherwise.

mdlHTMMLib_endOrderedList

```
Public StatusInt mdlHTMMLib_endOrderedList
(
    FILE          *pHTMLFile,      /* => handle to HTML text file */
    BoolInt       bNewLine         /* => newline char ? (file formatting) */
);
```

Description mdlHTMMLib_endOrderedList adds the closing tag for a numbered list to the HTML text file specified by *pHTMLFile*.

Returns mdlHTMMLib_endOrderedList returns SUCCESS if the tag was written to the file successfully, ERROR otherwise.

mdlHTMMLib_endParagraph

```
Public StatusInt mdlHTMMLib_endParagraph
(
    FILE          *pHTMLFile,      /* => handle to HTML text file */
    BoolInt       bNewLine         /* => newline char ? (file formatting) */
);
```

Description mdlHTMMLib_endParagraph writes the closing paragraph tag to the HTML text file specified by *pHTMLFile*.

Returns mdlHTMMLib_endParagraph returns SUCCESS if the tag was written to the file successfully, ERROR otherwise.

mdlHTMMLib_endTable

```
Public StatusInt mdlHTMMLib_endTable
(
    FILE          *pHTMLFile,      /* => handle to HTML text file */
    int            iTableAlign,     /* => table alignment within document */
    BoolInt       bNewLine         /* => newline char ? (file formatting) */
);
```

Description mdlHTMMLib_endTable writes the closing tag for a table to the HTML text file specified by *pHTMLFile*.

iTableAlign specifies table alignment within the HTML document using the values in `htmlib.h`: ALIGN_TABLECELL_LEFTTOP, ALIGN_TABLECELL_LEFTCENTER, ALIGN_TABLECELL_LEFTBOTTOM, ALIGN_TABLECELL_CENTERTOP, ALIGN_TABLECELL_CENTERCENTER, ALIGN_TABLECELL_CENTERBOTTOM, ALIGN_TABLECELL_RIGHTTOP, ALIGN_TABLECELL_RIGHTCENTER and ALIGN_TABLECELL_RIGHTBOTTOM.

Returns mdlHTMMLib_endTable returns SUCCESS if the tag was written to the file successfully, ERROR otherwise.

mdlHTMMLib_endTableCell

```
Public StatusInt mdlHTMMLib_endTableCell
(
    FILE          *pHTMLFile,      /* => handle to HTML text file */
    BoolInt       bNewLine         /* => newline char ? (file formatting) */
);
```

Description mdlHTMMLib_endTableCell writes the closing tag for a table cell to the HTML text file specified by *pHTMLFile*.

Returns mdlHTMMLib_endTableCell returns SUCCESS if the tag was written to the file successfully, ERROR otherwise.

mdlHTMMLib_endTableRow

```
Public StatusInt mdlHTMMLib_endTableRow
(
FILE          *pHTMLFile,      /* => handle to HTML text file */
BoolInt       bNewLine        /* => newline char ? (file formatting) */
);
```

Description mdlHTMMLib_endTableRow writes the closing tag for a table row to the HTML text file specified by *pHTMLFile*.

Returns mdlHTMMLib_endTableRow returns SUCCESS if the tag was written to the file successfully, ERROR otherwise.

mdlHTMMLib_endUnorderedList

```
Public StatusInt mdlHTMMLib_endUnorderedList
(
FILE          *pHTMLFile,      /* => handle to HTML text file */
BoolInt       bNewLine        /* => newline char ? (file formatting) */
);
```

Description mdlHTMMLib_endUnorderedList adds the closing tag for a bulleted list to the HTML text file specified by *pHTMLFile*.

Returns mdlHTMMLib_endUnorderedList returns SUCCESS if the tag was written to the file successfully, ERROR otherwise.

mdlHTMMLib_hrefListFromStringList

```
Public StatusInt mdlHTMMLib_hrefListFromStringList
(
FILE          *pHTMLFile,      /* => handle to HTML text file */
int           iListType,       /* => HTMMLIST_BULLETED, etc. */
StringList    *pDataList       /* => list of data for the table cells */
);
```

Description mdlHTMMLib_hrefListFromStringList adds a list of the specified type to the HTML text file specified by *pHTMLFile*.

iListType specifies the type of list to be created. The values from `htmlLib.h` are: HTMMLIST_BULLETED, HTMMLIST_NUMBERED, HTMMLIST_DEFINITION, HTMMLIST_DIRECTORY and HTMMLIST_MENU.

pDataList is the string list which specifies the data used in the list created. For all list types except the definition list, the string list must be created with two members for each list entry. The first member in the pair is the label displayed in the list, the second is a string specifying the HREF. For a definition list, each list item is specified by three string list members. The first is used as the term, the second is the definition, and the third is the string specifying the HREF. Both the term and definition will be linked to the HREF.

Returns mdlHTMMLib_hrefListFromStringList returns SUCCESS if the list was created successfully, ERROR otherwise.

mdlHTMMLib_htmlListFromStringList

```
Public StatusInt mdlHTMMLib_htmlListFromStringList
(
    FILE          *pHTMLFile,      /* => handle to HTML text file */
    int           iListType,        /* => HTMMLIST_BULLETED, etc. */
    StringList    *pDataList        /* => list of data for the table cells */
);
```

Description mdlHTMMLib_htmlListFromStringList adds a list of the specified type to the HTML text file specified by *pHTMLFile*.

iListType specifies the type of list to be created. The values from htmllib.h are: HTMMLIST_BULLETED, HTMMLIST_NUMBERED, HTMMLIST_DEFINITION, HTMMLIST_DIRECTORY and HTMMLIST_MENU.

pDataList is the string list which specifies the data used in the list created. For all list types except the definition list, the string list must be created with one member for each list entry. For a definition list, each list item is specified by two string list members. The first member is used as the term, the second member is the definition.

Returns mdlHTMMLib_htmlListFromStringList returns SUCCESS if the list was created successfully, ERROR otherwise.

mdlHTMMLib_imageTableFromList

```
Public StatusInt mdlHTMMLib_imageTableFromList
(
    FILE          *pHTMLFile,      /* => handle of HTML file */
    int           iTableAlign,      /* => alignment of table */
    int           iBorderWidth,     /* => width of border */
    StringList    *pCellDataList,   /* => list of data for table cells */
    int           iNumCols,         /* => # of table columns */
    int           iCellAlign,       /* => alignment of data within cells */
    StringList    *pHeaderList,     /* => column headers */
);
```

```
char          *psCaption,          /* => table caption */
int           iCaptionAlign        /* => where caption
appears wrt the table */
);
```

Description mdlHTMMLib_imageTableFromList adds a table of text entries with HREF links from a string list to the HTML text file specified by *pHTMLFile*.

iTableAlign specifies the alignment of the table within the document.

iBorderWidth specifies (in pixels) the width of the border. *iBorderWidth* equals 0 specifies “no border.”

pCellDataList specifies the string list which contains the data for table cells. *pCellDataList* is a string list containing four string members for each table cell. The first member of the set is a label for the cell, the second specifies the HREF link (NULLing this member will produce a table with no links), the third is the image file name, and the fourth specifies alternate text.

iNumcols specifies the number of columns in the table.

iCellAlign determines the alignment of the data within the cells.

pHeaderList is the string list which contains the text entries used for column headers. If non-NULL, a header row is written using the members in *pHeaderList*. When NULL, no column headers.

psCaption if non-NULL, specifies the table caption.

iCaptionAlign specifies alignment of the table caption.

Returns mdlHTMMLib_imageTableFromList returns SUCCESS if the tag was written to the file successfully, ERROR otherwise.

mdlHTMMLib_openFile

```
Public FILE *mdlHTMMLib_openFile
(
char      *psFullFileName,    /* <=> file name to open */
int       iFileAccess        /* => read/write access */
);
```

Description mdlHTMMLib_openFile opens an HTML file.

tpsFullFileName specifies the HTML file to open.

iFileAccess determines read/write access using the same values of mdlTextFile_open: TEXTFILE_READ, TEXTFILE_WRITE and TEXTFILE_APPEND.

Returns mdlHTMMLib_openFile returns a pointer to the file if opened successfully, NULL otherwise.

See Also mdlTextFile_open.

mdlHTMMLib_setDocTypeToHTML

```
Public StatusInt mdlHTMMLib_setDocTypeToHTML
(
FILE          *pHTMLFile,      /* => handle to HTML text file */
BoolInt       bNewLine         /* => newline char ? (file formatting) */
);
```

Description mdlHTMMLib_setDocTypeToHTML writes the doctype comment tag that indicates the content is HTML to the HTML text file specified by *pHTMLFile*.

Returns mdlHTMMLib_addDefinitionListItem returns SUCCESS if the tag was written to the file successfully, ERROR otherwise.

mdlHTMMLib_setMetaNameGenerator

```
Public StatusInt mdlHTMMLib_setMetaNameGenerator
(
FILE          *pHTMLFile      /* => handle to HTML text file */
);
```

Description mdlHTMMLib_setMetaNameGenerator sets the generator meta name tag to “MicroStation HTML Generator” with the version number to the HTML text file specified by *pHTMLFile*.

Returns mdlHTMMLib_setMetaNameGenerator returns SUCCESS if the tag was written to the file successfully, ERROR otherwise.

mdlHTMMLib_setPageTitle

```
Public StatusInt mdlHTMMLib_setPageTitle
(
FILE          *pHTMLFile,      /* => handle to HTML text file */
char          *psTitleString,  /* => title string */
BoolInt       bNewLine         /* => newline char ? (file formatting) */
);
```

Description mdlHTMMLib_setPageTitle writes a title tag to the HTML text file specified by *pHTMLFile*.

psTitleString specifies the title string.

Returns mdlHTMMLib_setPageTitle returns SUCCESS if the tag was written to the file successfully, ERROR otherwise.

mdlHTMMLib_startBody

```
Public StatusInt mdlHTMMLib_startBody
(
    FILE          *pHTMLFile,                /* => handle to HTML
    text file */
    BoolInt       bNewLine                    /* => newline char ?
    (file formatting) */
);
```

Description mdlHTMMLib_startBody writes the opening body tag to the HTML file specified by *pHTMLFile*.

Returns mdlHTMMLib_startBody returns SUCCESS if the tag was written to the file successfully, ERROR otherwise.

mdlHTMMLib_startCenter

```
Public StatusInt mdlHTMMLib_startCenter
(
    FILE          *pHTMLFile, /* => handle to HTML text file */
    BoolInt       bNewLine    /* => newline char ? (file formatting) */
);
```

Description mdlHTMMLib_startCenter writes the opening tag to begin centering HTML content to the HTML text file specified by *pHTMLFile*.

Returns mdlHTMMLib_startCenter returns SUCCESS if the tag was written to the file successfully, ERROR otherwise.

mdlHTMMLib_startDirectoryList

```
Public StatusInt mdlHTMMLib_startDirectoryList
(
    FILE          *pHTMLFile, /* => handle to HTML text file */
    BoolInt       bNewLine    /* => newline char ? (file formatting) */
);
```

Description mdlHTMMLib_startDirectoryList writes the header tag for directory list to the HTML text file specified by *pHTMLFile*.

Returns mdlHTMMLib_startDirectoryList returns SUCCESS if the tag was written to the file successfully, ERROR otherwise.

mdlHTMMLib_startDefinitionList

```
Public StatusInt mdlHTMMLib_startDefinitionList
(
    FILE          *pHTMLFile, /* => handle to HTML text file */
    BoolInt       bNewLine    /* => newline char ? (file formatting) */
);
```

Description mdlHTMMLib_startDefinitionList writes the header tag for a definition list to the HTML text file specified by *pHTMLFile*.

Returns mdlHTMMLib_startDefinitionList returns SUCCESS if the tag was written to the file successfully, ERROR otherwise.

mdlHTMMLib_startFileHeader

```
Public StatusInt mdlHTMMLib_startFileHeader
(
    FILE          *pHTMLFile, /* => handle to HTML text file */
    BoolInt       bNewLine    /* => newline char ? (file formatting) */
);
```

Description mdlHTMMLib_startFileHeader writes the opening file header tag to the HTML text file specified by *pHTMLFile*.

Returns mdlHTMMLib_startFileHeader returns SUCCESS if the tag was written to the file successfully, ERROR otherwise.

mdlHTMMLib_startForm

```
Public StatusInt mdlHTMMLib_startForm
(
    FILE          *pHTMLFile, /* => handle to HTML text file */
    int           iMethod,    /* => form method */
    char          *psAction,   /* => action string */
    BoolInt       bNewLine    /* => newline char ? (file formatting) */
);
```

Description mdlHTMMLib_startForm writes the ending tag for a form to the HTML text file specified by *pHTMLFile*.

iMethod specifies the form method using values defined in `htmlLib.h`:
FORMMETHOD_POST or FORMMETHOD_GET.

psAction specifies the form action string.

Returns mdlHTMMLib_startForm returns SUCCESS if the tag was written to the file successfully, ERROR otherwise.

mdlHTMMLib_startHREFAnchor

```
Public StatusInt mdlHTMMLib_startHREFAnchor
(
    FILE          *pHTMLFile, /* => handle to HTML text file */
    char          *psRefFileName, /* => string (filename) for HREF */
    char          *psTarget,     /* => target frame (optional) */
    BoolInt       bNewLine      /* => newline char ? (file formatting) */
);
```

Description mdlHTMMLib_startHREFAnchor writes the opening tag for an anchor with an optional target frame to the HTML text file specified by *pHTMLFile*.

Returns mdlHTMMLib_startHREFAnchor returns SUCCESS if the tag was written to the file successfully, ERROR otherwise.

mdlHTMMLib_startHTML

```
Public StatusInt mdlHTMMLib_startHTML
(
FILE          *pHTMLFile,          /* => handle to HTML text file */
BoolInt      bNewLine              /* => newline char ? (file formatting) */
);
```

Description mdlHTMMLib_startHTML writes the opening HTML tag to the HTML text file specified by *pHTMLFile*.

Returns mdlHTMMLib_startHTML returns SUCCESS if the tag was written to the file successfully, ERROR otherwise.

mdlHTMMLib_startMap

```
Public StatusInt mdlHTMMLib_startMap
(
FILE          *pHTMLFile,          /* => handle to HTML text file */
char          *psName,             /* => map name (required) */
BoolInt      bNewLine              /* => newline char ? (file formatting) */
);
```

Description mdlHTMMLib_startMap writes the opening tag for an image map to the HTML text file specified by *pHTMLFile*.

psName specifies the image map.

Returns mdlHTMMLib_startMap returns SUCCESS if the tag was written to the file successfully, ERROR otherwise.

mdlHTMMLib_startMenuList

```
Public StatusInt mdlHTMMLib_startMenuList
(
FILE          *pHTMLFile,          /* => handle to HTML text file */
BoolInt      bNewLine              /* => newline char ? (file formatting) */
);
```

Description mdlHTMMLib_startMenuList writes a menu list header tag to the HTML text file specified by *pHTMLFile*.

Returns mdlHTMMLib_startMenuList returns SUCCESS if the tag was written to the file successfully, ERROR otherwise.

mdlHTMMLib_startNameAnchor

```
Public StatusInt mdlHTMMLib_startNameAnchor
(
FILE          *pHTMLFile,    /* => handle to HTML text file */
char          *psRefLabel,    /* => string (label name) for link */
BoolInt       bNewLine       /* => newline char ? (file formatting) */
);
```

Description mdlHTMMLib_startNameAnchor writes the opening tag for a name anchor to the HTML text file specified by *pHTMLFile*.

psRefLabel specifies the text to be used for the link.

Returns mdlHTMMLib_startNameAnchor returns SUCCESS if the tag was written to the file successfully, ERROR otherwise.

mdlHTMMLib_startNoFrame

```
Public StatusInt mdlHTMMLib_startNoFrame
(
FILE          *pHTMLFile,    /* => handle to HTML text file */
BoolInt       bNewLine       /* => newline char ? (file formatting) */
);
```

Description mdlHTMMLib_startNoFrame writes the opening tag for the NOFRAMES portion of a frame document to the HTML text file specified by *pHTMLFile*.

Returns mdlHTMMLib_startNoFrame returns SUCCESS if the tag was written to the file successfully, ERROR otherwise.

mdlHTMMLib_startOrderedList

```
Public StatusInt mdlHTMMLib_startOrderedList
(
FILE          *pHTMLFile,    /* => handle to HTML text file */
BoolInt       bNewLine       /* => newline char ? (file formatting) */
);
```

Description mdlHTMMLib_startOrderedList writes the header tag for a numbered list to the HTML text file specified by *pHTMLFile*.

Returns mdlHTMMLib_startOrderedList returns SUCCESS if the tag was written to the file successfully, ERROR otherwise.

mdlHTMMLib_startParagraph

```
Public StatusInt mdlHTMMLib_startParagraph
(
    FILE          *pHTMLFile,          /* => handle to HTML
    text file */
    BoolInt       bNewLine              /* => newline char ?
    (file formatting) */
);
```

Description mdlHTMMLib_startParagraph writes the opening paragraph tag to the HTML text file specified by *pHTMLFile*.

Returns mdlHTMMLib_startParagraph returns SUCCESS if the tag was written to the file successfully, ERROR otherwise.

mdlHTMMLib_startStandardTable

```
Public StatusInt mdlHTMMLib_startStandardTable
(
    FILE          *pHTMLFile,          /* => handle to HTML text file */
    int           iBorderWidth,        /* => border width */
    boolInt       bNewLine              /* => newline char ? (file formatting) */
);
```

Description mdlHTMMLib_startStandardTable writes the opening tag for a table with no alignment information to the HTML text file specified by *pHTMLFile*.

iBorderWidth specifies border width in pixels. The default value of *iBorderWidth* is 0, which means “no border.”

Returns mdlHTMMLib_startStandardTable returns SUCCESS if the tag was written to the file successfully, ERROR otherwise.

mdlHTMMLib_startTable

```
Public StatusInt mdlHTMMLib_startTable
(
    FILE          *pHTMLFile,          /* => handle to HTML text file */
    int           iTableAlign,         /* => alignment of table on document */
    int           iBorderWidth,        /* => border width of table */
    BoolInt       bNewLine              /* => newline char ? (file formatting) */
);
```

Description mdlHTMMLib_startTable writes the opening tag for a table to the HTML text file specified by *pHTMLFile*.

iTableAlign specifies alignment of table within the document using the values in *htmlLib.h*.

iBorderWidth specifies border width in pixels. The default value of 0 means “no border.”

Returns mdlHTMMLib_startTableCell returns SUCCESS if the tag was written to the file successfully, ERROR otherwise.

mdlHTMMLib_startTableCell

```
Public StatusInt mdlHTMMLib_startTableCell
(
    FILE          *pHTMLFile,      /* => handle to HTML text file */
    int           iCellAlign,      /* => alignment of data within cell */
    BoolInt       bWrapData,       /* => allow data w/i this cell to wrap? */
    int           iRowSpan,        /* => rows cell spans */
    int           iColSpan,        /* => columns cell spans */
    BoolInt       bNewLine         /* => newline char ? (file formatting) */
);
```

Description mdlHTMMLib_startTableCell writes the opening tag for an HTML table cell to the HTML text file specified by *pHTMLFile*.

iCellAlign specifies alignment of data within the cell using values from htmllib.h.

bWrapData specifies whether wrapping of data within cell will be permitted.

iRowSpan specifies the number of rows the cell should span.

iColSpan specifies the number of columns the cell should span.



Passing 0 for any of these arguments will leave the property out of the HTML tag.

Returns mdlHTMMLib_startTableCell returns SUCCESS if the tag was written to the file successfully, ERROR otherwise.

mdlHTMMLib_startTableRow

```
Public StatusInt mdlHTMMLib_startTableRow
(
    FILE          *pHTMLFile,      /* => handle to HTML text file */
    BoolInt       bNewLine         /* => newline char ? (file formatting) */
);
```

Description mdlHTMMLib_startTableRow adds the opening tag for a table row to the HTML text file specified by *pHTMLFile*.

Returns mdlHTMMLib_startTableRow returns SUCCESS if the tag was written to the file successfully, ERROR otherwise.

mdlHTMMLib_startTableRowAlign

```
Public StatusInt mdlHTMMLib_startTableRowAlign
(
FILE          *pHTMLFile,    /* => handle to HTML text file */
int           iAlignValue,    /* => alignment value */
BoolInt       bNewLine       /* => newline char ? (file formatting) */
);
```

Description mdlHTMMLib_startTableRowAlign writes the opening tag for a table row to the HTML text file specified by *pHTMLFile*.

iAlignValue specifies the alignment using values found in `htmlLib.h`.

Returns mdlHTMMLib_startTableRowAlign returns SUCCESS if the tag was written to the file successfully, ERROR otherwise.

mdlHTMMLib_startUnorderedList

```
Public StatusInt mdlHTMMLib_startUnorderedList
(
FILE          *pHTMLFile,    /* => handle to HTML text file */
BoolInt       bNewLine       /* => newline char ? (file formatting) */
);
```

Description mdlHTMMLib_startUnorderedList adds the header tag for a bulleted list to the HTML text file specified by *pHTMLFile*.

Returns mdlHTMMLib_startUnorderedList returns SUCCESS if the tag was written to the file successfully, ERROR otherwise.

mdlHTMMLib_textHREFTableFromList

```
Public StatusInt mdlHTMMLib_textHREFTableFromList
(
FILE          *pHTMLFile,    /* => handle to HTML text file */
int           iTableAlign,    /* => alignment of table */
int           iBorderWidth,    /* => border width, 0=no border */
StringList    *pCellDataList, /* => data list for table cells */
int           iNumCols,       /* => # of table columns */
int           iCellAlign,      /* => alignment of data within cells */
StringList    *pHeaderList,    /* => column headers */
char          *psCaption,      /* => table caption */
int           iCaptionAlign    /* => where caption appears wrt the table */
);
```

Description mdlHTMMLib_textHREFTableFromList adds a table of text entries with HREF links, created from a string list, to the HTML text file specified by *pHTMLFile*.

iTableAlign specifies the alignment of the table within the document.

iBorderWidth specifies (in pixels) the width of the border.

pCellDataList is the string list which contains the text entries used for the table cells. This string list contains two string members for each table cell. The first member of the pair is the displayed data and the second specifies the HREF link.

iNumCols specifies the number of columns in the table.

iCellAlign determines the alignment of the data within the cells.

pHeaderList is the string list which contains the text entries used for column headers. If non-NULL, a header row is written using the members in *pHeaderList*. When NULL, no column headers.

psCaption (if non-NULL) specifies the table caption. When NULL, no table caption.

iCaptionAlign specifies alignment of the table caption, *psCaption*.

Returns mdlHTMMLib_textHREFTableFromList returns SUCCESS if the table was written to the file successfully, ERROR otherwise.

mdlHTMMLib_textTableFromList

```
Public StatusInt mdlHTMMLib_textTableFromList
(
FILE          *pHTMLFile,    /* => handle to HTML text file */
int           iTableAlign,   /* => alignment of table in document */
int           iBorderWidth, /* => border width in pixels, 0=no border */
StringList    *pCellDataList, /* => list of data for table cells */
int           iNumCols,      /* => # of table columns */
int           iCellAlign,    /* => alignment of data within cells */
StringList    *pHeaderList, /* => column headers (NULL=no headers) */
char          *psCaption,    /* => table caption (NULL=no caption) */
int           iCaptionAlign /* => where caption appears wrt the table */
);
```

Description mdlHTMMLib_textTableFromList adds a table of text entries, created from a string list, to the HTML text file specified by *pHTMLFile*.

iTableAlign specifies the alignment of the table within the document using values found in `htmlLib.h`.

iBorderWidth specifies (in pixels) the width of the border.

pCellDataList specifies the string list which contains the text entries. It contains one string member for each table cell.

iNumcols specifies the number of columns in the table.

iCellAlign determines the alignment of the data within the cells.

pHeaderList specifies the string list which contains column headers. If non-NULL, a header row is written using the members in *pHeaderList*.

psCaption if non-NULL, specifies the table caption.

iCaptionAlign specifies alignment of the table caption, *psCaption*.

Returns mdlHTMMLib_textTableFromList returns SUCCESS if the table was written to the file successfully, ERROR otherwise.

mdlTag_extractURL

```
Public BootInt mdlTag_extractURL
(
    char          *pStrURL,          /* <= actual URL, pass NULL if don't care*/
    char          *pStrDesc,        /* <= actual descr, Null if not needed */
    MSElement    *pElement,        /* => Element with tag*/
    int           iFileNum          /* => File number*/
);
```

Description mdlTag_extractURL returns the URL and Description from the tagged element.

Returns FALSE if the element has no tags.

mdlTag_hasInternetTagSet

```
Public BootInt mdlTag_hasInternetTagSet
(
    TagSetSpec    *pTagSetSpec      /* <= TagSetSpec if not NULL */
);
```

Description mdlTag_hasInternetTagSet checks for the Internet Tags in the current design file.

Returns TRUE if Internet Tags exists, FALSE if not.

See Also mdlTag_createInternetTagSet.

mdlTag_createInternetTagSet

```
Public int    mdlTag_createInternetTagSet
(
    TagSetSpec *pTagSetSpec    /* <= TagSetSpec if not NULL */
);
```

Description mdlTag_createInternetTagSet creates the Internet Tags in current design file.

Returns SUCCESS if Internet Tags were created successfully.

mdlTag_attachURL

```
Public Int    mdlTag_attachURL
(
    MSElement *element          /*<=> Element to attach tags to*/
    ULong      filePos          /* => File Position of element*/
    char       *URLString,      /* <= URL string for tag, cannot be NULL*/
    char       *descString      /* <= descr for tag, Null if no descr */
);
```

```
int          iFileNum          /* => File number*/
);
```

Description mdlTag_attachURL attaches tags for URL and description of the URL to element passed in and adds them to the design file. If the correct web tag specifications (Set and Tags) do not exist in the design file, they are created.

Returns SUCCESS if tags are successfully created.

mdlWeb_getUrlToFileBegin

```
Public WebJobH mdlWeb_getURLToFileBegin
(
char      *pStrURL,           /* => URL being requested */
char      *pStrFileName      /* => Filename to save URL in */
);
```

Description mdlWeb_getUrlToFileBegin prepares and sends a request via the http, ftp, gopher or file protocol. Since URL protocols are handled asynchronously, this function returns immediately upon submitting the request, and the application must track the progress of the request manually (perhaps via a Timer function). When complete, the application must call mdlWeb_getUrlFinish to allow the DLL to clean-up data associated with the request.

Returns mdlWeb_getUrlToFileBegin returns a handle to the URL request. This handle should be used in all later functions associated with the request.

See Also mdlWeb_getUrlProgress, mdlWeb_getUrlFinish.

mdlWeb_getUrlProgress

```
Public StatusInt mdlWeb_getUrlProgress
(
WebJobH      hJob,           /* => handle to URL request */
int           *nBytesSoFar,   /* <= number of bytes received */
int           *nTotalBytes,   /* <= total number of bytes expected */
int           *nStatus        /* <= status of URL request */
);
```

Description mdlWeb_getUrlProgress checks the status of a current URL request specified by *hJob*.

If the request is currently receiving data, *nBytesSoFar* and *nTotalBytes* will contain the progress information. If one of these parameters is unavailable or unknown, its value shall be -1.

nStatus specifies the status of the current URL request. It can return one of the following values: BSIWEB_UNKNOWNSTATUS, BSIWEB_PENDING, BSIWEB_RESOLVINGHOSTNAME, BSIWEB_OPENINGCONNECTION, BSIWEB_SENDINGREQUEST, BSIWEB_RECEIVINGDATA, BSIWEB_FINISHING, BSIWEB_ABORTED, BSIWEB_DONE.

Returns mdlWeb_getUrlProgress returns SUCCESS if the URL request exists, else returns ERROR.

See Also mdlWeb_getUrlToFileBegin, mdlWeb_getUrlFinish.

mdlWeb_getUrlFinish

```
Public StatusInt mdlWeb_getUrlFinish
(
WebJobH      hJob      /* => handle to URL request */
);
```

Description mdlWeb_getUrlFinish cleans up data associated with a URL request. After this function returns, the URL request will not be accessible.

Returns mdlWeb_getUrlFinish returns SUCCESS if the URL request existed and was removed, and ERROR if the URL request handle was invalid.

See Also mdlWeb_getUrlToFileBegin, mdlWeb_getUrlProgress.

mdlWeb_stopBrowser

```
Public StatusInt mdlWeb_stopBrowser
(
WebJobH      hJob      /* => handle to URL request */
);
```

Description mdlWeb_stopBrowser stops an existing URL request and cleans up the data associated with it.

hJob specifies the URL request to halt.

Returns mdlWeb_stopBrowser returns SUCCESS if the URL request existed and was stopped, and returns ERROR if the URL request handle was invalid.

See Also mdlWeb_getUrlToFileBegin, mdlWeb_getUrlProgress.

mdlWeb_getLastModified

```
Public long mdlWeb_getLastModified
(
WebJobH      hJob      /* => handle to URL request */
);
```

Description mdlWeb_getLastModified returns the last modified date of a remote file associated with an existing URL request. This data is not always provided by remote servers for all data types, and is not available until the status of the URL request is BSIWEB_RECEIVINGDATA or later.

hJob specifies the URL request that should return the last modification date of its associated remote file.

Returns `mdlWeb_getLastModified` returns the last modified date if available. Otherwise, returns 0.

See Also `mdlWeb_getUrlProgress`.

Example Source Code

This chapter contains the source code referenced throughout the MDL Function Reference Manual.

grphtest.mc

```

/*-----+
| Copyright (1995) Bentley Systems, Inc., All rights reserved.|
| "MicroStation" is a registered trademark and "MDL" and "MicroCSL"|
| are trademarks of Bentley Systems, Inc.      |
| Limited permission is hereby granted to reproduce and modify this|
| copyrighted material provided that the resulting code is used only |
| in conjunction with Bentley Systems products under the terms of the|
| license agreement provided therein, and that this notice is retained|
|
| in its entirety in any such reproduction or modification.|
+-----*/
/*-----+
|
| Function -
|   Place a trumpet in a design file
|   -----
|
| Public Routine Summary -
|
|   elemDscr_show - Display contents of element dscr
|   main - Main entry point
|   multi - MULTI command entry point
|   trumpet - TRUMPET command entry point
|   trumpet_appendElbowPipe - Append an elbow pipe to elem dscr
|   trumpet_appendReducer - Append a reducer to elem dscr
|   trumpet_appendStraightPipe - Append straight pipe to elem dscr
|
|   trumpet_createTrumpet - Create the trumpet
|   trumpet_placeMultiPoint - Define point for multi command plcmnt
|
|   trumpet_placeTrumpetPoint - Def point for trumpet command
|
plcmnt|

```

```

|         trumpet_setPoint - Set a point                                |
|         trumpet_startTrumpet - Start trumpet processing                |
+-----*/
#include    <mselems.h>
#include    <global.h>
#include    <mdl.h>
#include    <rsdefs.h>
#include    <math.h>
#include    <string.h>

#include    "trumpet.h"

#include    <dlogman.fdf>
#include    <mcurrtr.fdf>
#include    <mssystem.fdf>
#include    <msrsrc.fdf>
#include    <mstmatrix.fdf>
#include    <mselmdsc.fdf>
#include    <msoutput.fdf>
#include    <msstate.fdf>
#include    <mscell.fdf>
#include    <mselemen.fdf>

/*-----+
|   Private Global variables                                           |
+-----*/
MSElementDescr  *trumpetDP;

/*-----+
|   Private utility routines                                           |
+-----*/
/*-----+
| name          trumpet_setPoint                                       |
| author        BSI                                                    |
+-----*/
Private void trumpet_setPoint(Dpoint3d *pt, double x,
double y, double z)
{
    pt->x = x;
    pt->y = y;
    pt->z = z;
}

#if defined (DEBUG)
/*-----+
| name          elemDscr_show                                           |
| author        BSI                                                    |
|               5/90                                                    |
+-----*/
Public int elemDscr_show(MSElementDescr *elemDescr,
char *currentIndent)

```

```

    {
    char indent[128];
    int color, weight, style, level, ggNum, class, locked, new;
    int modified, viewIndepend, solidHole;

    if (elemDescr == NULL)
        return SUCCESS;

    strcpy(indent, currentIndent);
    strcat(indent, "|   ");
    do
    {
        mdlElement_getSymbology(&color, &weight, &style,
                                &elemDescr->el);
        mdlElement_getProperties (&level, &ggNum, &class, &locked, &new,
                                &modified, &viewIndepend, &solidHole, &elemDescr->el);
        printf ("%shdr=%d, typ=%d, cmplx=%d", currentIndent,
                elemDescr->h.isHeader, elemDescr->el.hdr.ehdr.type,
                elemDescr->el.hdr.ehdr.complex);
        printf ("c=%d,w=%d,s=%d,l=%d,gg=%d,cl=%d\n",
                color, weight, style, level, ggNum, class);

        if (elemDescr->h.isHeader)
        {
            elemDscr_show (elemDescr->h.firstElem, indent);
            printf ("%send of chain\n", currentIndent);
        }

        elemDescr = elemDescr->h.next;
    } while (elemDescr);
    }
#endif

/*-----+
| name      trumpet_startTrumpet      |
| author    BSI                        7/90      |
+-----*/
Private MSElementDescr *trumpet_startTrumpet(void)
{
    MSElementUnion trumpetHeader;

    mdlCell_create (&trumpetHeader, "trumpt", NULL, 0);
    mdlElmdscr_new (&trumpetDP, NULL, &trumpetHeader);

    return trumpetDP;
}

#ifdef CYLINDERS
/*-----+
| If the constant "CYLINDERS" is defined, we create the trumpet using |
| cones and cylinders, otherwise we use surfaces of projection and    |

```

```

| rotation. This is merely to illustrate various techniques that can |
| be used with element descriptors. |
+-----*/
/*-----+
| name trumpet_appendReducer - add a "reducer" to element descriptor. |
| author      BSI | 9/89 |
+-----*/
Private void trumpet_appendReducer(MSElementDescr *trumpetDP,
double rad1, double rad2, double len)
{
    Dpoint3d pt[3];
    MSElementUnion cone;

    memset(pt, 0, sizeof(pt));
    pt[1].z += len;

    mdlCone_create(&cone, NULL, rad2, rad1, &pt[0], &pt[1], NULL);
    mdlElmdscr_appendElement(trumpetDP, &cone);

    mdlCurrTrans_translateOrigin(&pt[1]);
}

/*-----+
| name      trumpet_appendStraightPipe |
| author    BSI | 9/89 |
+-----*/
Private void trumpet_appendStraightPipe(MSElementDescr *trumpetDP,
double rad, double len)
{
    Dpoint3d pt[2];
    MSElementUnion cylinder;

    memset(pt, 0, sizeof(pt));
    pt[1].z = len;

    mdlCone_createRightCylinder(&cylinder, NULL, rad, &pt[0], &pt[1]);
    mdlElmdscr_appendElement (trumpetDP, &cylinder);

    mdlCurrTrans_translateOrigin (&pt[1]);
}
#else
/*-----+
| name      trumpet_appendReducer |
| author    BSI | 9/89 |
+-----*/
Private void trumpet_appendReducer(MSElementDescr *trumpetDP,
double rad1, double rad2, double len)
{
    double      scale;
    Dpoint3d     pt[3];

```



```

MSElementDescr *ellipseDP, *pipeDP;
MSElementUnion el;
Transform        tMatrix;

mdlEllipse_create (&el, NULL, NULL, rad1, rad1, NULL, 0);
mdlElmdscr_new (&ellipseDP, NULL, &el);

memset (pt, 0, sizeof(pt));
pt[1].z += len;

scale = rad2 / rad1;
mdlTMatrix_getIdentity (&tMatrix);
mdlTMatrix_scale (&tMatrix, &tMatrix, scale, scale, scale);

mdlSurface_project (&pipeDP, ellipseDP, &pt[0], &pt[1], &tMatrix);
mdlElmdscr_freeAll (&ellipseDP);

mdlElmdscr_appendDscr (trumpetDP, pipeDP);
mdlCurrTrans_translateOrigin (&pt[1]);
}

/*-----+
| name      trumpet_appendStraightPipe      |
| author    BSI                             9/89      |
+-----*/
Private void trumpet_appendStraightPipe(MSElementDescr *trumpetDP,
double rad, double len)
{
    Dpoint3d pt[2];
    MSElementDescr *ellipseDP, *pipeDP;
    MSElementUnion el;

    mdlEllipse_create (&el, NULL, NULL, rad, rad, NULL, 0);
    mdlElmdscr_new (&ellipseDP, NULL, &el);

    memset (pt, 0, sizeof(pt));
    pt[1].z = len;

    mdlSurface_project (&pipeDP, ellipseDP, &pt[0], &pt[1], NULL);
    mdlElmdscr_freeAll (&ellipseDP);

    mdlElmdscr_appendDscr (trumpetDP, pipeDP);
    mdlCurrTrans_translateOrigin (&pt[1]);
}
#endif

/*-----+
| name      trumpet_appendElbowPipe          |
| author    BSI                             9/89      |
+-----*/
Private void trumpet_appendElbowPipe(MSElementDescr *trumpetDP,
double radius, double length, double bendRadius, double angle)
{

```

```

Dpoint3d pt;
MSElementUnion el;
MSElementDescr *ellipseDP, *pipeDP;
double radians;

radians = angle * fc_piover180;
trumpet_appendStraightPipe(trumpetDP, radius, length);

mdlEllipse_create (&el, NULL, NULL, radius, radius, NULL, -1);
mdlElmdscr_new (&ellipseDP, NULL, &el);

memset (&pt, 0, sizeof(pt));
pt.y = -1. * bendRadius;
mdlCurrTrans_translateOrigin (&pt);

mdlSurface_revolve (&pipeDP, ellipseDP, NULL, 0, radians);
mdlElmdscr_freeAll (&ellipseDP);
mdlElmdscr_appendDscr (trumpetDP, pipeDP);

pt.x = sin(radians) * bendRadius;
pt.y = cos(radians) * bendRadius;
mdlCurrTrans_translateOrigin (&pt);
mdlCurrTrans_rotateByAngles (radians, fc_zero, fc_zero);

trumpet_appendStraightPipe (trumpetDP, radius, length);
}

/*-----+
| name      trumpet_createTrumpet      |
| description creates an element descriptor with the elements that |
|           define a "trumpet" at current MDL origin and |
|           orientation |
| author      BSI                      9/89 |
+-----*/
Public int trumpet_createTrumpet(void)
{
    Dpoint3d    pt;
    double      length, bendRadius, radius, radius2;
    int         i;

    /* push the current transform */
    mdlCurrTrans_begin();

    /* create the element descriptor with only a cell header */
    trumpet_startTrumpet ();

    mdlCurrTrans_rotateByAngles (fc_zero, fc_piover2, fc_zero);

    /* MouthPeice */
    length = .750;
    radius = .25;
    trumpet_appendReducer(trumpetDP, radius * 2., radius, length);

    /* First Straight Section */

```

```

    trumpet_appendStraightPipe (trumpetDP, radius, 14.0);

/* Far Elbow */
    length = fc_onehalf;
    bendRadius = 1.500;
    trumpet_appendElbowPipe (trumpetDP, radius, length, bendRadius,
                             fc_180);

/* Second Straight Section */
    length = fc_10;
    trumpet_appendStraightPipe (trumpetDP, radius, length);

    mdlCurrTrans_rotateByAngles(fc_zero, fc_zero, 12.0 * fc_piover180);
/* Near Elbow */
    length = .500;
    trumpet_appendElbowPipe (trumpetDP, radius, length, bendRadius,
                             fc_180);

/* Third Straight Section (out to horn) */
    trumpet_appendStraightPipe (trumpetDP, radius, 12.0);

/* Horn */
    length = fc_2;
    trumpet_appendReducer (trumpetDP, radius, fc_1, length);
    trumpet_appendReducer (trumpetDP, fc_1, 2.0, length);

    trumpet_setPoint (&pt, fc_zero, -4., -12.);
    mdlCurrTrans_translateOrigin (&pt);

    mdlCurrTrans_rotateByAngles (-fc_piover2, fc_zero, fc_zero);
    for (i=0; i<3; i++)
    {
        length = fc_5;
        trumpet_appendStraightPipe (trumpetDP, radius, length);
        radius2 = .175;
        length = .250;
        trumpet_appendStraightPipe (trumpetDP, radius2, length);
        radius2 = .375;
        trumpet_appendStraightPipe (trumpetDP, radius2, length);
        trumpet_setPoint (&pt, fc_zero, fc_m1, -5.5);
        mdlCurrTrans_translateOrigin (&pt);
    }

/* set the range diagonal for the trumpet cell */
    mdlCell_setRange (trumpetDP);

/* restore original transformation */
    mdlCurrTrans_end();

    return SUCCESS;
}

/*-----+

```

```

| name          trumpet_placeMultiPoint
| author        BSI
|                                     2/90
+-----*/
Private void trumpet_placeMultiPoint(Dpoint3d *origin,int view)
{
    Transform tMatrix;
    int i;

    /* Clear any outstanding abort request */
    mdlSystem_abortRequested ();

    /* set up the current transform to be oriented about data point
    in the view the user selected */
    mdlCurrTrans_begin();
    mdlCurrTrans_masterUnitsIdentity (FALSE);
    mdlCurrTrans_rotateByView (view);
    mdlCurrTrans_translateOriginWorld (origin);

    /* create a trumpet element descriptor */
    trumpet_createTrumpet();

    /* set up a transformation matrix to rotate trumpet by 45 degrees */
    mdlTMatrix_getIdentity (&tMatrix);
    mdlTMatrix_rotateByAngles (&tMatrix, &tMatrix, fc_zero, fc_zero,
                               fc_piover4);

    /* make 8 copies, rotating between them */
    for (i=0; i<8; i++)
    {
        /* display and add this trumpet */
        mdlElmdscr_display (trumpetDP, 0, NORMALDRAW);
        mdlElmdscr_add (trumpetDP);

        if (mdlSystem_abortRequested ())
            break;

        /* rotate by 45 degrees */
        mdlElmdscr_transform (trumpetDP, &tMatrix);
    }

    /* free trumpet Element Descriptor (allocated by
    trumpet_createTrumpet) */
    mdlElmdscr_freeAll (&trumpetDP);
    mdlCurrTrans_end();
}

/*-----*/
| name          multi    (command)
| description    Place 8 trumpets in a circular pattern
| author        BSI
|                                     9/89
+-----*/
cmdName void multi(void)

```

```

{
  if (!mgds_modes.three_d)
  {
    mdlOutput_rscPrintf (MSG_ERROR, NULL, MESSAGELISTID_Messages,
                        ERRID_3D);
    return;
  }

  mdlState_startPrimitive (trumpet_placeMultiPoint, multi, 0, 0);
  mdlOutput_rscPrintf (MSG_MESSAGE, NULL, MESSAGELISTID_Messages,
                      MSGID_GroupOrigin);
  mdlCurrTrans_identity();
}

/*-----+
| name      trumpet_placeTrumpetPoint      |
| author    BSI                            2/90      |
+-----*/
Private void trumpet_placeTrumpetPoint(Dpoint3d *origin, int view)
{
  int status;
  Dpoint3d worldOrigin;

  /* set up the current transform to be oriented about data point
  in the view the user selected */
  mdlCurrTrans_begin();

  mdlCurrTrans_masterUnitsIdentity (FALSE);
  status = mdlCurrTrans_rotateByView (view);
  mdlCurrTrans_translateOriginWorld (origin);

  /* create trumpet element descriptor */
  trumpet_createTrumpet();

  /* display and add the trumpet to the file */
  mdlElmdscr_display (trumpetDP, 0, NORMALDRAW);
  mdlElmdscr_add (trumpetDP);

  /* free trumpet Element Descriptor (allocated by
  trumpet_createTrumpet) */
  mdlElmdscr_freeAll (&trumpetDP);
  mdlCurrTrans_end();
}

/*-----+
| name      trumpet (command)              |
| description Place single trumpet at location specified by user |
|           continues to place trumpets with additional datapoints |
| author    BSI                            9/89      |
+-----*/
cmdName void trumpet(void)

```

```

{
    if (!mgds_modes.three_d)
    {
        mdlOutput_rscPrintf (MSG_ERROR, NULL, MESSAGELISTID_Messages,
                             ERRID_3D);
        return;
    }

    mdlState_startPrimitive (trumpet_placeTrumpetPoint, trumpet, 0, 0);
    mdlOutput_rscPrintf (MSG_MESSAGE, NULL, MESSAGELISTID_Messages,
                         MSGID_TrumpetOrigin);
    mdlCurrTrans_identity();
}

/*-----+
| name      main                                     |
| description Perform initialization.                  |
| author    BSI                                     05/90 |
+-----*/
void main(void)
{
    RscFileHandle   rscFileH;

    mdlResource_openFile(&rscFileH, NULL, FALSE);

    /* Prevent an asynchronous abort that would leave partial
       trumpets in the design file. */
    mdlSystem_userAbortEnable (0);

    /* set up a current transformation in world coordinates */
    mdlCurrTrans_begin();
}

```

rasticon.mc

```

/*-----+
-+
|                                     |
| Copyright (1993) Bentley Systems, Inc., All rights reserved.|
|                                     |
| "MicroStation", "MDL", and "MicroCSL" are trademarks of Bentley|
| Systems, Inc. and/or Intergraph Corporation.|
|                                     |
| Limited permission is hereby granted to reproduce and modify this|
| copyrighted material provided that the resulting code is used only in
|
| conjunction with Bentley Systems products under the terms of the|
| license agreement provided therein, and that this notice is retained|
| in its entirety in any such reproduction or modification.|

```

```

|
+-----+
*/
/*-----+
|
| $Logfile: J:/mdl/examples/rasticon/rasticon.mcv $
| $Workfile: rasticon.mc $
| $Revision: 5.4 $
| $Date: 21 Jul 1993 09:10:08 $
|
+-----+
*/
/*-----+
|
| Function -
|
| Rasticon - MDL Raster Icon Editor|
|
| - - - - -
|
| Public Routine Summary -|
|
|
| main - main entry point
| rastIcon_allocateBitMap - Allocate bitmaps for icons|
| rastIcon_append - Append definition to a file|
| rastIcon_calculatePositions - Calculate cursor position in icon|
| rastIcon_clear - Clear an icon map|
| rastIcon_close - No-op
| rastIcon_compressIconName - Compress an icon name|
| rastIcon_create - Open a resource file|
| rastIcon_dialogHook - Dialog box hook|
| rastIcon_dialogOpenIconResourceHook - Open resource file db hook|
| rastIcon_dialogUserSizeHook - User size icon db hook|
| rastIcon_displayErrorMessage - Display an error message|
| rastIcon_drawSmallBitMap - Draw the small icon bit map|
| rastIcon_exit - Exit Rasticon|
| rastIcon_findHighestResource - Find the highest icon resource # |
| rastIcon_fitToIcon - Fit the window to the icon size|
| rastIcon_fmtMsgGet - Get a message from the resource file|
| rastIcon_freeBitMap - Free the icon bitmap|
| rastIcon_initDynamics - Initialize drawing dynamics|
| rastIcon_largeIconButtonHandler - Handle large icon button hdlr|
| rastIcon_largeIconDraw - Draw the large icon|
| rastIcon_largeIconHook - Large icon generic item hook|
| rastIcon_listHook - List box hook|

```

```

| rastIcon_openIcon - Open a icon for editing|
| rastIcon_pixelOp - Perform a pixel operation|
| rastIcon_pixelValue - Determine a pixel value|
| rastIcon_readIconResourceFile - Read the icon resource file|
| rastIcon_refreshPixel - Refresh the screen icons|
| rastIcon_reportPixelPosition - Report pixel position|
| rastIcon_saveAs - Save icon to a new file|
| rastIcon_setInRscId - Get a rsc id|
| rastIcon_setPixelExtent - Set extent of an item|
| rastIcon_setPositions - Set item position|
| rastIcon_shiftDown - Shift icon down|
| rastIcon_shiftLeft -Shift icon left|
| rastIcon_shiftRight - Shift icon right|
| rastIcon_shiftUp - shift icon up|
| rastIcon_showRscList - Show rsc list|
| rastIcon_sizeOptionHook - Size option button hook|
| rastIcon_smallIconDraw - Small icon draw function|
| rastIcon_smallIconHook - Small icon generic hook|
| rastIcon_stopDynamics - Stop drawing dynamics|
| rastIcon_toolBrush - Brush tool init|
| rastIcon_toolBrushPixelOp - Brush tool operations|
| rastIcon_toolFill - Fill tool init|
| rastIcon_toolFillDynamics - Fill dynamics|
| rastIcon_toolFillEndPoint - Fill end point processor|
| rastIcon_toolLine - Line tool init|
| rastIcon_toolLineDynamics - Line dynamics|
| rastIcon_toolLineEndPoint - Line end point processor|
| rastIcon_toolLineFirstPoint - Line 1st point processor|
| rastIcon_updateDynamics - General update|
| rastIcon_whichPixel - Determine if in a pixel|
| rastIcon_writeIconFile - Write to icon file|
|
+-----
*/
/* This program edits icon resources.  You can either start from
scratch,
    or read in icon menus from existing compiled resource files - either
    .rsc files, or .ma files that have had resources merged into them.
    If you want to want to test the ability to read in icons, you may want
    to load the icons from \ustn40\mdlapps\dlogdemo.ma (see instructions
in
    \ustn40\mdl\examples\dlogdemo\dlogread.me to compile this
application.)

```

Note that there are two different standard icon resource types -
small

and large icons. Small icons are 23 by 23 pixels, and large icons are 31 by 31 pixels. In addition to these standard sizes, this application can edit icons on any reasonable dimensions. There is an option button which allows you to choose one of three settings - the two standard sizes, and a user specified size. In each setting, a icon read operation initiated by the file/open pulldown will attempt to read icon resources with an appropriate type. However, you may want to change the third member of the <iconResourceTypes> array if you want to read user-customized resources.

While rasticon reads compiled resource files, it writes .r files so that the resulting data can be easily included in resource source files

```
for MDL applications. */
/*-----
-+
|                                     |
|   Include Files                     |
|                                     |
+-----
*/
#include <dlogbox.h>
#include <dlogitem.h>
#include <keys.h>
#include <cexpr.h>
#include <mdl.h>
#include <mdlio.h>
#include <stdarg.h>
#include <dlogman.fdf>
#include "rasticon.h"
#include "riconcmd.h"

/*-----
-+
|                                     |
|   Local defines                     |
|                                     |
+-----
*/
#define APPEND_INDEX    -1  /* Used during mdlStringList_InsertMember */
```

```

#define SEARCH_FROM_BEGINNING -1 /* Used during
mdlDialog_getNextSelection. */

/*-----
--+
|                                     |
|   Local type definitions   |
|                                     |
+-----
*/
typedef struct infofields
{
    ULONG    reservedForListManager;
    ULONG    id; /* resourceclass or resourceID */
} InfoFields;

/*-----
--+
|                                     |
|   Private Global variables|
|                                     |
+-----
*/
RastIconGlobals *rastIconP;
RscFileHandle   rscHandleRastIcon;

/*-----
--+
|                                     |
|   Local function declarations |
|                                     |
+-----
*/
void rastIcon_fmtMsgGet();

voidrastIcon_largeIconHook(), rastIcon_smallIconHook(),
    rastIcon_sizeOptionHook(), rastIcon_dialogHook(),
    rastIcon_dialogOpenIconResourceHook(),
    rastIcon_dialogUserSizeHook(), rastIcon_listHook();

voidrastIcon_largeIconDraw(), rastIcon_smallIconDraw(),
    rastIcon_drawSmallBitMap();

voidrastIcon_calculatePositions(), rastIcon_setPositions();
voidrastIcon_largeIconButtonHandler(), rastIcon_reportPixelPosition();
voidrastIcon_initDynamics(), rastIcon_stopDynamics();

```

```

voidrastIcon_updateDynamics();

voidrastIcon_setInRscID();

voidrastIcon_compressIconName();

intrastIcon_toolLineFirstPoint(), rastIcon_toolLineEndPoint(),
    rastIcon_toolLineDynamics();

intrastIcon_toolBrushPixelOp();

intrastIcon_toolFillEndPoint(), rastIcon_toolFillDynamics();

voidrastIcon_displayErrorMessage (int, ...);


inticonWidths[] = {23, 31, 0};
inticonHeights[] = {23, 31, 0};

/* This array contains the three resource classes that the File/Open
pull down
    will recognize. The first two map to small and large command icons.
The
    third will undoubtedly need to be changed to a resource type selected
    by someone who wants to read user-specified icon resources. If these
    resource types are changed, the format message strings in rasticon.r
    should be changed also. */

int    iconResourceTypes[] = {RTYPE_IconCmdSmallIcon,
RTYPE_IconCmdLargeIcon,
                                RTYPE_IconCmd};

#definePREDEFINED_SIZES 2

Private DialogHookInfo uHooks[] =
{
    {HOOKITEMID_Generic_IconLarge,rastIcon_largeIconHook},
    {HOOKITEMID_Generic_IconSmall,rastIcon_smallIconHook},
    {HOOKITEMID_Option_Size,rastIcon_sizeOptionHook},
    {HOOKITEMID_List_RscNum,rastIcon_listHook},
    {HOOKDIALOGID_Icon,rastIcon_dialogHook},
    {HOOKDIALOGID_OpenIconResource,
rastIcon_dialogOpenIconResourceHook},
    {HOOKDIALOGID_UserSize,rastIcon_dialogUserSizeHook}
};

```

```

/*-----
-+
|                                     |
|   Public Global variables |       |
|                                     |
+-----
*/
/*-----
-+
|                                     |
|   External variables      |       |
|                                     |
+-----
*/
/*ff Major Public Code Section */
/*-----
-+
|                                     |
|   Major Public Code Section|     |
|                                     |
+-----
*/
/*-----
-+
|                                     |
| name main                      |   |
|                                     |   |
| authorBSI      8/89      |     |
|                                     |
+-----
*/
main
(
int  argc,
char *argv[]
)
{
char      *setP;
int       rscFileH;
DialogBox *dbP;

/* open the resource file that we came out of */
mdlResource_openFile (&rscHandleRastIcon, NULL, 0);

/* load the command table */
if (mdlParse_loadCommandTable (NULL) == NULL)
rastIcon_displayErrorMessage (0);

```

```

    /* set up the variables we are going to set from dialog boxes */
    setP = mdlCEXpression_initializeSet (VISIBILITY_DIALOG_BOX, 0,
FALSE);
    mdlDialog_publishComplexPtr (setP, "rastIconGlobals",
                                "rastIconP", &rastIconP);

    /* publish our hooks */
    mdlDialog_hookPublish (sizeof(uHooks)/sizeof(DialogHookInfo),
uHooks);
    /* start the dialog box */
    if ( (dbP = (DialogBox *) mdlDialog_open (NULL, DIALOGID_Icon)) ==
NULL)
    rastIcon_displayErrorMessage (1);

    /* set up our initial upFunction and downFunction */
    rastIcon_toolLine();
}

/*-----
-+
|
|           Dialog Box Hooks
|
+-----
*/
/*-----
-+
|
| name rastIcon_dialogHook      |
| authorBSI          7/90      |
|
+-----
*/
Private voidrastIcon_dialogHook
(
DialogMessage*dmP
)
{
    Rectangle newOverall;
    Sextent  smallIcon,  largeIcon, positionLabel;
    Sextent  toolPopup, modePopup, sizePopup, nameText;

    dmP->msgUnderstood = TRUE;

    switch (dmP->messageType)

```

```

{
case DIALOG_MESSAGE_CREATE:
    dmP->u.create.interests.updates    = TRUE;
    dmP->u.create.interests.mouses     = TRUE;
    dmP->u.create.interests.resizes     = TRUE;
    rastIconP = malloc (sizeof(RastIconGlobals));
    if (!(dmP->u.create.createFailed = (rastIconP == NULL)))
    {
        memset (rastIconP, 0, sizeof(RastIconGlobals));
        rastIconP->dialogBoxP = dmP->db;
        rastIconP->onByteValue = BLACK_INDEX;
        rastIconP->offByteValue = LGREY_INDEX;

        rastIconP->onColorDescr =
            mdlWindow_fixedColorIndexGet (dmP->db, rastIconP-
>onByteValue);
        rastIconP->offColorDescr =
            mdlWindow_fixedColorIndexGet (dmP->db, rastIconP-
>offByteValue);

        rastIconP->lineColor =
            mdlWindow_fixedColorIndexGet (dmP->db,
                                           DGREY_INDEX);
        rastIconP->dynamicsColor =
            mdlWindow_fixedColorIndexGet (dmP->db,
                                           WHITE_INDEX);
        rastIconP->appendToFile = TRUE;
        rastIconP->inRscId = 0;
    }
    break;

case DIALOG_MESSAGE_INIT:
    /* set up the bitmap after the items have been created */
    rastIconP->width      = iconWidths[rastIconP->sizeIndex];
    rastIconP->height     = iconHeights[rastIconP->sizeIndex];
    if ((dmP->u.init.initFailed
        = rastIcon_allocateBitMap ()) == 0)
    {
        /* calculate the positions of all the items */
        rastIcon_calculatePositions (&newOverall, NULL, NULL, NULL,
                                     NULL, NULL, NULL, NULL, dmP->db);
        mdlWindow_resize (dmP->db, CORNER_LOWERRIGHT,
                           &newOverall.corner);
    }
    break;

case DIALOG_MESSAGE_RESIZE:
    /* we don't want to do this on an overall move */

```

```

        if (dmP->u.resize.whichCorners != CORNER_ALL)
        {
            rastIcon_calculatePositions (NULL, &smallIcon, &largeIcon,
                                         &positionLabel, &toolPopup, &modePopup,
                                         &sizePopup, &nameText, dmP->db);
            rastIcon_setPositions (dmP->db, &smallIcon, &largeIcon,
                                  &positionLabel, &toolPopup, &modePopup,
                                  &sizePopup, &nameText);
        }
        break;

    default:
        dmP->msgUnderstood = FALSE;
        break;
    }
}

/*-----
-+
| name rastIcon_dialogOpenIconResourceHook |
| authorBSI          7/90          |
|-----
*/
Private voidrastIcon_dialogOpenIconResourceHook
(
    DialogMessage*dmP
)
{
    dmP->msgUnderstood = TRUE;
    switch (dmP->messageType)
    {
    case DIALOG_MESSAGE_ACTIONBUTTON:
        if (dmP->u.actionButton.actionType == ACTIONBUTTON_OK)
        {
            rastIcon_setInRscId();
            dmP->u.actionButton.abortAction =
                rastIcon_readIconResourceFile();
        }
        break;

    default:
        dmP->msgUnderstood = FALSE;
        break;
    }
}

```

```

    }

/*-----
--+
| name rastIcon_dialogUserSizeHook      |
| authorBSI          7/90                |
|-----
*/
Private voidrastIcon_dialogUserSizeHook
(
DialogMessage*dmP
)
{
    dmP->msgUnderstood = TRUE;

    switch (dmP->messageType)
    {
    case DIALOG_MESSAGE_ACTIONBUTTON:
        if (dmP->u.actionButton.actionType == ACTIONBUTTON_OK)
            rastIcon_fitToIcon (rastIconP->dialogBoxP, rastIconP->width,
                                rastIconP->height);

        break;

    default:
        dmP->msgUnderstood = FALSE;
        break;
    }
}

/*-----
--+
| name rastIcon_sizeOptionHook          |
| authorBSI          7/90                |
|-----
*/
Public voidrastIcon_sizeOptionHook
(
DialogItemMessage *dimP
)
{
    Rectangle newOverall;

```



```

    dimP->msgUnderstood = TRUE;

    switch (dimP->messageType)
    {
    case DITEM_MESSAGE_STATECHANGED:
        /* Check for user-defined size */
        if (rastIconP->sizeIndex == (PREDEFINED_SIZES ))
        {
            /* do resize from the other dialog box */
            mdlDialog_open (NULL, DIALOGID_UserSize);
        }

        else if (dimP->u.stateChanged.reallyChanged)
        {
            rastIcon_fitToIcon (dimP->db,
                                iconWidths[rastIconP->sizeIndex],
                                iconHeights[rastIconP->sizeIndex]);
        }
        break;

    default:
        dimP->msgUnderstood = FALSE;
        break;
    }
}

/*-----
-+
| name rastIcon_largeIconHook      |
| authorBSI          7/90          |
|-----
*/
Public voidrastIcon_largeIconHook
(
DialogItemMessage  *dimP
)
{
    DialogItem *diP = dimP->dialogItemP;

    dimP->msgUnderstood = TRUE;

    switch (dimP->messageType)
    {

```

```

    case DITEM_MESSAGE_CREATE:
        diP->attributes.acceptsKeystrokes = FALSE;
        diP->attributes.mouseSensitive    = TRUE;
        break;

    case DITEM_MESSAGE_DRAW:
        /* draw large icon */
        rastIcon_largeIconDraw (dimP);
        break;

    case DITEM_MESSAGE_BUTTON:
        /* handle button according to current tool/mode */
        rastIcon_largeIconButtonHandler(dimP);
        break;

    default:
        dimP->msgUnderstood = FALSE;
        break;
    }
}

/*-----
-+
|
| name rastIcon_smallIconHook      |
|
| authorBSI          7/90          |
|
|-----
*/
Public void rastIcon_smallIconHook
(
DialogItemMessage *dimP
)
{
    DialogItem *diP = dimP->dialogItemP;

    dimP->msgUnderstood = TRUE;

    switch (dimP->messageType)
    {
    case DITEM_MESSAGE_CREATE:
        diP->attributes.acceptsKeystrokes = FALSE;
        diP->attributes.mouseSensitive    = FALSE;
        break;

        /* large icon sends us a user message that means "draw" */

```

```

    case DITEM_MESSAGE_DRAW:
        /* draw small icon */
        rastIcon_smallIconDraw (dimP);
        break;

    default:
        dimP->msgUnderstood = FALSE;
        break;
    }
}

/*-----
-+
|
|      Command Handling routines
|
+-----
*/
/*-----
-+
|
|      name          rastIcon_create
|
|      author        BSI
|
|
|      7/90
|
+-----
*/
cmdName rastIcon_create
(
)
cmdNumberCMD_MENUICON_OPENFILE
{
    DialogItem      *diP;
    Ditem_PulldownMenuItem  miP;
    Ditem_PulldownMenu      *mBarP;

    mdlDialog_fileOpen (rastIconP->inFileName, NULL, 0, NULL, "*.ma",
"MS_MDL",
"Resource File to Open");

/* Open file specified to make sure it is in fact a resource file */
    if (mdlResource_openFile (&rastIconP->rscHandleFileToRead,
        rastIconP->inFileName, 0))
    {
        rastIcon_displayErrorMessage (3, rastIconP->inFileName);
        return (ERROR);
    }
}

```

```

    /* Resource file must be closed at this stage, so that the Dialog
Box
    Manager does not confuse dialog items in this file from the
RastIcon
    resource files. Since standard items are used in RastIcon dialog
boxes, NULL must be specified in mdlDialogOpen statements so that
they
    are found. */

    mdlResource_closeFile (rastIconP->rsCHandleFileToRead);
    /* Now that a resource file has been successfully loaded, we can
enable
    the "Open Icon" command */
    diP = mdlDialog_itemGetByTypeAndId (rastIconP->dialogBoxP,
RTYPE_MenuBar,
                                MENUBARID_Icon, 0);

    mBarP = mdlDialog_menuBarFindMenu (diP->rawItemP, NULL,
                                PULLDOWNMENUID_IconFile);
    mdlDialog_menuBarItem (&miP, &mBarP, NULL, 0, 0, 1);
    mdlDialog_textPDMItemSetEnabled (&miP, TRUE);

    rastIcon_openIcon();
}

/*-----
-+
| name          rastIcon_openIcon
|
| author        BSI
|
| 7/90
|-----
*/
cmdName rastIcon_openIcon
(
)
cmdNumberCMD_MENUICON_OPENICON
{

/* Open dialog box which displays options for resource to load */
    mdlDialog_open (NULL, DIALOGID_OpenIconResource);
}

/*-----
-+
|
|

```

```

| name          rastIcon_close|
|
| author        BSI              7/90
|
+-----+
*/
cmdName rastIcon_close
(
)
cmdNumberCMD_MENUICON_CLOSE
{
    printf ("In rasticon close\n");
}

/*-----+
|
| name          rastIcon_append
|
| author        BSI              7/90
|
+-----+
*/
cmdName rastIcon_append
(
)
cmdNumberCMD_MENUICON_APPEND
{

    mdlDialog_fileOpen (rastIconP->outFileName, NULL, 0, NULL,"*.r",
        "MS_DBGSOURCE", /* Default to debugger source directory */
        "Append Resource Source to File:");

    rastIconP->appendToFile = TRUE;
    rastIcon_writeIconFile();
}

/*-----+
|
| name          rastIcon_saveAs
|
| author        BSI              7/90
|
+-----+
*/
cmdName rastIcon_saveAs

```

```

(
)
cmdNumberCMD_MENUICON_SAVEAS
{
    mdlDialog_fileCreate (rastIconP->outFileName, NULL, 0, NULL, "*.r",
        "MS_DBGSOURCE", /* Default to debugger source directory */
        "Create New File Named:");
    rastIconP->appendToFile = FALSE;
    rastIcon_writeIconFile();
}

/*-----
-+
| name          rastIcon_quit
|
| author        BSI
|
| 7/90
|
+-----
*/
cmdName rastIcon_exit
(
)
cmdNumberCMD_MENUICON_QUIT
{
    /* -----
        If the user has modified the bitmap, we should put out an alert
        box and warn him.
        -----
    */
    mdlStringList_destroy (rastIconP->stringListP);
    free (rastIconP->dataP);
    free (rastIconP);
    /* The above lines are not really necessary - see documentation for
        mdlSystem_exit */
    mdlSystem_exit(NULL, 1);
}

/*-----
-+
| name          rastIcon_toolLine
|
| author        BSI
|
| 7/90
|
+-----
*/

```

```

cmdName rastIcon_toolLine
(
)
cmdNumberCMD_MENUICON_TOOL_LINE
{
    /* -----
    -
        Set the down function to start the line, the motion function to
        show the prospective line and the up function to put it into the
        bitmap.
    -----
    - */
    rastIconP->downFunctionP    = rastIcon_toolLineFirstPoint;
    rastIconP->upFunctionP      = rastIcon_toolLineEndPoint;
    rastIconP->resetFunctionP    = rastIcon_toolLine;
    rastIconP->dynamicFuncP     = rastIcon_toolLineDynamics;
}

/*-----
-+
| name          rastIcon_toolBrush|
| author        BSI                7/90
|
+-----
*/
cmdName rastIcon_toolBrush
(
)
cmdNumberCMD_MENUICON_TOOL_BRUSH
{
    /* first point function does the same thing as line */
    rastIconP->downFunctionP    = rastIcon_toolLineFirstPoint;
    /* the up function does not really have to do anything */
    rastIconP->upFunctionP      = NULL;
    rastIconP->resetFunctionP    = rastIcon_toolBrush;
    rastIconP->dynamicFuncP     = rastIcon_toolBrushPixelOp;
}

/*-----
-+
| name          rastIcon_toolFill|
| author        BSI                7/90
|
|

```

```

+-----
*/
cmdName rastIcon_toolFill
(
)
cmdNumberCMD_MENUICON_TOOL_FILL
{
    /* first point function does the same thing as line */
    rastIconP->downFunctionP = rastIcon_toolLineFirstPoint;

    /* the up function does not really have to do anything */
    rastIconP->upFunctionP    = rastIcon_toolFillEndPoint;

    rastIconP->resetFunctionP = rastIcon_toolFill;
    rastIconP->dynamicFuncP   = rastIcon_toolFillDynamics;
}

/*-----
-+
|
|          Line Tool Routines |
|
+-----
*/
/*-----
-+
|
| name          rastIcon_toolLineFirstPoint
|
| author        BSI
|
|          7/90
|
+-----
*/
Private intrastIcon_toolLineFirstPoint
(
    DialogItem *diP,    /* => dialog Item */
    Point2d     *pixelP, /* => pixel id */
    int         pixelVal, /* => pixel value */
    DialogBox   *dbP    /* => dialog box */
)
{
    rastIconP->anchorPixel      = *pixelP;
    rastIconP->anchorPixelValue = pixelVal;

    /* anchorPos is the position on the screen of the center point */
    rastIconP->anchorPos.x      = diP->rect.origin.x + 1 +
                                pixelP->x * (FATBIT + 1) + FATBIT / 2;

```



```

        rastIconP->anchorPos.y      = diP->rect.origin.y + 1 +
        pixelP->y * (FATBIT + 1) + FATBIT / 2;
    }

/*-----
-+
|
| name          rastIcon_toolLineEndPoint|
|
| author        BSI                      7/90|
|
+-----
*/
Private intrastIcon_toolLineEndPoint
(
    DialogItem *diP,      /* => large icon item */
    Point2d     *pixelP,  /* => pixel id */
    int         pixelVal, /* => pixel value */
    DialogBox   *dbP      /* => dialog box */
)
{
    int xDelta, yDelta;
    int xInc=1, yInc=1;
    int error, iPixel;
    Point2d currentPixel;

    /* -----
    -
        Set pixels from beginning to end of line based on the mode and
        (possibly) the first pixel value, using a Bresenham algorithm.
    -----
    */
    if ( (xDelta = pixelP->x - rastIconP->anchorPixel.x) < 0)
        xInc = -1;
    if ( (yDelta = pixelP->y - rastIconP->anchorPixel.y) < 0)
        yInc = -1;
    /* make both xDelta and yDelta positive */
    xDelta *= xInc;
    yDelta *= yInc;
    currentPixel = rastIconP->anchorPixel;

    /* operate on first pixel */
    if (xDelta >= yDelta)
    {
        error = xDelta / 2;
        for (iPixel=0; iPixel<=xDelta; iPixel++)
        {

```

```

        /* do the operation */
        rastIcon_pixelOp (&currentPixel,
            rastIconP->modeOverride ? MODE_TOGGLE : rastIconP->mode, dbP,
            iPixel==xDelta);
        currentPixel.x += xInc;
        if ( (error = error + yDelta) > xDelta)
        {
            error        -= xDelta;
            currentPixel.y += yInc;
        }
    }
    else
    {
        error = yDelta / 2;
        for (iPixel=0; iPixel<=yDelta; iPixel++)
        {
            /* do the operation */
            rastIcon_pixelOp (&currentPixel,
                rastIconP->modeOverride ? MODE_TOGGLE : rastIconP->mode, dbP,
                iPixel==yDelta);
            currentPixel.y += yInc;
            if ( (error = error + xDelta) > yDelta)
            {
                error        -= yDelta;
                currentPixel.x += xInc;
            }
        }
    }
}

/*-----
-+
|
| name          rastIcon_toolLineDynamics
|
| author        BSI
|
| 7/90
|
+-----
*/
Private intrastIcon_toolLineDynamics
(
    DialogItem *diP, /* => dialog item */
    Point2d    *pointP, /* => position */
    Point2d    *pixelP, /* => pixel position */
    int        newData /* => this is a new (rather than erase) point */
)

```

```

    {
        DialogBox *dbP = rastIconP->dialogBoxP;

        mdlWindow_lineStyleSet (dbP, 0, rastIconP->dynamicsColor, XORDRAW,
0);
        mdlWindow_lineDraw (dbP, rastIconP->anchorPos.x,
            rastIconP->anchorPos.y, pointP->x, pointP->y, NULL);
        if (newData)
            rastIcon_reportPixelPosition (pixelP, dbP);
    }

/*-----
-+
|          Brush Tool routines          |
|-----
*/
/*-----
-+
| name          rastIcon_toolBrushPixelOp|
| author        BSI                      7/90
|-----
*/
Private intrastIcon_toolBrushPixelOp
(
    DialogItem *diP, /* => dialog item */
    Point2d    *pointP, /* => position */
    Point2d    *pixelP, /* => pixel position */
    int        newData /* => this is a new (rather than erase) point */
)
{
    DialogBox *dbP = rastIconP->dialogBoxP;
    int operation;

    if (newData)
    {
        /* set the pixel */
        if ((operation = rastIconP->mode) == MODE_TOGGLE ||
            rastIconP->modeOverride)
            operation = (rastIconP->anchorPixelValue ==
                rastIconP->offByteValue) ?
                MODE_SET : MODE_CLEAR;
    }
}

```

```

rastIcon_pixelOp (pixelP, operation, dbP, TRUE);

/* report the position */
rastIcon_reportPixelPosition (pixelP, dbP);
}
}

/*-----
-+
|
|      Fill Tool Routines |
|
|-----
*/
/*-----
-+
|
| name          rastIcon_toolFillEndPoint|
|
| author        BSI                      7/90
|
|-----
*/
Private intrastIcon_toolFillEndPoint
(
DialogItem *diP,    /* => large icon dialog item */
Point2d *pixelP,    /* => pixel id */
int pixelVal,       /* => pixel value */
DialogBox *dbP      /* => dialog box */
)
{
    int xDelta, yDelta;
    int xInc=1, yInc=1;
    intiRow, iColumn;
    Point2d currentPixel;

    /* -----
    -
        Set pixels enclosed in the rectangle based on the mode.
    -----
    - */
    if ( (xDelta = pixelP->x - rastIconP->anchorPixel.x) < 0)
        xInc = -1;
    if ( (yDelta = pixelP->y - rastIconP->anchorPixel.y) < 0)
        yInc = -1;
    /* make both xDelta and yDelta positive */
    xDelta *= xInc;

```

```

        yDelta *= yInc;

        /* operate on first pixel */
        for (currentPixel.y = rastIconP->anchorPixel.y, iRow=0;
             iRow <= yDelta; iRow++, currentPixel.y += yInc)
        {
            for (currentPixel.x = rastIconP->anchorPixel.x, iColumn=0;
                 iColumn <= xDelta; iColumn++, currentPixel.x += xInc)
            {
                rastIcon_pixelOp (&currentPixel,
                                   rastIconP->modeOverride ? MODE_TOGGLE : rastIconP->mode,
                                   dbP, ((iRow == yDelta) && (iColumn == xDelta)));
            }
        }
    }
}

/*-----
-+
|
| name          rastIcon_toolFillDynamics
|
| author        BSI
|
| 7/90
|
+-----
*/
Private intrastIcon_toolFillDynamics
(
    DialogItem *diP, /* => dialog item */
    Point2d    *pointP, /* => position */
    Point2d    *pixelP, /* => pixel position */
    int        newData /* => this is a new (rather than erase) point */
)
{
    DialogBox *dbP = rastIconP->dialogBoxP;
    Rectangle rect;

    rect.origin = rastIconP->anchorPos;
    rect.corner = *pointP;

    mdlWindow_lineStyleSet (dbP, 0, rastIconP->dynamicsColor, XORDRAW,
0);
    mdlWindow_rectDraw (dbP, &rect, NULL);
    if (newData)
        rastIcon_reportPixelPosition (pixelP, dbP);
}
/*-----
-+

```

```

|
|      Icon shift routines
|
+-----+
*/
/*-----+
|
| name          rastIcon_shiftLeft|
|
| author        BSI                7/90
|
+-----+
*/
cmdName rastIcon_shiftLeft
(
)
cmdNumberCMD_MENUICON_SHIFT_LEFT
{
    DialogItem *diP;
    byte        *bitP;
    int         iRow, iCol;

    bitP = rastIconP->dataP;

    for (iRow=0; iRow<rastIconP->height; iRow++)
    {
        for (iCol=0; iCol<rastIconP->width; iCol++, bitP++)
        {
            if (iCol < rastIconP->width - 1)
                *bitP = *(bitP+1);
            else
                *bitP = rastIconP->offByteValue;
        }
    }

    diP = mdlDialog_itemGetByTypeAndId (rastIconP->dialogBoxP,
RTYPE_Generic,
                                GENERICID_IconSmall, 0);
    mdlDialog_itemDraw (rastIconP->dialogBoxP, diP->itemIndex);

    diP = mdlDialog_itemGetByTypeAndId (rastIconP->dialogBoxP,
RTYPE_Generic,
                                GENERICID_IconLarge, 0);
    mdlDialog_itemDraw (rastIconP->dialogBoxP, diP->itemIndex);
}

```

```

/*-----
-+
| name          rastIcon_shiftRight      |
| author        BSI                      7/90 |
|-----
*/
cmdName rastIcon_shiftRight
(
)
cmdNumberCMD_MENUICON_SHIFT_RIGHT
{
    DialogItem *diP;
    byte      *bitP;
    int       iRow, iCol;

    bitP = &rastIconP->dataP[rastIconP->width * rastIconP->height - 1];;

    for (iRow=rastIconP->height; iRow>0; iRow--)
    {
        for (iCol=rastIconP->width; iCol>0; iCol--, bitP--)
        {
            if (iCol > 1)
                *bitP = *(bitP-1);
            else
                *bitP = rastIconP->offByteValue;
        }
        diP = mdlDialog_itemGetByTypeAndId (rastIconP->dialogBoxP,
        RTYPE_Generic,
            GENERICID_IconSmall, 0);
        mdlDialog_itemDraw (rastIconP->dialogBoxP, diP->itemIndex);

        diP = mdlDialog_itemGetByTypeAndId (rastIconP->dialogBoxP,
        RTYPE_Generic,
            GENERICID_IconLarge, 0);
        mdlDialog_itemDraw (rastIconP->dialogBoxP, diP->itemIndex);
    }
}

/*-----
-+
| name          rastIcon_shiftUp|
| author        BSI                      7/90 |
|-----

```

```

|
+-----+
|
| */
| cmdName rastIcon_shiftUp
| (
| )
| cmdNumberCMD_MENUICON_SHIFT_UP
| {
|     DialogItem *diP;
|     byte      *bitP;
|     int       iRow, iCol;
|
|     bitP = rastIconP->dataP;
|
|     for (iRow=0; iRow<rastIconP->height; iRow++)
|     {
|         for (iCol=0; iCol<rastIconP->width; iCol++, bitP++)
|         {
|             if (iRow < rastIconP->height - 1)
|                 *bitP = *(bitP+rastIconP->width);
|             else
|                 *bitP = rastIconP->offByteValue;
|         }
|     }
|     diP = mdlDialog_itemGetByTypeAndId (rastIconP->dialogBoxP,
|     RTYPE_Generic,
|                                     GENERICID_IconSmall, 0);
|     mdlDialog_itemDraw (rastIconP->dialogBoxP, diP->itemIndex);
|
|     diP = mdlDialog_itemGetByTypeAndId (rastIconP->dialogBoxP,
|     RTYPE_Generic,
|                                     GENERICID_IconLarge, 0);
|     mdlDialog_itemDraw (rastIconP->dialogBoxP, diP->itemIndex);
| }
|
| /*-----+
|
| name          rastIcon_shiftDown|
|
| author        BSI                | 7/90
|
|
+-----+
|
| */
| cmdName rastIcon_shiftDown
| (
| )

```



```

cmdNumberCMD_MENUICON_SHIFT_DOWN
{
    DialogItem *diP;
    byte      *bitP;
    int       iRow, iCol;

    bitP = &rastIconP->dataP[rastIconP->width * rastIconP->height - 1];;

    for (iRow=rastIconP->height; iRow>0; iRow--)
    {
        for (iCol=rastIconP->width; iCol>0; iCol--, bitP--)
        {
            if (iRow > 1)
                *bitP = *(bitP-rastIconP->width);
            else
                *bitP = rastIconP->offByteValue;
        }
    }
    diP = mdlDialog_itemGetByTypeAndId (rastIconP->dialogBoxP,
    RTYPE_Generic,
                                GENERICID_IconSmall, 0);
    mdlDialog_itemDraw (rastIconP->dialogBoxP, diP->itemIndex);

    diP = mdlDialog_itemGetByTypeAndId (rastIconP->dialogBoxP,
    RTYPE_Generic,
                                GENERICID_IconLarge, 0);
    mdlDialog_itemDraw (rastIconP->dialogBoxP, diP->itemIndex);
}

/*-----
-+
| name          rastIcon_clear|
| author        BSI           7/90|
|-----
*/
cmdName rastIcon_clear
(
)
cmdNumberCMD_MENUICON_CLEAR
{
    DialogItem *diP;

    memset (rastIconP->dataP, rastIconP->offByteValue,
            rastIconP->width * rastIconP->height);
}

```

```

rastIconP->empty = TRUE;

/* redraw */
diP = mdlDialog_itemGetByTypeAndId (rastIconP->dialogBoxP,
RTYPE_Generic,
                                GENERICID_IconSmall, 0);
mdlDialog_itemDraw (rastIconP->dialogBoxP, diP->itemIndex);

diP = mdlDialog_itemGetByTypeAndId (rastIconP->dialogBoxP,
RTYPE_Generic,
                                GENERICID_IconLarge, 0);
mdlDialog_itemDraw (rastIconP->dialogBoxP, diP->itemIndex);
}

/*-----
-+
|
|           Dialog box size calculation / specification
|
+-----
*/
/*-----
-+
|
| name           rastIcon_calculatePositions
|
| author         BSI
|
|                                     7/90
|
+-----
*/
Private voidrastIcon_calculatePositions
(
Rectangle    *overallP,
Sextent      *smallIconP,
Sextent      *largeIconP,
Sextent      *positionLabelP,
Sextent      *toolPopupP,
Sextent      *modePopupP,
Sextent      *sizePopupP,
Sextent      *nameTextP,
DialogBox    *dbP
)
{
    DialogItem*popupDiP, *menuBarDiP, *labelDiP, *textDiP;
    Rectangle largeIconRect;
    Rectangle overallRect;
    int totalWidth;

```

```

int popupWidth, popupTop, popupHeight;
int labelWidth, labelHeight, textWidth, textHeight;

/* get existing rectangle */
mdlWindow_contentRectGetGlobal (&overallRect, dbP);

/* -----
   Calculate all X components first.
   ----- */

/* get the width of the tool option button */
popupDiP = mdlDialog_itemGetByTypeAndId (dbP, RTYPE_OptionButton,
                                         OPTIONBUTTONID_Tool, 0);
popupWidth = mdlDialog_rectWidth (&popupDiP->rect);

/* get width of the label item */
labelDiP = mdlDialog_itemGetByTypeAndId (dbP, RTYPE_Label,
                                         LABELID_Position, 0);
labelWidth = mdlDialog_rectWidth (&labelDiP->rect);

/* get width of the text item */
textDiP = mdlDialog_itemGetByTypeAndId (dbP, RTYPE_Text,
                                         TEXTID_IconName, 0);

if (popupWidth < rastIconP->width)
largeIconRect.origin.x = rastIconP->width;
else
largeIconRect.origin.x = popupWidth;

/* add 10 pixels on either side of the wider of the small icon/
pops */
largeIconRect.origin.x += 20;
largeIconRect.corner.x = largeIconRect.origin.x +
                        rastIconP->width * (FATBIT + 1) + 1;
totalWidth = largeIconRect.corner.x + 10;
overallRect.corner.x = overallRect.origin.x + totalWidth;
if (smallIconP)
{
smallIconP->origin.x =
    (largeIconRect.origin.x - rastIconP->width) / 2 - 1;
smallIconP->width = rastIconP->width + 2;
}
if (largeIconP)
{
largeIconP->origin.x = largeIconRect.origin.x;
largeIconP->width = rastIconP->width * (FATBIT + 1) + 1;
}

```

```

        if (positionLabelP)
        {
            positionLabelP->origin.x = largeIconRect.origin.x;
            positionLabelP->width = labelWidth;
        }
        if (toolPopupP)
        {
            toolPopupP->origin.x = 10;
            toolPopupP->width = popupWidth;
        }
        if (modePopupP)
        {
            modePopupP->origin.x = 10;
            modePopupP->width = popupWidth;
        }
        if (sizePopupP)
        {
            sizePopupP->origin.x = 10;
            sizePopupP->width = popupWidth;
        }
        if (nameTextP)
        {
            nameTextP->origin.x = largeIconRect.origin.x;
            nameTextP->width = largeIconRect.corner.x -
largeIconRect.origin.x;
        }

        /* -----
        Now calculate all Y components.
        ----- */

        /* get the bottom of the menuBar item */
        menuBarDiP = mdlDialog_itemGetByTypeAndId (dbP, RTYPE_MenuBar,
                                                    MENUBARID_Icon, 0);
        largeIconRect.origin.y = menuBarDiP->rect.corner.y + 10;
        largeIconRect.corner.y = largeIconRect.origin.y +
            rastIconP->height * (FATBIT + 1);

        labelHeight = mdlDialog_rectHeight (&labelDiP->rect);
        textHeight = mdlDialog_rectHeight (&textDiP->rect);
        overallRect.corner.y = overallRect.origin.y + 24 +
            largeIconRect.corner.y +
            labelHeight + textHeight;

        /* get popup start position and height */
        popupTop = (largeIconRect.origin.y + largeIconRect.corner.y) / 2;
        popupHeight = mdlDialog_rectHeight (&popupDiP->rect);

```

```

        if (smallIconP)
        {
            /* centered on the top half of the large icon rect */
            smallIconP->origin.y = largeIconRect.origin.y +
                (FATBIT + 1) * rastIconP->height/4 -
                rastIconP->height/2 - 1;
            smallIconP->height = rastIconP->height + 2;
        }
        if (largeIconP)
        {
            largeIconP->origin.y = largeIconRect.origin.y;
            largeIconP->height = rastIconP->height * (FATBIT + 1) + 1;
        }
        if (positionLabelP)
        {
            positionLabelP->origin.y = largeIconRect.corner.y + 10;
            positionLabelP->height = labelHeight;
        }
        if (toolPopupP)
        {
            toolPopupP->origin.y = popupTop;
            toolPopupP->height = popupHeight;
        }
        if (modePopupP)
        {
            modePopupP->origin.y = popupTop + popupHeight + 5;
            modePopupP->height = popupHeight;
        }
        if (sizePopupP)
        {
            sizePopupP->origin.y = popupTop + 2 * popupHeight + 10;
            sizePopupP->height = popupHeight;
        }
        if (nameTextP)
        {
            nameTextP->origin.y = largeIconRect.corner.y + 10 + 4 + labelHeight;
            nameTextP->height = textHeight;
        }

        if (overallP)
        *overallP = overallRect;
    }

/*-----
-+
|

```

```

| name          rastIcon_setPixelExtent
|
| author        BSI
|
|
+-----+
*/
Private void rastIcon_setPixelExtent
(
DialogBox *dbP, /* => dialogBox that contains item */
int iItem, /* => index of item to set extent */
Sextent *sextentP /* => new extent of item */
)
{
    Sextent sextent;

    /* need to negate all the coordinates to indicate that sextent is
       specifying pixels not dialog coordinate units */
    sextent.origin.x = -sextentP->origin.x;
    sextent.origin.y = -sextentP->origin.y;
    sextent.width = -sextentP->width;
    sextent.height = -sextentP->height;

    mdlDialog_itemSetExtent (dbP, iItem, &sextent, TRUE);
}

/*-----+
|
| name          rastIcon_setPositions|
|
| author        BSI
|
|
+-----+
*/
Private void rastIcon_setPositions
(
DialogBox *dbP,
Sextent *smallIconP,
Sextent *largeIconP,
Sextent *positionLabelP,
Sextent *toolPopupP,
Sextent *modePopupP,
Sextent *sizePopupP,
Sextent *nameTextP
)
{
    DialogItem *diP;

```

```

#if defined (debug)
    printf ("smallIcon %d,%d %d %d\n",
        smallIconP->origin.x, smallIconP->origin.y,
        smallIconP->width, smallIconP->height);
    printf ("largeIcon %d,%d %d %d\n",
        largeIconP->origin.x, largeIconP->origin.y,
        largeIconP->width, largeIconP->height);
    printf ("positionLabel %d,%d %d %d\n",
        positionLabelP->origin.x, positionLabelP->origin.y,
        positionLabelP->width, positionLabelP->height);
    printf ("toolPopup %d,%d %d %d\n",
        toolPopupP->origin.x, toolPopupP->origin.y,
        toolPopupP->width, toolPopupP->height);
    printf ("modePopup %d,%d %d %d\n",
        modePopupP->origin.x, modePopupP->origin.y,
        modePopupP->width, modePopupP->height);
    printf ("sizePopup %d,%d %d %d\n",
        sizePopupP->origin.x, sizePopupP->origin.y,
        sizePopupP->width, sizePopupP->height);
#endif

/* set the small icon first */
diP = mdlDialog_itemGetByTypeAndId (dbP, RTYPE_Generic,
    GENERICID_IconSmall, 0);
rastIcon_setPixelExtent (dbP, diP->itemIndex, smallIconP);

/* set the large icon */
diP = mdlDialog_itemGetByTypeAndId (dbP, RTYPE_Generic,
    GENERICID_IconLarge, 0);
rastIcon_setPixelExtent (dbP, diP->itemIndex, largeIconP);

/* set the position label */
diP = mdlDialog_itemGetByTypeAndId (dbP, RTYPE_Label,
    LABELID_Position, 0);
rastIcon_setPixelExtent (dbP, diP->itemIndex, positionLabelP);

/* set the tool option button */
diP = mdlDialog_itemGetByTypeAndId (dbP, RTYPE_OptionButton,
    OPTIONBUTTONID_Tool, 0);
rastIcon_setPixelExtent (dbP, diP->itemIndex, toolPopupP);

/* set the mode popup */
diP = mdlDialog_itemGetByTypeAndId (dbP, RTYPE_OptionButton,
    OPTIONBUTTONID_Mode, 0);
rastIcon_setPixelExtent (dbP, diP->itemIndex, modePopupP);

/* set the size popup */

```

```

    diP = mdlDialog_itemGetTypeAndId (dbP, RTYPE_IconButton,
                                      OPTIONBUTTONID_Size, 0);
    rastIcon_setPixelExtent (dbP, diP->itemIndex, sizePopupP);

    /* set the name text */
    diP = mdlDialog_itemGetTypeAndId (dbP, RTYPE_Text,
                                      TEXTID_IconName, 0);
    rastIcon_setPixelExtent (dbP, diP->itemIndex, nameTextP);

    /* set the menubar item (NOTE: the extent you send is ignored) */
    diP = mdlDialog_itemGetTypeAndId (dbP, RTYPE_MenuBar,
                                      MENUBARID_Icon, 0);
    rastIcon_setPixelExtent (dbP, diP->itemIndex, nameTextP);
}

/*-----
-+
|
|      Dynamic Routines
|
+-----
*/
/*-----
-+
|
| name      rastIcon_largeIconButtonHandler
|
| author    BSI
|
|
+-----
*/
Private voidrastIcon_largeIconButtonHandler
(
DialogItemMessage  *dimP
)
{
    DialogItem *diP = dimP->dialogItemP;
    Point2dpixel;
    int pixelVal;
    Point2dcenter;

    if (dimP->u.button.buttonTrans == BUTTONTRANS_DOWN)
    {
        /* find out which pixel we are in */
        if (!rastIcon_whichPixel (&pixel, &center,
                                &diP->rect, &dimP->u.button.pt))

```



```

    {
        pixelVal = rastIcon_pixelValue (NULL, &pixel);
        /* call the down function */
        (*rastIconP->downFunctionP) (diP, &pixel, pixelVal, dimP->db);

        /* report the pixel position */
        rastIcon_reportPixelPosition (&pixel, dimP->db);

        rastIconP->modeOverride =
            ((dimP->u.button.qualifierMask & SHIFTKEY) != 0);
        if (rastIconP->dynamicFuncP)
        {
            dimP->u.button.motionFunc = rastIcon_updateDynamics;
            rastIcon_initDynamics (diP, &center, &pixel,
                                  &dimP->u.button.pt);
        }
    }
    else if (dimP->u.button.buttonTrans == BUTTONTRANS_UP)
    {
        rastIcon_stopDynamics (diP, &dimP->u.button.pt);

        /* find out which pixel we are in */
        if (!rastIcon_whichPixel (&pixel, &center,
                                  &diP->rect, &dimP->u.button.pt))
        {
            pixelVal = rastIcon_pixelValue (NULL, &pixel);
            if (rastIconP->upFunctionP)
                (*rastIconP->upFunctionP) (diP, &pixel, pixelVal, dimP->db);
        }
    }
}

/*-----
-+
| name          rastIcon_initDynamics
| author        BSI
| 7/90
+-----
*/
Private voidrastIcon_initDynamics
(
DialogItem  *diP,
Point2d     *centerP,
Point2d     *pixelP,

```

```

Point2d    *pointP
)
{
    mdlWindow_cursorTurnOff();
    (*rastIconP->dynamicFuncP)(diP, centerP, pixelP, TRUE);
    rastIconP->oldPos    = *centerP;
    rastIconP->oldPixel  = *pixelP;
}

```

```

/*-----
-+
| name          rastIcon_stopDynamics
| author        BSI
|
+-----
*/

```

```

Private voidrastIcon_stopDynamics
(
    DialogItem *diP,
    Point2d    *pointP
)
{
    if (rastIconP->oldPos.x > 0)
    {
        mdlWindow_cursorTurnOff();
        (*rastIconP->dynamicFuncP)(diP, &rastIconP->oldPos,
                                   &rastIconP->oldPixel, FALSE);
    }
}

```

```

/*-----
-+
| name          rastIcon_updateDynamics
| author        BSI
|
+-----
*/

```

```

Private voidrastIcon_updateDynamics (newPointP)
Point2d *newPointP;
{
    Point2dlocalPoint;
    Point2dpixel;
    Point2dcenter;
}

```

```

    DialogItem *largeIconDiP;
    int inPixel;

    largeIconDiP = mdlDialog_itemGetByTypeAndId (rastIconP->dialogBoxP,
                                                RTYPE_Generic, GENERICID_IconLarge, 0);

    mdlWindow_pointToLocal (&localPoint, rastIconP->dialogBoxP,
newPointP);

    inPixel = !rastIcon_whichPixel (&pixel, &center,
                                    &largeIconDiP->rect, &localPoint);

    /* if same pixel don't need to do anything */
    if ( (inPixel && (rastIconP->oldPixel.x == pixel.x) &&
          (rastIconP->oldPixel.y == pixel.y))
        || (!inPixel && rastIconP->oldPos.x < 0) )
    return;

    /* turn off the cursor */
    mdlWindow_cursorTurnOff();

    /* if necessary, erase old dynamics */
    if (rastIconP->oldPos.x > 0)
    (*rastIconP->dynamicFuncP)(largeIconDiP, &rastIconP->oldPos,
                              &rastIconP->oldPixel, FALSE);

    if (inPixel)
    {
        /* draw new dynamics */
        (*rastIconP->dynamicFuncP)(largeIconDiP, &center, &pixel, TRUE);
        rastIconP->oldPos    = center;
        rastIconP->oldPixel  = pixel;
    }
    else
        rastIconP->oldPos.x = -1;
    }

/*-----
-+
|
|          Utility routines
|
+-----
*/
/*-----
-+
|
|

```

```

| name          rastIcon_largeIconDraw
|
| author        BSI
|
|
+-----+
*/
Private void rastIcon_largeIconDraw
(
DialogItemMessage *dimP
)
{
    Rectangle *clipRectP;
    Rectangle fillRect;
    int iRect;
    int iRow, iCol;
    int startX, startY, endX, endY;
    byte *bitP;
    DialogItem *diP = dimP->dialogItemP;

    /* if there is only one rectangle, use it as the clipping rectangle */
    if (dimP->u.draw.nRects == 1)
        clipRectP = dimP->u.draw.rectList;
    else
        clipRectP = &diP->rect;

    /* if told to do so, erase first */
    if (dimP->u.draw.eraseFirst)
        mdlWindow_rectClear (dimP->db, &diP->rect, clipRectP);

    /* first draw the frame */
    mdlWindow_lineStyleSet (dimP->db, 0, rastIconP->onColorDescr,
NORMALDRAW, 0);
    mdlWindow_rectDraw (dimP->db, &diP->rect, clipRectP);

    /* now draw the interior lines */
    mdlWindow_lineStyleSet (dimP->db, 0, rastIconP->lineColor,
NORMALDRAW, 0);
    startY = diP->rect.origin.y + 1;
    endY = diP->rect.corner.y - 1;
    for (startX = diP->rect.origin.x + FATBIT+1;
        startX < diP->rect.corner.x; startX += FATBIT+1)
    {
        mdlWindow_lineDraw (dimP->db, startX, startY,
                            startX, endY, clipRectP);
    }
}

```

```

        startX = diP->rect.origin.x + 1;
        endX    = diP->rect.corner.x - 1;
        for (startY = diP->rect.origin.y + FATBIT+1;
            startY < diP->rect.corner.y; startY += FATBIT+1)
        {
            mdlWindow_lineDraw (dimP->db, startX, startY,
                                endX, startY, clipRectP);
        }

        fillRect.origin.y = diP->rect.origin.y + 1;
        fillRect.corner.y = diP->rect.origin.y + FATBIT;
        bitP = rastIconP->dataP;
        if (!rastIconP->empty)
        for (iRow=0; iRow<rastIconP->height; iRow++)
        {
            fillRect.origin.x = diP->rect.origin.x + 1;
            fillRect.corner.x = diP->rect.origin.x + FATBIT;
            for (iCol=0; iCol<rastIconP->width; iCol++, bitP++)
            {
                mdlWindow_rectFill (dimP->db, &fillRect,
                                    (*bitP == rastIconP->onByteValue)
                                    ? rastIconP->onColorDescr
                                    : rastIconP->offColorDescr,
                                    clipRectP);
                fillRect.corner.x += FATBIT+1;
                fillRect.origin.x += FATBIT+1;
            }
            fillRect.origin.y += FATBIT+1;
            fillRect.corner.y += FATBIT+1;
        }
    }

/*-----
-+
| name          rastIcon_smallIconDraw
|
| author        BSI
|
| 7/90
|
+-----
*/
Private voidrastIcon_smallIconDraw
(
DialogItemMessage *dimP
)
{
    Rectangle *clipRectP;

```



```

/*-----
-+
| name          rastIcon_allocateBitMap
| author        BSI
| 5/90
|
+-----
*/
Private intrastIcon_allocateBitMap
(
)
{
    int iconBytes;

    iconBytes = rastIconP->width * rastIconP->height;

    /* find the small icon and see what size it is */
    rastIconP->dataP = malloc (iconBytes);
    memset (rastIconP->dataP, rastIconP->offByteValue, iconBytes);
    rastIconP->empty = TRUE;
    return (FALSE);
}

/*-----
-+
| name          rastIcon_freeBitMap|
| author        BSI
| 5/90
|
+-----
*/
Private intrastIcon_freeBitMap
(
)
{
    free (rastIconP->dataP);
    rastIconP->dataP = NULL;
}

/*-----
-+
| name          rastIcon_whichPixel
|
|
|

```

```

| author          BSI                      7/90          |
|-----|-----|
*/
Private intrastIcon_whichPixel
(
Point2d      *pixelP, /* <= pixel we are on */
Point2d      *centerP, /* <= center of pixel we are on */
Rectangle     *rectP, /* => large icon rectangle */
Point2d      *pointP /* => input point */
)
{
    int returnCode;

    pixelP->x = (pointP->x - rectP->origin.x) / (FATBIT + 1);
    pixelP->y = (pointP->y - rectP->origin.y) / (FATBIT + 1);

    returnCode = (pixelP->x < 0) || (pixelP->x >= rastIconP->width) ||
        (pixelP->y < 0) || (pixelP->y >= rastIconP->height);

    if (!returnCode && centerP)
    {
        /* figure out the starting and ending points */
        centerP->x = rectP->origin.x + 1 +
            pixelP->x * (FATBIT + 1) + FATBIT / 2;
        centerP->y = rectP->origin.y + 1 +
            pixelP->y * (FATBIT + 1) + FATBIT / 2;
    }
    return (returnCode);
}

/*-----+
|
| name          rastIcon_pixelValue
|
| author        BSI                      7/90          |
|-----+
*/
Private intrastIcon_pixelValue
(
byte**bytePP, /* <= pointer to where data is stored */
Point2d *pixelP /* => pixel */
)
{
    byte *byteP;

```



```

        byteP = &rastIconP->dataP[pixelP->x + rastIconP->width * pixelP->y];
        if (bytePP)
            *bytePP = byteP;

        return (*byteP);
    }

/*-----
-+
| name          rastIcon_reportPixelPosition
| author        BSI                               7/90
|-----
*/
Private voidrastIcon_reportPixelPosition
(
    Point2d      *pixelP,      /* => pixel position */
    DialogBox     *dbP/* => dialog box pointer */
)
{
    DialogItem *labelDiP;
    int isNew;
    charoutMsg[80];
    charfmtMsg[80];

    mdlResource_loadFromStringList (fmtMsg, NULL, STRINGID_Messages, 0);

    sprintf (outMsg, fmtMsg, pixelP->x+1, pixelP->y+1);

    /* find the label */
    labelDiP = mdlDialog_itemGetByTypeAndId (dbP, RTYPE_Label,
                                             LABELID_Position, 0);
    mdlDialog_itemSetLabel (dbP, labelDiP->itemIndex, outMsg);
}

/*-----
-+
| name          rastIcon_pixelOp
| author        BSI                               7/90
|-----
*/

```

```

Private intrastIcon_pixelOp
(
    Point2d    *pixelP,
    int        operation,
    DialogBox  *dbP,
    int        lastOp
)
{
    byte *byteP;
    int  currentVal;

    currentVal = rastIcon_pixelValue (&byteP, pixelP);

    switch (operation)
    {
    case MODE_TOGGLE:
        *byteP = (currentVal == rastIconP->onByteValue) ?
            rastIconP->offByteValue : rastIconP->onByteValue;
        break;
    case MODE_SET:
        *byteP = rastIconP->onByteValue;
        break;
    case MODE_CLEAR:
        *byteP = rastIconP->offByteValue;
        break;
    }

    rastIconP->empty = FALSE;
    rastIconP->modified = TRUE;
    rastIcon_refreshPixel (pixelP,
        (*byteP == rastIconP->onByteValue) ?
            rastIconP->onColorDescr : rastIconP->offColorDescr,
            dbP, lastOp);
    }

/*-----
-+
|
| name          rastIcon_refreshPixel
|
| author        BSI
|
| 7/90
|
+-----
*/
Private intrastIcon_refreshPixel
(
    Point2d    *pixelP, /* => pixel to redraw */

```

```

int    value, /* => value to draw it as */
DialogBox *dbP, /* => dialog box */
int    refreshSmallIcon /* => true if small icon should be redrawn */
)
{
    DialogItem*largeIconDiP, *smallIconDiP;
    Rectangle fillRect;

    largeIconDiP = mdlDialog_itemGetByTypeAndId (dbP, RTYPE_Generic,
                                                GENERICID_IconLarge, 0);
    smallIconDiP = mdlDialog_itemGetByTypeAndId (dbP, RTYPE_Generic,
                                                GENERICID_IconSmall, 0);
    /* calculate the rectangle to fill */
    fillRect.origin.x = largeIconDiP->rect.origin.x + 1 +
        pixelP->x * (FATBIT + 1);
    fillRect.corner.x = fillRect.origin.x + FATBIT - 1;
    fillRect.origin.y = largeIconDiP->rect.origin.y + 1 +
        pixelP->y * (FATBIT + 1);
    fillRect.corner.y = fillRect.origin.y + FATBIT - 1;
    mdlWindow_rectFill (dbP, &fillRect, value, NULL);

    /* now tell the small icon to refresh also */
    if (refreshSmallIcon)
        rastIcon_drawSmallBitMap (&smallIconDiP->rect, dbP, NULL);
}

/*-----
-+
| name          rastIcon_writeIconFile|
| author        BSI                    7/90|
|-----
*/
Private intrastIcon_writeIconFile
(
{
    char    fileSpec[MAXFILELENGTH];
    char    fmtMsg[128];
    char    compressedIconName[16];
    byte    outBytes[8], *dataP;
    int     iOutByte, iByte, totalBytes, iLeftOver;
    int     append = rastIconP->appendToFile;
    int     currentByte;
    int     rscNum = rastIconP->inRscId;

```

```

FILE      *fP;
char      tmp[128];
char      tmpFileName[64];
char      *tmp1;

tmp1 = strchr (rastIconP->outFileName, '.');
if (tmp1)
{
    strcpy (tmp, tmp1);
   strupr (tmp);
    if (strcmp (tmp, ".R"))
    {
        mdlUtil_beep(1);
        mdlOutput_errorU ("invalid file type (must be .r)");
        return(TRUE);
    }
}

/* see if we can open the file */
if (!mdlFile_find (fileSpec, rastIconP->outFileName, "MS_DATA",
".r"))
{
    /* -----
    If we don't have append flag on, we should probably put out
    an alert box. (But we won't).
    ----- */
    if (append)
    {
        if(!rscNum)
        {
            /* find the highest resource in the file */
            rscNum = rastIcon_findHighestResource (fileSpec);
            rscNum++;
        }
        mdlOutput_rscPrintf (MSG_STATUS, NULL, STRINGID_Messages,
                            1, fileSpec);
    }
    else
    {
        /* no file, can't append */
        append = 0;
    }

    if ( (fP = fopen (fileSpec, append ? "a" : "w")) == NULL)

```

```
{
rastIcon_displayErrorMessage (2, fileSpec);
return (TRUE);
}

    if (append)
fseek (fP, 0, SEEK_END);
    else
    {
/* include rasticon.h to make sure it will compile */
rastIcon_fmtMsgGet (fmtMsg, 10);
fprintf (fP, fmtMsg);
    }

/* here we have an open file, write to it */
/* write out comment indicating the icon name */
rastIcon_fmtMsgGet (fmtMsg, 0);
fprintf (fP, fmtMsg, rastIconP->iconName);

/* compress the spaces out of the item name and put out resource id
*/
    rastIcon_compressIconName (compressedIconName, rastIconP-
>iconName);

/* #define the resourceId */
rastIcon_fmtMsgGet (fmtMsg, 11);
fprintf (fP, fmtMsg, compressedIconName, rscNum);

/* declaration */
rastIcon_fmtMsgGet (fmtMsg, 20 + rastIconP->sizeIndex);
fprintf (fP, fmtMsg, compressedIconName);

/* start the definition */
rastIcon_fmtMsgGet (fmtMsg, 2);
fprintf (fP, fmtMsg);

/* put out the size and format */
rastIcon_fmtMsgGet (fmtMsg, 3);
fprintf (fP, fmtMsg, rastIconP->width, rastIconP->height);

/* put out the icon name */
rastIcon_fmtMsgGet (fmtMsg, 4);
fprintf (fP, fmtMsg, rastIconP->iconName);

/* put out the bit map */
rastIcon_fmtMsgGet (fmtMsg, 5);
fprintf (fP, fmtMsg);
```

```

rastIcon_fmtMsgGet (fmtMsg, 6);

totalBytes = rastIconP->width * rastIconP->height;
for (iByte = 1, currentByte = iOutByte = 0, dataP=rastIconP->dataP;
    iByte <= totalBytes; iByte++, dataP++)
{
    currentByte |= (*dataP == rastIconP->offByteValue) ? 0 : 1;
    if (iByte % 8)
        currentByte <<= 1;
    else
    {
        outBytes[iOutByte++] = currentByte;
        currentByte = 0;
        if (!(iOutByte % 8))
        {
            fprintf (fP, fmtMsg,
                outBytes[0], outBytes[1], outBytes[2], outBytes[3],
                outBytes[4], outBytes[5], outBytes[6], outBytes[7]);
            iOutByte = 0;
        }
    }
}

/* flush out any that did not already get printed */
/* shift the last bits all the way to the left */
if (iByte % 8)
{
    for (; iByte % 8; iByte++)
        currentByte <<= 1;

    outBytes[iOutByte++] = currentByte;
}

if (iOutByte)
{
    fprintf (fP, "\t");
    rastIcon_fmtMsgGet (fmtMsg, 7);
    for (iLeftOver = 0; iLeftOver < iOutByte; iLeftOver++)
        fprintf (fP, fmtMsg, outBytes[iLeftOver]);
    fprintf (fP, "\n");
}

rastIcon_fmtMsgGet (fmtMsg, 8);
fprintf (fP, fmtMsg);
rastIcon_fmtMsgGet (fmtMsg, 9);
fprintf (fP, fmtMsg);

/* close the file */

```

```

fclose (fP);

/* bitmap has not been modified since write */
rastIconP->modified = FALSE;

return (SUCCESS);
}

/*-----
-+
| name          rastIcon_findHighestResource          |
| author        BSI                                  7/90 |
|-----
*/
Private intrastIcon_findHighestResource
(
char *fileSpec
)
{
    int    highestResource = -100000000;
    int    len;
    int    resourceNum;
    char    inLine[128];
    char    searchLine[128];
    char    scanLine[128];
    char    iconName[128];
    char    *percentP;
    FILE    *fP;

    rastIcon_fmtMsgGet (searchLine, 11);
    strcpy (scanLine, searchLine);
    if ( (percentP = strchr (searchLine, '%')) != NULL)
*percentP = '\0';
    len = strlen (searchLine);

    if ( (fP = fopen (fileSpec, "r")) != NULL)
    {
        for (;;)
        {
            if (!fgets (inLine, sizeof(inLine), fP))
                break;
            /* see if the line defines a resource ID */
            if (!strncmp (inLine, searchLine, len))
            {

```

```

        resourceNum = highestResource;
        sscanf (inLine, scanLine, iconName, &resourceNum);
        if (resourceNum > highestResource)
            highestResource = resourceNum;
    }
}
fclose (fP);
}

return (highestResource);
}

/*-----
-+
| name          rastIcon_readIconResourceFile|
| author        BSI                          7/90
|-----
*/
Private intrastIcon_readIconResourceFile
(
{
    int rscHandle;
    Point2d origin;
    DialogItem*smallIconDiP, *largeIconDiP, *sizePopupDiP;
    IconRsc*irP;
    byte*inDataP, *curDataP;
    int iSize, iBit, mask, totalSize;
    DialogBox *dbP = rastIconP->dialogBoxP;

mdlResource_openFile (&rastIconP->rscHandleFileToRead,
                      rastIconP->inFileName,0);

    /* try to get the specified resource */
    if ( (irP = (IconRsc *) mdlResource_load (rastIconP-
>rscHandleFileToRead,
        rastIconP->selectedType,
        rastIconP->inRscId)) == NULL)
    {
        rastIcon_displayErrorMessage (4, rastIconP->inRscId);
        return (ERROR);
    }
    mdlResource_closeFile (rastIconP->rscHandleFileToRead);

```

```

    /* now decompose it and put it into our window */
    if ( (irP->width != rastIconP->width) ||
        (irP->height != rastIconP->height) )
    {
        rastIcon_fitToIcon (rastIconP->dialogBoxP, irP->width, irP->height);
        /* see what sizeIndex we can use */
        for (iSize=0; iSize<sizeof(iconWidths)/sizeof(int); iSize++)
        {
            if ( (iconWidths[iSize] == irP->width) &&
                (iconHeights[iSize] == irP->height) )
                break;
        }
        if (iSize < sizeof(iconWidths)/sizeof(int))
        {
            if (rastIconP->sizeIndex != iSize);
            {
                rastIconP->sizeIndex = iSize;
                /* set the size option button */
                sizePopupDiP = mdlDialog_itemGetByTypeAndId (dbP,
                    RTYPE_OptionButton, OPTIONBUTTONID_Size, 0);
                mdlDialog_itemDraw (dbP, sizePopupDiP->itemIndex);
            }
        }
    }

    /* find where it goes */
    smallIconDiP = mdlDialog_itemGetByTypeAndId (dbP, RTYPE_Generic,
        GENERICID_IconSmall, 0);

    origin.x = smallIconDiP->rect.origin.x + 1;
    origin.y = smallIconDiP->rect.origin.y + 1;

    /* just as a test of iconDraw, put it out */
    mdlWindow_iconDraw (dbP, irP, &origin, rastIconP->onColorDescr,
        rastIconP->offColorDescr, ICON_STYLE_NORMAL, NULL);

    /* set up the data */
    totalSize = rastIconP->width * rastIconP->height;
    for (iBit=1, inDataP=irP->data, curDataP=rastIconP->dataP,
mask=0x80;
        iBit <= totalSize; curDataP++, iBit++)
    {
        *curDataP = (*inDataP & mask) ? rastIconP->onByteValue
            : rastIconP->offByteValue;
        if (!(iBit % 8))
        {
            mask = 0x0080;

```

```

        inDataP++;
    }
else
    mask >>= 1;
}

rastIconP->empty = FALSE;
/* draw both Icon items */
mdlDialog_itemDraw (dbP, smallIconDiP->itemIndex);
largeIconDiP = mdlDialog_itemGetByTypeAndId (dbP, RTYPE_Generic,
        GENERICID_IconLarge, 0);
mdlDialog_itemDraw (dbP, largeIconDiP->itemIndex);

return (SUCCESS);
}

/*-----
-+
| name          rastIcon_fitToIcon
| author        BSI
|                                     7/90
+-----
*/
Private voidrastIcon_fitToIcon
(
DialogBox *dbP,
int width,
int height
)
{
Rectangle newOverall;

/* set the width and height */
rastIconP->width = width;
rastIconP->height = height;

/* free the old bit map */
rastIcon_freeBitMap();

/* allocate the new bit map */
rastIcon_allocateBitMap ();
rastIcon_calculatePositions (&newOverall, NULL, NULL, NULL,
        NULL, NULL, NULL, NULL, dbP);
mdlWindow_resize (dbP, CORNER_LOWERRIGHT, &newOverall.corner);
mdlWindow_windowEventsProcessAll ();

```

```

    }

/*-----
-+
| name          rastIcon_compressIconName
| author        BSI
| 7/90
+-----
*/
Private voidrastIcon_compressIconName
(
char *outNameP,
char *inNameP
)
{
    for (; *inNameP; inNameP++)
    if (*inNameP != ' ')
        *outNameP++ = *inNameP;

    *outNameP = '\0';
}

/*-----
-+
| name          rastIcon_fmtMsgGet
| author        BSI
| 7/90
+-----
*/
Private voidrastIcon_fmtMsgGet
(
char *fmtMsg,
int  msgNum
)
{
    mdlResource_loadFromStringList (fmtMsg, NULL,
                                   STRINGID_FormatMsgs, msgNum);
}

/*-----
-+
|
|
|
+-----

```

```

| name          rastIcon_displayErrorMessage|
|
| author          BSI                          7/90
|
+-----+
*/
Private void rastIcon_displayErrorMessage
(
intmessageNumber,
...
)
{
    va_list ap;

    va_start (ap, messageNumber);

    mdlOutput_rscvPrintf (MSG_ERROR, NULL, STRINGID_Errors,
                          messageNumber, ap);

    va_end (ap);
}

/*-----+
-+
|
| name rastIcon_listHook
|
| authorBSI
|
+-----+
*/
Private voidrastIcon_listHook
(
DialogItemMessage*dimP
)
{
    DialogItem *diP = dimP->dialogItemP;
    StringList **strListP;
    int rscHandle;

    dimP->msgUnderstood = TRUE;
    switch (dimP->messageType)
    {
    case DITEM_MESSAGE_CREATE:
        {
            rastIconP->listBoxHdrP = diP->rawItemP;

```

```

        if ((rastIconP->stringListP = mdlStringList_create (0,2)) ==
NULL)
        dimP->u.create.createFailed = TRUE;

        if (mdlDialog_listBoxSetStrListP (diP->rawItemP,
            rastIconP->stringListP, 1) != SUCCESS)
            dimP->u.create.createFailed = TRUE;
        rastIcon_showRscList();
        break;
    }

case DITEM_MESSAGE_DESTROY:
    break;

case DITEM_MESSAGE_BUTTON:
    {
        break;
    }

default:
    dimP->msgUnderstood = FALSE;
}
}

/*-----
-+
| name          rastIcon_showRscList|
| author      BSI          11/90    |
|-----
+-----
*/
Private Void rastIcon_showRscList
(
{
    int          numResources, i, memberIndex, status;
    ULong        rscId;
    char         asciiResourceId[20];
    char         iconName[17];
    boolean      foundSelection;
    InfoFields   *rscInfoFields;
    IconRsc      *rscP;

    if (rastIconP->rscHandleFileToRead == FILE_UNUSED)
return;

```

```

mdlResource_openFile (&rastIconP->rscHandleFileToRead,
                      rastIconP->inFileName, 0);

rastIconP->selectedType = iconResourceTypes[rastIconP->sizeIndex];

/* Set appropriate fields in member. */
rastIconP->selectedTypeAsc [0] =
(char)((rastIconP->selectedType & 0xFF000000) >> 24);
rastIconP->selectedTypeAsc [1] =
(char)((rastIconP->selectedType & 0x00FF0000) >> 16);
rastIconP->selectedTypeAsc [2] =
(char)((rastIconP->selectedType & 0x0000FF00) >> 8);
rastIconP->selectedTypeAsc [3] =
(char)((rastIconP->selectedType & 0x000000FF));
rastIconP->selectedTypeAsc [4] = '\0';

mdlResource_queryClass (&numResources, rastIconP-
>rscHandleFileToRead,
                      rastIconP->selectedType, RSC_QRY_COUNT, NULL);

for (i=0; i<numResources; i++)
{
    mdlResource_queryClass (&rscId, rastIconP->rscHandleFileToRead,
                          rastIconP->selectedType, RSC_QRY_ID, &i);

    /* Temporarily load the resource to get the icon name. */
    rscP = mdlResource_load (rastIconP->rscHandleFileToRead,
                          rastIconP->selectedType, rscId);
    memset (iconName, '\0', sizeof (iconName));
    strncpy (iconName, rscP->name, sizeof (iconName)-1);
    mdlResource_free (rscP);

    status = mdlStringList_insertMember (&memberIndex,
                                       rastIconP->stringListP,
                                       APPEND_INDEX, 2);

    mdlStringList_getMember (NULL, &rscInfoFields,
                          rastIconP->stringListP, memberIndex);

    rscInfoFields->id = rscId;

    /* The following line would display the ids in hex rather than decimal
    */
    /* sprintf (asciiResourceId, "0x%08X", rscId); */

```

```

    sprintf (asciiResourceId, "%d", rscId);

    status = mdlStringList_setMember (rastIconP->stringListP,
                                      memberIndex, asciiResourceId,
                                      rscInfoFields);

    status = mdlStringList_setMember (rastIconP->stringListP,
                                      memberIndex+1, iconName,
                                      rscInfoFields);
}

    status = mdlDialog_listBoxNRowsChanged (rastIconP->listBoxHdrP);
    status = mdlDialog_listBoxDrawContents (rastIconP->listBoxHdrP,
                                            -1, -1);
    mdlResource_closeFile (rastIconP->rscHandleFileToRead);
}

/*-----
-+
| name      rastIcon_setInRscId|
|
| author    BSI          2/91  |
|
+-----
*/
rastIcon_setInRscId
(
)
{
    boolean      foundSelection;
    InfoFields   *rscInfoFields;
    int          row, col, selectedAliasIndex;

    /* Get currently selected resource. */
    row = col = SEARCH_FROM_BEGINNING;
    if (mdlDialog_listBoxGetNextSelection (&foundSelection,
                                           &row, &col,
                                           rastIconP->listBoxHdrP)
        != SUCCESS)
        return;

    if (mdlStringList_getMember (NULL, &rscInfoFields,
                                rastIconP->stringListP,
                                row * 2)
        == SUCCESS)

```

```

    {
        rastIconP->inRscId = rscInfoFields->id;
    }
}

```

resmover.mc

```

/*-----+
| Copyright (c) 1985-95; Bentley Systems, Inc., All rights reserved.|
| "MicroStation" is a registered trademark and "MDL" and "MicroCSL"|
| are trademarks of Bentley Systems, Inc.      |
| Limited permission is hereby granted to reproduce and modify this|
| copyrighted material provided that the resulting code is used only |
| in conjunction with Bentley Systems products under the terms of the|
| license agreement provided therein, and that this notice is retained|
| in its entirety in any such reproduction or modification.|
+-----*/
/*-----+
| $Workfile:   resmover.mc  $
| $Revision:   5.7  $
| $Date:      24 Oct 1995 09:38:14  $
+-----*/
/*-----+
| Function -                               |
| resmover.mc - example MDL program to manipulate resource files.|
| - - - - -|
| Public Routine Summary -|
|                               |
| main - main entry point      |
| resMover_aliasTextHook - Resource alias text item hook function|
| resMover_buildClassList - Build list of resource classes|
| resMover_buttonHook - Push button hook function|
| resMover_changeAlias - Change resource alias|
| resMover_clearRscList - Clear list of resources|
| resMover_close - Close a resource file|
| resMover_closeAndReopenFile - Close and reopen a resource file|
| resMover_closeCurrentFile - Close current destination rsc file|
| resMover_copyResource - Copy a resource from one file to another|
| resMover_create - Create a resource file|
| resMover_deleteResource - Delete a resource from a file|
| resMover_dialogHook - Dialog Box hook|
| resMover_displayError - Display an error message|
| resMover_getAndOpenFile - Ger a resource file and open it|
| resMover_getDataBasedOnLogicalName - Get a resource by alias|
| resMover_getSelClass - Get current resource class selection|

```



```

| resMover_getSelRscInfo - Get selected resource information|
| resMover_listHook - List Box hook function for all list boxes|
| resMover_openFile - Open a resource file|
| resMover_processClassListClick - Handle button events in|
|     resource class list box|
| resMover_processResourceListClick - Handle button events in|
|     resource list box |
| resMover_quit - Exit resmover|
| resMover_saveUsersFavoriteColor - Write/Update a resource|
| resMover_selectFile - Select a resource file|
| resMover_showRscList - Show list of resources in a file|
| resMover_unloadFunction - Unload async function|
| resMover_validateFileName - Make sure file name is good|
+-----*/
#include    <dlogbox.h>
#include    <dlogitem.h>
#include    <cexpr.h>
#include    <mdl.h>
#include    <mdlio.h>
#include    <userfnc.h>
#include    <string.h>
#include    <stdlib.h>
#include    <dlogids.h>

#include    "resmover.h"
#include    "rmovrcmd.h"

#include    <dlogman.fdf>
#include    <msrsrc.fdf>
#include    <mssystem.fdf>
#include    <msoutput.fdf>
#include    <mspase.fdf>
#include    <mscexpr.fdf>

/*-----+
|   Local defines           |
+-----*/
#define APPEND_INDEX -1 /* Used during mdlStringList_insertMember () */
#define DELETE_ALL -1 /* Used during mdlStringList_deleteMember () */
#define SEARCH_FROM_BEGINNING -1 /* Used during
mdlDialog_getNextSelection. */
#define NULL_ID ' '
#define RSC_LIST_COLUMNS 2

/*-----+
|   Local type definitions |
+-----*/
typedef struct infofields
{

```

```

        ULONG reservedForListManager;
        ULONG id;          /* resourceclass or resourceID */
    } InfoFields;

/*-----+
|   Private Global variables|
+-----*/
ResMoverGlobals *resMoverP;
char          *setP;

/*-----+
|   Local function declarations |
+-----*/
/*-----+
|   External variables          |
+-----*/
/*-----+
|   Major Public Code Section|
+-----*/
/*-----+
| name      resMover_displayError|
| author    BSI          12/90    |
+-----*/
int resMover_displayError(int errorMsgId)
{
    int actionButton;
    char alertMsg[80];

    mdlResource_loadFromStringList(alertMsg,resMoverP->resMoverRfHandle,
                                   STRINGID_Errors, errorMsgId);

    actionButton = mdlDialog_openMessageBox(DIALOGID_MsgBoxOKCancel,
                                             alertMsg, MSGBOX_ICON_WARNING);
    return (actionButton);
}

/*-----+
| name      resMover_clearRscList|
| author    BSI          11/90    |
+-----*/
Private void resMover_clearRscList
(
int currFile /* => File corresponding to list box being manipulated. */
)
{
    ResourceFileInfo*rscFileInfoP = &resMoverP->rscFileInfo[currFile];
    if (rscFileInfoP->rfHandle == FILE_UNUSED)
        return;

    mdlStringList_deleteMember (rscFileInfoP->rscListP, 0, DELETE_ALL);
}

```

```

mdlDialog_listBoxNRowsChanged (rscFileInfoP->rscListBoxHdrP);
mdlDialog_listBoxDrawContents (rscFileInfoP->rscListBoxHdrP, -1, -1);
rscFileInfoP->selRscId = NULL_ID;
rscFileInfoP->selAlias[0] = '\0';
}

/*-----+
| name      resMover_validateFileName|
| author    BSI          11/90      |
+-----*/
Private int resMover_validateFileName      /* <= SUCCESS or -1 */
(
char *filename      /* => See if this file is already open. */
)
{
    if (((resMoverP->rscFileInfo [0].rfHandle != FILE_UNUSED) &&
        (!strcmp (filename, resMoverP->rscFileInfo [0].fileName)))
        || ((resMoverP->rscFileInfo [1].rfHandle != FILE_UNUSED) &&
            (!strcmp (filename, resMoverP->rscFileInfo [1].fileName))))
        return (-1);
    return (SUCCESS);
}

/*-----+
| name      resMover_getSelClass|
| author    BSI          12/90      |
+-----*/
void resMover_getSelClass(int currFile)
{
    ResourceFileInfo*rscFileInfoP = &resMoverP->rscFileInfo [currFile];
    boolean          foundSelection;
    InfoFields       *classInfoFields;
    int              row, col;

/* Get currently selected resourceclass. */
    row = col = SEARCH_FROM_BEGINNING;
    if ((mdlDialog_listBoxGetNextSelection (&foundSelection,
        &row, &col, rscFileInfoP->classListBoxHdrP)!= SUCCESS)
        || (mdlStringList_getMember(NULL,
            (long **)&classInfoFields, rscFileInfoP->classListP, row)
            != SUCCESS))
        return;

    rscFileInfoP->selClass = classInfoFields->id;
}

/*-----+
| name      resMover_getSelRscInfo|
| author    BSI          12/90      |
+-----*/

```

```

void resMover_getSelRscInfo(int currFile)
{
    ResourceFileInfo *rscFileInfoP = &resMoverP->rscFileInfo[currFile];
    boolean foundSelection;
    InfoFields *rscInfoFields;
    int row, col, selectedAliasIndex;
    char *aliasP;

    /* Get currently selected resource. */
    row = col = SEARCH_FROM_BEGINNING;
    if (mdlDialog_listBoxGetNextSelection (&foundSelection,
        &row, &col, rscFileInfoP->rscListBoxHdrP)!= SUCCESS)
        return;

    selectedAliasIndex = (row * RSC_LIST_COLUMNS) + 1;
    if (mdlStringList_getMember (&aliasP, (long **)&rscInfoFields,
        rscFileInfoP->rscListP, selectedAliasIndex) == SUCCESS)
    {
        rscFileInfoP->selRscId = rscInfoFields->id;
        if (aliasP)
        {
            strncpy(rscFileInfoP->selAlias, aliasP,
                MAX_ALIAS_LENGTH-1);
            strncpy(resMoverP->editAlias, aliasP, MAX_ALIAS_LENGTH-1);
            mdlDialog_itemSynch ((DialogBox *)resMoverP->dialogBoxP,
                ALIASTEXTITEMNUM);
        }
    }
}

/*-----+
| name      resMover_changeAlias|
| author    BSI                2/91      |
+-----*/
void resMover_changeAlias(void)
{
    ResourceFileInfo *rscFileInfoP;
    int status;

    /* See if at least one file is open. */
    if ((resMoverP->rscFileInfo[0].rfHandle == FILE_UNUSED) &&
        (resMoverP->rscFileInfo[1].rfHandle == FILE_UNUSED))
    {
        resMover_displayError (15);
        return;
    }

    if (resMoverP->rscFileInfo [0].selRscId != NULL_ID)
        rscFileInfoP = &resMoverP->rscFileInfo [0];

```

```

else if (resMoverP->rscFileInfo [1].selRscId != NULL_ID)
    rscFileInfoP = &resMoverP->rscFileInfo [1];
else
{
    resMover_displayError (7);
    return;
}

/* Change the resource's alias. */
if (status = mdlResource_changeAlias (rscFileInfoP->rfHandle,
    rscFileInfoP->selClass, rscFileInfoP->selRscId,
    resMoverP->editAlias)!= SUCCESS)
{
    switch (status)
    {
        case MDLERR_RSCALREADYEXISTS:
            resMover_displayError (19);
            break;

        default:
            resMover_displayError (18);
            break;
    }
    return;
}
else
{
    boolean foundSelection;
    InfoFields *rscInfoFields;
    int row, col, selectedAliasIndex;

    /* Get currently selected resource. */
    row = col = SEARCH_FROM_BEGINNING;
    if (mdlDialog_listBoxGetNextSelection (&foundSelection,
        &row, &col, rscFileInfoP->rscListBoxHdrP)
        != SUCCESS)
    {
        int targetFile = rscFileInfoP-&resMoverP->rscFileInfo [0];
        resMover_clearRscList (targetFile);
        return;
    }

    selectedAliasIndex = (row * RSC_LIST_COLUMNS) + 1;
    if (mdlStringList_getMember (NULL, (long **)&rscInfoFields,
        rscFileInfoP->rscListP, selectedAliasIndex)
        == SUCCESS)
    {
        int topRowIndex;
        mdlStringList_setMember (rscFileInfoP->rscListP,

```

```

        selectedAliasIndex, resMoverP->editAlias,
        (long *)rscInfoFields);

    mdlDialog_listBoxGetDisplayRange (&topRowIndex, NULL,
        NULL, NULL,
        rscFileInfoP->rscListBoxHdrP);

    mdlDialog_listBoxDrawContents(
        rscFileInfoP->rscListBoxHdrP,
        row-topRowIndex, 1);
    }
}

/*-----+
| name      resMover_buildClassList|
| author    BSI          11/90      |
+-----*/
Private int resMover_buildClassList(int currFile)
{
    int numClasses, status, i, memberIndex;
    ULONG *classes = NULL;
    char asciiClass[10];
    InfoFields *classInfoFields;
    ResourceFileInfo *rscFileInfoP=&resMoverP->rscFileInfo[currFile];
    if ((status = mdlResource_queryFile (&numClasses,
        rscFileInfoP->fileName,
        RSC_QRY_NUMTYPES )) != SUCCESS)
    {
        rscFileInfoP->rfHandle = FILE_UNUSED;
        return (-1);
    }

    if (numClasses)
    {
        if ((classes = malloc(numClasses * sizeof(ULONG))) == NULL)
            return (-1);

        if ((status = mdlResource_queryFile(classes,
            rscFileInfoP->fileName, RSC_QRY_CLASS )) != SUCCESS)
        {
            free (classes);
            return (-1);
        }
    }

    for (i=0; i<numClasses; i++)
    {
        status=mdlStringList_insertMember(&memberIndex,
            rscFileInfoP->classListP, APPEND_INDEX, 1);
    }
}

```

```

        mdlStringList_getMember(NULL, (long **)&classInfoFields,
                                rscFileInfoP->classListP, memberIndex);

        /* Set appropriate fields in member. */
        asciiClass [0] = (char)((classes[i] & 0xFF000000) >> 24);
        asciiClass [1] = (char)((classes[i] & 0x00FF0000) >> 16);
        asciiClass [2] = (char)((classes[i] & 0x0000FF00) >> 8);
        asciiClass [3] = (char)((classes[i] & 0x000000FF));
        asciiClass [4] = '\0';
        classInfoFields->id = classes[i];

        status = mdlStringList_setMember(rscFileInfoP->classListP,
                                        memberIndex, asciiClass, (long *)classInfoFields);
    }

    if (numClasses)
    {
        free (classes);
        status=mdlDialog_listBoxNRowsChanged(
            rscFileInfoP->classListBoxHdrP);
        status = mdlDialog_listBoxDrawContents(
            rscFileInfoP->classListBoxHdrP, -1, -1);
    }

    return (SUCCESS);
}

/*-----+
| name      resMover_openFile|
| author    BSI              |
+-----*/
void resMover_openFile
(
char *fileName,    /* => Specific file to open. */
int currFile      /* => file number (0-left or 1-right). */
)
{
    int fileNameLength;
    RscFileHandle *rfHandleP;
    char *fileNameP = fileName;

    if (resMoverP->rscFileInfo [currFile].rfHandle != FILE_UNUSED)
    {
        resMover_displayError (4);
        return;
    }

    /* Make sure the file is not already open. */
    if (resMover_validateFileName(fileNameP))
    {
        resMover_displayError (3);
    }
}

```

```

        return;
    }

    rfHandleP = &resMoverP->rscFileInfo [currFile].rfHandle;
    if ((mdlResource_openFile (rfHandleP, fileNameP, RSC_READWRITE))
        == SUCCESS)
    {
        int itemNumber = (currFile == 0)
            ? CLASSLIST1ITEMNUMBER : CLASSLIST2ITEMNUMBER;

        /* Save name of successfully opened file. */
        strcpy (resMoverP->rscFileInfo [currFile].fileName, fileNameP);
        if ((fileNameLength = strlen(fileName)) > (LISTWIDTHS-2))
        {
            char tempfilename [LISTWIDTHS];

            strncpy (tempfilename, fileNameP, LISTWIDTHS - 17);
            tempfilename [LISTWIDTHS - 17] = '\0';
            strcat (tempfilename, "...");
            strcat (tempfilename, &fileNameP [fileNameLength-12]);
            fileNameP = tempfilename;
        }

        /* Display the newly opened file's name. */
        mdlDialog_itemSetLabel ((DialogBox *)resMoverP->dialogBoxP,
            itemNumber, fileNameP);
        resMover_buildClassList (currFile);
    }
else /* File open failed. */
{
    char errorMsgTemplate[80];
    char alertMsg[80];
    mdlResource_loadFromStringList (errorMsgTemplate,
        resMoverP->resMoverRfHandle,
        STRINGID_Errors, 2);
    sprintf (alertMsg, errorMsgTemplate, fileNameP);
    mdlDialog_openMessageBox(DIALOGID_MsgBoxOK,
        alertMsg, MSGBOX_ICON_WARNING);
}
}

/*-----+
| name      resMover_close          |
|          Close a resource file.    |
| author    BSI                     |
+-----*/
void resMover_close(int targetFile/* => file to close. */)
{
    int itemNumber;

```



```

ResourceFileInfo *rscFileInfoP =
    &resMoverP->rscFileInfo [targetFile];
char label[64];

if (rscFileInfoP->rfHandle == FILE_UNUSED)
    return;

mdlResource_closeFile (rscFileInfoP->rfHandle);
rscFileInfoP->fileName[0] = '\0';
rscFileInfoP->rfHandle = FILE_UNUSED;
mdlStringList_deleteMember (rscFileInfoP->classListP, 0, DELETE_ALL);
mdlStringList_deleteMember (rscFileInfoP->rscListP, 0, DELETE_ALL);
mdlDialog_listBoxNRowsChanged (rscFileInfoP->classListBoxHdrP);
mdlDialog_listBoxDrawContents(rscFileInfoP->classListBoxHdrP, -1, -1);
mdlDialog_listBoxNRowsChanged (rscFileInfoP->rscListBoxHdrP);
mdlDialog_listBoxDrawContents(rscFileInfoP->rscListBoxHdrP,
                                -1, -1);

if (targetFile == 0)
{
    mdlResource_loadFromStringList(label, NULL,
                                    STRINGID_Messages, 4);
    itemNumber = CLASSLIST1ITEMNUMBER;
}
else
{
    mdlResource_loadFromStringList (label, NULL,
                                    STRINGID_Messages, 5);
    itemNumber = CLASSLIST2ITEMNUMBER;
}

mdlDialog_itemSetLabel((DialogBox *)resMoverP->dialogBoxP,
                        itemNumber, label);
}

/*-----+
| name      resMover_quit          |
| author    BSI                    |
+-----*/
Private void resMover_quit(void)
cmdNumber CMD_RESMOVER_QUIT
{
    mdlSystem_exit(0, 1);
}

/*-----+
| Supporting Routines              |
+-----*/
/*-----+
| name      resMover_showRscList|

```

```

| author      BSI              11/90      |
+-----*/
Private void resMover_showRscList
(
int      currFile      /* => File whose list box is being manipulated. */
)
{
    int numResources, i, memberIndex, status;
    ULONG selectedClass, rscId;
    char asciiResourceId[20];
    ResourceFileInfo *rscFileInfoP = &resMoverP->rscFileInfo[currFile];
    boolean foundSelection;
    InfoFields *rscInfoFields;

    if (rscFileInfoP->rfHandle == FILE_UNUSED)
        return;

    selectedClass = rscFileInfoP->selClass;
    mdlResource_queryClass(&numResources, rscFileInfoP->rfHandle,
        selectedClass, RSC_QRY_COUNT, NULL);
    for (i=0; i<numResources; i++)
    {
        mdlResource_queryClass (&rscId, rscFileInfoP->rfHandle,
            selectedClass, RSC_QRY_ID, &i);

        mdlResource_queryClass (rscFileInfoP->selAlias,
            rscFileInfoP->rfHandle,
            selectedClass, RSC_QRY_ALIAS, &i);

        status = mdlStringList_insertMember(&memberIndex,
            rscFileInfoP->rscListP,
            APPEND_INDEX, 2);

        mdlStringList_getMember(NULL, (long *)&rscInfoFields,
            rscFileInfoP->rscListP, memberIndex);

        rscInfoFields->id = rscId;
        sprintf (asciiResourceId, "0x%08X", rscId);

        status = mdlStringList_setMember (rscFileInfoP->rscListP,
            memberIndex, asciiResourceId,
            (long *)rscInfoFields);

        status = mdlStringList_setMember (rscFileInfoP->rscListP,
            memberIndex+1, rscFileInfoP->selAlias,
            (long *)rscInfoFields);
    }

    status = mdlDialog_listBoxNRowsChanged(
        rscFileInfoP->rscListBoxHdrP);
    status = mdlDialog_listBoxDrawContents (rscFileInfoP->rscListBoxHdrP,

```

```

        -1, -1);
    }

/*-----+
| name      resMover_closeAndReopenFile|
| author    BSI                        |
+-----*/
int resMover_closeAndReopenFile(int targetFile)
{
    char saveFileName[128];

    /* Save the destination file name. */
    strcpy (saveFileName, resMoverP->rscFileInfo [targetFile].fileName);
    resMover_close (targetFile);
    resMover_openFile (saveFileName, targetFile);
    resMoverP->rscFileInfo [targetFile].selClass =
        resMoverP->rscFileInfo [targetFile].selClass;
    resMover_showRscList (targetFile);

    return SUCCESS;
}

/*-----+
| name      resMover_selectFile|
| author    BSI                11/90    |
+-----*/
Private int resMover_selectFile(DialogItemMessage *dimP)
{
    if (dimP->u.button.buttonTrans == BUTTONTRANS_UP)
    {
        switch (resMoverP->destinationFile)
        {
            case 0: /* Switching from left file to right file. */
                /* Set up to manipulate the first file. */
                resMoverP->sourceFile = 0;
                resMoverP->destinationFile = 1;
                mdlDialog_itemSetLabel((DialogBox *)
                    resMoverP->dialogBoxP,
                    BUTTON_ITEMNUM, ">>");
                mdlDialog_focusItemIndexSet((DialogBox *)
                    resMoverP->dialogBoxP,
                    CLASSLIST2ITEMNUMBER, FALSE);
                break;

            case 1: /* Switching from right file to left file. */
                /* Set up to manipulate the second file. */
                resMoverP->sourceFile = 1;
                resMoverP->destinationFile = 0;
                mdlDialog_itemSetLabel((DialogBox *)
                    resMoverP->dialogBoxP,

```

```

        BUTTON_ITEMNUM, "<<");
        mdlDialog_focusItemIndexSet((DialogBox *)
            resMoverP->dialogBoxP,
            CLASSLIST1ITEMNUMBER, FALSE);
        break;
    }
}
return SUCCESS;
}

/*-----+
| name      resMover_copyResource|
| author    BSI                  11/90      |
+-----*/
Private void resMover_copyResource(DialogItemMessage *dimP)
{
    int status, errMsgNumber, rscSize,
    src = resMoverP->sourceFile,
    dest = resMoverP->destinationFile;

void *rscP;
char *origAlias;

if (dimP->u.button.buttonTrans != BUTTONTRANS_UP)
    return;

if ((resMoverP->rscFileInfo[0].rfHandle == FILE_UNUSED) ||
    (resMoverP->rscFileInfo[1].rfHandle == FILE_UNUSED))
{
    resMover_displayError (5);
    return;
}

if (resMoverP->rscFileInfo [src].selRscId == NULL_ID)
{
    resMover_displayError (7);
    return;
}

/* Attempt to load the specified resource. */
if ((rscP = mdlResource_load (
    resMoverP->rscFileInfo [src].rfHandle,
    resMoverP->rscFileInfo [src].selClass,
    resMoverP->rscFileInfo [src].selRscId))
    == NULL)
{
    resMover_displayError (8);
    return;
}

mdlResource_query (&rscSize, rscP, RSC_QRY_SIZE);

```

```

    mdlResource_query (&origAlias, rscP, RSC_QRY_ALIAS);

status = mdlResource_add (
    resMoverP->rscFileInfo [dest].rfHandle,
    resMoverP->rscFileInfo [src].selClass,
    resMoverP->rscFileInfo [src].selRscId,
    rscP, rscSize, origAlias);

mdlResource_free (rscP);

if (status == SUCCESS)
{
    /* -----
       Note: Upon a successful "mdlResource_add", it IS NOT necessary
       to close and reopen the resource file. The file is reopened
       below as a simple way of rebuilding the class list and
       resource list in the dialog box.
       ----- */
    resMover_closeAndReopenFile (dest);
}
else
{
    switch (status)
    {
        case MDLERR_RSCWRITEERROR:
        case MDLERR_RSCFILEERROR:
            errMsgNumber = 9;
            break;

        case MDLERR_RSCWRITEVIOLATION:
            errMsgNumber = 10;
            break;

        case MDLERR_RSCALREADYEXISTS:
            errMsgNumber = 11;
            break;

        case MDLERR_RSCINSFMEM:
            errMsgNumber = 12;
            break;

        case MDLERR_RSCTYPEINVALID:
            errMsgNumber = 13;
            break;

        default:
            errMsgNumber = 14;
            break;
    }
    resMover_displayError (errMsgNumber);
}

```

```

    }

/*-----+
| name      resMover_deleteResource|
| author    BSI          11/90    |
+-----*/
Private void resMover_deleteResource(DialogItemMessage *dimP)
{
    int status, errMsgNumber, targetFile;

    if (dimP->u.button.buttonTrans != BUTTONTRANS_UP)
        return;

    /* See if at least one file is open. */
    if ((resMoverP->rscFileInfo[0].rfHandle == FILE_UNUSED) &&
        (resMoverP->rscFileInfo[1].rfHandle == FILE_UNUSED))
    {
        resMover_displayError (15);
        return;
    }

    if (resMoverP->rscFileInfo [0].selRscId != NULL_ID)
        targetFile = 0;
    else if (resMoverP->rscFileInfo [1].selRscId != NULL_ID)
        targetFile = 1;
    else
    {
        resMover_displayError (7);
        return;
    }

    if (resMover_displayError (17) == ACTIONBUTTON_CANCEL)
        return;

    /* Attempt to delete the specified resource. */
    status = mdlResource_delete (
        resMoverP->rscFileInfo [targetFile].rfHandle,
        resMoverP->rscFileInfo [targetFile].selClass,
        resMoverP->rscFileInfo [targetFile].selRscId);

    if (status == SUCCESS)
    {
        /* -----
        Note: Upon a successful "mdlResource_delete", it IS NOT
        necessary to close and reopen the resource file. The file is
        reopened below as a simple way of rebuilding the class list
        and resource list in the dialog box.
        ----- */
        resMover_closeAndReopenFile (targetFile);
    }
}
else

```

```

        {
        switch (status)
        {
            case MDLERR_RSCINUSE:
                errMsgNumber = 16;
                break;

            case MDLERR_RSCWRITEVIOLATION:
                errMsgNumber = 10;
                break;

            default:
                errMsgNumber = 14;
                break;
        }
        resMover_displayError (errMsgNumber);
    }
}

/*-----+
| name      resMover_processClassListClick|
| author    BSI          11/90      |
+-----*/
int resMover_processClassListClick(DialogItemMessage *dimP)
{
    DialogItem *diP = dimP->dialogItemP;
    ResourceFileInfo*rscFileInfoP;
    switch (dimP->u.button.upNumber)
    {
        case 1: /* Single click. */
            resMover_clearRscList (0);
            resMover_clearRscList (1);
            break;

        case 2: /* Double click. */
            if (dimP->u.button.clicked == TRUE)
            {
                switch (diP->id)
                {
                    case LISTID_File1_Classes:
                        resMover_getSelClass (0);
                        resMover_showRscList (0);
                        break;

                    case LISTID_File2_Classes:
                        resMover_getSelClass (1);
                        resMover_showRscList (1);
                        break;
                }
            }
    }
}

```

```

        }
        break;
    }
    return SUCCESS;
}

/*-----+
| name      resMover_processResourceListClick|
| author    BSI          11/90          |
+-----*/
int resMover_processResourceListClick(DialogItemMessage *dimP)
{
    DialogItem *diP = dimP->dialogItemP;
    switch (dimP->u.button.upNumber)
    {
        case 1: /* Single click */
        case 2: /* or double click. */
            switch (diP->id)
            {
                case LISTID_File1_Resources:
                    resMover_clearRscList (1);
                    resMover_getSelRscInfo (0);
                    break;

                case LISTID_File2_Resources:
                    resMover_clearRscList (0);
                    resMover_getSelRscInfo (1);
                    break;

            }
            break;
    }
    return SUCCESS;
}

/*-----+
|   Other Resource Manager Routines|
|   The routines listed below do not do anything with regard to   |
|   the functionality of the RESMOVER program. They are shown below|
|   as examples of the remaining Resource Manager calls not shown in|
|   the program, above.   |
+-----*/
/*-----+
| name      Example of "mdlResource_getRscIdByAlias"|
| author    BSI          8/90          |
+-----*/
Public int resMover_getDataBasedOnLogicalName
(
    void *resourceInfoP, /* <= Copy Resource Info to this memory area */

```



```

RscFileHandle rfHandle, /* => Resource file handle from earlier open */
ULONG resourceclass, /* => Resource type */
char *aliasName /* => Logical Name of resource */
)
{
    void *rscP;
    long rscID;
    int rscSize;

    /* Using the resource's alias name, retrieve actual resourceID. */
    if (mdlResource_getRscIdByAlias (&rscID, rfHandle, resourceclass,
        aliasName))
        return (ERROR);

    /* Now use the resourceID to load the resource. */
    if (rscP = mdlResource_load (rfHandle, resourceclass, rscID))
    {
        /* Find out how many bytes need to be copied.
        Note: The caller of this routine MUST BE SURE THAT THE MEMORY
        POINTED TO BY "*resourceInfoP" IS LARGE ENOUGH TO ACCEPT THE
        FULL RESOURCE. */
        if (mdlResource_query (&rscSize, rscP, RSC_QRY_SIZE) == SUCCESS)
        {
            memcpy (resourceInfoP, rscP, rscSize);
            mdlResource_free (rscP);
            return (SUCCESS);
        }
    }

    return (ERROR);
}

/*-----+
| name      Example of "mdlResource_write" and|
|           "mdlResource_resize"|
| author    BSI          8/90      |
+-----*/
Public int resMover_saveUsersFavoriteColor
(
    char *usersFavoriteColorP, /* => This is the user's favorite color. */
    int  maxResourceSize, /* => Maimum size of input string, above. */
    RscFileHandle rfHandle /* => Resource file where data is to be stored. */
)
{
    /* arbitrary resourceclass for color info. */
#define RTYPE_COLOR_CLASS 'COLR'

    /* arbitrary resourceID for favorite color. */
#define RSCID_FAVORITE_COLOR 0

```

```

char *existingColorP;
int rscSize;

/* First see if this user preference already exists in file. */
if ((existingColorP = mdlResource_load (rfHandle, RTYPE_COLOR_CLASS,
    RSCID_FAVORITE_COLOR)) != NULL)
{
    if (mdlResource_query(&rscSize, existingColorP, RSC_QRY_SIZE)
        != SUCCESS)
        return (ERROR);

    if (rscSize < maxResourceSize)
        existingColorP = mdlResource_resize (existingColorP,
            maxResourceSize);
    if (existingColorP != NULL)
    {
        memcpy(existingColorP, usersFavoriteColorP,
            maxResourceSize);
        mdlResource_write (existingColorP);
        mdlResource_free (existingColorP);
    }
    else
        return (ERROR);
}

else /* Since "favorite color" resource was not already in file,
    we will add it now. */
{
    return (mdlResource_add (rfHandle, RTYPE_COLOR_CLASS,
        RSCID_FAVORITE_COLOR, usersFavoriteColorP,
        maxResourceSize, NULL));
}

return (ERROR);
}

/*-----+
| name      resMover_unloadFunction|
| author    BSI          2/91      |
+-----*/
Private int resMover_unloadFunction()
{
    if (resMoverP->rscFileInfo [0].classListP != NULL)
        mdlStringList_destroy (resMoverP->rscFileInfo [0].classListP);
    if (resMoverP->rscFileInfo [1].classListP != NULL)
        mdlStringList_destroy (resMoverP->rscFileInfo [1].classListP);
    if (resMoverP->rscFileInfo [0].rscListP != NULL)
        mdlStringList_destroy (resMoverP->rscFileInfo [0].rscListP);
    if (resMoverP->rscFileInfo [1].rscListP != NULL)
        mdlStringList_destroy(resMoverP->rscFileInfo [1].rscListP);
    return SUCCESS;
}

```

```

    }

/*-----+
|       Dialog Box Hooks                               |
+-----*/
/*-----+
| name resMover_dialogHook      |
| authorBSI                     |
+-----*/
Private void resMover_dialogHook(DialogMessage *dmP)
{
    dmP->msgUnderstood = TRUE;
    switch (dmP->messageType)
    {
        case DIALOG_MESSAGE_CREATE:
            dmP->u.create.interests.mouses = TRUE;
            resMoverP->dialogBoxP = (int) dmP->db;
            resMoverP->sourceFile = 1;
            resMoverP->destinationFile = 0;
            resMoverP->rscFileInfo[0].rfHandle =
            resMoverP->rscFileInfo[1].rfHandle = FILE_UNUSED;
            break;

        case DIALOG_MESSAGE_INIT:
            break;

        case DIALOG_MESSAGE_DESTROY:
            mdlCEXpression_freeSet (setP);
            break;

        default:
            dmP->msgUnderstood = FALSE;
            break;
    }
}

/*-----+
| name resMover_listHook        |
| authorBSI                     |
+-----*/
Private void resMover_listHook(DialogItemMessage *dimP)
{
    DialogItem *diP = dimP->dialogItemP;
    StringList **strListP;

    dimP->msgUnderstood = TRUE;
    switch (dimP->messageType)
    {
        case DITEM_MESSAGE_CREATE:
            {

```

```

switch (diP->id)
{
case LISTID_File1_Classes:
case LISTID_File2_Classes:
{
int index;

if (diP->id == LISTID_File1_Classes)
    index = 0;
else
    index = 1;

resMoverP->rscFileInfo[index].classListBoxHdrP =
    diP->rawItemP;
strListP=
    &resMoverP->rscFileInfo[index].classListP;
if ((*strListP = mdlStringList_create (0, 2))
    == NULL)
    dimP->u.create.createFailed = TRUE;
if (mdlDialog_listBoxSetStrListP (diP->rawItemP,
    *strListP, 1)!= SUCCESS)
    dimP->u.create.createFailed = TRUE;
break;
}

case LISTID_File1_Resources:
case LISTID_File2_Resources:
{
int index;

if (diP->id == LISTID_File1_Resources)
    index = 0;
else
    index = 1;

resMoverP->rscFileInfo [index].rscListBoxHdrP =
    diP->rawItemP;
strListP =
    &resMoverP->rscFileInfo[index].rscListP;
resMoverP->rscFileInfo[index].selRscId =
    NULL_ID;
resMoverP->rscFileInfo[index].selAlias[0] =
    '\0';

if ((*strListP = mdlStringList_create (0, 2))
    == NULL)
    dimP->u.create.createFailed = TRUE;

if (mdlDialog_listBoxSetStrListP (diP->rawItemP,
    *strListP, RSC_LIST_COLUMNS)!= SUCCESS)
    dimP->u.create.createFailed = TRUE;
}
}

```

```

        break;
    }

    default:
        dimP->u.create.createFailed = TRUE;
        return;
    }
    break;
}

case DITEM_MESSAGE_DESTROY:
{
switch (diP->id)
{
case LISTID_File1_Classes:
    mdlStringList_destroy (
        resMoverP->rscFileInfo [0].classListP);
    resMoverP->rscFileInfo [0].classListP = NULL;
    break;
case LISTID_File2_Classes:
    mdlStringList_destroy (
        resMoverP->rscFileInfo [1].classListP);
    resMoverP->rscFileInfo [1].classListP = NULL;
    break;
case LISTID_File1_Resources:
    mdlStringList_destroy (
        resMoverP->rscFileInfo [0].rscListP);
    resMoverP->rscFileInfo [0].rscListP = NULL;
    break;
case LISTID_File2_Resources:
    mdlStringList_destroy (
        resMoverP->rscFileInfo [1].rscListP);
    resMoverP->rscFileInfo [1].rscListP = NULL;
    break;
}
break;
}

case DITEM_MESSAGE_BUTTON:
{
    switch (diP->id)
    {
        case LISTID_File1_Classes:
        case LISTID_File2_Classes:
            if (dimP->u.button.buttonTrans == BUTTONTRANS_UP)
                resMover_processClassListClick (dimP);
            break;
    }
}

```

```

        case LISTID_File1_Resources:
        case LISTID_File2_Resources:
            if (dimP->u.button.buttonTrans == BUTTONTRANS_UP)
                resMover_processResourceListClick (dimP);
            break;
        }
        break;
    }

default:
    dimP->msgUnderstood = FALSE;
}
}

/*-----
-+
|
| name resMover_buttonHook          |
|
| authorBSI                        |
|
+-----
*/
Public void resMover_buttonHook
(
DialogItemMessage    *dimP
)
{
    DialogItem *diP = dimP->dialogItemP;

    dimP->msgUnderstood = TRUE;
    switch (dimP->messageType)
    {
    case DITEM_MESSAGE_BUTTON:
        {
            switch (diP->id)
            {
            case PUSHBUTTONID_FilePtr:
                resMover_selectFile (dimP);
                break;

            case PUSHBUTTONID_Copy:
                resMover_copyResource (dimP);
                break;

            case PUSHBUTTONID_Del:
                resMover_deleteResource (dimP);

```

```

        break;
    }
    break;
}

default:
    dimP->msgUnderstood = FALSE;
    break;
}
}

/*-----+
| name      resMover_aliasTextHook|
| author    BSI          2/91      |
+-----*/
Public void resMover_aliasTextHook(DialogItemMessage *dimP)
{
    DialogItem *diP = dimP->dialogItemP;
    DialogBox *dbP = dimP->db;

    dimP->msgUnderstood = TRUE;
    switch (dimP->messageType)
    {
        case DITEM_MESSAGE_CREATE:
        {
            dimP->u.create.createFailed = FALSE;
            break;
        }

        case DITEM_MESSAGE_STATECHANGED:
        {
            switch (dimP->dialogItemP->id)
            {
                case TEXTID_ChangeAlias:
                    resMover_changeAlias ();
                    break;

                default:
                    dimP->msgUnderstood = FALSE;
                    break;
            }
            break;
        }

        case DITEM_MESSAGE_DESTROY:
        {
            break;
        }

        default:

```

```

        dimP->msgUnderstood = FALSE;
    }
}

/*-----+
|       Command Handling routines                               |
+-----*/
/*-----+
| name          resMover_create                                |
|              Create a resource file.                        |
| author        BSI                                           |
+-----*/
Private void resMover_create()
cmdNumber CMD_RESMOVER_CREATE
{
    int status, currFile = resMoverP->destinationFile;
    char titleString[80], filename[128];

    if (resMoverP->rscFileInfo [currFile].rfHandle != FILE_UNUSED)
    {
        resMover_displayError (4);
        return;
    }

    mdlResource_loadFromStringList(titleString,
        resMoverP->resMoverRfHandle, STRINGID_Messages, 0);

    status = mdlDialog_fileCreate (filename, NULL, 0, NULL, "*.rsc",
        "MS_RSRCPATH", titleString);

    if ((status == SUCCESS) && (filename[0] != '\0'))
    {
        mdlResource_createFile (filename, "", 0L);
        resMover_openFile (filename, currFile);
    }
}

/*-----+
| name          resMover_closeCurrentFile                      |
|              Close the currently selected resource file.    |
| author        BSI                                           |
+-----*/
Private void resMover_closeCurrentFile()
cmdNumber CMD_RESMOVER_CLOSE
{
    resMover_close (resMoverP->destinationFile);
}

/*-----+
| name          resMover_getAndOpenFile                        |
|              Get a resource file name and open it.          |

```



```

| author          BSI |
+-----*/
Private void resMover_getAndOpenFile()
cmdNumber CMD_RESMOVER_OPEN
{
    int      status, currFile = resMoverP->destinationFile;
    char      titleString[80], filename[128];

    if (resMoverP->rscFileInfo [currFile].rfHandle != FILE_UNUSED)
    {
        resMover_displayError (4);
        return;
    }

    mdlResource_loadFromStringList (titleString,
        resMoverP->resMoverRfHandle, STRINGID_Messages, 1);

    /* Obtain a file name from the user. */
    status = mdlDialog_fileOpen (filename, NULL, 0, NULL,
        "*.rsc", "MS_RSRCPATH", titleString);

    if ((status == SUCCESS) && (filename[0] != '\0'))
        resMover_openFile (filename, currFile);
}

/*-----+
| name main          |
| authorBSI          |
+-----*/
Private DialogHookInfo uHooks[] =
{
    {HOOKDIALOGID_ResMover,resMover_dialogHook},
    {HOOKITEMID_List_ResMover,      resMover_listHook},
    {HOOKITEMID_Button_ResMover,    resMover_buttonHook},
    {HOOKITEMID_ChangeAlias,      resMover_aliasTextHook},
};

int main(int argc, char *argv[])
{
    DialogBox      *dbP;

    if ((resMoverP = malloc (sizeof(ResMoverGlobals))) == NULL)
        mdlSystem_exit (1, 1);
    memset (resMoverP, 0, sizeof(ResMoverGlobals));

    /* -----
        Open the resource file from which this application was loaded.
        This is accomplished by specifying "NULL" as the filename to the
        mdlResource_openFile call.
        ----- */
    if (mdlResource_openFile (&resMoverP->resMoverRfHandle, NULL, 0)

```

```

        != SUCCESS)
    {
        char buffer[128];

        mdlResource_loadFromStringList(buffer, NULL,
                                       STRINGID_Errors, 20);

        mdlDialog_openMessageBox(DIALOGID_MsgBoxOK,
                                buffer, MSGBOX_ICON_WARNING);

        mdlSystem_exit (1, 1);
    }

/* -----
Load the command table for this application. Specifying NULL as the
file will cause the Resource Manager to look for the command table
in the Resource File opened above.
----- */
if (mdlParse_loadCommandTable (NULL) == NULL)
    mdlOutput_rscPrintf(MSG_ERROR,NULL, STRINGID_Errors, 0);

/* set up the variables we are going to set from dialog boxes */
setP = mdlCEExpression_initializeSet (VISIBILITY_DIALOG_BOX, 0, TRUE);
mdlDialog_publishComplexPtr (setP, "resmoverglobals",
                             "resMoverP", &resMoverP);

/* publish our hooks */
mdlDialog_hookPublish(
    sizeof(uHooks)/sizeof(DialogHookInfo), uHooks);

/* start the dialog box */
if ((dbP=(DialogBox *)mdlDialog_open(NULL,DIALOGID_ResMover))== NULL)
    mdlOutput_rscPrintf(MSG_ERROR,NULL, STRINGID_Errors, 1);

mdlSystem_setFunction(SYSTEM_UNLOAD_PROGRAM,
                     resMover_unloadFunction);

return SUCCESS;
}

```

parse.mc

```

/*-----
-+
|                                     |
| Copyright (c) 1985-91; Bentley Systems, Inc., All rights reserved.|
|                                     |
| "MicroStation", "MDL", and "MicroCSL" are trademarks of Bentley|
| Systems, Inc. and/or Intergraph Corporation. |
|                                     |
| This program is proprietary and unpublished property of Bentley |

```

```

| Systems Inc. It may NOT be copied in part or in whole on any medium,
|
| either electronic or printed, without the express written consent|
| of Bentley Systems, Inc. |
|
+-----+
*/
/*-----+
+
|
|      $Workfile:  scanexec.tmp  $
|      $Revision:  5.1  $  $Date:  30 Jul 1991 14:24:46  $
|
+-----+
*/
/*-----+
+
|
|      parse.mc - examples for the mdlParse_ functions.|
|
|      This file is intended as an adjunct to the MDL manual to|
|      illustrate MDL built-in function calling conventions and parameter|
|      values. While it can be compiled, it does NOT, on its own,|
|      constitute a workable MDL example.|
|
+-----+
*/
/*-----+
+
|
|      Include Files
|
+-----+
*/
#include <mdl.h>      /* system include files */
#include <userfnc.h>
#include <msinputq.h>
#include <cmdlist.h>
#include <cmdclass.h>

/*-----+
+
|
|      Function declarations
|
+-----+
*/

```

```

/*-----
-+
|
| name main
|
| authorBSI          12/90
|
+-----
*/
main (void)
{
    int    userParseFunction ();

    mdlParse_setFunction (PARSE_HANDLE_STRING, userParseFunction);
}

/*-----
-+
|
| Private Utility Routines|
|
+-----
*/
/*-----
-+
|
| name handleCommands
|
| authorBSI          12/90
|
+-----
*/
void commandHandler
(
char *unparsedP
) cmdNumber 1, 2
{
    mdlOutput_printf (MSG_MESSAGE, "Command %d, unparsed %s",
        mdlCommandNumber, unparsedP);
}

/*-----
-+
|
| name createCommandIQe1
|
+-----

```

```

| authorBSI          12/90    |
|                               |
+-----+
*/
Private void createCommandIQel
(
Inputq_element*queueElementP,
int      commandNumber,
boolean  microStationCommand,
int      commandClass,/* PLACEMENT, VIEWING, etc. */
char     *unparsedP
)
{
    memset (queueElementP, '\0', sizeof (Inputq_header) +
            sizeof (Inputq_command));
    queueElementP->hdr.cmdtype= CMDNUM;

    queueElementP->hdr.bytes= sizeof (Inputq_header) +
        sizeof (Inputq_command);

    /* The size already includes the EOS. Add in the count of non-zero
       bytes in the string.
    */
    if (unparsedP)
        queueElementP->hdr.bytes += strlen (unparsedP);

    queueElementP->hdr.source= FROM_MDL;

    /*  commandNumber should be a MicroStation command number taken
        from cmdlist.h or should be a number associated with an MDL
        application with a cmdNumber directive
    */
    queueElementP->u.cmd.command = commandNumber;

    /*  Identify type of functionality provided by the command. This
        information may be used by command filters. */
    queueElementP->u.cmd.class = commandClass;

    /*-----+
    +-
    the
    Set the task ID in the command structure to tell MicroStation what
    task should execute the command. Note that nothing is stored in
    the
    task ID in the header of the queue element. The task ID in the
    header is used to address the queue element to a specific task.

```

```

+-----+
-*/
    /* Set the task ID */
    if (microStationCommand)
        strcpy (queueElementP->u.cmd.taskId, ustnTaskId);
    else
        strcpy (queueElementP->u.cmd.taskId, mdlSystem_getCurrTaskID ());

    strcpy (queueElementP->u.cmd.unparsed, unparsedP);
}

/*-----+
+
|
| name userParseFunction          |
|
| authorBSI          12/90      |
|
| userParseFunction illustrates how MicroStation responds to      |
| different return values from the parse function.|
|
| AA - queues command number 1  |
| A  - queue command number 2   |
| DX - returns ambiguous command |
| ZZ - returns ambiguous command |
|
+-----+
*/
Private int userParseFunction
(
Inputq_element*queueElementP, /* <= MicroStation will queue it */
char    *stringP
)
{
    char    buffer [5];

    strncpy (buffer, stringP, 5);
    buffer [4] = '\0';
    strupr (buffer);

    if (!strcmp (buffer, "AA=", 3))
/* Replaces a MicroStation command. Note that this can only be
   done for the 2-character codes. */
createCommandIQel (queueElementP, 1, FALSE, PARAMETERS, stringP+3);
    else if (!strcmp (buffer, "A=", 2))
/* Overrides MicroStation's ambiguous error. */
createCommandIQel (queueElementP, 2, FALSE, PARAMETERS, stringP+2);

```

```

        else if (!strncmp (buffer, "DX=", 3))
/* MicroStation will override the parse functions ambiguous error */
return -2;
        else if (!strncmp (buffer, "ZZ=", 3))
/* MicroStation will display an "ambiguous command" */
return -2;
        else
return -1;

return SUCCESS;
}

```

chngtxt.mc

```

/*-----+
| Copyright (1993-1995) Bentley Systems, Inc., All rights reserved.|
| "MicroStation" is a registered trademark and "MDL" and "MicroCSL"|
| are trademarks of Bentley Systems, Inc.      |
| Limited permission is hereby granted to reproduce and modify this|
| copyrighted material provided that the resulting code is used only |
| in conjunction with Bentley Systems products under the terms of the|
| license agreement provided therein, and that this notice is retained|
|                                     |
| in its entirety in any such reproduction or modification.|
+-----*/
/*-----+
| Function -                               |
| chngtxt.mc - example MDL program to change text in a design file.|
+-----*/

#include    < tcb.h>
#include    < mselems.h>
#include    < global.h>
#include    < scanner.h>
#include    < msinputq.h>
#include    < userfnc.h>
#include    < mdl.h>
#include    < cexpr.h>
#include    < rscdefs.h>
#include    < dlogitem.h>
#include    < cmdlist.h>
#include    < string.h>
#include    < dlogids.h>

#include    "chtxtcmd.h"/* Generated by resource compiler (rcomp) */
#include    "chtxtdlg.h"/* Need to know dialog id to open */

```

```

#include    <dlogman.fdf>
#include    <msview.fdf>
#include    <msrmatrix.fdf>
#include    <msscan.fdf>
#include    <msrsrc.fdf>
#include    <msparse.fdf>
#include    <mssystem.fdf>
#include    <msoutput.fdf>
#include    <mslocate.fdf>
#include    <msstate.fdf>
#include    <mselemen.fdf>
#include    <msselect.fdf>
#include    <mscexpr.fdf>
#include    <mselmdsc.fdf>
#include    <msmisc.fdf>
#include    <msvec.fdf>
#include    <msstring.fdf>

/*-----+
|   Type Definitions       |
+-----*/
/*-----+
|   Private Global variables|
+-----*/
static ChangeTextInfo chTextInfo;
static ULONG nextElm, currAddr[2];
static long currOffset;
static int commandName;
static DialogBox*changeTextDBP;
Public int    stopState;

/*-----+
|   Local function declarations |
+-----*/
/*-----+
|   Major Public Code Section|
+-----*/
/*-----+
| name  compareTextStrings    |
| authorBSI                    8/90    |
+-----*/
Private int compareTextStrings(char *searchStr, char *patternStr,
char **start, char **end)
{
    static char *space = " ";
    char  searchBuf [256], patternBuf [256];

    /*-----+
        Regular expression search takes priority if requested
    */

```



```

-----*/
if (chTextInfo.regularExps)
{
    if (mdlString_matchREExtended (searchStr, patternStr, start,
                                   end, &stopState))
        return ERROR;
    else
        return SUCCESS;
}

/*-----
Prepare for Whole Word search if necessary
-----*/
searchBuf [0] = 0;
patternBuf [0] = 0;

if (chTextInfo.wholeWords)
{
    strcat (searchBuf, space);
    strcat (patternBuf, space);
}

strcat (searchBuf, searchStr);
strcat (patternBuf, patternStr);

if (chTextInfo.wholeWords)
{
    strcat (searchBuf, space);
    strcat (patternBuf, space);
}

/*-----
Prepare for case insensitive search if necessary
-----*/
if (!chTextInfo.matchCase)
{
    strupr (searchBuf);
    strupr (patternBuf);
}

/*-----
Attempt to match substrings
-----*/
if ((*start = strstr (searchBuf, patternBuf)) == NULL)
    return ERROR;

*start = searchStr + (*start - searchBuf);
*end   = *start + strlen(patternStr);
return SUCCESS;
}

```

```

/*-----+
| name turnChangeButton      |
| author      BSI      6/93|
+-----*/
Private void turnChangeButton(int onOff)
{
    DialogItem *changeDiP = mdlDialog_itemGetByTypeAndId (changeTextDBP,
        RTYPE_PushButton, PUSHBUTTONID_Change, 0);

    mdlDialog_itemSetEnabledState (changeTextDBP, changeDiP->itemIndex,
        onOff, FALSE);
}

/*-----+
| name startNewFind          |
| authorBSI      6/93      |
+-----*/
Private void startNewFind(void)
{
    nextElm = 0;
    turnChangeButton (FALSE);
}

/*-----+
| name unloadFunction        |
| authorBSI      8/90      |
| MicroStation calls this function prior to unloading the |
| application CHNGTXT. MicroStation knows about this function |
| because it was identified in the call: |
|      mdlSystem_setFunction (SYSTEM_UNLOAD_APP, unloadFunction); |
+-----*/
Private int unloadFunction(void)
{
    RscFileHandle    userPrefsH;
    ChangeTextInfo   *textRscP;

    /* Open userpref.rsc to hold our small pref resource */
    mdlDialog_userPrefFileOpen (&userPrefsH, TRUE);
    textRscP = (ChangeTextInfo *)mdlResource_load (NULL, RTYPE_chText,
        RSCID_chTextPrefs);

    if (!textRscP)
    {
        /* Our pref resource does not exist, so add it */
        mdlResource_add (userPrefsH, RTYPE_chText, RSCID_chTextPrefs,
            &chTextInfo, sizeof(ChangeTextInfo), NULL);
    }
    else
    {
        /* There is a lot of wasted space in this resource */

```

```

        *textRscP = chTextInfo;

        /* Write out and free the updated resource */
        mdlResource_write (textRscP);
        mdlResource_free (textRscP);
    }

    /* Clean up */
    mdlResource_closeFile (userPrefsH);
    return FALSE;
}

/*-----+
| name  changeText_dialogHook      |
| authorBSI          1/93          |
+-----*/
Private void changeText_dialogHook(DialogMessage *dmP)
{
    dmP->msgUnderstood = TRUE;
    switch (dmP->messageType)
    {
        case DIALOG_MESSAGE_DESTROY:
        {
            /* unload this mdl task when the Change Text is closed */
            mdlDialog_cmdNumberQueue (FALSE, CMD_MDL_UNLOAD,
                                     mdlSystem_getCurrTaskID(), TRUE);
            break;
        }

        default:
            dmP->msgUnderstood = FALSE;
            break;
    }
}

/*-----+
| name  changeText_wholeWordsHook  |
| authorBSI          6/93          |
+-----*/
Public void changeText_wholeWordsHook(DialogItemMessage *dimP)
{
    dimP->msgUnderstood = TRUE;
    switch (dimP->messageType)
    {
        case DITEM_MESSAGE_STATECHANGED:
        {
            /* turn off Regular Expressions if either
            other toggle pressed */

            DialogItem *regularExpsDiP =

```

```

        mdlDialog_itemGetByTypeAndId (dimP->db,
            RTYPE_ToggleButton, TOGGLEID_RegularExps, 0);
        chTextInfo.regularExps = 0;
        mdlDialog_itemSynch (dimP->db, regularExpsDiP->itemIndex);
        break;
    }
    default:
    {
        dimP->msgUnderstood = FALSE;
        break;
    }
}

}

/*-----+
| name  changeText_matchCaseHook|
| authorBSI      6/93      |
+-----*/
Private void changeText_matchCaseHook(DialogItemMessage *dimP)
{
    dimP->msgUnderstood = TRUE;
    switch (dimP->messageType)
    {
        case DITEM_MESSAGE_STATECHANGED:
        {
            /* turn off Regular Expressions if either
            other toggle pressed */

            DialogItem *regularExpsDiP =
                mdlDialog_itemGetByTypeAndId (dimP->db,
                    RTYPE_ToggleButton, TOGGLEID_RegularExps, 0);
            chTextInfo.regularExps = 0;
            mdlDialog_itemSynch (dimP->db, regularExpsDiP->itemIndex);
            break;
        }
        default:
        {
            dimP->msgUnderstood = FALSE;
            break;
        }
    }
}

/*-----+
| name  changeText_regularExpsHook|
| authorBSI      6/93      |
+-----*/
Private void changeText_regularExpsHook(DialogItemMessage *dimP)
{

```

```

dimP->msgUnderstood = TRUE;
switch (dimP->messageType)
{
    case DITEM_MESSAGE_STATECHANGED:
    {
        /* turn on Match Case and turn off Whole Words when
        /* Regular Expressions is turned on */
        if (chTextInfo.regularExps)
        {
            DialogItem *matchCaseDiP =
            mdlDialog_itemGetByTypeAndId (dimP->db,
            RTYPE_ToggleButton, TOGGLEID_MatchCase, 0);
            DialogItem *wholeWordsDiP =
            mdlDialog_itemGetByTypeAndId (dimP->db,
            RTYPE_ToggleButton, TOGGLEID_WholeWords, 0);
            chTextInfo.matchCase = TRUE;
            mdlDialog_itemSynch (dimP->db,
            matchCaseDiP->itemIndex);
            chTextInfo.wholeWords = 0;
            mdlDialog_itemSynch (dimP->db,
            wholeWordsDiP->itemIndex);
        }
        break;
    }
    default:
    {
        dimP->msgUnderstood = FALSE;
        break;
    }
}

}

/*-----+
| name  changeText_inCellsHook |
| authorBSI          6/93      |
+-----*/
Private void changeText_inCellsHook(DialogItemMessage *dimP)
{
    dimP->msgUnderstood = TRUE;
    switch (dimP->messageType)
    {
        case DITEM_MESSAGE_STATECHANGED:
        {
            startNewFind ();
            break;
        }
        default:

```

```

        {
            dimP->msgUnderstood = FALSE;
            break;
        }
    }

/*-----+
| name  changeText_findHook      |
| authorBSI      6/93            |
+-----*/
Private void changeText_findHook(DialogItemMessage *dimP)
{
    switch (dimP->messageType)
    {
        case DITEM_MESSAGE_KEYSTROKE:
        {
            startNewFind ();
            break;
        }
    }
}

/*-----+
| name  startChangeTextCommand   |
| authorBSI      7/90            |
+-----*/
Private void startChangeTextCommand(int doNewFind)
{
    /* The text strings are always taken from the dialog box. Display
    the dialog box if it it not already displayed. */
    if (! mdlDialog_find (DIALOGID_ReplaceText, NULL))
        changeTextDBP = mdlDialog_open (NULL, DIALOGID_ReplaceText);
    if (doNewFind)
        startNewFind ();
}

/*-----+
| name  changeTextElement        |
| authorBSI      8/89            |
+-----*/
Private int changeTextElement(MSElementUnion *element,int doChange)
{
    char tempText[256], oldText[256], newText[256];
    char message [512], *strP, *start, *end;
    int status, stringChanged=FALSE, strlenNewString;
    int retval = SUCCESS, nChars, nEDFs;

    if (mdlElement_getType (element) != TEXT_ELM

```

```

        || mdlText_extractString (oldText, element))
    return ERROR;

if (mdlText_extract (NULL, NULL, /* origin, userOrigin */
    &nEDFs, NULL, /* numEdfields, edFields */
    oldText, /* string */
    NULL, NULL, NULL, NULL, NULL,
    element) != SUCCESS)
    return ERROR;

if (*chTextInfo.oldString == 0)
    return ERROR;

if (chTextInfo.fractions)
    mdlText_expandString (oldText, oldText,
        sizeof(oldText), FRACTIONS);

nChars = strlen(oldText);
strlenNewString = strlen (chTextInfo.newString);

strcpy (newText, oldText);
strP = newText;
stopState = 0;
while (!stopState && (!compareTextStrings (strP,
    chTextInfo.oldString, &start, &end)))
{
    if ((strlen (newText) + strlenNewString + strlen (end)) >=
        sizeof (newText))
    {
        retval = ERROR;
        sprintf (message, "Unable to insert '%s' after '%s'",
            chTextInfo.newString, newText);

        mdlDialog_openMessageBox(DIALOGID_MsgBoxOK,
            message, MSGBOX_ICON_WARNING);

        break;
    }

    strcpy (tempText, end);
    strcpy (start, chTextInfo.newString);
    if (tempText[0])
        strcat (strP, tempText);
    strP = start + strlen (chTextInfo.newString);
    stringChanged = TRUE;
}

if (!stringChanged)
    return ERROR;

if (nEDFs && strlen(newText) < nChars)
{
    int nNewLen = strlen (newText);

```

```

        strP = newText + nNewLen;
        while (nNewLen < nChars)
        {
            *strP = ' ';
            ++strP;
            ++nNewLen;
        }
        *strP = '\0';
    }

    if (chTextInfo.fractions)
        mdlText_compressString (newText, newText, sizeof(oldText),
                                FRACTIONS);

    if (doChange)
        mdlText_create (element, element, newText, NULL, NULL,
                        NULL, NULL, NULL);

    return  retval;
}

/*-----+
| name  editText                                |
| authorBSI          7/90          |
| MicroStation calls editText to modify an element because|
| the address of editText was passed to MicroStation in a |
| call to mdlModify_elementSingle or mdlModify_elementMulti. |
+-----*/
Private int editText(MSElementUnion *el, int *numChanged)
{
    int status;

    status = changeTextElement (el, TRUE);

    /* if the caller wants to know how many text elements were changed
    increment his counter here */
    if (status == SUCCESS && numChanged)
        (*numChanged)++;

    return (status ? 0 : MODIFY_STATUS_REPLACE);
}

/*-----+
| name  changeText_accept - called for data point on change|
|                               text single command|
| authorBSI          3/89          |
+-----*/
Private void changeText_accept(void)
{
    ULong    filePos, compOffset;
    int      currFile;

```



```

filePos = mdlElement_getFilePos (FILEPOS_CURRENT, &currFile);

/* If there is a selection set active, process all elements in the
set. Of course, this processes all component text elements of a text
node. If no selection set is active, process ONLY the identified
element. This makes this command not apply to graphic groups (by
design). */
if (mdlSelect_isActive())
{
    mdlModify_elementMulti (currFile, filePos,
        MODIFY_REQUEST_NOHEADERS, MODIFY_ORIG, editText,
        NULL, TRUE);
}
else
{
    compOffset = mdlElement_getFilePos(FILEPOS_COMPONENT,
        NULL) - filePos;
    mdlModify_elementSingle (currFile, filePos,
        MODIFY_REQUEST_ONLYONE, MODIFY_ORIG, editText,
        NULL, compOffset);
}

/* restart the element location process */
mdlLocate_restart (FALSE);
}

/*-----+
| name setTextSearchType          |
| authorBSI          7/90          |
+-----*/
Private void setTextSearchType(void)
{
    int nSearchTypes = 2, searchType [3];
    searchType [0] = TEXT_ELM;
    searchType [1] = TEXT_NODE_ELM;
    if (chTextInfo.cellText)
    {
        searchType [2] = CELL_HEADER_ELM;
        nSearchTypes = 3;
    }

    /* set search criteria to find nothing */
    mdlLocate_noElemNoLocked ();

    /* then add text elements and text nodes to list */
    mdlLocate_setElemSearchMask (nSearchTypes, searchType);
}

/*-----+
| name doChangeSingle              |

```

```

| authorBSI          8/89      |
+-----*/
Private void doChangeSingle(void)
{
    /* set up the MicroStation search mask to find only text elements */
    setTextSearchType();

    /* tell MicroStation we want to start a "modification" command */
    mdlState_startModifyCommand (doChangeSingle, changeText_accept, NULL,
                                NULL, NULL, commandName, 1, TRUE, FALSE);

    /* set the internal locate pointers to start at beginning of file */
    mdlLocate_init ();
}

/*-----+
| name doChangeCurrent          |
| authorBSI          6/93      |
+-----*/
Private void doChangeCurrent(void)
{
    /*-----+
    Call mdlState_startPrimitive to terminate the current command,
    identify the new command's name, and set undo's command boundary.
    +-----*/
    mdlState_startPrimitive (NULL, NULL, 4, 0);

    mdlModify_elementSingle (0, currAddr[0], MODIFY_REQUEST_ONLYONE,
                            MODIFY_ORIG, editText, NULL, currOffset);
    turnChangeButton (FALSE);

    /*-----+
    There is no logical way to continue the command, so start the
    default command.
    +-----*/
    mdlState_startDefaultCommand ();
}

/*-----+
| name changeAllElements        |
| authorBSI          8/89      |
+-----*/
Private int changeAllElements(void)
{
    ULong elemAddr[50], eofPos, filePos;
    int scanWords, numChanged=0, status, i, numAddr;
    Scanlist scanList;

    mdlScan_initScanlist (&scanList);
    mdlScan_noRangeCheck (&scanList);

```

```

scanList.scantype      = ELEMTYPE | NESTCELL;
scanList.extendedType  = FILEPOS;
scanList.typmask[0]    = TMSK0_TEXT_NODE;
scanList.typmask[1]    = TMSK1_TEXT;

if (chTextInfo.cellText)
    scanList.typmask[0] |= TMSK0_CELL_HEADER;

eofPos = mdlElement_getFilePos (FILEPOS_EOF, NULL);
filePos = 0L;
mdlScan_initialize (0, &scanList);

/* loop through all text elements in file */
do
{
    scanWords = sizeof(elemAddr)/sizeof(short);
    status = mdlScan_file (elemAddr, &scanWords, sizeof(elemAddr),
        &filePos);
    numAddr  = scanWords / sizeof(short);
    for (i=0; i<numAddr; i++)
    {
        if (elemAddr[i] >= eofPos)
            break;

        mdlModify_elementSingle (0, elemAddr[i],
            MODIFY_REQUEST_NOHEADERS,
            MODIFY_ORIG, editText, &numChanged, 0L);
    }
} while (status == BUFF_FULL);
return numChanged;
}

/*-----+
| name doChangeAll                               |
| authorBSI      8/89      |                     |
+-----*/
Private void doChangeAll(void)
{
    int numChanged;

    /*-----+
    Call mdlState_startPrimitive to terminate the current command,
    identify the new command's name, and set undo's command boundary.
    +-----*/
    mdlState_startPrimitive (NULL, NULL, 3, 0);

    /* Now change the elements. */
    numChanged = changeAllElements ();

    /*-----+
    Generate a message using string 3 in MessageList 1 in chtxtmsg.r

```

```

as the format string.
+-----*/
mdlOutput_rscPrintf (MSG_ERROR, NULL, 1, 3, numChanged);

/*-----+
There is no logical way to continue the command, so start the
default command.
+-----*/
mdlState_startDefaultCommand ();
}

/*-----+
| name showTextElement          |
| authorBSI          8/90      |
+-----*/
Private int showTextElement(MSElementUnion *element)
{
    char msgStr[80];
    Dpoint3d textOrigin, viewRange[2];
    MTextSize textSize;
    RotMatrix textRMatrix, invTextMatrix;

    /* Find text element's position and orientation */
    mdlText_extract (&textOrigin, NULL, NULL, NULL, NULL, &textRMatrix,
        NULL, NULL, NULL, &textSize, element);

    /* Set up view to show text element */
    viewRange[0].x = -2.0 * textSize.height;
    viewRange[0].y = -4.0 * textSize.height;
    viewRange[1].x = textSize.width + 2.0 * textSize.height;
    viewRange[1].y = textSize.height * 4.0;
    viewRange[0].z = viewRange[1].z = 0.0;

    mdlRMatrix_rotatePointArray (viewRange, &textRMatrix, 2);
    mdlVec_addPointArray (viewRange, &textOrigin, 2);

    /* Display text element */
    mdlRMatrix_invert (&invTextMatrix, &textRMatrix);
    mdlView_setArea (tcb->lsvw, viewRange, &viewRange[0],
        textSize.width, textSize.width/2.0, &invTextMatrix);
    if (mdlView_updateSingle (tcb->lsvw))
    {
        /* Unable to update view. */
        mdlOutput_rscPrintf (MSG_ERROR, NULL, 1, 4, tcb->lsvw+1);
    }

    return SUCCESS;
}

/*-----+
| name matchTextElement          |

```

```

| authorBSI                      8/93      |
+-----*/
Private int matchTextElement(MSElement *pElement, int *params,
int op, long offset)
{
    if (changeTextElement (pElement, FALSE) == SUCCESS)
    {
        currOffset = offset;
        showTextElement (pElement);
        turnChangeButton (TRUE);
        return 1;
    }
    return 0;
}

/*-----+
| name findNextElement          |
| authorBSI                    6/93      |
+-----*/
Private void findNextElement(void)
{
    Scanlist    scanList;
    int scanWords, status;
    MSElementDescr *pElmD;

    mdlScan_initScanlist (&scanList);
    mdlScan_noRangeCheck (&scanList);

    scanList.scantype      = ELEMTYPE | NESTCELL | ONEELEM;
    scanList.extendedType  = FILEPOS;
    scanList.typmask[0]    = TMSK0_TEXT_NODE;
    scanList.typmask[1]    = TMSK1_TEXT;
    scanList.sector        = DGN_BLOCK (nextElm);
    scanList.offset        = DGN_OFFSET (nextElm);

    if (chTextInfo.cellText)
        scanList.typmask[0] |= TMSK0_CELL_HEADER;

    mdlScan_initialize (0, &scanList);

    while (1)
    {
        scanWords = sizeof(currAddr) / sizeof(short);
        status= mdlScan_file (currAddr, &scanWords,
            sizeof(currAddr), &nextElm);

        if (status != BUFF_FULL)
        {
            char msg [128];
            mdlResource_loadFromStringList (msg, NULL, 1, 5);

```

```

        mdlDialog_openMessageBox(DIALOGID_MsgBoxOK,
                                msg, MSGBOX_ICON_INFORMATION);
        startNewFind ();
        break;
    }

    nextElm = mdlElmdscr_read (&pElmD, currAddr[0], 0,
                              FALSE, &currAddr[0]);

    status = mdlElmdscr_operation (pElmD, matchTextElement,
                                  NULL, ELMD_ALL_ONCE);

    mdlElmdscr_freeAll (&pElmD);

    /* if we got a match, quit */
    if (status)
    {
        break;
    }
}

/*-----+
| name  changeTextContentOp      |
| authorBSI      8/89           |
+-----*/
Private int changeTextContentOp(void)
{
    ULong filePos;

    filePos = mdlElement_getFilePos (FILEPOS_CURRENT, NULL),
    mdlModify_elementSingle (0, filePos, MODIFY_REQUEST_NOHEADERS,
                            MODIFY_ORIG, editText, NULL, 0L);

    return  SUCCESS;
}

/*-----+
| name  doChangeFence            |
| authorBSI      8/89           |
+-----*/
Private void doChangeFence(void)
{
    /* set the internal search filter to only find text and text nodes */
    setTextSearchType();

    startNewFind ();
    mdlState_startFenceCommand (changeTextContentOp, NULL, NULL,
                               NULL, 2, commandName, FENCE_CLIP_ORIG);
}

/*-----+
| name  confirmTextChange        |

```

```
| authorBSI          8/90      |
+-----*/
Private int confirmTextChange(MSElementUnion *element,
char *oldString, char *newString)
{
    int actionButton;
    char buffer[300];

    showTextElement(element);

    sprintf(buffer, "Replace %s with %s?", oldString, newString);

    /* Display an alert and return result to caller */
    actionButton = mdlDialog_openMessageBox(DIALOGID_MsgBoxOKCancel,
        buffer, MSGBOX_ICON_QUESTION);
    if (actionButton != ACTIONBUTTON_OK)
        return ERROR;

    return SUCCESS;
}

/*-----+
| name singleChangeText      |
| authorBSI          8/89      |
+-----*/
Private voidsingleChangeText(void)
cmdNumberCMD_CHANGE_TEXT_SINGLE
{
    /* This function is called when CHANGE TEXT SINGLE is keyed into the
    command window */

    commandName = 1;
    startChangeTextCommand (TRUE);

    doChangeSingle();
}

/*-----+
| name fenceChangeText      |
| authorBSI          8/89      |
+-----*/
Private voidfenceChangeText(void)
cmdNumberCMD_CHANGE_TEXT_FENCE, CMD_FENCE_CHANGE_TEXT
{
    commandName = 2;
    startChangeTextCommand (TRUE);

    doChangeFence();
}
```

```

/*-----+
| name allChangeText          |
| authorBSI      8/89        |
+-----*/
Private voidallChangeText(void)
cmdNumberCMD_CHANGE_TEXT_ALL
{
    commandName = 3;
    startChangeTextCommand (TRUE);
    doChangeAll ();
}

/*-----+
| name changeChangeText      |
| authorBSI      6/93        |
+-----*/
Private voidchangeChangeText(void)
cmdNumber      CMD_CHANGE_TEXT_CHANGE
{
    commandName = 4;

    startChangeTextCommand (TRUE);
    doChangeCurrent ();
}

/*-----+
| name findChangeText        |
| authorBSI      6/93        |
+-----*/
Private voidfindChangeText(void)
cmdNumber      CMD_CHANGE_TEXT_FIND
{
    startChangeTextCommand (FALSE);
    findNextElement ();
}

/*-----+
| name main                  |
| authorBSI      8/89        |
+-----*/
Private DialogHookInfo uHooks[] =
{
    {HOOKID_Dialog_ReplaceText,changeText_dialogHook},
    {HOOKID_Item_MatchCase,changeText_matchCaseHook},
    {HOOKID_Item_WholeWords,changeText_wholeWordsHook},
    {HOOKID_Item_RegularExps,changeText_regularExpsHook},
    {HOOKID_Item_InCells,changeText_inCellsHook},

```



```

        {H00KID_Item_Find,changeText_findHook},
    };

Public intmain(int      argc,char      *argv[])
{
    RscFileHandle      rfHandle, userPrefsH;
    ChangeTextInfo      *textRscP;
    char      *setP;

    /* Tell MicroStation that ommands start in message list 0,
       and prompts start in string 1 */
    mdlState_registerStringIds (0, 1);

    /* Prepare to read resource. The resource file was used to save
       information the last time chngtxt was used. */
    textRscP      = NULL;
    userPrefsH      = NULL;

    mdlDialog_userPrefFileOpen(&userPrefsH, TRUE);
    if (userPrefsH)
    textRscP = (ChangeTextInfo *)mdlResource_load (NULL, RTYPE_chText,
                                                    RSCID_chTextPrefs);

    if (!textRscP)
    {
        /* No resource was found */
        chTextInfo.wholeWords      = 0;
        chTextInfo.matchCase      = 0;
        chTextInfo.regularExps      = 0;
        chTextInfo.cellText      = 1;
        chTextInfo.oldString[0] = chTextInfo.newString[0] = '\0';
    }
    else
    {
        /* Copy resource into internal structure */
        chTextInfo = *textRscP;

        /* This is unnecessary because the closeFile will free all resources,
           * but it is recommended practice */
        mdlResource_free (textRscP);
    }

    if (userPrefsH)
    mdlResource_closeFile (userPrefsH);

    /* Open our file for access to command table and dialog */
    mdlResource_openFile (&rfHandle, NULL, FALSE);

```

```

/* Set up and Publish chTextInfo for access by the dialog manager */
setP = mdlCEExpression_initializeSet (VISIBILITY_DIALOG_BOX, 0, 0);
mdlDialog_publishComplexVariable (setP, "changetextinfo",
    "chTextInfo",
        &chTextInfo);

/* publish our hook functions */
mdlDialog_hookPublish (sizeof(uHooks)/sizeof(DialogHookInfo),
uHooks);

/* Make sure our function gets called at unload time */
mdlSystem_setFunction (SYSTEM_UNLOAD_PROGRAM, unloadFunction);

/* Be sure to reset the Find data when new design file is opened */
mdlSystem_setFunction (SYSTEM_NEW_DESIGN_FILE, startNewFind);

/* Load the command table */
if (mdlParse_loadCommandTable (NULL) == NULL)
mdlOutput_error ("Unable to load command table.");

/* Open the dialog */
changeTextDBP = mdlDialog_open (NULL, DIALOGID_ReplaceText);

return 0;
}

```

input.mc

```

/*-----+
| Copyright (1995) Bentley Systems, Inc., All rights reserved.|
| "MicroStation" is a registered trademark and "MDL" and|
| "MicroCSL" are trademarks of Bentley Systems, Inc. |
| Limited permission is hereby granted to reproduce and modify |
| this copyrighted material provided that the resulting code is |
| used only in conjunction with Bentley Systems products under |
| the terms of the license agreement provided therein, and that |
| this notice is retained in its entirety in any such reproduction |
| or modification. |
+-----*/
/*-----+
| input.mc - examples for the mdlInput_ functions.|
| This file is intended as an adjunct to the MDL manual to|
| illustrate MDL built-in function calling conventions and parameter|
| values. While it can be compiled, it does NOT, on its own,|
| constitute a workable MDL example.|

```

```

+-----*/
#include <mdl.h> /* system include files */
#include <userfnc.h>
#include <msinputq.h>
#include <cmdlist.h>
#include <cmdclass.h>
/*-----+
| Function declarations |
+-----*/
void clearOutputFields();
/*-----+
| name coordinates |
| authorBSI 9/90 |
| This function illustrates the functions for getting and |
| releasing "active command" status. It also illustrates |
| functions for receiving and sending MicroStation queue |
| elements. |
+-----*/
cmdName void coordinates()
{
    Inputq_element queueElement;
    int messageType;
    mdlState_clear ();
    /* Tell MicroStation that all queue elements should be processed
       by this task. */
    if (mdlInput_startCommand () != TRUE)
    {
        mdlOutput_error ("Could not get active application status.");
        return;
    }
    for (;;)
    {
        mdlOutput_printf(MSG_PROMPT, "Enter datapoint or reset");
        /* Wait for MicroStation queue element. */
        if ((messageType = mdlInput_waitForMessage())==RESET)
            break;
        /* The user is starting another command. Just exit. */
        if (messageType==CMDNUM)
        {
            /* Requeue the command so that it will be reprocessed
               after this command releases active command status. */
            mdlInput_requeueLastInput(0);
            break;
        }
        if (messageType==DATAPNT)
        {
            mdlInput_getMessage (&queueElement);

```

```

        mdlOutput_printf (MSG_STATUS, "UORS = %d, %d, %d",
                           queueElement.u.data.pnt.uors.x,
                           queueElement.u.data.pnt.uors.y,
                           queueElement.u.data.pnt.uors.z);
        mdlOutput_printf (MSG_MESSAGE, "RAW UORS = %d, %d, %d",
                           queueElement.u.data.pnt.rawUors.x,
                           queueElement.u.data.pnt.rawUors.y,
                           queueElement.u.data.pnt.rawUors.z);
    }

    /* Release "active command" status so that this task will
       not receive all queue elements. */
    mdlInput_endCommand();
    clearOutputFields();
    mdlOutput_message("\nDisplay coordinates\ exited.");
}

/*-----+
| The following functions illustrate the use of user hooks|
| available with the input queue handling functions. The |
| functions "commandFilter", "messageReceived",          |
| "preprocessKeyin", and "monitorQueue" are all user functions. |
| The function installUserFunctions makes them active. |
| The function removeUserFunctions makes them inactive. |
+-----*/
/*-----+
| name  commandFilter          |
| authorBSI      9/90          |
| This is an example of command filter user function. See |
| userInput_commandFilter in the MDL manual for more information. |
| This function prevents all MicroStation delete commands. |
+-----*/
Private int commandFilter(Inputq_element *queueElementP)
{
    /* If this is not a MicroStation command, just return. */
    if ((queueElementP->u.cmd.taskId != '\0') &&
        (strcmp (queueElementP->u.cmd.taskId, ustrnTaskId) != 0))
        return INPUT_COMMAND_ACCEPT;
    switch (queueElementP->u.cmd.command)
    {
        case CMD_DELETE:
        case CMD_DELETE_ELEMENT:
        case CMD_DELETE_VERTEX:
        case CMD_DELETE_PARTIAL:
        case CMD_FENCE_DELETE:
        #if defined (LET_OLD_COMMAND_CONTINUE)
            /* Select the icon of the old command */
            mdlDialog_selectIconsByCmd (NULL, 0, NULL, -1);

```

```

#else
    /* Usually selects the chooser command */
    mdlState_startDefaultCommand ();
#endif
    mdlOutput_error ("Delete request rejected.");
    return INPUT_COMMAND_REJECT;
}
    return INPUT_COMMAND_ACCEPT;
}

/*-----+
| name messageReceived          |
| authorBSI          9/90      |
| messageReceived is an example of a receive user function. |
| See userInput_receive in the MDL manual for more information. |
+-----*/
Private void messageReceived(Inputq_element*queueElementP)
{
    mdlOutput_printf (MSG_STATUS, "messageReceived: message from %x",
        queueElementP->hdr.sourcepid);
}

/*-----+
| name preprocessKeyin          |
| authorBSI          9/90      |
| preprocessKeyin is an example of a preprocess-keyin user |
| function. See userInput_preprocessKeyin in the MDL manual |
| for more information. |
| See the "OR" example for a more extensive example |
| of a preprocessor user hook. |
+-----*/
Private int preprocessKeyin(char *keyinString)
{
    /* If the key-in starts with '$', replace it with "PLACE LINE" */
    if (*keyinString == '$')
        strcpy (keyinString, "PLACE LINE");
    return INPUT_ACCEPT;
}

/*-----+
| name monitorQueue            |
| authorBSI          9/90      |
| monitorQueue is an example of a input-monitor user function. |
+-----*/
Private int monitorQueue(Inputq_element*queueElementP)
{
    mdlOutput_printf (MSG_MESSAGE, "monitor: %d queue element",
        queueElementP->hdr.cmdtype);
    return INPUT_ACCEPT;
}

```

```

/*-----+
| name installUserFunctions          |
| authorBSI          9/90          |
+-----*/
cmdName void installUserFunctions(void)
{
    mdlInput_setFunction (INPUT_COMMAND_FILTER, commandFilter);
    mdlInput_setFunction (INPUT_MESSAGE_RECEIVED, messageReceived);
    mdlInput_setFunction (INPUT_KEYIN_PREPROCESS, preprocessKeyin);
    mdlInput_setMonitorFunction (MONITOR_ALL, monitorQueue);
}

/*-----+
| name removeUserFunctions          |
| authorBSI          9/90          |
+-----*/
cmdName void removeUserFunctions(void)
{
    mdlInput_setFunction (INPUT_COMMAND_FILTER, NULL);
    mdlInput_setFunction (INPUT_MESSAGE_RECEIVED, NULL);
    mdlInput_setFunction (INPUT_KEYIN_PREPROCESS, NULL);
    mdlInput_setMonitorFunction (MONITOR_ALL, NULL);
}

/*-----+
| name sendApplicationMessage          |
| authorBSI          9/90          |
| If the receive-user-function is not installed, the message          |
| will be ignored. Otherwise, the user receive user hook |
| will receive the message.          |
+-----*/
cmdName void sendApplicationMessage()
{
    Inputq_element queueElement;
    memset (&queueElement, '\0', sizeof (queueElement));
    queueElement.hdr.bytes = sizeof (Inputq_header);
    strcpy (queueElement.hdr.taskId, "INPUT");
    queueElement.hdr.cmdtype = APPLICATION_EVENT;
    queueElement.hdr.source = FROM_MDL;
    mdlInput_sendMessage (&queueElement, 0);
}

/*-----+
| This following functions illustrate how a macro can be |
| implemented using the input functions.          |
+-----*/
/*-----+
| name sendKeyin          |
| authorBSI          9/90          |
+-----*/

```

```

cmdName void sendKeyin(void)
{
    Dpoint3d point, point1;
    int position = 0;
    point.x = 200;
    point.y = 200;
    point.z = 0;
    point1.x = 1400;
    point1.y = 1400;
    point1.z = 0;
    /* Start the window area command */
    mdlInput_sendCommand (CMD_WINDOW_AREA, NULL,
                          position++, NULL, VIEWING);
    /* Specify the origin */
    mdlInput_sendUORPoint (&point, 0, position++, NULL);
    /* Specify the other corner */
    mdlInput_sendUORPoint (&point1, 0, position++, NULL);
    /* Specify the view */
    mdlInput_sendUORPoint (&point, 0, position++, NULL);
    /* Exit the viewing command */
    mdlInput_sendReset (position++, NULL);
    /* Now all of the requests are queued. Pause to let MicroStation
       interpret them. */
    waitForProcessing (position);
    /* Pause until the operator enters a keystroke */
    mdlOutput_prompt ("HIT A KEYSTROKE TO CONTINUE");
    mdlInput_pause ();
    position = 0;
    /* Start the PLACE TEXT command */
    mdlInput_sendKeyin ("PLACE TEXT", 0, position++, NULL);
    /* Send the text as a literal */
    mdlInput_sendKeyin ("This is the text to place", 1, position++,
NULL);
    /* Position the text */
    mdlInput_sendUORPoint (&point, 0, position++, NULL);
    /* Use the NULL command to terminate PLACE TEXT */
    mdlInput_sendCommand (CMD_NULL, NULL, position++, NULL, 0);
    /* MicroStation will interpret the queued elements when
       this function returns to MicroStation. */
}

/*-----+
| name  waitForProcessing      |
| authorBSI          9/90      |
+-----*/
Private waitForProcessing(int position)
{
    mdlInput_sendResume (position);
}

```

```

        /* MicroStation will interpret all of the queue elements that are
           in front of the "resume" queue element. Then it will send the
           "resume" queue element to this task. That will cause this task
           to resume after returning from mdlInput_waitForMessage. */
        mdlInput_waitForMessage ();
    }

    /*-----+
    | The functions "disable" and "enable" illustrate the input      |
    | functions that enable and disable command classes.           |
    +-----*/
    /*-----+
    | name disable                                          |
    | authorBSI          9/90          |                    |
    +-----*/
    cmdName void disable(void)
    {
        long    mask;
        /* Disable fence and manipulation commands. */
        mask = 1 << (FENCE-1);
        mask |= 1 << (MANIPULATION-1);
        mdlInput_disableCommandClass (mask, 0L);
    }

    /*-----+
    | name enable                                          |
    | authorBSI          9/90          |                    |
    +-----*/
    cmdName void enable(void)
    {
        long    mask;
        /* Enable fence and manipulation commands */
        mask = 1 << (FENCE-1);
        mask |= 1 << (MANIPULATION-1);
        mdlInput_enableCommandClass (mask, 0L);
    }

    /*-----+
    | name miscellaneous                                  |
    | authorBSI          9/90          |                    |
    | This function illustrates the calling sequences of input-    |
    | queue-handling functions not illustrated anywhere else.     |
    | This function does no useful work.                        |
    +-----*/
    Private void miscellaneous()
    {
        if (mdlInput_getTabletType () == 0)
            mdlOutput_message ("MicroStation is not using a tablet");
        if (mdlInput_commandState () == VIEW_COMMAND)
            mdlOutput_message ("MicroStation is executing a view command");
    }

```



```

    }
/*-----+
|   Private Utility Routines|
+-----*/
/*-----+
| name clearOutputFields      |
| authorBSI          9/90    |
+-----*/
Private void clearOutputFields(void)
{
    mdlOutput_status ("");
    mdlOutput_prompt ("");
    mdlOutput_message ("");
}

```

output.mc

```

/*-----+
-+
|                                     |
|   Copyright (c) 1985-91; Bentley Systems, Inc., All rights reserved.|
|                                     |
|   "MicroStation", "MDL", and "MicroCSL" are trademarks of Bentley|
|   Systems, Inc. and/or Intergraph Corporation. |
|                                     |
|   This program is proprietary and unpublished property of Bentley   |
|   Systems Inc. It may NOT be copied in part or in whole on any medium,
|
|   either electronic or printed, without the express written consent|
|   of Bentley Systems, Inc. |
|                                     |
+-----+
*/
/*-----+
-+
|                                     |
|   $Workfile:  scanexec.tmp  $      |
|   $Revision:  5.1  $  $Date:  30 Jul 1991 14:23:28  $  |
|                                     |
+-----+
*/
/*-----+
-+
|                                     |
|   output.mc - examples for the mdlOutput_ functions.|
|                                     |

```

```

|   This file is intended as an adjunct to the MDL manual to|
|   illustrate MDL built-in function calling conventions and parameter|
|   values. While it can be compiled, it does NOT, on its own,|
|   constitute a workable MDL example.|
|
+-----+
*/
/*-----+
+
|
|   Include Files
|
+-----+
*/
#include    <mdl.h>      /* system include files */
#include    <stdarg.h>
#include    <tcbl.h>

/*-----+
+
|
|   Function declarations
|
+-----+
*/

void messagePrintf (char *, ...);

/*-----+
+
|
|   name outputU
|
|   authorBSI      9/90
|
|   This function illustrates use of the output functions that|
|   display messages regardless of whether messages are inhibited.
|
+-----+
*/
cmdName void outputU
(
)
{
    mdlOutput_errorU ("UNINHIBITED ERROR");
    mdlOutput_promptU ("UNINHIBITED PROMPT");
    mdlOutput_messageU ("UNINHIBITED MESSAGE");
}

```

```

    mdlOutput_statusU ("UNINHIBITED STATUS");
    mdlOutput_commandU ("UNINHIBITED COMMAND");
    mdlOutput_keyinU ("UNINHIBITED KEYIN");
}

/*-----
-+
|
| name output
|
| authorBSI          9/90
|
| This function illustrates use of the output functions that
| are affected by whether the messages are inhibited.
|
+-----
*/
cmdName void output
(
{
    mdlOutput_error ("ERROR");
    mdlOutput_prompt ("PROMPT");
    mdlOutput_message ("MESSAGE");
    mdlOutput_status ("STATUS");
    mdlOutput_command ("COMMAND");
    mdlOutput_keyin ("KEYIN");
}

/*-----
-+
|
| name messages
|
| authorBSI          9/90
|
| This function toggles the messages-inhibit bit in the TCB.
|
+-----
*/
cmdName messages
(
void
)
{
    tcb->control.inh_msg ^= 1;
    if (tcb->control.inh_msg)

```

```

        mdlOutput_statusU ("Status messages are inhibited.");
        else
        mdlOutput_statusU ("Status messages are not inhibited.");
    }

/*-----
-+
|
| name errors
|
| authorBSI          9/90
|
| This function toggles the error-inhibit bit in the TCB.
|
+-----
*/
cmdName errors
(
void
)
{
    tcb->control.inh_err ^= 1;
    if (tcb->control.inh_err)
        mdlOutput_statusU ("Error messages are inhibited.");
    else
        mdlOutput_statusU ("Error messages are not inhibited.");
}

/*-----
-+
|
| name testPrintf
|
| authorBSI          9/90
|
| This function illustrates mdlOutput_printf.  mdlOutput_printf
| is like printf, except that it also takes a paramter that
| specifies a field in the MicroStation Command Window.
|
+-----
*/
cmdName testPrintf
(
char*echoP
)
{
    mdlOutput_printf (MSG_STATUS, "Testing testPrintf");
}

```

```

        mdlOutput_printf (MSG_MESSAGE, "ECHOED: \"%s\"", echoP);
    }

/*-----
-+
|
| name callMessage
|
| authorBSI          9/90
|
| This command just calls messagePrintf.
|
+-----
*/
cmdName callMessage
(
void
)
{
    messagePrintf ("INT %d DOUBLE %g STRING %s", 100, 200., "Hello");
}

/*-----
-+
|
| name messagePrintf
|
| authorBSI          9/90
|
| This function illustrates the use of mdlOutput_vprintf.
| mdlOutput_vprintf is like vprintf, except that it also takes
| a paramter that specifies a field in the MicroStation Command
| Window.
|
| messagePrintf is essentially a printf that always displays
| messages to the the message field of the Command Window.
|
+-----
*/
Private void messagePrintf
(
char*formatP,
...
)
{
    va_listap;

```

```

/* Make ap point to the first named argument. */
va_start (ap, formatP);

mdlOutput_vprintf (MSG_MESSAGE, formatP, ap);

va_end (ap);
}

```

cexpr.mc

```

/*-----
--+
|
| Copyright (c) 1985-91; Bentley Systems, Inc., All rights reserved.
|
| "MicroStation", "MDL", and "MicroCSL" are trademarks of Bentley
| Systems, Inc. and/or Intergraph Corporation.
|
| This program is proprietary and unpublished property of Bentley
| Systems Inc. It may NOT be copied in part or in whole on any medium,
|
| either electronic or printed, without the express written consent
| of Bentley Systems, Inc.
|
+-----
*/
/*-----
--+
|
| $Workfile: scanexec.tmp $
| $Revision: 5.1 $ $Date: 30 Jul 1991 14:23:10 $
|
+-----
*/
/*-----
--+
|
| cexpr.mc - examples for the mdlCExpression_ functions.
|
| This file is intended as an adjunct to the MDL manual to
| illustrate MDL built-in function calling conventions and parameter
| values. While it can be compiled, it does NOT, on its own,
| constitute a workable MDL example.
|
| See the files in mdl\examples\calculat for a useful example of

```

```

|   the mdlCExpression_ functions.      |
|                                     |
+-----+
*/
/*-----+
|                                     |
|   Include Files                    |
|                                     |
+-----+
*/

#include    <mdl.h>
#include    <cexpr.h>

/*-----+
+
|                                     |
|   Private variables                |
|                                     |
+-----+
*/

Private void*setP;
Private CType  *intPointerTypeP, *doubleArrayTypeP;

/*-----+
+
|                                     |
|   Variables to be published        |
|                                     |
+-----+
*/

int intVariable;
int*intVariable1P, *intVariable2P;
double*doubleArrayP;

/*-----+
+
|                                     |
|   name  displayError                |
|                                     |
|   authorBSI          9/90          |
|                                     |
|   The MDL Debugger clobbers the information used by the system      |

```

```

| to generate the message. If you are stepping through this      |
| with the MDL debugger, the message that is generated will be  |
| incorrect.                                                     |
|                                                                 |
+-----+
*/
displayError (void)
{
    char    messageBuffer [128];    /* The message is guaranteed to be
less
                                than 128 characters */

    mdlCEExpression_generateMessage (messageBuffer, mdlErrno);
    mdlOutput_error (messageBuffer);
}

/*-----+
-+
| name hideSymbols          |
| authorBSI          9/90   |
|                                                                 |
+-----+
*/
cmdName hideSymbols
(
)
{
    if (setP != NULL)
    {
        /* The call to mdlCEExpression_freeSet frees all memory
        associated with the symbol set. It frees all memory
        that had been allocated by calls to
        mdlCEExpression_typeFromRsc, mdlCEExpression_typePointer, etc */
        mdlCEExpression_freeSet (setP);
        setP = NULL;
    }
}

/*-----+
-+
| name publishSymbols      |
| authorBSI          9/90   |
|                                                                 |
|                                                                 |

```



```

+-----
*/
cmdName publishSymbols
(
)
{
    /* Initialize the symbol set. Set it up so that symbols that are
    added to it can be recognized by the calculator. */
    setP = mdlCEXpression_initializeSet (VISIBILITY_CALCULATOR, 0,
FALSE);

    /* Create a simple variable */
    mdlCEXpression_symbolPublish (setP, "intVariable",
SYMBOL_CLASS_VAR,
    &intType, &intVariable);

    /* Create a pointer variable. First define a type that can be used
    for all subsequent definitions of variables with type pointer
    to int.
    */
    intPointerTypeP = mdlCEXpression_typePointer (setP, &intType);
    mdlCEXpression_symbolPublish (setP, "pointer1", SYMBOL_CLASS_VAR,
    intPointerTypeP, &intVariable1P);
    mdlCEXpression_symbolPublish (setP, "pointer2", SYMBOL_CLASS_VAR,
    intPointerTypeP, &intVariable2P);

    /* Create an array variable. */
    doubleArrayP = malloc (sizeof (double) * 10);
    doubleArrayTypeP = mdlCEXpression_typeArray (setP, &doubleType, 10);
    mdlCEXpression_symbolPublish (setP, "doubleArray",
SYMBOL_CLASS_VAR,
    doubleArrayTypeP, doubleArrayP);
}

/*-----
-+
|
| name evaluateExpressions      |
|
| authorBSI          9/90      |
|
|
+-----
*/
cmdName evaluateExpressions
(
)

```

```

{
    CExprResult    result;    /* Detailed result */
    CExprValue     value;    /* Similar to result, but easier to
                               interpret. */

    if (mdlCEXpression_getValue (&value, &result, "intVariable*20",
    VISIBILITY_CALCULATOR))
        displayError ();

    if (mdlCEXpression_getValue (&value, &result, "INTVariable-10",
    VISIBILITY_CALCULATOR))
        displayError ();

    /* The following sequence sets doubleArray [2] to 5.2 */
    value.type = CEXPR_TYPE_DOUBLE;
    value.val.valDouble = 5.2;
    mdlCEXpression_setValue (&value, NULL, "doubleArray [2]",
    VISIBILITY_CALCULATOR);

    /* The following sequence sets doubleArray [2] to 7.5 */
    mdlCEXpression_getValue (&value, &result, "doubleArray [2] = 7.5",
    VISIBILITY_CALCULATOR);
}

```

calculat.mc

```

/*-----+
| Copyright (1995) Bentley Systems, Inc., All rights reserved.|
| "MicroStation" is a registered trademark and "MDL" and "MicroCSL"|
| are trademarks of Bentley Systems, Inc.      |
| Limited permission is hereby granted to reproduce and modify this|
| copyrighted material provided that the resulting code is used only |
| in conjunction with Bentley Systems products under the terms of the|
| license agreement provided therein, and that this notice is retained|
| in its entirety in any such reproduction or modification.|
+-----*/
/*-----+
| Function -                               |
| Calculator Dialog Hooks                   |
|                                         |
+-----*/
#include <dlogbox.h>
#include <dlogitem.h>
#include <dlogids.h>

```

```
#include <cexpr.h>
#include <rsdefs.h>
#include <vartypes.h>

#include "calculat.h"

#include <mscexpr.fdf>

/*-----+
|   External variables           |
+-----*/
extern char*dialogSetP;
extern inttotUserVars;
extern CalcStateInfocalcState;

/*-----+
| name          calcdlog_dialogHook|
| author        BSI                  8/90      |
+-----*/
Public void calcdlog_dialogHook(DialogMessage *dmP)
{
    dmP->msgUnderstood = TRUE;

    switch (dmP->messageType)
    {
        /* When dialog box is destroyed, cleanup C
        expression processor */
        case DIALOG_MESSAGE_DESTROY:
            calcState.dlogActive = FALSE;
            mdlCEXpression_freeSet (dialogSetP);
            dialogSetP = NULL;
            break;

        default:
            dmP->msgUnderstood = FALSE;
            break;
    }
}
```

system.mc

```
/*-----+
|   Copyright (1995) Bentley Systems, Inc., All rights reserved. |
|   "MicroStation" is a registered trademark and "MDL" and |
|   "MicroCSL" are trademarks of Bentley Systems, Inc.      |
|   Limited permission is hereby granted to reproduce and modify |
|   this copyrighted material provided that the resulting code is |
|   used only in conjunction with Bentley Systems products under |
|   the terms of the license agreement provided therein, and that |
|   this notice is retained in its entirety in any such reproduction |
```

```

| or modification. |
+-----*/
/*-----+
|
| system.mc - examples for the mdlSystem_ functions. |
|
| This file is intended as an adjunct to the MDL manual to |
| illustrate MDL built-in function calling conventions and |
| parameter values. While it can be compiled, it does NOT, on |
| its own, constitute a workable MDL example. |
|
+-----*/
#include <mdl.h> /* system include files */
#include <system.h>
#include <userfnc.h>
#include <mselems.h>
#include <tcb.h>
#include <msdefs.h>

/*-----+
|
| Global data |
|
+-----*/
Private int timerHandle;
Private int startTicks, seconds;
Private char* status[3] = { "Active", "Expired", "Not Found" };
/*-----+
|
| Function declarations |
|
+-----*/
/* The following functions are all used as user hooks in this program */
void mdlChildTerminated(), reloadProgram(), newDesignFile(),
timerExpired();
int unloadProgramHook();
/*-----+
|
| name main |
|
| authorBSI 9/90 |
|
+-----*/
main(int argc, char*argv[])
{
    /* Programs specified via the MS_INITAPPS environment variable
       are started before the design file is opened. */
    if (strcmp (argv [1], "MS_INITAPPS") == 0)

```

```

        {
            if (strcmp (argv [argc-1], "-idefault") == 0)
            {
                mdlSystem_enterGraphics ();
                if (mdlSystem_newDesignFile ("test2d") != SUCCESS)
                    exit (-1);
            }
        }
        reloadProgram(argc, argv);
    }

/*-----+
|
|   The sections that follow illustrate the mdlSystem_ user functions.
|
|   installUserFunctions installs all of the user functions.      |
|   removeUserFunctions terminates the user functions.            |
|
|-----*/
/*-----+
|
|   name installUserFunctions                                     |
|
|   authorBSI          9/90          |
|
|-----*/
cmdName void installUserFunctions(void)
{
    /* All of the mdlSystem_set... functions take a function pointer
       as an argument. These functions are called to handle certain
       events in MicroStation. For example, the first
       mdlSystem_setFunction call tells MicroStation to call
       newDesignFile whenever a new design file is opened. */
    mdlSystem_setFunction (SYSTEM_NEW_DESIGN_FILE, newDesignFile);
    mdlSystem_setFunction (SYSTEM_UNLOAD_PROGRAM, unloadProgramHook);
    mdlSystem_setFunction (SYSTEM_MDLCHILD_TERMINATED,
mdlChildTerminated);
    mdlSystem_setFunction (SYSTEM_RELOAD_PROGRAM, reloadProgram);
    if (timerHandle == 0)
    {
        mdlSystem_setTimerFunction (&timerHandle, 30, timerExpired, 0,
TRUE);
        startTicks = mdlSystem_getTicks ();
        seconds = 0;
    }
}

/*-----+
|
|
|-----*/

```

```

| name removeUserFunctions
|
| authorBSI          9/90
|
+-----*/
cmdName void removeUserFunctions(void)
{
    mdlSystem_setFunction(SYSTEM_NEW_DESIGN_FILE, NULL);
    mdlSystem_setFunction(SYSTEM_UNLOAD_PROGRAM, NULL);
    mdlSystem_setFunction(SYSTEM_MDLCHILD_TERMINATED, NULL);
    mdlSystem_setFunction(SYSTEM_RELOAD_PROGRAM, NULL);
    if (timerHandle!=0)
    {
        mdlSystem_cancelTimer(timerHandle);
        timerHandle=0;
    }
}

/*-----+
|
| name mdlChildTerminated
|
| authorBSI          9/90
|
| mdlChildTerminated is an example of an "MDL-child-terminated"
| user function. See userSystem_mdlChildTerminated in the
| manual for more information.
|
+-----*/
Private void mdlChildTerminated(MdlChildTerminated *childInfoP)
{
    char buffer[40];
    sprintf(buffer, "%s terminated, status = %d",
        childInfoP->pName, childInfoP->exitStatus);
    mdlDialog_openAlert(buffer);
}

/*-----+
|
| name reloadProgram
|
| authorBSI          9/90
|
| reloadProgram is an example of a "reload-program"
| user function. See userSystem_reloadProgram in the
| manual for more information.
|
+-----*/
Private void reloadProgram(int argc, char *argv[])

```

```

    {
        mdlOutput_printf(MSG_STATUS, "%s loaded by %s",
            argv[0], argv[1]);
    }
}

/*-----+
| name unloadProgramHook                                     |
| authorBSI          9/90                                   |
| unloadProgramHook is an example of a "unload-program"|
| user function. See userSystem_unloadProgram in the|
| manual for more information.                            |
|-----*/
Private int unloadProgramHook(int unloadReason)
{
    if ((unloadReason == SYSTEM_TERMINATED_COMMAND) ||
        (unloadReason == SYSTEM_TERMINATED_BY_APP))
    /* Reject the unload request */
    return 1;

    /* Let MicroStation unload the application */
    return 0;
}

/*-----+
| name newDesignFile                                         |
| authorBSI          9/90                                   |
| newDesignFile is an example of a "new-design file"|
| user function. See userSystem_newDesignFile in the|
| manual for more information.                            |
|-----*/
Private void newDesignFile(char *fileP, int state)
{
    if (state == SYSTEM_NEWFILE_CLOSE)
    {
        if (mdlDialog_openAlert ("Save settings?") == 3)
            mdlSystem_fileDesign ();
        mdlOutput_printf (MSG_COMMAND,
            "newDesignFile: %s is being closed.\n", fileP);
    }
    else
        mdlOutput_printf (MSG_PROMPT,

```



```
|   This marks the end of the user functions.|
|
+-----*/
/*-----+
|
|   name  loadProgram
|
|   authorBSI          9/90
|
+-----*/
cmdName void loadProgram(char *programNameP)
{
    /* If the user did not specify a program to start, just return */
    if (*programNameP == '\0')
        return;

    /* Start the program with no arguments. Assume that the task
       name can be derived from the application file name. Pass
       "TEST as the argument.*/
    if (mdlSystem_loadMdlProgram (programNameP, NULL, "TEST") !=
        SUCCESS)
        mdlOutput_printf (MSG_ERROR, "Unable to load %s", programNameP);
}
/*-----+
|
|   name  unloadProgram
|
|   authorBSI          9/90
|
+-----*/
cmdName void unloadProgram(char*taskIdP)
{
    /* If the user did not specify a program to start, just return */
    if (*taskIdP == '\0')
        return;

    if (mdlSystem_unloadMdlProgram (taskIdP) != SUCCESS)
        mdlOutput_printf (MSG_ERROR, "Unable to unload %s", taskIdP);
}
/*-----+
|
|   name  addStartup
|
|   authorBSI          9/90
|
|   addStartup creates type 66 element to start the application
|   "SYSTEM". It then writes that to the design file.
|
```

```

| Afterwards, everytime MicroStation tries loads the design |
| file, it will see the type 66 element and try to load |
| the program SYSTEM. |
|
+-----*/
cmdName void addStartup(void)
{
    MSElementUnion element;

    mdlSystem_createStartupElement (&element, "SYSTEM");
    mdlElement_add (&element);
}

/*-----+
|
| name removeStartup |
|
| authorBSI          9/90 |
|
| If the design file has a type 66 startup element that would |
| start the application SYSTEM, then removeStartup deletes it. |
|
+-----*/
cmdName void removeStartup(void)
{
    mdlSystem_deleteStartupElement("SYSTEM");
}

/*-----+
|
| name closeDgn |
|
| authorBSI      9/90 |
|
+-----*/
cmdName void closeDgn(void)
{
    char buffer[MAXFILELENGTH];

    strncpy(buffer, tcb->dgnfilenm, MAXFILELENGTH);
    mdlSystem_closeDesignFile();
    mdlSystem_enterGraphics();
    mdlSystem_newDesignFile(buffer);
}

/*-----+
|
| name miscellaneous |
|
| authorBSI          9/90 |
|

```

```

| This function illustrates the calling sequences of mdlSystem |
| functions not documented anyplace else. |
| This function does no useful work. |
|-----*/
cmdName misc(void)
{
    MdlAppStatistics appStatistics;
    char* applicationListP;
    char envVariable [80];
    char serverName[40], buffer[40];
    int timeLeft;
    int nlaModeChangeFunc();
    mdlOutput_prompt ("Enter any character to continue");
    while (mdlSystem_getChar () == 0)
;

    /* Get the task ID's all all installed applications */
    applicationListP = mdlSystem_getMdlTaskList ();
    /* Collect information on the first task in the list. */
    mdlSystem_getTaskStatistics (&appStatistics, applicationListP);
    /* mdlSystem_getMdlTaskList mallocs the memory for the task list.
       Free it now. */
    free (applicationListP);

    mdlOutput_prompt ("Waiting 10 seconds.");

    /* Make the message display. Pausing is not enough to call
       MicroStation to flush automatically. */
    mdlWindow_flush (NULL);

    /* The parameter is in units of sixtieths of a second. */
    mdlSystem_pauseTicks (600);

    /* Turn off the automatic abort */
    mdlSystem_userAbortEnable (FALSE);
    mdlOutput_prompt ("Hit Control C or Control Break to continue.");
    /* abortRequested will return TRUE when the operator hits control C
*/
    while (!mdlSystem_abortRequested ())
;
    mdlOutput_prompt ("");
    mdlSystem_putenv ("MY_VARIABLE", "TESTVALUE");
    mdlSystem_getenv (envVariable, "MY_VARIABLE", sizeof (envVariable));
    if (strcmp ("TESTVALUE", envVariable) != 0)
/* Something went wrong. Lets enter the debugger */

```

```

mdlSystem_enterDebug ();

#if defined (msdos)
    /* This shows the use of the NLA functionality provided */
    if (mdlSystem_nlaStartKeyChecking ("PROD001", nlaModeChangeFunc) ==
SUCCESS)
    {
        mdlSystem_networkKeyInfo ("PROD001", &timeLeft, serverName);
        mdlOutput_status (serverName);
        sprintf (buffer, "Time Left = %d hours, %d minutes", (timeLeft /
3600),
                (timeLeft % 3600));
        mdlOutput_message (buffer);
        mdlSystem_nlaStopKeyChecking ("PROD001");
    }
#endif
    /* This closes all files and frees all memory used by the
application. */
    mdlSystem_cleanUp ();
}

/*-----+
|
| name done
|
| authorBSI          9/90
|
|-----*/
cmdName done(void)
{
    char buffer[80];

    sprintf(buffer, "Do you want to terminate %s?",
            mdlSystem_getCurrTaskID ());
    if (mdlDialog_openAlert(buffer)==3)
        mdlSystem_exit (1, 1);
}

```

excfgar.mc

```

/*-----+
-+
|
| Copyright (1995) Bentley Systems, Inc., All rights reserved.|
|
| "MicroStation" is a registered trademark and "MDL" and "MicroCSL"|

```

```

| are trademarks of Bentley Systems, Inc. |
|
| Limited permission is hereby granted to reproduce and modify this|
| copyrighted material provided that the resulting code is used only |
| in conjunction with Bentley Systems products under the terms of the|
| license agreement provided therein, and that this notice is retained
|
| in its entirety in any such reproduction or modification.|
|
+-----+
*/
/*-----+
|
| $Workfile: excfgvar.mc $
| $Revision: 5.5 $
| $Date: 25 Jul 1995 12:50:14 $
|
+-----+
*/
/*-----+
|
| Function -
|
| Example demonstrating mdlSystem_cfgVarXXXX calls.|
|
+-----+
*/
/*-----+
|
| Include Files
|
+-----+
*/
#include <mdl.h> /* MDL Library funcs structures & constants */
#include <dlogitem.h> /* Dialog Box Manager structures & constants */
*/
#include <cexpr.h> /* C Expression structures & constants */
#include <cmdlist.h> /* MicroStation command numbers */
#include <msdefs.h> /* MicroStation common defines */
#include <stdarg.h> /* Variable argument defines */
#include <string.h> /* String function defines */
#include <stdio.h> /* Standard function defines */
#include <stdlib.h> /* Function prototype for free() */

```

```

#include <dlogman.fdf> /* dialog box manager function prototypes */
#include <msrsrc.fdf> /* mdlResource_xxx prototypes */
#include <mssystem.fdf> /* mdlSystem_xxx prototypes */
#include <msstrngl.fdf> /* mdlStringList_xxx prototypes */

#include "excfgvar.h"

/*-----
-+
|
| Nameexcfgvar_rscSprintf |
|
| AuthorBSI 3/93 |
|
+-----
*/
Private void excfgvar_rscSprintf
(
char *stringP, /* <= Result of sprintf from resource*/
intmessageNumber, /* => Index into msg list for format str*/
... /* => Any other optional arguments*/
)
{
va_list ap;
char tempStr[1024];

va_start (ap, messageNumber);

*stringP = tempStr[0] = '\0';
mdlResource_loadFromStringList (tempStr, NULL, MESSAGELISTID_Msgs,
messageNumber);
vsprintf (stringP, tempStr, ap);

va_end (ap);
}

/*-----
-+
|
| name main |
|
| authorBSI 5/93 |
|
+-----
*/
Public int main
(

```

```

int argc,      /* => number of args in next array */
char  *argv[] /* => array of cmd line arguments */
)
{
    int      cfgLevel, i;
    char      cfgLevelName[128];
    char      cfgVarValue[128];
    char      tmpString[128];
    char      *cfgVarExpansion = NULL;
    char      *name;
    RscFileHandle  rscHandle;

    mdlResource_openFile (&rscHandle, NULL, FALSE);

    /* Get an already existing configuration variable */
    mdlSystem_getCfgVar (cfgVarValue, "MS_DEF", sizeof(cfgVarValue));
    sprintf (tmpString, "MS_DEF = %s", cfgVarValue);
    mdlDialog_dmsgsPrint (tmpString);

    /* Expands the configuration variable */
    cfgVarExpansion = mdlSystem_expandCfgVar (cfgVarValue);

    if (cfgVarExpansion != NULL)
    {
        excfgvar_rscPrintf (tmpString, MSGID_MSDEFEXPANDED,
cfgVarExpansion);
        mdlDialog_dmsgsPrint (tmpString);

        /* free when done */
        free (cfgVarExpansion);
    }

    /* Get an undefined configuration variable */
    if (mdlSystem_getCfgVar
        (cfgVarValue, "TMP_NEW", sizeof(cfgVarValue)) != SUCCESS)
    {
        mdlResource_loadFromStringList (tmpString, NULL, MESSAGELISTID_Msgs,
MSGID_TMPNEWNOTYET);
        mdlDialog_dmsgsPrint (tmpString);
    }

    /* Define a new configuration variable */
    mdlSystem_defineCfgVar ("TMP_NEW", "$(MSDIR)test.txt",
CFGVAR_LEVEL_USER);

    /* Clear the string & get the newly created configuration variable
*/

```

```

    if (mdlSystem_getCfgVar
        (cfgVarValue, "TMP_NEW", sizeof(cfgVarValue)) == SUCCESS)
    {
        sprintf (tmpString, "TMP_NEW = %s", cfgVarValue);
        mdlDialog_dmsgsPrint (tmpString);

        /* Expand the configuration variable to get current value */
        cfgVarExpansion = mdlSystem_expandCfgVar (cfgVarValue);

        excfgvar_rscPrintf (tmpString, MSGID_TMPNEWEXPANDED,
            cfgVarExpansion);
        mdlDialog_dmsgsPrint (tmpString);

        /* free when done */
        free (cfgVarExpansion);
    }

#ifdef MSVERSION
    /* Starting with 5.5.1, it is possible to get expanded value in one
    step */
    cfgVarExpansion = mdlSystem_getExpandedCfgVar ("MS_DEF");

    if (cfgVarExpansion != NULL)
    {
        int          count;
        StringList   *strListP;

        /* in 5.5.1, it is possible to create a list from a cfg vars value */
        count = mdlSystem_createListFromCfgVarValue (&strListP,
            cfgVarExpansion);

        if (count > 0)
        {
            char      *valueEntryP;
            int        j;

            for (j=0; j<count; j++)
            {
                mdlStringList_getMember (&valueEntryP, NULL, strListP, j);
                sprintf (tmpString, "(%d) MS_DEF = %s", j, valueEntryP);
                mdlDialog_dmsgsPrint (tmpString);
            }

            /* free string list created by createListFromCfgVarValue */
            mdlStringList_destroy (strListP);
        }
    }

```



```

        /* free when done */
        free (cfgVarExpansion);
    }

#endif

    /* Delete the newly created configuration variable */
    mdlSystem_deleteCfgVar ("TMP_NEW");
    mdlSystem_getCfgVar (cfgVarValue, "TMP_NEW", sizeof(cfgVarValue));
    excfgvar_rscPrintf(tmpString, MSGID_TMPNEWDELETED, cfgVarValue);
    mdlDialog_dmsgsPrint (tmpString);

    /* Define another new configuration variable */
    mdlSystem_defineCfgVar ("TMP_ANOTHERNEW", "$(MSDIR)new.txt",
CFGVAR_LEVEL_USER);

    /* Check to see if the configuration variable is locked */
    if (mdlSystem_isCfgVarLocked ("TMP_ANOTHERNEW") == TRUE)
    {
        mdlResource_loadFromStringList (tmpString, NULL, MESSAGELISTID_Msgs,
MSGID_TMPANOTHERNEWLOCKED);
        mdlDialog_dmsgsPrint (tmpString);
    }
    else
    {
        mdlResource_loadFromStringList (tmpString, NULL, MESSAGELISTID_Msgs,
MSGID_TMPANOTHERNEWNOWLOCKED);
        mdlDialog_dmsgsPrint (tmpString);
    }

    /* Lock the new configuration variable then try to change it */
    mdlSystem_lockCfgVar ("TMP_ANOTHERNEW");
    if (mdlSystem_isCfgVarLocked("TMP_ANOTHERNEW") == TRUE)
    {
        mdlResource_loadFromStringList (tmpString, NULL, MESSAGELISTID_Msgs,
MSGID_TMPANOTHERNEWNOWLOCKED);
        mdlDialog_dmsgsPrint (tmpString);
    }

    /* Clear the string & get the newly created configuration variable
*/
    mdlSystem_getCfgVar (cfgVarValue, "TMP_ANOTHERNEW",
sizeof(cfgVarValue));
    sprintf (tmpString, "TMP_ANOTHERNEW = %s", cfgVarValue);
    mdlDialog_dmsgsPrint (tmpString);

```

```

    /* Unsuccessfully delete the newly created & locked config variable
    */
    mdlSystem_deleteCfgVar ("TMP_NEW");
    if (mdlSystem_getCfgVar
        (cfgVarValue, "TMP_ANOTHERNEW", sizeof(cfgVarValue)) == SUCCESS)
    {
        excfgvar_rscPrintf (tmpString, MSGID_TMPANOTHERNEWCANTDELETE,
                            cfgVarValue);
        mdlDialog_dmsgsPrint (tmpString);
    }

    /* prints out the level that _USTN_SYSTEM IS */
    mdlSystem_getCfgVarLevel(&cfgLevel,"_USTN_SYSTEM");

    /* convert level number to level name (msdefs.h) */
    switch (cfgLevel)
    {
    case CFGVAR_LEVEL_SYSENV:
        mdlResource_loadFromStringList (cfgLevelName, NULL,
                                         MESSAGELISTID_Msgs, MSGID_SysEnv);

        break;

    case CFGVAR_LEVEL_SYSTEM:
        mdlResource_loadFromStringList (cfgLevelName, NULL,
                                         MESSAGELISTID_Msgs, MSGID_System);

        break;

    case CFGVAR_LEVEL_APPL:
        mdlResource_loadFromStringList (cfgLevelName, NULL,
                                         MESSAGELISTID_Msgs,
                                         MSGID_Application);

        break;

    case CFGVAR_LEVEL_SITE:
        mdlResource_loadFromStringList (cfgLevelName, NULL,
                                         MESSAGELISTID_Msgs, MSGID_Site);

        break;

    case CFGVAR_LEVEL_PROJECT:
        mdlResource_loadFromStringList (cfgLevelName, NULL,
                                         MESSAGELISTID_Msgs, MSGID_Project);

        break;

    case CFGVAR_LEVEL_USER:
        mdlResource_loadFromStringList (cfgLevelName, NULL,
                                         MESSAGELISTID_Msgs, MSGID_User);

        break;
    }

```

```

default:
    mdlResource_loadFromStringList (cfgLevelName, NULL,
                                    MESSAGELISTID_Msgs, MSGID_Unknown);
}

excfgvar_rscPrintf (tmpString, MSGID_USTNSYSTEM, cfgLevelName);
mdlDialog_dmsgsPrint (tmpString);

/* prints out all of the config variables starting with MS_C */
mdlDialog_dmsgsPrint (" ");
mdlResource_loadFromStringList (tmpString, NULL,
MESSAGELISTID_Msgs,
                                MSGID_MSC);
mdlDialog_dmsgsPrint (tmpString);

i=0;
while (TRUE)
{
    if (mdlSystem_getCfgVarByIndex (&name, NULL, NULL, NULL, i++) !=
SUCCESS)
        break;

/* Only print those beginning with MS_C */
if (strncmp (name, "MS_C", 4) == 0)
{
    mdlDialog_dmsgsPrint (name);
}
}

return (SUCCESS);
}

```

create.mc

```

/*-----+
|
| Copyright (1995) Bentley Systems, Inc., All rights reserved.|
|
| "MicroStation" is a registered trademark and "MDL" and|
| "MicroCSL" are trademarks of Bentley Systems, Inc.    |
|
|

```

```

| Limited permission is hereby granted to reproduce and modify |
| this copyrighted material provided that the resulting code is |
| used only in conjunction with Bentley Systems products under |
| the terms of the license agreement provided therein, and that |
| this notice is retained in its entirety in any such reproduction |
| or modification. |
+-----*/
/*-----+
|
|   $Logfile:   J:/mdl/examples/doc/create.mcv  $
|   $Workfile:  create.mc  $
|   $Revision:  5.5  $
|   $Date:     20 Jun 1995 08:49:38  $
|
+-----*/
/*-----+
|
|   create.mc - examples for the mdlXXX_create functions. |
|
|   This file is intended as an adjunct to the MDL manual to |
|   illustrate MDL built-in function calling conventions and |
|   parameter values. While it can be compiled, it does NOT, on |
|   its own, constitute a workable MDL example. |
|
+-----*/
/*-----+
|
|   Include Files |
|
+-----*/
#include    <mdl.h>    /* system include files */
#include    <global.h>
#include    <mselems.h>
#include    <tcb.h>
#include    <userpref.h>
#include    <wchar.h>
#include    <mselemen.fdf>
/*-----+
|
|   name placeAShape |
|
|   authorBSI         8/90 |
|
+-----*/
Private void placeAShape(void)
{
    Dpoint3dshapePts[3];

```

```

        MSElementUnion shape;
        shapePts[0].x = fc_zero;
        shapePts[0].y = 1.0;
        shapePts[0].z = fc_zero;
        shapePts[1].x = 100.0;
        shapePts[1].y = 200.0;
        shapePts[1].z = fc_zero;
        shapePts[2].x = 300.0;
        shapePts[2].y = 400.0;
        shapePts[2].z = fc_zero;
        if (mdlShape_create (&shape, NULL, shapePts, 3, -1) == SUCCESS)
        {
            mdlElement_display (&shape, NORMALDRAW);
            mdlElement_add (&shape);
        }
    }

/*-----+
|
| name placeALineString
|
| authorBSI      8/90
|
|-----*/
Private void placeALineString(Dpoint3d *lsPoints, int numVerts)
{
    MSElementUnion lineString;
    if (mdlLineString_create(&lineString, NULL, lsPoints,
        numVerts) == SUCCESS)
    {
        mdlElement_display(&lineString, NORMALDRAW);
        mdlElement_add(&lineString);
    }
}

/*-----+
|
| name placeAnArc
|
| authorBSI      8/90
|
|-----*/
Private void placeAnArc(MSElementUnion *existingArc)
{
    Dpoint3d center;
    MSElementUnion arc;

    center.x = 100.0;
    center.y = 100.0;

```

```

        center.z = 100.0;

        if (mdlArc_create (&arc, existingArc, &center, 100.0, 200.0,
            NULL, fc_zero, fc_pi) == SUCCESS)
        {
            mdlElement_display (&arc, NORMALDRAW);
            mdlElement_add (&arc);
        }
    }

/*-----+
|
| name placeAnArcByPoints          |
|
| authorBSI          8/90          |
|
|-----*/
Private void placeAnArcByPoints(void)
{
    MSElementUnion arc;
    Dpoint3d arcPt[3];

    arcPt[0].x = 20.;    /* start point */
    arcPt[0].y = 30.;
    arcPt[0].z = fc_zero;
    arcPt[1].x = 50.;    /* mid point */
    arcPt[1].y = 60.;
    arcPt[1].z = fc_zero;
    arcPt[2].x = 20.;    /* end point */
    arcPt[2].y = 80.;
    arcPt[2].z = fc_zero;
    if (mdlArc_createByPoints(&arc, NULL, arcPt)==SUCCESS)
    {
        mdlElement_display (&arc, NORMALDRAW);
        mdlElement_add (&arc);
    }
}

/*-----+
|
| name placeAnArcByCenter          |
|
| authorBSI          8/90          |
|
|-----*/
Private void placeAnArcByCenter(int view)
{
    MSElementUnion arc;
    Dpoint3d arcPt[3];

```

```

    arcPt[0].x = 20.;    /* start point */
    arcPt[0].y = 30.;
    arcPt[0].z = fc_zero;
    arcPt[1].x = 50.;    /* center */
    arcPt[1].y = 30.;
    arcPt[1].z = fc_zero;
    arcPt[2].x = 80.;    /* end point */
    arcPt[2].y = 30.;
    arcPt[2].z = fc_zero;
    /* create an arc with a radius of 44.0 and center at [50,30,0] */
    if (mdlArc_createByCenter(&arc, NULL, arcPt, TRUE,
        44.0, view)== SUCCESS)
    {
        mdlElement_display(&arc, NORMALDRAW);
        mdlElement_add(&arc);
    }
}

/*-----+
| name placeEllipse                                |
| authorBSI            8/90                        |
|-----*/
Private void placeEllipse(void)
{
    MSElementUnion ellipse;
    Dpoint3d center;
    center.x = 100.0;
    center.y = 100.0;
    center.z = 100.0;
    if (mdlEllipse_create(&ellipse, NULL, &center, 10.0, 15.0,
        NULL, -1)==SUCCESS)
    {
        mdlElement_display (&ellipse, NORMALDRAW);
        mdlElement_add (&ellipse);
    }
}

/*-----+
| name placeCircleByPoints                          |
| authorBSI            8/90                        |
|-----*/
Private void placeCircleByPoints(void)

```

```

    {
        MSElementUnion circle;
        Dpoint3d circlePt[3];
        circlePt[0].x = 200.;
        circlePt[0].y = 300.;
        circlePt[0].z = 000.;
        circlePt[1].x = 500.;
        circlePt[1].y = 600.;
        circlePt[1].z = 000.;
        circlePt[2].x = 200.;
        circlePt[2].y = 800.;
        circlePt[2].z = 000.;
        if (mdlCircle_createBy3Pts (&circle, NULL, circlePt, -1) ==
SUCCESS)
    {
        mdlElement_display (&circle, NORMALDRAW);
        mdlElement_add (&circle);
    }
}

/*-----+
|
| name placeText
|
| authorBSI          8/90
|
|-----*/
Private void placeText(MSElementUnion *currentText, long textPos)
{
    MSElementUniontext;
    Dpoint3dpt[3];
    TextSizeParamtxtSize;
    TextParamtxtParam;
    TextEDFieldedFields[2];
    TextEDParamedParam;
    pt[0].x = 285.;/* origin of text element */
    pt[0].y = 450.;
    pt[0].z = fc_zero;
    txtSize.mode= TXT_BY_TEXT_SIZE;
    txtSize.size.width= fc_2;
    txtSize.size.height= 3.;
    txtParam.font= 2;
    txtParam.just= -1;/* use active justification */
    txtParam.style= -1;/* use active style */
    txtParam.viewIndependent = FALSE;
    edFields[0].start= 1;
    edFields[0].len= 2;
    edFields[0].just= 2;

```



```

    edFields[1].start= 5;
    edFields[1].len= 1;
    edFields[1].just= 3;
    edParam.numEDFields= 2;
    edParam.edField= edFields;
    /* create a new text element */
    if (mdlText_create (&text, NULL, "this text placed in MDL", pt,
        &txtSize, NULL, &txtParam, &edParam) == SUCCESS)
    {
        mdlElement_display (&text, NORMALDRAW);
        mdlElement_add (&text);
    }
    /* overwrite an existing text element without changing parameters */
    if (mdlText_create (&text, currentText, "overwritten text",
        NULL, NULL, NULL, NULL, NULL) == SUCCESS)
    {
        mdlElement_rewrite (&text, currentText, textPos);
        mdlElement_display (&text, NORMALDRAW);
    }
}

/*-----+
| name placeTextNode |
| authorBSI          8/90 |
|-----*/
Private void placeTextNode(void)
{
    MSElementUnion textNode;
    Dpoint3d origin;
    MTextSize nodeSize;
    TextParam txtParam;
    double lineSpacing;
    origin.x= 100.;
    origin.y= 200.;
    origin.z = 0.;
    nodeSize.width = 10.;
    nodeSize.height= 13.;
    lineSpacing = 1.1;
    txtParam.font= 2;
    txtParam.just= -1;/* use active justification */
    txtParam.style= -1;/* use active style */
    txtParam.viewIndependent = FALSE;
    if (mdlTextNode_create (&textNode, NULL, &origin, NULL,
&lineSpacing,
        &nodeSize, &txtParam) == SUCCESS)

```

```

    {
        mdlElement_display (&textNode, NORMALDRAW);
        mdlElement_add (&textNode);
    }
}

/*-----+
| name          text_getExtendedParam                               |
|                                                       |
|      sets flags and values for extended text attributes|
|      (slant, underline, character spacing, and direction)|
|      according to tcb and userpref                        |
|                                                       |
| author        BSI                      4/93                |
|                                                       |
+-----*/
Public void text_setExtendedParam
(
    TextParamWide  *textParam/* <= flags, slant, spacing, etc. */
)
{
    /* TextDrawFlags tells the existence of slant, inter-character
    spacing,
    underline, and vertical direction.    Set all flags to zero first.
    */
    memset (&textParam->flags, 0, sizeof(TextDrawFlags));
    /* Convert tcb->textSlant from degree to radian.
    Set flags.slant only if slant value is not zero */
    if (fabs(tcb->textSlant) > fc_epsilon)
    {
        textParam->flags.slant = TRUE;
        textParam->slant = tcb->textSlant * fc_piover180;
    }
    if (tcb->textDirection & TXTDIR_VERTICAL)/* vertical direction */
        textParam->flags.vertical = TRUE;
    /* user preference tells if fixed-width mode or inter-character mode
    */
    if (userPrefsP->extFlags.fixedWidthCharSpace)
    {
        textParam->flags.fixedWidthSpacing = TRUE;
        /* if char spacing is 0, ignore it otherwise characters are
        stacked together */
        if (fabs(tcb->textAboveSpacing) < fc_epsilon)
            textParam->characterSpacing =
                textParam->flags.vertical ? tcb->chheight : tcb->chwidth;
        else
            textParam->characterSpacing = tcb->textAboveSpacing;
    }
}

```

```

    }
    else if (fabs(tcb->textAboveSpacing) > fc_epsilon)
    {
        textParam->flags.interCharSpacing = TRUE;
        textParam->characterSpacing = tcb->textAboveSpacing;
    }
    if (tcb->textUnderline)
    {
        textParam->flags.underline = TRUE;
        textParam->underlineSpacing = tcb->chheight *
            userPrefsP->textUnderlineSpace * fc_p01;
    }
}

/*-----+
| name placeTextWide |
| authorBSI          7/93 |
|-----*/
Private void placeTextWide(MSElementUnion *currentText, long textPos)
{
    MSElementUniontext;
    Dpoint3dpt[3];
    TextSizeParamtxtSize;
    TextParamWidetxtParam;
    TextEDFieldedFields[2];
    TextEDParamedParam;
    char*string1 = "this text placed in MDL";
    char*string2 = "overwritten text";
    MSWideCharwString1[50], wString2[50];
    /* convert to wide-character string - the third argument is the size
       of wString, see "stdlib.h" for details */
    mbstowcs (wString1, string1, 50);
    mbstowcs (wString2, string2, 50);
    pt[0].x = 285.;/* origin of text element */
    pt[0].y = 450.;
    pt[0].z = fc_zero;
    txtSize.mode= TXT_BY_TEXT_SIZE;
    txtSize.size.width= fc_2;
    txtSize.size.height= 3.;
    txtParam.font= 2;
    txtParam.just= -1;/* use active justification */
    txtParam.style= -1;/* use active style */
    txtParam.viewIndependent = FALSE;
    /* set extended params (slant, underline, char spacing, and
    direction) */

```

```

    text_setExtendedParam (&txtParam);
    edFields[0].start= 1;
    edFields[0].len= 2;
    edFields[0].just= 2;
    edFields[1].start= 5;
    edFields[1].len= 1;
    edFields[1].just= 3;
    edParam.numEDFields= 2;
    edParam.edField= edFields;
    /* create a new text element */
    if (mdlText_createWide (&text, NULL, wString1, pt, NULL,
        &txtSize, &txtParam, &edParam) == SUCCESS)
    {
        mdlElement_display (&text, NORMALDRAW);
        mdlElement_add (&text);
    }
    /* overwrite an existing text element without changing parameters */
    if (mdlText_createWide (&text, currentText, wString2,
        NULL, NULL, NULL, NULL, NULL) == SUCCESS)
    {
        mdlElement_rewrite (&text, currentText, textPos);
        mdlElement_display (&text, NORMALDRAW);
    }
}

/*-----+
|
| name placeTextNodeWide
|
| authorBSI 7/93
|
|-----*/
Private void placeTextNodeWide(void)
{
    MSElementUnion textNode;
    Dpoint3d origin;
    TextSizeParam nodeSize;
    TextParamWide txtParam;
    origin.x= 100.;
    origin.y= 200.;
    origin.z = 0.;
    nodeSize.mode= TXT_BY_TEXT_SIZE;
    nodeSize.size.width= 10.;
    nodeSize.size.height= 13.;
    txtParam.font= 2;
    txtParam.just= -1; /* use active justification */
    txtParam.style= -1; /* use active style */
    txtParam.viewIndependent = FALSE;

```

```

        txtParam.lineSpacing = tcb->nodespace;
        /* set extended params (slant, underline, char spacing, and
direction) */
        text_setExtendedParam (&txtParam);
        if (mdlTextNode_createWide (&textNode, NULL, &origin, NULL,
            &nodeSize, &txtParam) == SUCCESS)
        {
            mdlElement_display (&textNode, NORMALDRAW);
            mdlElement_add (&textNode);
        }
    }
}

/*-----+
| name placeCone |
| authorBSI      8/90 |
|-----*/
Private void placeCone(RotMatrix *rMatrix)
{
    MSElementUnion cone;
    Dpoint3d conePts[2];
    conePts[0].x = 100.;
    conePts[0].y = 200.;
    conePts[0].z = 0.;
    conePts[1].x = 100.;
    conePts[1].y = 200.;
    conePts[1].z = 10.;
    if (mdlCone_create (&cone, NULL, 2.0, 1.0, &conePts[0], &conePts[1],
        rMatrix) == SUCCESS)
    {
        mdlElement_display (&cone, NORMALDRAW);
        mdlElement_add (&cone);
    }
}

/*-----+
| name . |
| authorBSI      8/90 |
|-----*/
Private void placeCylinder(void)
{
    MSElementUnion cone;
    Dpoint3d conePts[2];
    conePts[0].x = 100.;

```

```

        conePts[0].y = 200.;
        conePts[0].z = 0.;
        conePts[1].x = 100.;
        conePts[1].y = 200.;
        conePts[1].z = 10.;
        if (mdlCone_createRightCylinder (&cone, NULL, 2.0, &conePts[0],
                                         &conePts[1]) == SUCCESS)
        {
            mdlElement_display (&cone, NORMALDRAW);
            mdlElement_add (&cone);
        }
    }
}

```

element.mc

```

/*-----
-+
|                                     |
| Copyright (c) 1985-91; Bentley Systems, Inc., All rights reserved.|
|                                     |
| "MicroStation", "MDL", and "MicroCSL" are trademarks of Bentley|
| Systems, Inc. and/or Intergraph Corporation. |
|                                     |
| This program is proprietary and unpublished property of Bentley  |
| Systems Inc. It may NOT be copied in part or in whole on any medium,
|
| either electronic or printed, without the express written consent|
| of Bentley Systems, Inc. |
|                                     |
+-----
*/
/*-----
-+
|                                     |
| $Workfile: scanexec.tmp $         |
| $Revision: 5.1 $ $Date: 30 Jul 1991 14:24:10 $ |
|                                     |
+-----
*/
/*-----
-+
|                                     |
| element.mc - examples for the mdlElement_ functions.|
|                                     |
| This file is intended as an adjunct to the MDL manual to|

```

```
| illustrate MDL built-in function calling conventions and parameter|
| values. While it can be compiled, it does NOT, on its own,|
| constitute a workable MDL example.|
|
+-----+
*/
/*-----+
+
|
| Include Files
|
+-----+
*/
#include <mdl.h> /* system include files */
#include <global.h>
#include <mselems.h>

/*-----+
+
|
| local defines
|
+-----+
*/
#define MASTER_FILE 0

/*-----+
+
|
| name incrementColor
|
| authorBSI 8/90
|
+-----+
*/
Public void incrementColor
(
MSElement *element
)
{
int color;
int true=TRUE, false=FALSE;

/* get the current color from the element */
mdlElement_getSymbology (&color, NULL, NULL, element);

/* increment it (wrap if necessary) */
```

```

        if (++color >= 256)
            color = 0;

        /* set the new color in the element */
        mdlElement_setSymbology (element, &color, NULL, NULL);

        /* mark the element as "not-new" and "modified" */
        mdlElement_setProperties (element, NULL, NULL, NULL, NULL,
                                &false, &true, NULL, NULL);
    }

/*-----
-+
|
| name incrementLevel          |
|
| authorBSI          8/90      |
|
|-----
+-----
*/
Public void incrementLevel
(
MSElement    *element
)
{
    int    level, gg, locked;
    int    true=TRUE, false=FALSE;

    /* get the current level from the element */
    mdlElement_getProperties (&level, &gg, NULL, &locked, NULL, NULL,
NULL,
    NULL, element);

    /* don't do members of a graphic group or locked elements */
    if (gg==0 && !locked)
    {
        if (++level > 63)
        {
            /* set the level and mark element as "not-new" and "modified" */
            mdlElement_setProperties (element, level, NULL, NULL, NULL,
                                    &false, &true, NULL, NULL);
        }
    }
}

/*-----
-+

```



```

| name placeNewLine |
| authorBSI 8/90 |
|-----
*/
Public voidplaceNewLine
(
Dpoint3d*linePts,/* => points defining line */
ULong *filePos/* <= filePosition of new element */
)
{
MSElementline;

if (mdlLine_create (&line, NULL, linePts) == SUCCESS)
{
mdlElement_display (&line, NORMALDRAW);
if ((*filePos = mdlElement_add (&line)) == 0L)
{
mdlOutput_printf (MSG_ERROR,"error adding element, %d",
mdlErrno);
}
}
}

/*-----
-+
| name changeLine |
| authorBSI 8/90 |
|-----
*/
Public voidchangeLine
(
Dpoint3d *linePts,/* => points defining line */
MSElement *oldLine,/* => existing line element */
ULong *filePos/* <=> file position of line */
)
{
MSElementUnion line;

/* make sure they passed us a line element */
if (mdlElement_getType (oldLine) != LINE_ELM)
return;

```

```

        if (mdlLine_create (&line, oldLine, linePts) == SUCCESS)
        {
            if ((*filePos = mdlElement_rewrite (&line, oldLine, *filePos)) == 0L)
            {
                mdlOutput_printf (MSG_ERROR, "error changing element, %d",
                                   mdlErrno);
            }
        }
        else
        {
            mdlElement_display (oldLine, ERASE);
            mdlElement_display (&line,  NORMALDRAW);
        }
    }
}

/*-----
-+
|                                     |
| name deletePoints - delete "point" elements |
|                                     |
| authorBSI          8/90          |
|                                     |
+-----
*/
Public voiddeletePoints
(
MSElement    *oldLine, /* => existing line element */
ULong        filePos, /* => file position of line */
int          fileNum
)
{
    Dpoint3dlinePts[MAX_VERTICES];
    int numVerts;

    /* delete any "point" elements (those with 2 vertices the same) */
    if (mdlLinear_extract (linePts, &numVerts, oldLine, fileNum) ==
        SUCCESS)
    {
        if (numVerts == 2 && mdlVec_pointEqual (linePts, linePts+1))
        {
            mdlElement_undoableDelete (oldLine, filePos, TRUE);
        }
    }
}

```

```

/*-----
-+
|
| name moveAttributes |
|
| authorBSI          8/90 |
|
|-----
*/
Public voidmoveAttributes
(
MSElement*oldElem,/* => old element */
MSElement*newElem/* <=> new element */
)
{
    shortattributes[MAX_ATTRIBSIZE];
    int attribLen;

    /* copy all of the attributes from the old element to the
    new one, appending them to any existing attributes. Then
    clear the attributes from the original element. */
    mdlElement_extractAttributes (&attribLen, attributes, oldElem);
    mdlElement_appendAttributes (newElem, attribLen, attributes);

    mdlElement_stripAttributes (oldElem, oldElem);
}

/*-----
-+
|
| name resetWorkingWindow - clear working window (if present) |
|
| authorBSI          7/90 |
|
|-----
*/
Public intresetWorkingWindow
    /* <= returns SUCCESS if WW was set, ERROR otherwise */
(
void
)
{
    /* get current "working window" (don't need fileNumber) */
    if (mdlElement_getFilePos (FILEPOS_WORKING_WINDOW, NULL) != 0L)
    {
        mdlElement_setFilePos (FILEPOS_WORKING_WINDOW, MASTER_FILE, 0L);
        mdlOutput_message ("Working window cleared");
    }
}

```

```

        return    SUCCESS;
    }
    return ERROR;
}

/*-----
-+
|
| name  displayOneElementInOneView|
|
| authorBSI          1/91      |
|
+-----
*/
Private void displayOneElementInOneView
(
    int fileNum,
    ULong filePos,
    int view
)
{
    MElement element;

    /* NOTE: This function would probably be better written using the
    mdlElmdscr_ functions so that complex elements would work as well. */

    if (mdlElement_read (&element, fileNum, filePos) == SUCCESS)
    {
        if (view >= 0 && view <= 7)
            mdlElement_displayInSelectedViews (&element, NORMALDRAW,
            1 << view);
    }
}

```

elemdscr.mc

doc.mc

view.mc

```

/*-----
-+
|
|      view - MDL function to save and restore up 10 view positions
|              per view. It also provides a command to allow users
|              to shift their view using datapoints to define the
|              distance to shift. To load the application keyin
|              MDL L VIEW. The following commands are then available:
|
|              WINDOW PREVIOUS
|              WINDOW PAN
|
|-----
|
|      This is FREE software. It may be used, copied, and distributed
|      as long as there is no charge. This software was written by
|      Bill Steinbock for US Army Corps of Engineers.
|
+-----
*/
/*-----
-+
|
|      Private Global variables|
|
|-----
+-----
*/
#include    <mdlio.h>
#include    <mdl.h>
#include    <basetype.h>
#include    <global.h>
#include    <tcb.h>
#include    <userfnc.h>
#include    <rscldefs.h>
#include    <mselems.h>
#include    <keys.h>
#include    "view.h"

/*-----
-+
|
|      Local defines
|
|-----

```

```

+-----
*/
#define MAX_TO_SAVE      10
#define NUM_WINDOWS      8
#define NORMAL           0
#define PREVIOUS         1

#pragma      Version      2:0:0

/*-----
-+
|                                     |
|   Local functions                                     |
|                                     |
+-----
*/
/*-----
-+
|                                     |
|   Private Global variables|
|                                     |
+-----
*/
Private Dpoint3d    point1;
Private int         view1;
Private int         numPushed[NUM_WINDOWS]      = {-1,-1,-1,-1,-1,-1,-1,-1,-1,-1};
Private int         viewPrevious[NUM_WINDOWS]   = {0,0,0,0,0,0,0,0,0,0};

Private struct
{
    Dpoint3d    vieworg;
    Dpoint3d    viewdelta;
    RotMatrix   rotmatrix;
    double      activeZ;
} currentView[NUM_WINDOWS],
savedViews[NUM_WINDOWS][MAX_TO_SAVE];

/*-----
-+
|                                     |
|   name          view_exitViewCmd                                     |
|                                     |
+-----
*/
Private void        view_exitViewCmd
(

```

```

void
)
{
    mdlState_exitViewCommand (TRUE);
}

/*-----
-+
|
| name          view_handleViewUpdates
|
+-----
*/
Private int      view_handleViewUpdates
(
    int          preUpdate,
    int          eraseMode,
    long         *fileMask,
    int          numberRegions,
    Asynch_update_view regions[]
)
{
    int          count, view, offset;
    int          check1, check2, i;
    Dpoint3d     vieworg;
    Dpoint3d     viewdelta;
    RotMatrix    rotmatrix;
    double       activeZ;

    if (preUpdate == TRUE)
    {
        for (count=0; count<numberRegions; count++)
        {
            view = regions[count].viewnum;

            if (numPushed[view] < 0)
            {
                view_loadViewSettings (view);
                numPushed[view]=0;
                continue;
            }

            mdlView_getParameters (&vieworg, NULL,
                                   &viewdelta,
                                   &rotmatrix,
                                   &activeZ, view);
        }
    }
}

```

```

        /* --- compare to previous saved area --- */
        check1 = mdlVec_pointEqual (&vieworg,
&currentView[view].vieworg);
        check2 = mdlVec_pointEqual (&vieworg,
&currentView[view].vieworg);
        if (check1 && check2)
            continue;

/* --- check to see if we already have saved max. # of views
--- */
    if (numPushed[view]+1 > MAX_TO_SAVE)
    {
        /* --- push saved views back --- */
        for (i=0; i<(MAX_TO_SAVE-1); i++)
        {
            savedViews[view][i].vieworg =
                savedViews[view][i+1].vieworg;
            savedViews[view][i].viewdelta =
                savedViews[view][i+1].viewdelta;
            savedViews[view][i].rotmatrix =
                savedViews[view][i+1].rotmatrix;
            savedViews[view][i].activeZ =
                savedViews[view][i+1].activeZ;
        }

        /* --- reset number of saved views --- */
        numPushed[view] = numPushed [view]-1;
    }

    /* --- save old current view settings --- */
    offset = numPushed[view];
    savedViews[view][offset].vieworg =
currentView[view].vieworg;
    savedViews[view][offset].viewdelta =
currentView[view].viewdelta;
    savedViews[view][offset].rotmatrix =
currentView[view].rotmatrix;
    savedViews[view][offset].activeZ =
currentView[view].activeZ;

    /* --- save current view info --- */
    currentView[view].vieworg = vieworg;
    currentView[view].viewdelta = viewdelta;
    currentView[view].rotmatrix = rotmatrix;
    currentView[view].activeZ = activeZ;

    numPushed[view] = numPushed[view]+1;

```



```

        viewPrevious[view] = numPushed[view];
    }
}
return (SUCCESS);
}

/*-----
-+
|
| name          view_loadViewSettings          |
|
|-----
*/
Private int      view_loadViewSettings
(
    int          view
)
{
    mdlView_getParameters (&currentView[view].vieworg, NULL,
                           &currentView[view].viewdelta,
                           &currentView[view].rotmatrix,
                           &currentView[view].activeZ, view);
}

/*-----
-+
|
| name          view_showPreviousWindow          |
|
|-----
*/
Private void      view_showPreviousWindow
(
    Dpoint3d      *pt,
    int           view
)
{
    {
        int          offset;
        Dpoint3d      range[2], viewOffset;

        if (viewPrevious[view] > 0)
        {
            offset = viewPrevious[view]-1;

            /* --- rotate point by view rotation --- */
            viewOffset = savedViews[view][offset].viewdelta;
            mdlRMatrix_unrotatePoint (&viewOffset,

```

```

                                &savedViews[view][offset].rotmatrix);
range[0]  = savedViews[view][offset].vieworg;
range[1].x = range[0].x + viewOffset.x;
range[1].y = range[0].y + viewOffset.y;
range[1].z = range[0].z + viewOffset.z;
mdlView_setArea (view, range, &savedViews[view][offset].vieworg,
                savedViews[view][offset].viewdelta.z,
                savedViews[view][offset].activeZ,
                &savedViews[view][offset].rotmatrix);

/* --- update pointer to previous view info --- */
viewPrevious[view] = viewPrevious[view] - 1;

/* --- pop view from stack --- */
numPushed[view] = numPushed [view]-1;

/* --- don't store the view if previous command issued --- */
mdlView_setFunction (UPDATE_PRE, NULL);

/* --- update current view --- */
mdlView_updateSingle (view);

/* --- reset up function to handle update events --- */
mdlView_setFunction (UPDATE_PRE, view_handleViewUpdates);

/* --- reset current view settings --- */
mdlView_getParameters (&currentView[view].vieworg, NULL,
                      &currentView[view].viewdelta,
                      &currentView[view].rotmatrix,
                      &currentView[view].activeZ, view);
mdlOutput_printf (MSG_MESSAGE, " ");
}
else
mdlOutput_rscPrintf (MSG_MESSAGE, NULL, 0, 8);
}

/*-----
-+
| name          view_windowPreviousCmd          |
|-----+-----|
*/
cmdName          view_windowPreviousCmd
(
void

```

```

)
cmdNumber      CMD_WINDOW_PREVIOUS
{
    mdlState_startViewCommand (view_showPreviousWindow, 2, 3);
    mdlState_setFunction (STATE_RESET, view_exitViewCmd);
}

/*-----
-+
|
| name          view_panWindow
|
|-----
*/
Private void    view_panWindow
(
Dpoint3d      *pt,
int           view
)
{
    Dpoint3d    offset, viewOffset;
    Dpoint3d    range[2];
    Dpoint3d    vieworg;
    Dpoint3d    viewdelta;
    RotMatrix    rotmatrix;
    double      activeZ;

    offset.x = pt->x - point1.x;
    offset.y = pt->y - point1.y;
    offset.z = pt->z - point1.z;

    /* --- read current view settings --- */
    mdlView_getParameters (&vieworg, NULL, &viewdelta,
                          &rotmatrix, &activeZ, view1);

    /* --- rotate point by view rotation --- */
    viewOffset = viewdelta;
    mdlRMMatrix_unrotatePoint (&viewOffset, &rotmatrix);

    /* --- modify view settings --- */
    vieworg.x = vieworg.x - offset.x;
    vieworg.y = vieworg.y - offset.y;
    vieworg.z = vieworg.z - offset.z;

    /* --- set new view settings --- */
    range[0] = vieworg;
    range[1].x = range[0].x + viewOffset.x;

```

```

    range[1].y = range[0].y + viewOffset.y;
    range[1].z = range[0].z + viewOffset.z;
    mdlView_setArea (view, range, &vieworg, viewdelta.z,
                    activeZ, &rotmatrix);

    /* --- update current view --- */
    mdlView_updateSingle (view1);

    /* --- restart the command --- */
    view_windowPanCmd ();
}

/*-----
--+
| name          view_showLine          |
|-----
*/
Private void    view_showLine
(
Dpoint3d      *pt,
int           viewNum
)
{
    Dpoint3d    linePnts[2];
    linePnts[0] = point1;
    linePnts[1] = *pt;

    if (mdlLine_create (dgnBuf, NULL, linePnts) != SUCCESS)
        *((short *) dgnBuf) = 0;
    else
    {
        dgnBuf->hdr.dhdr.symb.b.color = 0;
        dgnBuf->hdr.dhdr.symb.b.weight = 0;
        dgnBuf->hdr.dhdr.symb.b.style = 0;
    }
}

/*-----
--+
| name          view_saveFirstPoint    |
|-----
*/
Private int     view_saveFirstPoint

```

```

(
Dpoint3d    *pt,
int         view
)
{
    point1 = *pt;
    view1 = view;
    mdlState_setFunction (STATE_DATAPOINT, view_panWindow);
    mdlOutput_rscPrintf (MSG_PROMPT, NULL, 0, 7);
    mdlState_dynamicUpdate (view_showLine, FALSE);
}

/*-----
-+
|                                     |
| name          view_windowPanCmd    |
|                                     |
+-----
*/
cmdName          view_windowPanCmd
(
void
)
cmdNumber        CMD_WINDOW_PAN
{
    mdlState_startViewCommand (view_saveFirstPoint, 5, 6);
    mdlState_setFunction (STATE_RESET, view_exitViewCmd);
}

/*-----
-+
|                                     |
| name main      |
|                                     |
+-----
*/
main
(
int  argc,
char *argv[]
)
{
    RscFileHandle  rfHandle;

    /* --- load our command table --- */
    if (mdlParse_loadCommandTable (NULL) == NULL)
        mdlOutput_error ("Unable to load command table.");
}

```

```

/* --- open resource files to get messages from --- */
mdlResource_openFile (&rfHandle, NULL, FALSE);

/* --- set up function to handle update events --- */
mdlView_setFunction (UPDATE_PRE, view_handleViewUpdates);
}

```

trumpet.mc

```

/*-----
-+
|
| Copyright (1995) Bentley Systems, Inc., All rights reserved.|
|
| "MicroStation" is a registered trademark and "MDL" and "MicroCSL"|
| are trademarks of Bentley Systems, Inc.      |
|
| Limited permission is hereby granted to reproduce and modify this|
| copyrighted material provided that the resulting code is used only |
| in conjunction with Bentley Systems products under the terms of the|
| license agreement provided therein, and that this notice is retained|
|
| in its entirety in any such reproduction or modification.|
|
+-----
*/
/*-----
-+
|
| $Workfile:  trumpet.mc  $
| $Revision:  5.7  $
| $Date:    26 Jul 1995 13:47:10  $
|
+-----
*/
/*-----
-+
|
| Function -
|
| Place a trumpet in a design file
|
| -----
|

```

```

Public Routine Summary -

    elemDscr_show - Display contents of element dscr
    main - Main entry point
    multi - MULTI command entry point
    trumpet - TRUMPET command entry point
    trumpet_appendElbowPipe - Append an elbow pipe to elem dscr
    trumpet_appendReducer - Append a reducer to elem dscr
    trumpet_appendStraightPipe - Append straight pipe to elem dscr

    trumpet_createTrumpet - Create the trumpet
    trumpet_placeMultiPoint - Define point for multi command plcmnt

    trumpet_placeTrumpetPoint - Def point for trumpet command
plcmnt|
    trumpet_setPoint - Set a point
    trumpet_startTrumpet - Start trumpet processing
+-----+
*/
/*-----+
-+
|
|   Include Files
|
+-----+
*/
#include    <mselems.h>
#include    <global.h>
#include    <mdl.h>
#include    <rsdefs.h>
#include    <math.h>
#include    <string.h>

#include    "trumpet.h"

#include    <dlogman.fdf>
#include    <mcurrtr.fdf>
#include    <mssystem.fdf>
#include    <msrsrc.fdf>
#include    <mstmtrx.fdf>
#include    <mse1mdsc.fdf>
#include    <msoutput.fdf>
#include    <msstate.fdf>
#include    <mscell.fdf>
#include    <mselemen.fdf>

```

```

/*-----
-+
|
|   Private Global variables
|
+-----
*/
MSElementDescr  *trumpetDP;

/*-----
-+
|
|   Private utility routines
|
+-----
*/
/*-----
-+
|
|   name          trumpet_setPoint
|
|   author        BSI
|
+-----
*/
Private void      trumpet_setPoint
(
Dpoint3d      *pt,
double        x,
double        y,
double        z
)
{
    pt->x = x;
    pt->y = y;
    pt->z = z;
}

#if defined (DEBUG)
/*-----
-+
|
|   name          elemDscr_show
|
|   author        BSI
|
+-----
5/90

```



```
+-----
*/
Public int      elemDscr_show
(
MSElementDescr *elemDescr,
char            *currentIndent
)
{
    char    indent[128];
    int     color, weight, style, level, ggNum, class, locked, new,
           modified, viewIndepend, solidHole;

    if (elemDescr == NULL)
return SUCCESS;

    strcpy (indent, currentIndent);
    strcat (indent, "|  ");

    do
    {
mdlElement_getSymbology (&color, &weight, &style, &elemDescr->el);
mdlElement_getProperties (&level, &ggNum, &class, &locked, &new,
                        &modified, &viewIndepend, &solidHole, &elemDescr->el);
printf ("%shdr=%d, typ=%d, cmplx=%d", currentIndent,
        elemDescr->h.isHeader, elemDescr->el.hdr.ehdr.type,
        elemDescr->el.hdr.ehdr.complex);
printf ("c=%d,w=%d,s=%d,l=%d,gg=%d,cl=%d\n",
        color, weight, style, level, ggNum, class);

    if (elemDescr->h.isHeader)
    {
        elemDscr_show (elemDescr->h.firstElem, indent);
        printf ("%send of chain\n", currentIndent);
    }

    elemDescr = elemDescr->h.next;
    } while (elemDescr);
    }
#endif

/*-----
-+
|
| name            trumpet_startTrumpet
|
| author          BSI
|
| 7/90
|
```

```

+-----
*/
Private MElementDescr *trumpet_startTrumpet
(
void
)
{
    MElementUnion      trumpetHeader;

    mdlCell_create (&trumpetHeader, "trumpt", NULL, 0);
    mdlElmdscr_new (&trumpetDP, NULL, &trumpetHeader);

    return trumpetDP;
}

#ifdef (CYLINDERS)
/*-----
-+
|
|   If the constant "CYLINDERS" is defined, we create the trumpet using
|   cones and cylinders, otherwise we use surfaces of projection and
|   rotation. This is merely to illustrate various techniques that can
|   be used with element descriptors.
|
+-----
*/
/*-----
-+
|
|   name          trumpet_appendReducer - add a "reducer" to element
|                  descriptor
|
|   author        BSI                      9/89
|
+-----
*/
Private void      trumpet_appendReducer
(
MElementDescr    *trumpetDP,
double           rad1,
double           rad2,
double           len
)
{

```

```

    Dpoint3d      pt[3];
    MSElementUnion cone;

    memset (pt, 0, sizeof(pt));
    pt[1].z += len;

    mdlCone_create (&cone, NULL, rad2, rad1, &pt[0], &pt[1], NULL);
    mdlElmdscr_appendElement (trumpetDP, &cone);

    mdlCurrTrans_translateOrigin (&pt[1]);
}

/*-----
-+
| name          trumpet_appendStraightPipe
| author        BSI
|               9/89
|-----
*/
Private void      trumpet_appendStraightPipe
(
MSElementDescr  *trumpetDP,
double          rad,
double          len
)
{
    Dpoint3d      pt[2];
    MSElementUnion cylinder;

    memset (pt, 0, sizeof(pt));
    pt[1].z = len;

    mdlCone_createRightCylinder (&cylinder, NULL, rad, &pt[0], &pt[1]);
    mdlElmdscr_appendElement (trumpetDP, &cylinder);

    mdlCurrTrans_translateOrigin (&pt[1]);
}
#else

/*-----
-+
| name          trumpet_appendReducer
| author        BSI
|               9/89
|-----

```

```

|
+-----+
*/
Private void    trumpet_appendReducer
(
MSElementDescr *trumpetDP,
double          rad1,
double          rad2,
double          len
)
{
    double          scale;
    Dpoint3d        pt[3];
    MSElementDescr *ellipseDP, *pipeDP;
    MSElementUnion  el;
    Transform        tMatrix;

    mdlEllipse_create (&el, NULL, NULL, rad1, rad1, NULL, 0);
    mdlElmdscr_new (&ellipseDP, NULL, &el);

    memset (pt, 0, sizeof(pt));
    pt[1].z += len;

    scale = rad2 / rad1;
    mdlTMatrix_getIdentity (&tMatrix);
    mdlTMatrix_scale (&tMatrix, &tMatrix, scale, scale, scale);

    mdlSurface_project (&pipeDP, ellipseDP, &pt[0], &pt[1], &tMatrix);
    mdlElmdscr_freeAll (&ellipseDP);

    mdlElmdscr_appendDscr (trumpetDP, pipeDP);
    mdlCurrTrans_translateOrigin (&pt[1]);
}

/*-----+
+
|
| name          trumpet_appendStraightPipe
|
| author        BSI
|
| 9/89
|
+-----+
*/
Private void    trumpet_appendStraightPipe
(
MSElementDescr *trumpetDP,
double          rad,

```

```

double      len
)
{
    Dpoint3d      pt[2];
    MSElementDescr *ellipseDP, *pipeDP;
    MSElementUnion el;

    mdlEllipse_create (&el, NULL, NULL, rad, rad, NULL, 0);
    mdlElmdscr_new (&ellipseDP, NULL, &el);

    memset (pt, 0, sizeof(pt));
    pt[1].z = len;

    mdlSurface_project (&pipeDP, ellipseDP, &pt[0], &pt[1], NULL);
    mdlElmdscr_freeAll (&ellipseDP);

    mdlElmdscr_appendDscr (trumpetDP, pipeDP);
    mdlCurrTrans_translateOrigin (&pt[1]);
}
#endif

/*-----
-+
|
| name          trumpet_appendElbowPipe
|
| author        BSI
|
| 9/89
|
+-----
*/
Private void      trumpet_appendElbowPipe
(
MSElementDescr *trumpetDP,
double          radius,
double          length,
double          bendRadius,
double          angle
)
{
    Dpoint3d      pt;
    MSElementUnion el;
    MSElementDescr *ellipseDP, *pipeDP;
    double          radians;

    radians = angle * fc_piover180;
    trumpet_appendStraightPipe (trumpetDP, radius, length);

```

```

mdlEllipse_create (&el, NULL, NULL, radius, radius, NULL, -1);
mdlElmdscr_new (&ellipseDP, NULL, &el);

memset (&pt, 0, sizeof(pt));
pt.y = -1. * bendRadius;
mdlCurrTrans_translateOrigin (&pt);

mdlSurface_revolve (&pipeDP, ellipseDP, NULL, 0, radians);
mdlElmdscr_freeAll (&ellipseDP);
mdlElmdscr_appendDscr (trumpetDP, pipeDP);

pt.x = sin(radians) * bendRadius;
pt.y = cos(radians) * bendRadius;
mdlCurrTrans_translateOrigin (&pt);
mdlCurrTrans_rotateByAngles (radians, fc_zero, fc_zero);

trumpet_appendStraightPipe (trumpetDP, radius, length);
}

/*-----
--
|
| name          trumpet_createTrumpet
|
| description    creates an element descriptor with the elements that
|
|                define a "trumpet" at current MDL origin and
|                orientation
|
| author         BSI                      9/89
|
+-----
*/
Public int      trumpet_createTrumpet
(
void
)
{
    Dpoint3d    pt;
    double      length, bendRadius, radius, radius2;
    int         i;

    /* push the current transform */
    mdlCurrTrans_begin();

    /* create the element descriptor with only a cell header */
    trumpet_startTrumpet ();

```

```

mdlCurrTrans_rotateByAngles (fc_zero, fc_piover2, fc_zero);

/* MouthPeice */
length      = .750;
radius      = .25;
trumpet_appendReducer (trumpetDP, radius * 2., radius, length);

/* First Straight Section */
trumpet_appendStraightPipe (trumpetDP, radius, 14.0);

/* Far Elbow */
length      = fc_onehalf;
bendRadius  = 1.500;
trumpet_appendElbowPipe (trumpetDP, radius, length, bendRadius,
fc_180);

/* Second Straight Section */
length = fc_10;
trumpet_appendStraightPipe (trumpetDP, radius, length);

mdlCurrTrans_rotateByAngles (fc_zero, fc_zero, 12.0 * fc_piover180);

/* Near Elbow */
length      = .500;
trumpet_appendElbowPipe (trumpetDP, radius, length, bendRadius,
fc_180);

/* Third Straight Section (out to horn) */
trumpet_appendStraightPipe (trumpetDP, radius, 12.0);

/* Horn */
length = fc_2;
trumpet_appendReducer (trumpetDP, radius, fc_1, length);
trumpet_appendReducer (trumpetDP, fc_1, 2.0, length);

trumpet_setPoint (&pt, fc_zero, -4., -12.);
mdlCurrTrans_translateOrigin (&pt);

mdlCurrTrans_rotateByAngles (-fc_piover2, fc_zero, fc_zero);
for (i=0; i<3; i++)
{
length      = fc_5;
trumpet_appendStraightPipe (trumpetDP, radius, length);
radius2 = .175;
length    = .250;
trumpet_appendStraightPipe (trumpetDP, radius2, length);

```

```

radius2 = .375;
trumpet_appendStraightPipe (trumpetDP, radius2, length);
trumpet_setPoint (&pt, fc_zero, fc_m1, -5.5);
mdlCurrTrans_translateOrigin (&pt);
}

/* set the range diagonal for the trumpet cell */
mdlCell_setRange (trumpetDP);

/* restore original transformation */
mdlCurrTrans_end();

return SUCCESS;
}

/*-----
-+
| name          trumpet_placeMultiPoint
| author        BSI                                2/90
|-----
*/
Private void    trumpet_placeMultiPoint
(
Dpoint3d      *origin,
int           view
)
{
    Transform   tMatrix;
    int         i;

    /* Clear any outstanding abort request */
    mdlSystem_abortRequested ();

    /* set up the current transform to be oriented about data point
in the view the user selected */
    mdlCurrTrans_begin();
    mdlCurrTrans_masterUnitsIdentity (FALSE);
    mdlCurrTrans_rotateByView (view);
    mdlCurrTrans_translateOriginWorld (origin);

    /* create a trumpet element descriptor */
    trumpet_createTrumpet();

    /* set up a transformation matrix to rotate trumpet by 45 degrees */

```



```

    mdlTMatrix_getIdentity (&tMatrix);
    mdlTMatrix_rotateByAngles (&tMatrix, &tMatrix, fc_zero, fc_zero,
                               fc_piover4);

    /* make 8 copies, rotating between them */
    for (i=0; i<8; i++)
    {
        /* display and add this trumpet */
        mdlElmdscr_display (trumpetDP, 0, NORMALDRAW);
        mdlElmdscr_add (trumpetDP);

        if (mdlSystem_abortRequested ())
            break;

        /* rotate by 45 degrees */
        mdlElmdscr_transform (trumpetDP, &tMatrix);
    }

    /* free trumpet Element Descriptor (allocated by
    trumpet_createTrumpet) */
    mdlElmdscr_freeAll (&trumpetDP);
    mdlCurrTrans_end();
}

/*-----
-+
| name          multi   (command)
| description    Place 8 trumpets in a circular pattern
| author         BSI                      9/89
|-----
*/
cmdName void    multi
(
void
)
{
    if (!mgds_modes.three_d)
    {
        mdlOutput_rscPrintf (MSG_ERROR, NULL, MESSAGELISTID_Messages,
                              ERRID_3D);
        return;
    }
}

```

```

        mdlState_startPrimitive (trumpet_placeMultiPoint, multi, 0, 0);
        mdlOutput_rscPrintf (MSG_MESSAGE, NULL, MESSAGELISTID_Messages,
                               MSGID_GroupOrigin);
        mdlCurrTrans_identity();
    }

/*-----
-+
|
| name          trumpet_placeTrumpetPoint
|
| author        BSI
|
| 2/90
|
+-----
*/
Private void    trumpet_placeTrumpetPoint
(
Dpoint3d      *origin,
int           view
)
{
    int         status;
    Dpoint3d    worldOrigin;

    /* set up the current transform to be oriented about data point
    in the view the user selected */
    mdlCurrTrans_begin();

    mdlCurrTrans_masterUnitsIdentity (FALSE);
    status = mdlCurrTrans_rotateByView (view);
    mdlCurrTrans_translateOriginWorld (origin);

    /* create trumpet element descriptor */
    trumpet_createTrumpet();

    /* display and add the trumpet to the file */
    mdlElmdscr_display (trumpetDP, 0, NORMALDRAW);
    mdlElmdscr_add (trumpetDP);

    /* free trumpet Element Descriptor (allocated by
    trumpet_createTrumpet) */
    mdlElmdscr_freeAll (&trumpetDP);
    mdlCurrTrans_end();
}

/*-----
-+

```

```

| name          trumpet (command) |
| description    Place single trumpet at location specified by user |
|               continues to place trumpets with additional datapoints |
| author         BSI                      9/89 |
+-----+
*/
cmdName void    trumpet
(
void
)
{
    if (!mgds_modes.three_d)
    {
        mdlOutput_rscPrintf (MSG_ERROR, NULL, MESSAGELISTID_Messages,
                               ERRID_3D);
        return;
    }

    mdlState_startPrimitive (trumpet_placeTrumpetPoint, trumpet, 0, 0);
    mdlOutput_rscPrintf (MSG_MESSAGE, NULL, MESSAGELISTID_Messages,
                          MSGID_TrumpetOrigin);
    mdlCurrTrans_identity();
}

/*-----+
| name          main |
| description    Perform initialization. |
| author         BSI                      05/90 |
+-----+
*/
void          main
(
void
)
{
    RscFileHandle    rscFileH;

```

```

mdlResource_openFile (&rscFileH, NULL, FALSE);

/* Prevent an asynchronous abort that would leave partial
trumpets in the design file. */
mdlSystem_userAbortEnable (0);

/* set up a current transformation in world coordinates */
mdlCurrTrans_begin();
}

```

locate.mc

```

/*-----
-+
|                                     |
| Copyright (1995) Bentley Systems, Inc., All rights reserved.|
|                                     |
| "MicroStation" is a registered trademark and "MDL" and "MicroCSL"|
| are trademarks of Bentley Systems, Inc.      |
|                                     |
| Limited permission is hereby granted to reproduce and modify this|
| copyrighted material provided that the resulting code is used only |
| in conjunction with Bentley Systems products under the terms of the|
| license agreement provided therein, and that this notice is retained|
|                                     |
| in its entirety in any such reproduction or modification.|
|                                     |
+-----
*/
/*-----
-+
|                                     |
|   $Logfile:   J:/mdl/examples/doc/locate.mcv  $
|   $Workfile:  locate.mc  $
|   $Revision:  1.2  $
|   $Date:     20 Jun 1995 08:49:52  $
|                                     |
+-----
*/
/*-----
-+
|                                     |
| locate.mc - examples for the mdlLocate_ functions.|
|                                     |

```

```

|   This file is intended as an adjunct to the MDL manual to|
|   illustrate MDL built-in function calling conventions and parameter|
|   values. While it can be compiled, it does NOT, on its own,|
|   constitute a workable MDL example.|
|
+-----+
*/
/*-----+
|
|   Include Files
|
+-----+
*/
#include    <mdl.h>
#include    <mselems.h>
#include    <userfnc.h>

/*-----+
|
|   Local function declarations
|
+-----+
*/
inteditText(), replaceComponentDone();

/*-----+
|
|   name  setTextSearchType
|
|   authorBSI      7/90
|
+-----+
*/
Private voidsetTextSearchType
(
void
)
{
    static int  searchType[] = {TEXT_ELM, TEXT_NODE_ELM};
    /* set search criteria to find nothing */
    mdlLocate_noElemNoLocked ();
    /* then add text elements and text nodes to list */
    mdlLocate_setElemSearchMask (sizeof(searchType)/
sizeof(searchType[0]),

```

```

        searchType);
    }
}
/*-----
-+
|
| name  changeText_accept - called for data point on change|
|                               text single command|
|
| authorBSI          3/89      |
|
+-----
*/
Private voidchangeText_accept
(
void
)
{
    ULong   filePos, compOffset;
    int     currFile;
    filePos = mdlElement_getFilePos (FILEPOS_CURRENT, &currFile);
    /* If there is a selection set active, process all elements in the
set.
Of course, this processes all component text elements of a text node.
If no selection set is active, process ONLY the identified element.
This makes this command not apply to graphic groups (by design).
*/
    if (mdlSelect_isActive())
    {
        mdlModify_elementMulti (currFile, filePos, MODIFY_REQUEST_NOHEADERS,
MODIFY_ORIG, editText, NULL, TRUE);
    }
    else
    {
        compOffset = mdlElement_getFilePos(FILEPOS_COMPONENT, NULL) -
filePos;
        mdlModify_elementSingle (currFile, filePos, MODIFY_REQUEST_ONLYONE,
MODIFY_ORIG, editText, NULL, compOffset);
    }
    /* restart the element location process */
    mdlLocate_restart (FALSE);
}
/*-----
-+
|
| name  doChangeSingle
|
| authorBSI          8/89      |
|

```

```

|                                     |
+-----+
*/
Private voiddoChangeSingle
(
void
)
{
    /* set up the MicroStation search mask to find only text elements */
    setTextSearchType();
    /* tell MicroStation we want to start a "modification" command */
    mdlState_startModifyCommand (doChangeSingle, changeText_accept,
NULL,
        NULL, NULL, 0, 1, TRUE, FALSE);
    /* set the internal locate pointers to start at beginning of file */
    mdlLocate_init ();
}
/*-----+
-+
|                                     |
| name            irrigationComponentsFilter                               |
|                                     |
|         Locate filter for REPLACE COMPONENT command. Permits|
|         only irrigation elements to be selected. The|
|         user attribute data on the element is checked to|
|         see if the element is an irrigation component.|
|                                     |
| author            BSI                                                    4/91                               |
|                                     |
+-----+
*/
Private intirrigationComponentsFilter
(
int          preLocate,    /* => TRUE if pre-locate */
MSElementUnion *elementP    /* => current element */
)
{
    UShort*wordOffsetP;
    int componentData;

    /* check for irrigation component element */
    if (!irrattrib_extractComponentUserData (&componentData,
        &wordOffsetP, elementP))
returnLOCATE_ELEMENT_NEUTRAL;

    return LOCATE_ELEMENT_REJECT;
}

```

```

/*-----
~+
| name          replaceComponentStart|
|          Start function for REPLACE COMPONENT command.|
| author        BSI                  4/91
|-----
*/
Private voidreplaceComponentStart
(
char*unparsed/* => unparsed part of command */
)
{
    /* we need to locate any existing irrigation component */
    mdlState_startModifyCommand (replaceComponentStart,
replaceComponentDone,
        NULL, NULL, NULL, 0, 0, TRUE, 0);

    initializeComponentLocate ();

    /* filter elements based on user attribute data */
    mdlLocate_setFunction (LOCATE_POSTLOCATE,
irrigationComponentsFilter);
    if (!mdlSelect_isActive ())
        util_displayPrompt (122);
}
/*-----
~+
| name dataButtonHit
| authorBSI          2/91
|-----
*/
Private voiddataButtonHit
(
Dpoint3d    *point,
int         view
)
{
    Dpoint3dclosestPoint;
    int segment;

```



```

        if (!mdlLocate_findElement (point, view, 0, 0, FALSE))
        mdlOutput_printf (MSG_STATUS, "Element not found");
        else
        {
        mdlOutput_printf (MSG_STATUS, "ELEMENT FOUND: type=%d",
            dgnBuf->ehdr.type);

        if (dgnBuf->ehdr.type == LINE_STRING_ELM)
        {
            mdlLocate_getProjectedPoint (&closestPoint, &segment, NULL);
            mdlOutput_printf (MSG_PROMPT,
                "Locate point=[%d,%d,%d], segment=%d",
                closestPoint.x, closestPoint.y, closestPoint.z, segment);
        }
        }
    }
}

/*-----
-+
| name      locateElement - print out element type of elements pointed|
| to by user.                                     |
| author    BSI      10/90      |
|-----
*/
cmdName    locateElement
(
void
)
{
    mdlState_startPrimitive (dataButtonHit, locateElement, 0, 0);
    /* reset the location logic */
    mdlLocate_init();

    /* allow any element */
    mdlLocate_allowLocked();

    /* change the cursor to be the "locate" cursor */
    mdlLocate_setCursor();
}

```

select.mc

fence.mc

```

/*-----
-+
|
| Copyright (1995) Bentley Systems, Inc., All rights reserved.|
|
| "MicroStation" is a registered trademark and "MDL" and "MicroCSL"|
| are trademarks of Bentley Systems, Inc.      |
|
| Limited permission is hereby granted to reproduce and modify this|
| copyrighted material provided that the resulting code is used only |
| in conjunction with Bentley Systems products under the terms of the|
| license agreement provided therein, and that this notice is retained|
|
| in its entirety in any such reproduction or modification.|
|
+-----
*/
/*-----
-+
|
| $Logfile:   J:/mdl/examples/fence/fence.mcv  $
| $Workfile:  fence.mc  $
| $Revision:  1.7  $
| $Date:    25 Jul 1995 14:26:08  $
|
+-----
*/
/*-----
-+
|
| Function -
|
| FENCE MDL Example Application
|
+-----
*/
/*-----
-+
|
| Include Files
|
+-----
*/
#include    <mselems.h>
#include    <mdl.h>

```

```

#include    <global.h>
#include    <tcb.h>
#include    <userfnc.h>
#include    <rsdefs.h>
#include    <string.h>

#include    "mdlfence.h"
#include    "fenccmd.h"

#include    <msoutput.fdf>
#include    <msparse.fdf>
#include    <msrsrc.fdf>
#include    <msundo.fdf>
#include    <msstate.fdf>
#include    <mslocate.fdf>
#include    <msmisc.fdf>
#include    <mselemen.fdf>
#include    <msvec.fdf>

/*-----
-+
|                                     |
|   Local type definitions   |
|                                     |
+-----
*/
/*-----
-+
|                                     |
|   Private Global variables|
|                                     |
+-----
*/
/*-----
-+
|                                     |
|   Public Global variables |
|                                     |
+-----
*/
Dpoint3d   anchorPoint;
int        copyMode;

/*-----
-+
|                                     |
|   Local function declarations |
|                                     |

```

```

|
+-----|
*/
/*-----
--+
|
| name  eraseFence          |
|
| authorBSI          6/91   |
|
+-----|
*/
Public voideraseFence
(
void
)
{
    /* convert the current fence to a shape in dgnbuf */
    mdlFence_toShape (dgnBuf);

    /* save the modification to the fence in the undo buffer */
    mdlUndo_saveElement (0L, dgnBuf, UNDO_MODIFYFENCE);

    /* erase the existing fence */
    mdlElement_display (dgnBuf, ERASE);
}

/*-----
--+
|
| name  redrawFence          |
|
| authorBSI          6/91   |
|
+-----|
*/
Public voidredrawFence
(
void
)
{
    mdlElement_display (dgnBuf, NORMALDRAW);
    mdlFence_fromShape (dgnBuf);
}

/*-----
--+

```

```

|
| name endFenceCommand
|
| authorBSI          6/91
|
+-----+
*/
Public voidendFenceCommand
(
void
)
{
    mdlState_startDefaultCommand ();
}

/*-----+
-+
|
| name elementModify_transform - used by scale, rotate, spin,|
| mirror, etc.|
|
| authorBSI          6/91
|
+-----+
*/
Public intelementModify_transform
(
MSElementUnion*el,
FenceCopyParams*mp
)
{
    if (mdlElement_transform (el, el, &mp->transform))
    {
        /* if it went off the design plane, dont do modification */
        mdlOutput_rscPrintf (MSG_ERROR, NULL, MESSAGEID_Messages,
            MSGID_OffDesignPlane);

        return MODIFY_STATUS_ERROR;
    }
    return  MODIFY_STATUS_REPLACE;
}

/*-----+
-+
|
| name moveFunction - function to actually change an element|
| in the fence. Should change the element in memory|

```

```

|          and return a status indicating what it did|
|
| authorBSI          2/91      |
|
+-----+
*/
Private intmoveFunction
(
MSElement    *el,
FenceCopyParams    *cp
)
{
    /* offset the element by the distance passed in our structure */
    if (mdlElement_offset (el, el, &cp->distance))
    {
        /* if it went off the design plane, dont do modification */
        mdlOutput_rscPrintf (MSG_ERROR, NULL, MESSAGEID_Messages,
                             MSGID_OffDesignPlane);
        return MODIFY_STATUS_ERROR;
    }

    /* tell mdlModify_elementSingle that we changed it */
    return MODIFY_STATUS_REPLACE;
}

/*-----+
+
|
| name  fenceModify_transformOrig |
|
| authorBSI          5/90      |
|
+-----+
*/
Public intfenceModify_transformOrig
(
FenceCopyParams    *cp
)
{
    ULong    filePos;
    int    fileNum;

    /* get file position of current element */
    filePos = mdlElement_getFilePos (FILEPOS_CURRENT, &fileNum);

    /* do actual modification */

```

```

        mdlModify_elementSingle (fileNum, filePos, MODIFY_REQUEST_HEADERS,
                                MODIFY_ORIG, elementModify_transform, cp, 0L);

    return  SUCCESS;
}

/*-----
-+
|
| name  fenceModify_transformCopy |
|
| authorBSI          5/90      |
|
|-----
*/
Public intfenceModify_transformCopy
(
FenceCopyParams*cp
)
{
    ULong   filePos;
    int     fileNum, modifyFlag = MODIFY_COPY;

    /* get file position of current element */
    filePos = mdlElement_getFilePos (FILEPOS_CURRENT, &fileNum);

    /* do actual modification */
    mdlModify_elementSingle (fileNum, filePos, MODIFY_REQUEST_HEADERS,
                            modifyFlag, elementModify_transform, cp, 0L);

    return  SUCCESS;
}

/*-----
-+
|
| name  copyContent0p - called for each element satisfying|
|          fence criteria      |
|
| authorBSI          2/91      |
|
|-----
*/
Private intcopyContent0p
(
FenceCopyParams    *cp
)

```

```

    {
        ULong    filePos;
        int      fileNum, modifyFlag = MODIFY_COPY;

        /* get file position of current element */
        filePos = mdlElement_getFilePos (FILEPOS_CURRENT, &fileNum);

        /* do actual modification */
        mdlModify_elementSingle (fileNum, filePos, MODIFY_REQUEST_HEADERS,
                                modifyFlag, moveFunction, cp, 0L);

        return  SUCCESS;
    }

/*-----
-+
|                                     |
| name moveContentOp - called for every element in fence. |
|                                     |
| authorBSI              7/86      |
|                                     |
+-----
*/
Public intmoveContentOp
(
    FenceCopyParams    *cp
)
{
    ULong    filePos;
    int      fileNum;

    /* get file position of current element */
    filePos = mdlElement_getFilePos (FILEPOS_CURRENT, &fileNum);

    /* do actual modification */
    mdlModify_elementSingle (fileNum, filePos, MODIFY_REQUEST_HEADERS,
                            MODIFY_ORIG, moveFunction, cp, 0L);

    return  SUCCESS;
}

/*-----
-+
|                                     |
| name moveOrCopyFence                |
|                                     |
| authorBSI              5/90      |
|                                     |

```



```

|
+-----+
*/
Private voidmoveOrCopyFence
(
Dpoint3d    *currPoint
)
{
    Dpoint3dpt1;
    FenceCopyParams cp;

    /* zero out all copy parameters structure */
    memset (&cp, 0, sizeof(cp));

    /* get the distance we are going to move */
    mdlVec_subtractPoint (&cp.distance, currPoint, &anchorPoint);

    /* now do the fence procesing */
    mdlFence_process (&cp);

    /* free memory allocated for graphic group remapping */
    mdlModify_freeGGMap ((MdlCopyParams *)&cp);
}

/*-----+
-+
| name dragElement - dynamic function for fence copy/move|
| authorBSI          2/91          |
|-----+
*/
Private voiddragElement
(
Dpoint3d    *currPoint
)
{
    Dpoint3ddragDist;

    /* get distance from anchor point */
    mdlVec_subtractPoint (&dragDist, currPoint, &anchorPoint);

    /* offset element in dgnBuf (will be the fence) */
    mdlElement_offset (dgnBuf, dgnBuf, &dragDist);
}

```

```

/*-----
-+
|
| name moveFenceCont_firstPt - define point (origin) for move|
|
| authorBSI              7/86      |
|
+-----
*/
Private voidmoveFenceCont_firstPt
(
Dpoint3d    *point
)
{
    /* save anchor point */
    anchorPoint = *point;

    /* dont do dynamics if he is using a working set */
    if (!tcb->wssect)
    {
        mdlFence_toShape (dgnBuf);
        mdlState_dynamicUpdate (dragElement, 1);
    }

    /* change data point function to one that processes fence */
    mdlState_setFunction (STATE_DATAPOINT, moveOrCopyFence);

    mdlOutput_rscPrintf (MSG_PROMPT, NULL, MESSAGEID_Messages,
                        MSGID_DefinedDistancePrompt);
}

/*-----
-+
|
| name moveFence - moves the actual fence to a new position|
|
| authorBSI              7/86      |
|
+-----
*/
PrivatevoidmoveFence
(
Dpoint3d    *currPoint
)
{
    Dpoint3ddistance;

```

```

    /* erase current fence */
    eraseFence();

    /* offset the fence by the same distance we offset all components
    */
    mdlVec_subtractPoint (&distance, currPoint, &anchorPoint);
    mdlElement_offset (dgnBuf, dgnBuf, &distance);

    /* re-display the new fence */
    redrawFence();

    /* save new anchor point */
    anchorPoint = *currPoint;
}

/*-----
-+
|
| name  copyFenceContents          |
|
| authorBSI          2/91          |
|
|-----
*/
PrivatevoidcopyFenceContents
(
void
)
cmdNumberCMD_MFENCE_COPY
{
    /* set search masks to allow any element type */
    mdlLocate_allowLocked ();

    /* set up fence state functions */
    mdlState_startFenceCommand (copyContentOp, moveFence,
                                moveFenceCont_firstPt, copyFenceContents,
                                -108, -141, FENCE_CLIP_COPY);
}

/*-----
-+
|
| name  moveFenceContents          |
|
| authorBSI          7/86          |
|
|-----

```

```

+-----
*/
Private void moveFenceContents
(
void
)
cmdNumberCMD_MFENCE_MOVE
{
    /* don't allow locked elements */
    mdlLocate_normal ();

    /* set up fence state functions */
    mdlState_startFenceCommand (moveContentOp, moveFence,
                                moveFenceCont_firstPt, moveFenceContents,
                                -107, -141, FENCE_CLIP_ORIG);
}

/*-----
-+
|
| name main
|
| authorBSI      8/89
|
+-----
*/
int main
(
int argc,
char *argv[]
)
{
    RscFileHandle rfHandle;

    /* Open our file for access to command table and dialog */
    mdlResource_openFile (&rfHandle, NULL, FALSE);

    /* Load the command table */
    if (mdlParse_loadCommandTable (NULL) == NULL)
        mdlOutput_rscPrintf (MSG_ERROR, NULL, MESSAGEID_Messages,
                             MSGID_CmdTblOpenErr);

    return 0;
}

```

trumpet

dynamic.mc

```

/*-----+
|
| Copyright (1995) Bentley Systems, Inc., All rights reserved.|
|
| "MicroStation" is a registered trademark and "MDL" and "MicroCSL"|
| are trademarks of Bentley Systems, Inc.    |
|
| Limited permission is hereby granted to reproduce and modify this|
| copyrighted material provided that the resulting code is used only|
| in conjunction with Bentley Systems products under the terms of the|
| license agreement provided therein, and that this notice is retained|
|
| in its entirety in any such reproduction or modification.|
|
+-----+
*/
/*-----+
-+
|
| $Logfile:   J:/mdl/examples/doc/dynamic.mcv  $
| $Workfile:  dynamic.mc  $
| $Revision:  5.3  $
| $Date:      20 Jun 1995 08:49:56  $
|
+-----+
*/
/*-----+
-+
|
| dynamic.mc
|
| Example program that contains a few MDL commands that illustrate|
| various techniques to use the concept of "simple dynamics".|
| These are not intended to be true application commands (since they|
| have no resource files associated with them and they really do not|
| do sufficient prompting or error checking and reporting). Rather,|
| they are intended to show various aspects of MicroStation's|
| dynamics and how MDL primitive functions interact with same.|
|
+-----+
*/
/*-----+
-+
|
| Include Files
|

```

```

|
+-----+
*/
#include    <mselems.h>
#include    <global.h>
#include    <mdl.h>
#include    <tcb.h>
#include    <userfnc.h>

/*-----+
|
|   Private function declarations
|
+-----*/
void moveAnElement();
void placeACell();

/*-----+
|
|   MOVE EXISTING ELEMENT example
|
+-----*/
/*-----+
|
|   name  getMoveDistance- get distance from anchor point to
|           current point.
|
|   authorBSI          10/90
|
+-----*/
Private void getMoveDistance
(
Dpoint3d    *distance,          /* <= distance from anchor */
Dpoint3d    *pt                /* => current point */
)
{
    Dpoint3d anchor;
    /*-----
    The element location logic stores the point where the user located
    the element in the global variable "statedata.pointstack[0]". We
    use that point for our anchor point. Since it is stored in integer
    world coordinates we need to convert it to Dpoint3d first.
    -----*/
    mdlCnv_IPointToPoint(&anchor, statedata.pointstack);

    /* subtract anchor point from current point to get the distance */
    mdlVec_subtractPoint (distance, pt, &anchor);
}

```

```

/*-----+
|
| name offsetElement - dynamic function for "moveAnElement".
|
| authorBSI          10/90
|
|-----*/
Private void offsetElement
(
Dpoint3d *pt      /* => current location of cursor */
)
{
    Dpoint3d distance;
    getMoveDistance (&distance, pt);
    mdlElement_offset (dgnBuf, dgnBuf, &distance);
}

/*-----+
|
| name elementModify_move - called indirectly for each element
|                          to be moved by mdlModify_elementMulti.
|
| authorBSI          10/90
|
|-----*/
Private int elementModify_move
(
MSElementUnion  *el,      /* <> element to be modified */
Dpoint3d         *distance /* => from params in mdlModify_element... */
)
{
    /* offset the element by specified distance */
    if (mdlElement_offset (el, el, distance))
        return MODIFY_STATUS_ERROR;

    /* indicate that we did something to the element */
    return MODIFY_STATUS_REPLACE;
}

/*-----+
|
| name moveElement_accept - data point function for
|                          "moveAnElement" example.
|
| authorBSI          10/90
|
|-----*/
Private void moveElement_accept

```

```

(
Dpoint3d      *pt          /* => final point for move element */
)
{
    Dpoint3d distance;
    int fileNum, modifyFlags = MODIFY_ORIG;
    ULong filePos;

    /* Get the distance by which to move each element. */
    getMoveDistance (&distance, pt);

    /* Get the file position and file number of the element to move. */
    filePos = mdlElement_getFilePos (FILEPOS_CURRENT, &fileNum);

    /*-----
    If we are using a selection set we want to draw the element(s) in
    their new location in their normal color. If we are moving a single
    element, we want to draw them in the hilite color.
    -----*/
    if (!mdlSelect_isActive())
        modifyFlags |= MODIFY_DRAWINHILITE;

    /* Now move each element the user accepted. */
    mdlModify_elementMulti (fileNum, filePos, MODIFY_REQUEST_HEADERS,
                            modifyFlags, elementModify_move, &distance, 1);

    /* Save new anchor point (the current acceptance point) */
    mdlCnv_DPointToIPoint (statedata.pointstack, pt);

    /* Reload the dynamic buffer with the new element. */
    mdlDynamic_loadElement (NULL, fileNum, filePos);

    /* Change the reset function to restart the whole command rather
    than to continue to look for additional element about the last
    data point. */
    mdlState_setFunction (STATE_RESET, moveAnElement);

    /* See if we were single-shotted, or if we acted upon a selection set
    (in which case we don't want to keep performing the same action). */
    mdlState_checkSingleShot ();
}

/*-----+
|
| name moveAnElement - emulate (almost) the MicroStation
|           "MOVE ELEMENT" command.
|
| authorBSI           10/90
|
|-----*/
cmdName void moveAnElement(void)
{

```



```

/*-----
Start a "modification" command. This command will use groups
(either selection sets or graphic groups), and requires 2
data points to complete.
-----*/
mdlState_startModifyCommand (moveAnElement, moveElement_accept,
                             offsetElement, NULL, NULL, 0, 0, TRUE, 2);

/* initialize the element location logic so that it starts at the
beginning of the file finding all displayable element types */
mdlLocate_init();
}

/*-----+
|
| CELL PLACEMENT example
|
|-----*/
Dpoint3d cellOrigin; /* origin of cell to be placed,
                     set by first data point. */
int cellView;         /* view of cell to be placed,
                     set by first data point. */

/*-----+
|
| name  getCellTransform - return transformation matrix based
|       on the current view and active scales.
|
| authorBSI          10/90
|
|-----*/
Private void getCellTransform
(
Transform *cellTrans, /* <= transformation to be set */
int view              /* => view in which to draw cell */
)
{
    RotMatrix rMatrix;

    /* get the inverse of the rotation matrix for the given view */
    mdlRMatrix_fromView (&rMatrix, view, FALSE);
    mdlRMatrix_invert (&rMatrix, &rMatrix);

    /* convert to a transformation matrix and scale the columns */
    mdlTMatrix_fromRMatrix (cellTrans, &rMatrix);
    mdlTMatrix_scale (cellTrans, cellTrans, tcb->xactscle, tcb->yactscle,
                     tcb->zactscle);
}

/*-----+

```

```

|
| name getCursorAngle - find the angle to rotate the cell
|                      for the "placeACell" command.
|
| authorBSI            8/91
|
+-----*/
Private double getCursorAngle
(
Dpoint3d *pt          /* => current location of cursor */
)
{
    double  zRotation;
    if (pt->x - cellOrigin.x == 0)
        zRotation = fc_piover2;
    else
        zRotation=atan((pt->y-cellOrigin.y)/(pt->x-cellOrigin.x));
    if (pt->x - cellOrigin.x < 0)
        zRotation += fc_pi;

    return (zRotation);
}

/*-----+
|
| name drawRotatingCell - dynamic function for rotating the
|                      cell for the "placeACell" command.
|
| authorBSI            8/91
|
+-----*/
Private int drawRotatingCell
(
Dpoint3d *pt,          /* => current location of cursor */
int view               /* => current view */
)
{
    Transform tMatrix;
    getCellTransform (&tMatrix, cellView);

    /* rotate the transformation matrix to current cursor position. */
    mdlMatrix_rotateByAngles (&tMatrix, &tMatrix, fc_zero, fc_zero,
        getCursorAngle (pt));
    mdlMatrix_setTranslation (&tMatrix, &cellOrigin);
    mdlElement_transform (dgnBuf, dgnBuf, &tMatrix);
}

/*-----+
|
|

```

```

| name drawCell - dynamic function for the "placeACell"
|             command.
|
| authorBSI           10/90
|
+-----*/
Private intdrawCell
(
Dpoint3d *pt,      /* => current location of cursor */
int view           /* => current view */
)
{
    Transform tMatrix;
    getCellTransform (&tMatrix, view);
    /* rotate the transformation matrix by the active angle */
    mdlTMatrix_rotateByAngles (&tMatrix, &tMatrix, fc_zero, fc_zero,
                               tcb->actangle * fc_piover180);
    mdlTMatrix_setTranslation (&tMatrix, pt);
    mdlElement_transform (dgnBuf, dgnBuf, &tMatrix);
}

/*-----+
|
| name acceptPlaceCell - second data point function for
|             "placeACell".
|
| authorBSI           10/90
|
+-----*/
Private void acceptPlaceCell
(
Dpoint3d *pt,      /* => final data point */
int view           /* => view for same */
)
{
    char activeCell[10];
    TransformtMatrix;
    RotMatrixrMatrix;

    getCellTransform (&tMatrix, cellView);
    /* rotate the transformation matrix to current cursor position. */
    mdlTMatrix_rotateByAngles (&tMatrix, &tMatrix, fc_zero, fc_zero,
                               getCursorAngle (pt));
    mdlRMatrix_fromTMatrix (&rMatrix, &tMatrix);

    mdlParams_getActive (activeCell, ACTIVEPARAM_CELLNAME);
    mdlCell_placeCell (0L, &cellOrigin, NULL, &rMatrix, NULL, 0, FALSE,
0,

```

```

        tcb->ext_locks.sharedCells, activeCell));

    mdlState_restartCurrentCommand();
}

/*-----
-+
|
| name placeACellPt1 - first data point function for|
|               "placeACell".                      |
|
| authorBSI      8/91      |                        |
|
|-----
*/
Private voidplaceACellPt1
(
    Dpoint3d      *pt,      /* => first data point */
    int           view      /* => view for same */
)
{
    /* Set the cell origin and view */
    cellOrigin = *pt;
    cellView = view;

    /* Set function to use second data point */
    mdlState_setFunction (STATE_DATAPOINT, acceptPlaceCell);

    /* establish a new "simple dynamics" function to rotate the cell
during
    cursormovement */
    mdlState_dynamicUpdate (drawRotatingCell, TRUE);
}

/*-----
-+
|
| name restartDefault      |
|
| authorBSI      10/90      |
|
|-----
*/
Private voidrestartDefault
(
    void
)

```

```

    {
        mdlState_startDefaultCommand();
    }

/*-----
-+
|
| name          placeACell - simple example of how an MDL application can|
|                emulate the MicroStation "PLACE CELL ABSOLUTE"|
|                command.
|
| author        BSI          10/90
|
+-----*/
cmdNamevoidplaceACell
(
void
)
{
    MSElementDescr *cellDscrP;
    char    activeCell[10];
    mdlState_startPrimitive (placeACellPt1, placeACell, 0, 0);
    mdlState_setFunction (STATE_RESET, restartDefault);

    /* first get the name of the active cell */
    mdlParams_getActive (activeCell, ACTIVEPARAM_CELLNAME);

    /* get an element descriptor to hold the active cell (for use with
       the dynamic function) */
    mdlCell_getElmDscr (&cellDscrP, NULL, 0L, NULL, NULL, NULL, NULL, 0,
        tcb->ext_locks.sharedCells, activeCell);

    /*-----
    -----
        Load the element descriptor into MicroStation's dynamic buffer.
        This is the only place in MDL where an application creates an
    element
        descriptor and does not have to free it. Once an element
    descriptor
        is passed to mdlDynamic_setElmDscr, MicroStation owns the
    descriptor
        and will free it when finished with it.
    -----*/
    mdlDynamic_setElmDscr (cellDscrP);

```

```

        /* establish a "simple dynamics" function to draw the cell during
cursor
movement */
        mdlState_dynamicUpdate (drawCell, TRUE);
    }

```

bspline.mc

```

/*-----+
|
| Copyright (1995) Bentley Systems, Inc., All rights reserved.
|
| "MicroStation" is a registered trademark and "MDL" and "MicroCSL"
| are trademarks of Bentley Systems, Inc.
|
| Limited permission is hereby granted to reproduce and modify this
| copyrighted material provided that the resulting code is used only
| in conjunction with Bentley Systems products under the terms of the
| license agreement provided therein, and that this notice is retained
|
| in its entirety in any such reproduction or modification.
|
+-----*/
/*-----+
|
| $Logfile: J:/mdl/examples/doc/bspline.mcv $
| $Workfile: bspline.mc $
| $Revision: 5.3 $
| $Date: 20 Jun 1995 08:49:42 $
|
+-----*/
/*-----+
|
| bspline.mc - examples for the mdlBspline_ functions.
|
| This file is intended as an adjunct to the MDL manual to
| illustrate MDL built-in function calling conventions and parameter
| values. While it can be compiled, it does NOT, on its own,
| constitute a workable MDL example.
|
+-----*/
/*-----+
|
| Include Files
|

```

```

|
+-----*/
#include <mdl.h> /* system include files */
#include <global.h>
#include <mselems.h>
#include <mdlbspln.h>

/*-----+
| name      defineACurve      |
| author    BSI                1/91 |
|
+-----*/
Private int defineACurve
(
MSBsplineCurve *curve
)
{
    int status;
    MSElementDescr *edP;

    memset (curve, 0, sizeof (*curve));

    curve->params.order = 3;
    curve->params.numPoles = 5;

    if ((status = mdlBspline_allocateCurve (&curve)) != SUCCESS)
return (status);

    curve->poles[0].x = 0.0;
    curve->poles[0].y = 10.0;
    curve->poles[0].z = 5.0;
    curve->poles[1].x = 1.0;
    curve->poles[1].y = 10.0;
    curve->poles[1].z = 10.0;
    curve->poles[2].x = 2.0;
    curve->poles[2].y = 10.0;
    curve->poles[2].z = 15.0;
    curve->poles[3].x = 3.0;
    curve->poles[3].y = 10.0;
    curve->poles[3].z = 20.0;
    curve->poles[4].x = 4.0;
    curve->poles[4].y = 10.0;
    curve->poles[4].z = 25.0;

    mdlBspline_computeKnotVector (curve->knots, &curve->params, NULL);

```

```

        if ((status = mdlBspline_createCurve (&edP, NULL, curve)) ==
SUCCESS)
    {
        mdlElmdscr_display (edP, 0, NORMALDRAW);
        mdlElmdscr_add (edP);
        mdlElmdscr_freeAll (&edP);
    }

    return (SUCCESS);
}

/*-----
-+
|
| name          defineASurface |
|
| author        BSI            1/91      |
|
|-----
+*/
Private int defineASurface
(
MSBsplineSurface    *surface
)
{
    int            status;
    MSElementDescr    *edP;

    memset (surface, 0, sizeof (*surface));

    surface->uParams.order    = 3;
    surface->uParams.numPoles = 4;
    surface->vParams.order    = 2;
    surface->vParams.numPoles = 2;

    if ((status = mdlBspline_allocateSurface (&surface)) != SUCCESS)
return (status);

    surface->poles[0].x = 0.0;
    surface->poles[0].y = 10.0;
    surface->poles[0].z = 5.0;
    surface->poles[1].x = 1.0;
    surface->poles[1].y = 10.0;
    surface->poles[1].z = 10.0;
    surface->poles[2].x = 2.0;
    surface->poles[2].y = 10.0;

```



```

        surface->poles[2].z = 15.0;
        surface->poles[3].x = 3.0;
        surface->poles[3].y = 10.0;
        surface->poles[3].z = 20.0;
        surface->poles[4].x = 0.0;
        surface->poles[4].y = 30.0;
        surface->poles[4].z = 25.0;
        surface->poles[5].x = 1.0;
        surface->poles[5].y = 30.0;
        surface->poles[5].z = 25.0;
        surface->poles[6].x = 2.0;
        surface->poles[6].y = 30.0;
        surface->poles[6].z = 25.0;
        surface->poles[7].x = 3.0;
        surface->poles[7].y = 30.0;
        surface->poles[7].z = 25.0;

        mdlBspline_computeKnotVector (surface->uKnots, &surface->uParams,
        NULL);
        mdlBspline_computeKnotVector (surface->vKnots, &surface->vParams,
        NULL);

        if ((status = mdlBspline_createSurface (&edP, NULL, surface)) ==
        SUCCESS)
        {
            mdlElmdscr_display (edP, 0, NORMALDRAW);
            mdlElmdscr_add (edP);
            mdlElmdscr_freeAll (&edP);
        }

        return (SUCCESS);
    }

/*-----
-+
|          createASpiral          |
| name          |                  |
| author    BSI          1/91    |
|          |                  |
+-----
*/
Public void createASpiral
(
void
)
{

```

```

int          status;
double       initialRadius, finalRadius, sweepAngle;
Dpoint3d     startPt, tangentPt, directionPt;
MSBsplineCurve curve;
MSElementDescr *edP;

memset (&curve, 0, sizeof(curve));

initialRadius = 1000.0;
finalRadius   = 100.0;
sweepAngle    = fc_pi;
startPt.x     = 0.0;
startPt.y     = 0.0;
startPt.z     = 0.0;
tangentPt.x   = 10.0;
tangentPt.y   = 10.0;
tangentPt.z   = 0.0;
directionPt.x = 10.0;
directionPt.y = 20.0;
directionPt.z = 0.0;

if (status = mdlBspline_spiral (&curve, initialRadius, finalRadius,
                                sweepAngle, &startPt, &tangentPt,
                                &directionPt))
{
    mdlOutput_printf (MSG_ERROR, "ERROR : # %d", status);
    return;
}

if (status = mdlBspline_createCurve (&edP, NULL, &curve))
{
    mdlOutput_printf (MSG_ERROR, "ERROR : # %d", status);
    mdlBspline_freeCurve (&curve);
    return;
}

mdlElmdscr_display (edP, 0, NORMALDRAW);
mdlElmdscr_add (edP);
mdlElmdscr_freeAll (&edP);

mdlBspline_freeCurve (&curve);
}

/*-----
-+
|
| name          createAHelix          |
|

```

```

|
| author      BSI              1/91      |
|
+-----+
*/
Public void createAHelix
(
void
)
{
    int          status;
    double        initialRadius, finalRadius, pitchHeight;
    Dpoint3d      startPt, axis1Pt, axis2Pt;
    MSBsplineCurve curve;
    MSElementDescr *edP;

    memset (&curve, 0, sizeof(curve));

    initialRadius = 1000.0;
    finalRadius   = 500.0;
    pitchHeight   = 10.0;

    startPt.x     = 1000.0;
    startPt.y     = 0.0;
    startPt.z     = 0.0;
    axis1Pt.x     = 0.0;
    axis1Pt.y     = 0.0;
    axis1Pt.z     = 0.0;
    axis2Pt.x     = 0.0;
    axis2Pt.y     = 10.0;
    axis2Pt.z     = 0.0;

    if (status = mdlBspline_helix (&curve, initialRadius, finalRadius,
                                   pitchHeight, &startPt,
                                   &axis1Pt, &axis2Pt, TRUE))
    {
        mdlOutput_printf (MSG_ERROR, "ERROR : # %d", status);
        return;
    }

    if (status = mdlBspline_createCurve (&edP, NULL, &curve))
    {
        mdlOutput_printf (MSG_ERROR, "ERROR : # %d", status);
        mdlBspline_freeCurve (&curve);
        return;
    }
}

```

```

    mdlElmdscr_display (edP, 0, NORMALDRAW);
    mdlElmdscr_add (edP);
    mdlElmdscr_freeAll (&edP);

    mdlBspline_freeCurve (&curve);
}

/*-----
-+
|
| name      displayKnotInfo |
|
| author    BSI             1/91   |
|
+-----
*/
Public void displayKnotInfo
(
    MSBsplineCurve    *curve,
    double             parameter
)
{
    int    i, status, index, totalKnots, numDistinct,
           multiplicities[MAX_KNOTS], numInflects;
    double    knotTolerancedistinctKnots[MAX_KNOTS]greville[MAX_POLES];

    knotTolerance = mdlBspline_knotTolerance (curve);

    totalKnots = curve->params.closed ?
        curve->params.numPoles + 2 * curve->params.order - 1 :
        curve->params.numPoles + curve->params.order;
    if (totalKnots > MAX_KNOTS)
        return;

    mdlBspline_getKnotMultiplicity (distinctKnots, multiplicities,
                                    &numDistinct,
                                    curve->knots, curve->params.numPoles,
                                    curve->params.order, curve->params.closed,
                                    knotTolerance);

    for (i=0; i < numDistinct; i++)
        printf ("Knot [#i] = %3.4f\n", i, distinctKnots[i]);

    mdlBspline_computeGrevilleAbscissa (greville, curve->knots,
                                        curve->params.numPoles,
                                        curve->params.order,
                                        curve->params.closed,

```

```

        knotTolerance);

    for (i=0; i < curve->params.numPoles; i++)
        printf ("Greville Abscissa [#d] = %3.4f\n", i, greville[i]);

    mdlBspline_findSpan (&index, curve->knots, curve->params.numPoles,
        curve->params.order, curve->params.closed,
        parameter);
    printf ("Parameter value %3.4f is in span %d\n", parameter, index);

    mdlBspline_inflectionPoints (NULL, NULL, &numInflects, curve, NULL);
    printf ("The curve contains %d inflection points \n", numInflects);
}

/*-----
-+
|
| name          convertElementDescr|
|
| author      BSI          1/91    |
|
|-----
*/
Private int convertElementDescr
(
MSElementDescr**newEdP,
MSElementDescr*edP
)
{
    int          status;
    MSBsplineCurve    curve;
    MSBsplineSurface  surface;
    MSElementDescr    *endCaps;

    if ((edP->el.ehdr.type == LINE_ELM) ||
        (edP->el.ehdr.type == LINE_STRING_ELM) ||
        (edP->el.ehdr.type == SHAPE_ELM) ||
        (edP->el.ehdr.type == CURVE_ELM) ||
        (edP->el.ehdr.type == CPLX_STRING_ELM) ||
        (edP->el.ehdr.type == CPLX_SHAPE_ELM) ||
        (edP->el.ehdr.type == ELLIPSE_ELM) ||
        (edP->el.ehdr.type == ARC_ELM))
    {
        if ((status = mdlBspline_convertToCurve (&curve, edP)) ||
            (status = mdlBspline_createCurve (newEdP, NULL, &curve)))
            return (status);
        mdlBspline_freeCurve (&curve);
    }
}

```

```

    }
    else if ((edP->el.ehdr.type == SURFACE_ELM) ||
             (edP->el.ehdr.type == CONE_ELM) ||
             (edP->el.ehdr.type == SOLID_ELM))
    {
        if ((status = mdlBspline_convertToSurface (&surface, edP)) ||
            (status = mdlBspline_createSurface (newEdP, NULL, &surface)))
            return (status);
        mdlBspline_freeSurface (&surface);

        if (edP->el.ehdr.type == SOLID_ELM ||
            (edP->el.ehdr.type == CONE_ELM && ! edP->el.cone_3d.b.surf))
        {
            if (status = mdlBspline_convertToEndcaps (&endCaps, edP))
                return (status);
            mdlElmdscr_addToChain (*newEdP, endCaps);
        }
    }
    return (SUCCESS);
}

/*-----
-+
| name      sampleCurvePoints|
|
| author    BSI              1/91    |
|
+-----
*/
Private void sampleCurvePoints
(
void
)
{
    int          i, status, numPoints;
    Dpoint3d     *points, line[2];
    MSBsplineCurve curve;
    MSElementUnion u;

    if (status = defineACurve (&curve))
    {
        mdlOutput_printf (MSG_ERROR, "ERROR : # %d", status);
        return;
    }

    numPoints = 50;

```

```

        points = NULL;

        if ((status = mdlBspline_evaluateCurve (&points, NULL, &numPoints,
&curve))
            != SUCCESS)
        {
            mdlOutput_printf (MSG_ERROR, "ERROR : # %d", status);
            return;
        }

        for (i = 0; i < numPoints; i++)
        {
            line[0] = line[1] = points[i];
            mdlLine_create (&u, NULL, line);
            u.line_3d.dhdr.symb.b.weight = 5;
            mdlElement_display (&u, NORMALDRAW);
        }

        /* Free the memory allocated by mdlBspline_evaluateCurve */
        free (points);
        mdlBspline_freeCurve (&curve);
    }

/*-----
-+
|
| name      sampleSurfacePoints|
|
| author    BSI          1/91   |
|
|-----
*/
Private void sampleSurfacePoints
(
void
)
{
    int          i, status, numPoints[2];
    Dpoint3d     *points, line[2];
    MSBsplineSurface  surface;
    MSElementUnion  u;

    if (status = defineASurface (&surface))
    {
        mdlOutput_printf (MSG_ERROR, "ERROR : # %d", status);
        return;
    }
}

```

```

    numPoints[0] = 50;
    numPoints[1] = 100;
    points = NULL;

    if ((status = mdlBspline_evaluateSurface (&points, NULL, numPoints,
                                              &surface))
        != SUCCESS)
    {
        mdlOutput_printf (MSG_ERROR, "ERROR : # %d", status);
        return;
    }

    for (i = 0; i < numPoints[0]*numPoints[1]; i++)
    {
        line[0] = line[1] = points[i];
        mdlLine_create (&u, NULL, line);
        u.line_3d.dhdr.symb.b.weight = 5;
        mdlElement_display (&u, NORMALDRAW);
    }

    /* Free the memory allocated by mdlBspline_evaluateSurface */
    free (points);

    mdlBspline_freeSurface (&surface);
}

/*-----
-+
| name      modifyBsplineCurve|
|
| author    BSI          1/91  |
|
+-----
*/
Private void modifyBsplineCurve
(
void
)
{
    int      i, status, action, intValue, numKnots;
    double   doubleValue, knotTolerance;
    MSBsplineCurvecurve, result;
    MSElementDescr *edP=NULL;

    memset (&result, 0, sizeof (result));

```



```
    if (status = defineACurve (&curve))
    {
        mdlOutput_printf (MSG_ERROR, "ERROR : # %d", status);
        return;
    }

    action = 0;
    switch (action)
    {
        /* Reverse direction of curve */
        case 0:
            status = mdlBspline_reverseCurve (&result, &curve);
            break;

        /* Make into a multiple segment Bezier curve */
        case 1:
            status = mdlBspline_makeBezier (&result, &curve);
            break;

        /* Close curve */
        case 2:
            status = mdlBspline_closeCurve (&result, &curve);
            break;

        /* Open curve */
        case 3:
            doubleValue = fc_zero;
            status = mdlBspline_openCurve (&result, &curve, doubleValue);
            break;

        /* Elevate the degree of the curve */
        case 4:
            intValue = 4;
            status = mdlBspline_elevateDegree (&result, &curve, intValue);
            break;

        /* Add a knot to the knot vector of the curve */
        case 5:
            doubleValue = fc_onehalf;
            status = mdlBspline_copyCurve (&result, &curve);
            if (status == SUCCESS)
            {
                knotTolerance = mdlBspline_knotTolerance (&result);
                status = mdlBspline_addKnot (&result, doubleValue,
                                            knotTolerance,
                                            intValue, TRUE);
            }
    }
```

```

        break;

/* Change the weights of the curve */
case 6:
    doubleValue = fc_onehalf;
    status = mdlBspline_makeRational (&result, &curve);
    if (status == SUCCESS)
    {
        mdlBspline_unWeightPoles (result.poles, result.poles,
                                   result.weights, result.params.numPoles);
        for (i=0; i < result.params.numPoles; i++)
            result.weights[i] *= doubleValue;
        mdlBspline_weightPoles (result.poles, result.poles,
                                result.weights, result.params.numPoles);
    }
    break;

/* Partially delete the curve */
case 7:
    doubleValue = fc_onehalf;
    status = mdlBspline_segmentCurve (&result, &curve,
                                       fc_zero, doubleValue);

    break;

/* Change the parameterization of the curve */
case 8:
    status = mdlBspline_copyCurve (&result, &curve);
    if (status == SUCCESS)
    {
        numKnots = result.params.closed ?
            result.params.numPoles + 2 * result.params.order-1:
            result.params.numPoles + result.params.order;

        for (i=0; i < numKnots; i++)
            result.knots[i] *= doubleValue;

        mdlBspline_normalizeKnotVector (result.knots,
                                         result.params.numPoles,
                                         result.params.order,
                                         result.params.closed);
    }
    break;

}

if (status)
    mdlOutput_printf (MSG_ERROR, "ERROR : # %d", status);

```

```

        else
        {
            mdlBspline_createCurve (&edP, NULL, &result);
            mdlElmdscr_display (edP, 0, NORMALDRAW);
        }

        mdlBspline_freeCurve (&result);
        mdlBspline_freeCurve (&curve);
        mdlElmdscr_freeAll (&edP);
    }

/*-----
-+
| name      modifyBsplineSurface|
| author    BSI          1/91   |
|-----
*/
Private void modifyBsplineSurface
(
void
)
{
    int      i, status, action, intValue;
    double   doubleValue;
    Dpoint2d start, finish;
    MSBsplineSurfacesurface, result;
    MSElementDescr *edP=NULL;

    memset (&result, 0, sizeof (result));
    if (status = defineASurface (&surface))
    {
        mdlOutput_printf (MSG_ERROR, "ERROR : # %d", status);
        return;
    }

    action = 0;
    switch (action)
    {
        /* Make into a multiple patch Bezier surface */
        case 0:
            status = mdlBspline_makeBezierSurface (&result, &surface);
            break;

        /* Open surface in U direction */

```

```

case 1:
    doubleValue = fc_zero;
    status = mdlBspline_openSurface (&result, &surface,
                                     doubleValue, BSSURF_U);

    break;

/* Elevate the degree of the surface in V direction */
case 2:
    intValue = 4;
    status = mdlBspline_elevateDegreeSurface (&result, &surface,
                                              intValue, BSSURF_V);

    break;

/* Partially delete the surface */
case 3:
    start.x = 0.2;
    start.y = 0.0;
    finish.x = 0.8;
    finish.y = 0.8;
    status = mdlBspline_segmentSurface (&result, &surface,
                                       &start, &finish);

    break;

}

if (status)
mdlOutput_printf (MSG_ERROR, "ERROR : # %d", status);
else
{
mdlBspline_createSurface (&edP, NULL, &result);
mdlElmdscr_display (edP, 0, NORMALDRAW);
}

mdlBspline_freeSurface (&result);
mdlBspline_freeSurface (&surface);
mdlElmdscr_freeAll (&edP);
}

/* -----
--+
| name      makeDerivedSurface|
|
| author    BSI              1/91    |
|
+-----
*/

```

```

Private void makeDerivedSurface
(
MSBsplineCurve*curve1,
MSBsplineCurve*curve2,
MSBsplineCurve*curve3,
MSBsplineCurve*curve4,
int          surfaceType
)
{
    int          i, status;
    double        start, sweep;
    Dpoint3d      center, axis, delta;
    Dvector3d     orientation1, orientation2;
    MSBsplineCurve  curves[4];
    MSBsplineSurface surface;
    MSElementDescr *edP=NULL;

    memset (&surface, 0, sizeof(surface));
    memset (curves, 0, sizeof(curves));

    switch (surfaceType)
    {
    case 0:
        status = mdlBspline_ruledSurface (&surface, curve1, curve2);
        break;

    case 1:
        if (!(status = mdlBspline_copyCurve (&curves[0], curve1)) ||
            (status = mdlBspline_copyCurve (&curves[1], curve2)) ||
            (status = mdlBspline_copyCurve (&curves[2], curve3)) ||
            (status = mdlBspline_copyCurve (&curves[3], curve4)))
        {
            status = mdlBspline_coonsPatch (&surface, curves);
        }
        for (i=0; i < 4; i++)
            mdlBspline_freeCurve (&curves[i]);
        break;

    case 2:
        center.x = 0.0;
        center.y = 0.0;
        center.z = 0.0;
        axis.x = 0.0;
        axis.y = 0.0;
        axis.z = 100.0;

        start = fc_zero;
    }
}

```

```

        sweep = fc_pi;

        status = mdlBspline_surfaceOfRevolution (&surface, curve1,
                                                &center, &axis, start, sweep);
        break;

    case 3:
        delta.x = 0.0;
        delta.y = 0.0;
        delta.z = 100.0;

        start = fc_zero;
        sweep = fc_pi;

        status = mdlBspline_surfaceOfProjection (&surface, curve1,
&delta);
        break;

    case 4:
        orientation1.org.x = 0.0;    orientation1.org.x = 0.0;
        orientation1.org.y = 0.0;    orientation1.org.y = 1.0;
        orientation1.org.z = 0.0;    orientation1.org.z = 0.0;

        orientation2.org.x = 0.0;    orientation2.org.x = 1.0;
        orientation2.org.y = 0.0;    orientation2.org.y = 0.0;
        orientation2.org.z = 0.0;    orientation2.org.z = 0.0;

        status = mdlBspline_skinPatch (&surface, curve1, curve2, curve3,
                                      &orientation1, &orientation2,
                                      TRUE);

        break;
    }

    if (status)
        mdlOutput_printf (MSG_ERROR, "ERROR : # %d", status);
    else
    {
        mdlBspline_createSurface (&edP, NULL, &surface);
        mdlElmdscr_display (edP, 0, NORMALDRAW);
    }

    mdlBspline_freeSurface (&surface);
    mdlElmdscr_freeAll (&edP);
}

/*-----
-+

```

```

| name      makeCompatible |
|
| author    BSI            1/91    |
|
+-----+
*/
Private int makeCompaible
(
MSBsplineCurve*curve1,
MSBsplineCurve*curve2,
MSBsplineCurve*curve3,
MSBsplineCurve*curve4,
int      numberOfCurves
)
{
    int      i, status;
    MSBsplineCurve  *inputPtr[4];compatiblePtr[4];compatibleCurves[4];

    if (numberOfCurves > 4)
return (ERROR);
    if (numberOfCurves == 1)
return (SUCCESS);

    memset (compatibleCurves, 0, sizeof(compatibleCurves));

    if (numberOfCurves == 2)
return mdlBspline_make2CurvesCompatible (curve1, curve2);

    inputPtr[0] = curve1;    inputPtr[1] = curve2;
    inputPtr[2] = curve3;    inputPtr[3] = curve4;

    for (i=0; i < numberOfCurves; i++)
compatiblePtr[i] = &compatibleCurves[i];

    if (status = mdlBspline_makeCurvesCompatible (compatiblePtr,
inputPtr,
                                numberOfCurves))
return (status);

    for (i=0; i < numberOfCurves; i++)
    {
        mdlBspline_freeCurve (inputPtr[i]);
        *inputPtr[i] = *compatiblePtr[i];
    }

    return (SUCCESS);
}

```

```

    }

/*-----
--+
|
| name      imposeBoundary |
|
| author    BSI             1/91   |
|
|-----
*/
Private int imposeBoundary
(
MSBsplineSurface *surface,
MSBsplineCurve *boundary
)
{
    int i, status, numPts;
    double tolerance;
    Dpoint3d *points=NULL, *pP;
    Dvector3d direction;
    MSElementUnion u;
    MSElementDescr *edP=NULL;

    tolerance = fc_10000;

    if (status = mdlBspline_createCurve (&edP, NULL, boundary))
return (status);
    else
    {
        mdlElmdscr_extractNormal (&direction.end, &direction.org, edP, NULL);
        mdlElmdscr_freeAll (&edP);
    }

    if (status = mdlBspline_imposeBoundary (surface, boundary,
tolerance,
                                &direction, &points, &numPts))
return (status);

    for (i=0, pP=points; i < numPts; i++, pP++)
    {
        mdlLine_create (&u, NULL, pP);
        u.line_3d.dhdr.symb.b.weight = 5;
        mdlElement_display (&u, NORMALDRAW);
    }

    /* Free memory allocated in mdlBspline_imposeBoundary */

```



```

    free (points);

    if (status = mdlBspline_extractBoundary (&edP, surface, tolerance))
return (status);

    mdlElmdscr_display (edP, 0, NORMALDRAW);
    mdlElmdscr_freeAll (&edP);
}

/*-----
-+
|          |
| name      extractSurfaceCurves |
|          |
| author    BSI          1/91      |
|          |
+-----
*/
Private int extractSurfaceCurves
(
MSBsplineSurface  *surface,
Dpoint3d          *pt
)
{
    int      i, status;
    double   distance;
    Dpoint2d  parameter;
    Dpoint3d  closestPt;
    MSBsplineCurve  curve;
    MSElementDescr  *edP=NULL;

    if (status = mdlBspline_minimumDistanceToSurface (&distance,
&closestPt,
                                     &parameter, pt, surface))
return (status);

    if (status = mdlBspline_extractIsoCurve (&edP, surface,
parameter.x, BSSURF_U))
return (status);
    else
    {
        mdlElmdscr_display (edP, 0, NORMALDRAW);
        mdlElmdscr_freeAll (&edP);
    }

    if (status = mdlBspline_extractIsoCurve (&edP, surface,
parameter.y, BSSURF_V))

```

```

        return (status);
    else
    {
        mdlElmdscr_display (edP, 0, NORMALDRAW);
        mdlElmdscr_freeAll (&edP);
    }

    memset (&curve, 0, sizeof (curve));
    if (status = mdlBspline_extractProfile (&curve, surface))
        return (status);
    else
    {
        mdlElmdscr_display (edP, 0, NORMALDRAW);
        mdlElmdscr_freeAll (&edP);
    }

    return (SUCCESS);
}

/*-----
-+
|
| name      curveInfo          |
|
| author    BSI                1/91    |
|
|-----
*/
Private int curveInfo
(
    MSBsplineCurve    *curve,
    Dpoint3d          *pt
)
{
    int                i, status;
    double              distance, parameter, wtDerv[3], curvature, torsion;
    Dpoint3d            closestPt, frame[3], line[2];
    MSElementUnion     u;

    if (status = mdlBspline_minimumDistanceToCurve (&distance,
&closestPt,
&parameter, pt, curve))
        return (status);

    if (status = mdlBspline_computeDerivatives (frame, wtDerv, curve, 2,
&parameter))

```

```

    return (status);

    if (status = mdlBspline_frenetFrame (frame, line, &curvature,
&torsion,
                                curve, parameter))
    return (status);

    for (i=0; i < 3; i++)
    {
        mdlVec_projectPoint (line+1, line, frame+i, fc_10000);
        mdlLine_create (&u, NULL, line);
        mdlElement_display (&u, NORMALDRAW);
    }

    return (SUCCESS);
}

/*-----
-+
| name      leastSquaresCurve|
| author    BSI              1/91    |
|-----
*/
Private void leastSquaresCurve
(
Dpoint3d    *data,
int         numDataPoints
)
{
    int      status;
    MSBsplineCurvecurve;
    MSElementDescr    *edP=NULL;

    memset (&curve, 0, sizeof (curve));

    curve.params.order    = 3;
    curve.params.numPoles = 5;

    if (status = mdlBspline_leastSquaresToCurve (&curve, NULL, NULL,
data,
                                NULL, numDataPoints))
    {
        mdlOutput_printf (MSG_ERROR, "ERROR : # %d", status);
        return;
    }
}

```

```

    }

    mdlBspline_createCurve (&edP, NULL, &curve);
    mdlElmdscr_display (edP, 0, NORMALDRAW);
    mdlElmdscr_freeAll (&edP);

    mdlBspline_freeCurve (&curve);
}

/*-----
-+
| name      leastSquaresSurface|
| author    BSI          1/91   |
|-----
*/
Private void leastSquaresSurface
(
    Dpoint3d    *data,
    int          *numPointsInEachRow,
    int          numberOfRows
)
{
    int          status;
    MSBsplineSurface surface;
    MSElementDescr *edP=NULL;

    memset (&surface, 0, sizeof (surface));

    surface.uParams.order    = 3;
    surface.uParams.numPoles = 5;
    surface.vParams.order    = 2;
    surface.vParams.numPoles = 4;

    if (status = mdlBspline_leastSquaresToSurface (&surface, NULL, NULL,
                                                    data, NULL,
                                                    numPointsInEachRow,
                                                    numberOfRows))
    {
        mdlOutput_printf (MSG_ERROR, "ERROR : # %d", status);
        return;
    }

    mdlBspline_createSurface (&edP, NULL, &surface);
    mdlElmdscr_display (edP, 0, NORMALDRAW);

```

```

    mdlElmdscr_freeAll (&edP);

    mdlBspline_freeSurface (&surface);
}

/*-----
-+
|
| name      convertBackToNonBspline|
|
| author    BSI          1/91      |
|
|-----
*/
Private int convertBackToNonBspline
(
MSElementDescr    **out,
MSElementDescr    *in
)
{
    int      status;
    MSBsplineCurve    curve;
    MSBsplineSurface    surface;

    if ((status = mdlBspline_convertToCurve (&curve, in)) == SUCCESS)
    {
        status = mdlBspline_extractFromCurve (out, &curve);
        mdlBspline_freeCurve (&curve);
        return (status);
    }
    else if ((status = mdlBspline_convertToSurface (&surface, in)) ==
SUCCESS)
    {
        status = mdlBspline_extractFromSurface (out, &surface);
        mdlBspline_freeSurface (&surface);
        return (status);
    }
    else
        return (status);
}

/*-----
-+
|
| name      extractCurveInfo|
|
| author    BSI          1/91      |
|
|-----

```

```

|
+-----|
*/
Private void extractCurveInfo
(
MSElementDescr    *in
)
{
    int            type, rational;
    double         *weights, *knots;
    Dpoint3d       *poles;
    BsplineParam    params;
    MSElementUnion header;

    poles = (Dpoint3d *) 0;
    knots = weights = (double *) 0;

    mdlBspline_extractCurve (&header, &type, &rational, NULL, &params,
                             &poles, &knots, &weights, in);

    if (poles)    free (poles);
    if (knots)    free (knots);
    if (weights)  free (weights);
}

/*-----
-+
|
| name      extractSurfaceInfo|
|
| author    BSI              1/91    |
|
+-----
*/
Private void extractSurfaceInfo
(
MSElementDescr    *in
)
{
    int            type, rational, numBounds, holeOrigin;
    double         *weights, *uKnots, *vKnots;
    Dpoint3d       *poles;
    BsplineParam    uParams, vParams;
    MSElementUnion header;

    poles = (Dpoint3d *) 0;
    uKnots = vKnots = weights = (double *) 0;

```

```

        mdlBspline_extractSurface (&header, &type, &rational, NULL,
&uParams,
                                &vParams, &poles, &uKnots, &vKnots, &weights,
                                &holeOrigin, &numBounds, NULL, in);

        if (poles)      free (poles);
        if (uKnots)      free (uKnots);
        if (vKnots)      free (vKnots);
        if (weights)     free (weights);
    }

/*-----
-+
|
| name      appendCurves  |
|
| author    BSI           1/91   |
|
|-----
*/
Private int appendCurve
(
MSBsplineCurve  *curve1,
MSBsplineCurve  *curve2
)
{
    int          status;
    MSBsplineCurve  result;
    MSElementDescr  *edP=NULL;

    if (status = mdlBspline_appendCurves (&result, curve1, curve2))
return (status);

    mdlBspline_createCurve (&edP, NULL, &result);
    mdlElmdscr_display (edP, 0, NORMALDRAW);

    mdlElmdscr_freeAll (&edP);
    mdlBspline_freeCurve (&result);

    return (SUCCESS);
}

/*-----
-+
|
|

```

```

| name      appendSurfaces |
|
| author    BSI              1/91      |
|
+-----+
*/
Private int appendSurface
(
MSBsplineSurface *surface1,
MSBsplineSurface *surface2
)
{
    int      status;
    MSBsplineSurface result;
    MSElementDescr *edP=NULL;

    if (status = mdlBspline_appendSurfaces (&result, surface1,
surface2,
                                         BSSURF_U))
        return (status);

    mdlBspline_createSurface (&edP, NULL, &result);
    mdlElmdscr_display (edP, 0, NORMALDRAW);

    mdlElmdscr_freeAll (&edP);
    mdlBspline_freeSurface (&result);

    return (SUCCESS);
}

```

mline.mc

```

/*-----+
-+
|
| Copyright (1993) Bentley Systems, Inc., All rights reserved.|
|
| "MicroStation", "MDL", and "MicroCSL" are trademarks of Bentley|
| Systems, Inc. and/or Intergraph Corporation.|
|
| Limited permission is hereby granted to reproduce and modify this|
| copyrighted material provided that the resulting code is used only in
|

```



```
| conjunction with Bentley Systems products under the terms of the|
| license agreement provided therein, and that this notice is retained|
| in its entirety in any such reproduction or modification.|
|
+-----+
*/
/*-----+
+
|
| $Logfile: I:/mdl/examples/mline/mline.mcv $
| $Workfile: mline.mc $
| $Revision: 5.2 $
| $Date: 01 Apr 1993 17:52:42 $
|
+-----+
*/
/*-----+
+
|
| Function -
|
| This example MDL application adds one new command to MicroStation.|
|
| PLACE BLOCK MLINE - Place Multi-line block.|
| Command operation is the same as the standard|
| PLACE BLOCK command.|
| The Multi-line produced uses the active|
| multi-line definition.|
|
| To use this application: |
| 1. Compile the application -> bmake mline.mke|
| 2. Start MicroStation |
| 3. Load the application -> keyin MDL LOAD MLINE|
| 4. Use the new command listed above|
|
| Note:
|
| If multiline compatibility is ON (SET COMPATIBLE MLINE ON),|
| multi-lines will be saved as MicroStation 3.x (IGDS 8.8)|
compatible|
| primitive elements. |
|
| - - - - -|
|
| Public Routine Summary -|
|
| main - main entry point |
```

```
| mline_blockFirstPoint - First data point function|
| mline_blockLastPoint - Last data point function|
| mline_blockComplete - Process multi-line placement in dgn file|
| mline_generateBlock - Multi-line generation/dynamics function|
| mline_blockStart - Place multi-line command function|
| mline_addBreaks - insert breaks into multi-line segments|
| mline_writeMline - write to dgn file|
|
```

```
+-----|
*/
```

```
/*-----
--+
```

```
|
```

```
| Include Files |
```

```
|
```

```
+-----|
*/
```

```
#include <mdl.h>
```

```
#include <global.h>
```

```
#include <mselems.h>
```

```
#include <userfnc.h>
```

```
#include <rsdefs.h>
```

```
#include "mlineids.h"
```

```
#include "mlinecmd.h"
```

```
/*-----
--+
```

```
|
```

```
| Local function declarations |
```

```
|
```

```
+-----|
*/
```

```
Private void mline_blockFirstPoint ();
```

```
Private void mline_blockLastPoint ();
```

```
Private void mline_blockComplete ();
```

```
Private void mline_generateBlock ();
```

```
/*-----
--+
```

```
|
```

```
| name main |
```

```
|
```

```
| authorBSI 3/90 |
```

```
|
```

```
+-----|
*/
```

```

main ()
{
    RscFileHandle    rfHandle;

    /*-----
--
    Load the application command table and resources
-----
*/
    if (mdlParse_loadCommandTable (NULL) == NULL)
        mdlOutput_rscPrintf (MSG_ERROR, NULL, 0, 4);

    mdlResource_openFile (&rfHandle, NULL, FALSE);

    mdlCurrTrans_begin ();
}

/*-----
-+
|
| ** PLACE BLOCK MLINE command|
|
|-----
*/
/*-----
-+
|
| name          mline_blockStart - called for PLACE BLOCK MLINE keyin|
|
| author      BSI          11/90      |
|
|-----
*/
Private void mline_blockStart
(
)
cmdNumberCMD_PLACE_BLOCK_MLINE
{
    mdlState_startPrimitive
(mline_blockFirstPoint,/* Data point function*/
 mline_blockStart,/* Reset function*/
 MSGID_PlaceMultiLine,/* Tool / function name*/
 MSGID_EnterFirstPoint);/* Prompt number*/

    /*-----
--
    Set the current transform to identity so that the point passed to

```

```

        mline_blockFirstPoint is in a known coordinate system (world
coordinates)
-----
*/
    mdlCurrTrans_identity ();
}

/*-----
--
|
| name      mline_blockFirstPoint|
|
| author    BSI          11/90    |
|
|-----
+-----
*/
Private void mline_blockFirstPoint
(
Dpoint3d *dataPoint,      /* => Line start point (origin)      */
int      view
)
{
    /*-----
    --
    Set the current transform to the view of the first data point.
    Then transform the first data point from world coordinates to
    the current transform.
    -----
    */
    mdlCurrTrans_rotateByView (view);
    mdlCurrTrans_invtransPointArray (statedata.dPointStack, dataPoint,
1);

    /*-----
    --
    Set dynamic and data point functions.
    -----
    */
    mdlState_setFunction (STATE_DATAPOINT, mline_blockLastPoint);
    mdlState_setFunction (STATE_COMPLEX_DYNAMICS, mline_generateBlock);

    mdlOutput_rscPrintf (MSG_PROMPT, NULL, STRINGID_Messages,
        MSGID_EnterCorner);
}

/*-----
--

```

```

| name      mline_blockLastPoint|
|
| author    BSI          11/90   |
|
+-----+
*/
Private void mline_blockLastPoint
(
DPoint3d *dataPoint      /* => Cursor location      */
)
{
    mline_generateBlock (dataPoint, 0, NORMALDRAW, TRUE);

    mdlState_restartCurrentCommand ();
}

/*-----+
-+
| name      mline_generateBlock|
|
| author    BSI          11/90   |
|
+-----+
*/
Private void mline_generateBlock
(
Dpoint3d *dataPoint,      /* => Cursor location      */
int view,                 /* => view (not used)      */
int drawMode,             /* => Current draw / erase mode */
int writeMode             /* => If TRUE, add elements to file */
)
{
    MSElementUnion mline;
    DPoint3d dPoints[5];

    /*-----
    --
    Define 5 points for the multiline based on the two corners entered.
    Note: The last point must be a repeat of the first point in order
           for the multiline to be closed.
    -----
    */
    dPoints[0] = dPoints[1] = dPoints[3] = dPoints[4] =
        statedata.dPointStack[0];
    dPoints[2] = *dataPoint;

```

```

    dPoints[1].x = dPoints[2].x;
    dPoints[3].y = dPoints[2].y;

    if (mdlVec_pointEqualUOR (dPoints, dPoints+1) ||
        mdlVec_pointEqualUOR (dPoints, dPoints+3))
    return;

    if (mdlMline_create (&mline, NULL, NULL, dPoints, 5))
    return;

    mdlMline_setClosure (&mline, TRUE);

#ifdef ADD_BREAKS
    mline_addBreaks (&mline);
#endif

    mdlElement_display (&mline, drawMode);

    if (writeMode)
        mline_writeMline (&mline);
    }

#ifdef ADD_BREAKS
/*-----
-+
|
| name      mline_addBreaks |
|
| author    BSI             11/90   |
|
|-----
*/
Private void mline_addBreaks
(
MSElementUnion *mline
)
{
    DPoint3d dPoints[2];
    double   segLength;
    int      i, nPoints;

    /*-----
--
    Insert a break into each segment of the multi-line. (done only for
    an example of inserting breaks)

```

```

-----
*/
    mdlMline_getInfo (&nPoints, NULL, NULL, NULL, NULL, mline);

    for (i=0; i<nPoints-1; i++)
    {
        mdlMline_extractPoints (dPoints, mline, MASTERFILE, i, 2);
        segLength = mdlVec_distance (dPoints, dPoints+1);
        mdlMline_insertBreak (mline, i, segLength/4.0, segLength/2.0,
                               0xff, 0);
    }
}
#endif

/*-----
-+
| name          mline_writeMline|
| author      BSI          11/90  |
|-----
*/
Private void mline_writeMline
(
MSElementUnion *mline
)
{
    MSElementDescr *edP;
    int      compatMode;

    mdlParams_getActive (&compatMode, ACTIVEPARAM_MLINECOMPAT);

    /*-----
    --
    Check the setting of the current multiline compatibility mode.
    If multiline compatibility is ON, write the multiline to the file
    as IGDS 8.8 compatible primitives.
    If multiline compatibility is OFF, validate the multiline element
    and add it to the file.
    -----
    */
    if (compatMode)
    {
        if (mdlMline_getElementDescr (&edP, mline, MASTERFILE, TRUE))
            return;
    }
}

```

```

mdlElmdscr_add (edP);
mdlElmdscr_freeAll (&edP);
}
else
{
    if (mdlMline_validate (mline))
        return;

    mdlElement_add (mline);
}
}

```

cell.mc

patrnmdl.mc

params.mc

lvlnames.mc

```

/*-----
-+
|
| Copyright (1995) Bentley Systems, Inc., All rights reserved.
|
| "MicroStation" is a registered trademark and "MDL" and "MicroCSL"
| are trademarks of Bentley Systems, Inc.
|
| Limited permission is hereby granted to reproduce and modify this
| copyrighted material provided that the resulting code is used only
| in conjunction with Bentley Systems products under the terms of the
| license agreement provided therein, and that this notice is retained
|
| in its entirety in any such reproduction or modification.
|
+-----
*/
/*-----
-+
|
| $Logfile: J:/mdl/examples/doc/lvlnames.mcv $
| $Workfile: lvlnames.mc $
| $Revision: 5.2 $
| $Date: 20 Jun 1995 08:49:50 $
|
|

```



```

+-----+
*/
/*-----+
|
|   lvlnames.mc -- Named Level Dialog Hooks |
|                                           |
+-----+
*/
/*-----+
|
|   Include Files                          |
|                                           |
+-----+
*/
#include <mdl.h>
#include <dlogbox.h>
#include <dlogitem.h>
#include <dlogids.h>
#include <cexpr.h>
#include <rscdefs.h>
#include <system.h>
#include <global.h>
#include <levels.h>

#include <dlogman.fdf>

/*-----+
|
|   Local defines                          |
|                                           |
+-----+
*/
#define LEVELS_DISPLAY_ON29 /* cmdname.r resource index */
#define LEVELS_DISPLAY_OFF30 /* cmdname.r resource index */
#define SELECT_VIEW      2059 /* errors.r resource index */

/*-----+
|
|   Local type definitions |
|                                           |
+-----+
*/
typedef struct groupListData

```

```

    {
        char    **listPtrs;
        int      numRows; /* number of rows in list */
        int      rowSelected; /* row index currently selected */
        int      parentID; /* parentID of groups in list */
    } GroupListData;

typedef struct levelListData
{
    char    **listPtrs;
    int      numRows; /* number of rows in list */
    int      rowSelected; /* row index currently selected */
    int      parentID; /* parentID of levels in list */
    int      groupID; /* groupID of levels in list */
} LevelListData;

/*-----
-+
|                                     |
|   Private Global variables|       |
|                                     |
+-----
*/
Privatechar *parentString = "[..]";

/*-----
-+
|                                     |
|   Public Global variables |       |
|                                     |
+-----
*/
boolean turnLevelsOn;
short  levelMask[MAX_LEVELS/16];
short  sortOrder;
char   **editListP;
boolean isDialogClosing; /* Kludge. No way to avoid it. */

/*-----
-+
|                                     |
|   External variables      |       |
|                                     |
+-----
*/
/*-----
-+

```

```

|                                     |
|   Local function declarations |   |
|                                     |
+-----+
*/
voidchangeViewLevels();
voidgetLevelsList(), getGroupList();
intlevelSort();

/*ff Major Public Code Section */
/*-----+
|                                     |
|   Major Public Code Section|   |
|                                     |
+-----+
*/
/*-----+
|                                     |
|   name          handleChildDied|   |
|   author          ezs          11/90|   |
|                                     |   |
+-----+
*/
Private void handleChildDied
(
DialogBox*dbP,
int      childID/* => ID of child which died */
)
{
    DialogItem      *diP;
    GroupListData    **groupListP;
    LevelListData    **levelListP;

    if (childID == DIALOGID_LevelEdit)
    {
        diP = mdlDialog_itemGetByIndex (dbP, NAMEDLEVELS_LevelsList);
        levelListP = &diP->rawItemP->userDataP;

        loadLevelsList (dbP, (**levelListP).groupID);
    }
    else if (childID == DIALOGID_GroupEdit)
    {
        diP = mdlDialog_itemGetByIndex (dbP, NAMEDLEVELS_GroupList);
        groupListP = &diP->rawItemP->userDataP;
    }
}

```

```

        loadGroupList (dbP, (**groupListP).parentID);
    }
}

/*-----
-+
|
| name          levels_dialogHook
|
| author        slk
|
| 8/90
|
+-----
*/
Public void levels_dialogHook
(
    DialogMessage*dmP
)
{
    dmP->msgUnderstood = TRUE;

    switch (dmP->messageType)
    {
    case DIALOG_MESSAGE_CREATE:
        dmP->u.create.interests.keystrokes = TRUE;
        dmP->u.create.interests.mouses = TRUE;
        break;

    case DIALOG_MESSAGE_CHILDDESTROYED:
        if (dmP->u.childDestroyed.actionType == ACTIONBUTTON_OK)
            handleChildDied (dmP->db, dmP->u.childDestroyed.childDialogId);
        break;

    case DIALOG_MESSAGE_DESTROY:
        break;

    default:
        dmP->msgUnderstood = FALSE;
        break;
    }
}

/*-----
-+
|
| name          levels_levelsListHook
|
|
|
+-----

```

```

| author          slk                                8/90          |
|-----|-----|-----|-----|-----|-----|-----|-----|
*/
Public void levels_levelsListHook
(
    DialogItemMessage    *dimP
)
{
    int    status;

    dimP->msgUnderstood = TRUE;

    switch (dimP->messageType)
    {
    case DITEM_MESSAGE_CREATE:
        break;

    case DITEM_MESSAGE_DESTROY:
        freeLevelList (dimP->db);
        break;

    case DITEM_MESSAGE_BUTTON:
        if (dimP->u.button.buttonTrans == BUTTONTRANS_UP)
        {
            GroupListData    **listP;
            DialogItem*diP;

            diP = mdlDialog_itemGetByIndex (dimP->db,
                                           NAMEDLEVELS_GroupList);
            mdlDialog_listClearSelection (diP->rawItemP);
            listP = &diP->rawItemP->userDataP;
            (**listP).rowSelected = -1;
        }
        break;

    case DITEM_MESSAGE_STATECHANGED:
        doLevelListPick (dimP);
        break;

    default:
        dimP->msgUnderstood = FALSE;
        break;
    }
}

```

```

/*-----
-+
|
| name          levels_groupListHook
|
| author        slk
|
| 8/90
+-----
*/
Public void levels_groupListHook
(
DialogItemMessage *dimP
)
{
    static boolean doubleClick = FALSE;

    dimP->msgUnderstood = TRUE;

    switch (dimP->messageType)
    {
    case DITEM_MESSAGE_CREATE:
        loadGroupList (dimP->db, 0);
        loadLevelsList (dimP->db, 0);
        break;

    case DITEM_MESSAGE_DESTROY:
        freeGroupList (dimP->db);
        break;

    case DITEM_MESSAGE_BUTTON:
        if (dimP->u.button.buttonTrans == BUTTONTRANS_UP)
        {
            LevelListData **listP;
            DialogItem*diP;

            diP = mdlDialog_itemGetByIndex (dimP->db,
                                           NAMEDLEVELS_LevelsList);
            mdlDialog_listClearSelection (diP->rawItemP);
            listP = &diP->rawItemP->userDataP;
            (**listP).rowSelected = -1;
        }

        if (dimP->u.button.buttonTrans == BUTTONTRANS_UP &&
            dimP->u.button.upNumber == 2)
        {
            doubleClick = TRUE;

```

```

        doGroupListPick (dimP, TRUE);
    }
else
if (dimP->u.button.buttonTrans == BUTTONTRANS_TIMEOUT &&
    dimP->u.button.upNumber == BUTTONTIMEOUT_DOUBLECLICK)
{
    /* Make sure we got only a single click */
    if (doubleClick)
    {
        doubleClick = FALSE;
        break;
    }
    doubleClick = FALSE;

    mdlWindow_cursorTurnOff ();
    doGroupListPick (dimP, FALSE);
}
break;

default:
    dimP->msgUnderstood = FALSE;
    break;
}
}

/*-----
-+
| name          levels_pulldownHook
| author        slk
|               8/90
|-----
*/
Public void levels_pulldownHook
(
DialogItemMessage *dimP
)
{
    dimP->msgUnderstood = TRUE;

    switch (dimP->messageType)
    {
case DITEM_MESSAGE_BUTTON:
    {
        int subItem;
        longmenuID, itemArg;

```

```
mdlDialog_menuBarGetSelection (NULL, &menuID, &subItem,  
    &itemArg, dimP->dialogItemP->rawItemP);  
switch (itemArg)  
{  
    case MENUSEARCHID_NamedLevels_Open:  
        doFileOpen (dimP);  
        break;  
  
    case MENUSEARCHID_NamedLevels_Save:  
        doFileSave (dimP);  
        break;  
  
    case MENUSEARCHID_NamedLevels_AddLevel:  
        doAddLevel (dimP);  
        break;  
  
    case MENUSEARCHID_NamedLevels_DeleteLevel:  
        doDeleteLevel (dimP);  
        break;  
  
    case MENUSEARCHID_NamedLevels_EditLevel:  
        doEditLevel (dimP);  
        break;  
  
    case MENUSEARCHID_NamedLevels_AddGroup:  
        doAddGroup (dimP);  
        break;  
  
    case MENUSEARCHID_NamedLevels_DeleteGroup:  
        doDeleteGroup (dimP);  
        break;  
  
    case MENUSEARCHID_NamedLevels_EditGroup:  
        doEditGroup (dimP);  
        break;  
  
    case MENUSEARCHID_NamedLevels_OpenGroup:  
        doOpenGroup (dimP);  
        break;  
  
    case MENUSEARCHID_NamedLevels_DisplayOn:  
        turnLevelsOn = ON;  
        doDisplayObject (dimP);  
        break;  
  
    case MENUSEARCHID_NamedLevels_DisplayOff:
```



```

        turnLevelsOn = FALSE;
        doDisplayObject (dimP);
        break;

    default:
        break;
    }
}
break;

default:
    dimP->msgUnderstood = FALSE;
    break;
}
}

/*-----
-+
| name          levels_menuBarHook
| author        slk                      8/90
|-----
*/
Public void levels_menuBarHook
(
DialogItemMessage *dimP
)
{
    dimP->msgUnderstood = TRUE;

    switch (dimP->messageType)
    {
    case DITEM_MESSAGE_BUTTON:
        {
            RawItemHdr *menuBarP = dimP->dialogItemP->rawItemP;

            if (dimP->u.button.buttonTrans != BUTTONTRANS_DOWN)
                break;

            /* maintain state of menus */
            doMaintainMenus (menuBarP, dimP);
            break;
        }
    }

    default:

```

```

        dimP->msgUnderstood = FALSE;
        break;
    }
}

/*-----
-+
|
| name          doMaintainMenus
|
| author        slk
|                                     9/90
|
+-----
*/
Private doMaintainMenus
(
RawItemHdr          *menuBarP,    /* => dialog box menu bar */
DialogItemMessage *dimP          /* => menu bar item */
)
{
    boolean    levelSelected, groupSelected, levelsLoaded;
    int        rowSelected;
    Ditem_PulldownMenuItem mbarItem;
    DialogItem *diP;
    RawItemHdr *rihP;

    /* named level definitions in memory ? */
    levelsLoaded = (designLevels.levels || designLevels.groups);

    /* check for currently selected group */
    diP = mdlDialog_itemGetByIndex (dimP->db, NAMEDLEVELS_GroupList);
    rihP = diP->rawItemP;
    mdlDialog_listGetSelection (&rowSelected, NULL, rihP);
    groupSelected = (rowSelected >= 0) ? TRUE : FALSE;

    /* check for currently selected level */
    diP = mdlDialog_itemGetByIndex (dimP->db, NAMEDLEVELS_LevelsList);
    rihP = diP->rawItemP;
    mdlDialog_listGetSelection (&rowSelected, NULL, rihP);
    levelSelected = (rowSelected >= 0) ? TRUE : FALSE;

    /* File/Save menu item */
    if (!mdlDialog_menuBarFindItem (&mbarItem, NULL, menuBarP, NULL,
                                    PULLDOWNMENUID_NamedLevelsFile,
                                    MENUSEARCHID_NamedLevels_Save))
        mdlDialog_textPDMItemSetEnabled (&mbarItem, levelsLoaded);
}

```

```

/* Levels/Delete menu item */
if (!mdlDialog_menuBarFindItem (&mbarItem, NULL, menuBarP, NULL,
                                PULLDOWNMENUID_NamedLevelsLevels,
                                MENUSEARCHID_NamedLevels_DeleteLevel))
mdlDialog_textPDMItemSetEnabled (&mbarItem, levelSelected);

/* Levels/Edit menu item */
if (!mdlDialog_menuBarFindItem (&mbarItem, NULL, menuBarP, NULL,
                                PULLDOWNMENUID_NamedLevelsLevels,
                                MENUSEARCHID_NamedLevels_EditLevel))
mdlDialog_textPDMItemSetEnabled (&mbarItem, levelSelected);

/* Groups/Delete menu item */
if (!mdlDialog_menuBarFindItem (&mbarItem, NULL, menuBarP, NULL,
                                PULLDOWNMENUID_NamedLevelsGroups,
                                MENUSEARCHID_NamedLevels_DeleteGroup))
mdlDialog_textPDMItemSetEnabled (&mbarItem, groupSelected);

/* Groups/Edit menu item */
if (!mdlDialog_menuBarFindItem (&mbarItem, NULL, menuBarP, NULL,
                                PULLDOWNMENUID_NamedLevelsGroups,
                                MENUSEARCHID_NamedLevels_EditGroup))
mdlDialog_textPDMItemSetEnabled (&mbarItem, groupSelected);

/* Groups/Open menu item */
if (!mdlDialog_menuBarFindItem (&mbarItem, NULL, menuBarP, NULL,
                                PULLDOWNMENUID_NamedLevelsGroups,
                                MENUSEARCHID_NamedLevels_OpenGroup))
mdlDialog_textPDMItemSetEnabled (&mbarItem, groupSelected);

/* Display/On menu item */
if (!mdlDialog_menuBarFindItem (&mbarItem, NULL, menuBarP, NULL,
                                PULLDOWNMENUID_NamedLevelsDisplay,
                                MENUSEARCHID_NamedLevels_DisplayOn))
mdlDialog_textPDMItemSetEnabled (&mbarItem,
                                levelSelected || groupSelected);

/* Display/Off menu item */
if (!mdlDialog_menuBarFindItem (&mbarItem, NULL, menuBarP, NULL,
                                PULLDOWNMENUID_NamedLevelsDisplay,
                                MENUSEARCHID_NamedLevels_DisplayOff))
mdlDialog_textPDMItemSetEnabled (&mbarItem,
                                levelSelected || groupSelected);
}

/*-----
-+

```

```

| name          levels_sortHook
|
| author        slk
|
|
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
*/
Public void levels_sortHook
(
DialogItemMessage *dimP
)
{
    dimP->msgUnderstood = TRUE;

    switch (dimP->messageType)
    {
    case DITEM_MESSAGE_CREATE:
        sortOrder = LEVELEDIT_LevelNumberField;
        break;

    case DITEM_MESSAGE_INIT:
        {
            ValueUnionvalue;

            value.sLongFormat = 0;
            mdlDialog_itemSetValue (NULL, FMT_LONG, &value,
                                   NULL, dimP->db, dimP->itemIndex);

            break;
        }

    case DITEM_MESSAGE_STATECHANGED:
        {
            ValueUnion    value;
            DialogItem    *diP;
            LevelListData **listP;

            mdlDialog_itemGetValue (NULL, &value, NULL, dimP->db,
                                   dimP->itemIndex, 0);

            sortOrder = value.sWordFormat;
            diP = mdlDialog_itemGetByIndex (dimP->db,
                                           NAMEDLEVELS_LevelsList);
            listP = &diP->rawItemP->userDataP;

            if ((*listP).numRows > 1)
                mdlUtil_quickSort ((*listP).listPtrs, (*listP).numRows,
                                   sizeof (char *) * LEVEL_COLUMNS, levelSort);
        }
    }
}

```

```

        mdlDialog_listSetContents (diP->rawItemP, (**listP).numRows,
                                   LEVEL_COLUMNS, (**listP).listPtrs);
        mdlDialog_itemDraw (dimP->db, NAMEDLEVELS_LevelsList);
    }
    break;

default:
    dimP->msgUnderstood = FALSE;
    break;
}

}

/*-----
-+
|
| name          doFileOpen
|
| author        slk
|
| 9/90|
|
+-----
*/
Private void doFileOpen
(
DialogItemMessage *dimP
)
{
    char    levelResourceFile[256];
    char    tempString[80];
    int     status;

    mdlRsrc_fetchString (tempString, 716);
    status = mdlDialog_fileOpen (levelResourceFile, 0, 0, NULL,
                                "*.lvl", "MS_DATA", tempString);
    if (status != SUCCESS) return;

    mdlLevel_loadAllFromResource (levelResourceFile);

    /* create the group list starting from the root */
    loadGroupList (dimP->db, 0);

    /* create the level list starting from the root */
    loadLevelsList (dimP->db, 0);
}

/*-----
-+

```

```

| name          doFileSave |
| author        slk        9/90 |
|-----|
*/
Private void doFileSave
(
DialogItemMessage *dimP
)
{
    char    levelResourceFile[256];
    char    tempString[80];
    int     status;

    mdlRsrc_fetchString (tempString, 717);
    status = mdlDialog_fileCreate (levelResourceFile, 0, 0, NULL,
                                   "*.lvl", "MS_DATA", tempString);
    if (status != SUCCESS) return;

    mdlLevel_saveAllToResource (levelResourceFile);
}

/*-----
-+
| name          doDeleteLevel |
| author        slk        9/90 |
|-----|
*/
Private void doDeleteLevel
(
DialogItemMessage *dimP
)
{
    int     rowSelected, index;
    LevelListData*levelListP;
    DialogItem*diP;
    RawItemHdr*rihP;

    diP = mdlDialog_itemGetByIndex (dimP->db, NAMEDLEVELS_LevelsList);
    rihP = diP->rawItemP;
    if (!(levelListP = rihP->userDataP)) return;

```

```

    mdlDialog_listGetSelection (&rowSelected, NULL, rihP);
    if (rowSelected < 0) return;

    index = (rowSelected * LEVEL_COLUMNS) + 1;
    mdlLevel_deleteLevel (levelListP->listPtrs[index], levelListP-
>groupID);
    loadLevelsList (dimP->db, levelListP->groupID);
}

/*-----
-+
| name          doAddLevel                                     |
| author        slk                                           9/90|
|-----
*/
Private void doAddLevel
(
DialogItemMessage *dimP
)
{
    DialogItem*diP;
    RawItemHdr*rihP;
    int rowSelected;

    diP = mdlDialog_itemGetByIndex (dimP->db, NAMEDLEVELS_LevelsList);
    rihP = diP->rawItemP;
    editListP = &rihP->userDataP;

    (**((LevelListData **) editListP)).rowSelected = -1;

    mdlDialog_open (NULL, DIALOGID_LevelEdit);
}

/*-----
-+
| name          doEditLevel                                     |
| author        slk                                           9/90|
|-----
*/
Private void doEditLevel
(

```

```

DialogItemMessage    *dimP
)
{
    DialogItem*diP;
    RawItemHdr*rihP;
    int   rowSelected;

    diP = mdlDialog_itemGetByIndex (dimP->db, NAMEDLEVELS_LevelsList);
    rihP = diP->rawItemP;
    editListP = &rihP->userDataP;

    mdlDialog_listGetSelection (&rowSelected, NULL, rihP);
    (**((LevelListData **) editListP)).rowSelected = rowSelected;

    mdlDialog_open (NULL, DIALOGID_LevelEdit);
}

/*-----
-+
|
| name          doAddGroup
|
| author        slk          9/90|
|
+-----
*/
Private void doAddGroup
(
DialogItemMessage    *dimP
)
{
    DialogItem*diP;
    RawItemHdr*rihP;
    int   rowSelected;

    diP = mdlDialog_itemGetByIndex (dimP->db, NAMEDLEVELS_GroupList);
    rihP = diP->rawItemP;
    editListP = &rihP->userDataP;

    (**((GroupListData **) editListP)).rowSelected = -1;

    mdlDialog_open (NULL, DIALOGID_GroupEdit);
}

/*-----
-+
|
|

```



```

| name          doEditGroup                                     |
|                                                       |
| author        slk                                           9/90|
|                                                       |
+-----+
*/
Private void doEditGroup
(
DialogItemMessage *dimP
)
{
    DialogItem*diP;
    RawItemHdr*rihP;
    int rowSelected;

    diP = mdlDialog_itemGetByIndex (dimP->db, NAMEDLEVELS_GroupList);
    rihP = diP->rawItemP;
    editListP = &rihP->userDataP;

    mdlDialog_listGetSelection (&rowSelected, NULL, rihP);
    (((GroupListData **) editListP)).rowSelected = rowSelected;

    mdlDialog_open (NULL, DIALOGID_GroupEdit);
}

/*-----+
-+
| name          doDeleteGroup                                   |
|                                                       |
| author        slk                                           9/90|
|                                                       |
+-----+
*/
Private void doDeleteGroup
(
DialogItemMessage *dimP
)
{
    int rowSelected, index, groupID;
    GroupListData*groupListP;
    DialogItem*diP;
    RawItemHdr*rihP;

    diP = mdlDialog_itemGetByIndex (dimP->db, NAMEDLEVELS_GroupList);
    rihP = diP->rawItemP;
    if (!(groupListP = rihP->userDataP)) return;

```

```

mdlDialog_listGetSelection (&rowSelected, NULL, rihP);
if (rowSelected < 0) return;
/* skip parent group ([..])*/
if (!rowSelected && groupListP->parentID) return;

index = rowSelected;
mdlLevel_getGroupIDFromNameAndParent (&groupID,
    groupListP->listPtrs[index], groupListP->parentID);

/* delete selected group */
mdlLevel_deleteGroup (groupID, TRUE);

loadGroupList (dimP->db, groupListP->parentID);
}

/*-----+
| name          doOpenGroup                               |
| author        slk                                     9/90|
+-----*/
Private void doOpenGroup (DialogItemMessage *dimP)
{
    char parentName[64];
    int      rowSelected, parentID, groupID;
    GroupListData*groupListP, *newGroupListP;
    DialogItem*diP;
    RawItemHdr*rihP;

diP = mdlDialog_itemGetByIndex (dimP->db, NAMEDLEVELS_GroupList);
    rihP = diP->rawItemP;

if ((groupListP = rihP->userDataP))
{
    mdlDialog_listGetSelection (&rowSelected, NULL, rihP);
    /* move up to parent group */
    if (!rowSelected && groupListP->parentID)
    {
        if (!mdlLevel_getParentGroup (&parentID, parentName,
            groupListP->parentID, NULL))
        {
            loadGroupList (dimP->db, parentID);
            /* refresh our pointer to the group list */
            getGroupList (&newGroupListP, dimP->db);
            loadLevelsList (dimP->db, newGroupListP->parentID);
        }
    }
}

/* descend into group */
else if (rowSelected >= 0)
{

```

```

        if (!mdlLevel_getGroupIDFromNameAndParent (&groupID,
            groupListP->listPtrs[rowSelected], groupListP-
>parentID))
        {
            loadGroupList (dimP->db, groupID);
            loadLevelsList (dimP->db, groupID);
        }
    }
}

/*-----+
| name      groupSort  |
| author    EZS              11/90      |
+-----*/
Private int groupSort
(
char**string1,
char**string2
)
{
    if (**string1 == '[')
        return (-1);
    else if (**string2 == '[')
        return (1);

    return (strcmp (*string1, *string2));
}

/*-----+
| name      loadGroupList|
| author    slk              9/90      |
+-----*/
Private void loadGroupList
(
DialogBox    *db,
int          parentID/* => parentID of groups to load */
)
{
    char      levelPath[256];
    int       index;
    DialogItem *itemP;
    GroupListData **listP;
    LevelGroup parentGroup;

    itemP = mdlDialog_itemGetByIndex (db, NAMEDLEVELS_GroupList);
    listP = &itemP->rawItemP->userDataP;

    /* list already exists ? */

```

```

        if (*listP)
        {
            mdlDialog_listSetContents (itemP->rawItemP, 0, 0, NULL);
            mdlDialog_itemDraw (db, NAMEDLEVELS_GroupList);
            freeGroupList (db);
        }

*listP = calloc (sizeof (GroupListData), 1);
    if (!(*listP))
return;

(**listP).parentID    = parentID;
    (**listP).rowSelected = -1;

/* create a "pseudo" group for the parent of this list */
    if (parentID)
    {
        memset (&parentGroup, 0, sizeof (LevelGroup));
        parentGroup.parentID = parentID;
        parentGroup.groupID = -1;
        parentGroup.name = parentString;
        addGroupToList (listP, &parentGroup);
    }

for (index = 0; index < designLevels.numGroups; index++)
{
    if (designLevels.groups[index].parentID == parentID)
        addGroupToList (listP, &designLevels.groups[index]);
}

if ((**listP).numRows > 1)
    mdlUtil_quickSort ((**listP).listPtrs, (**listP).numRows,
        sizeof (char *), groupSort);

    mdlDialog_listSetContents (itemP->rawItemP, (**listP).numRows,
        1, (**listP).listPtrs);
    mdlDialog_itemDraw (db, NAMEDLEVELS_GroupList);

/* update dynamic path label */
    mdlLevel_buildLevelPath (levelPath, parentID, NULL);
    mdlDialog_itemSetLabel (db, NAMEDLEVELS_PathLabel, levelPath);
}

/*-----+
| name      levelSort  |
| author          EZS          11/90          |
+-----*/
Private int levelSort(char **string1, char **string2)
{
    if (sortOrder == LEVELEDIT_LevelNumberField)
    {

```

```

    int temp1, temp2;

    sscanf (string1[sortOrder], "%d", &temp1);
    sscanf (string2[sortOrder], "%d", &temp2);
    return ((temp1 < temp2) ? -1 : ((temp1 == temp2) ? 0 : 1));
}
else
return (strcmp (string1[sortOrder], string2[sortOrder]));
}

/*-----+
| name          loadLevelsList|
| author        slk           9/90      |
+-----*/
Private void loadLevelsList
(
DialogBox *db,
int groupID/* => groupID of levels to load */
)
{
    char parentName[64];
    int index, parentID;
    DialogItem *itemP;
    LevelListData **listP;

    itemP = mdlDialog_itemGetByIndex (db, NAMEDLEVELS_LevelsList);
    listP = &itemP->rawItemP->userDataP;

    /* list already exists ? */
    if (*listP)
    {
        mdlDialog_listSetContents (itemP->rawItemP, 0, 0, NULL);
        mdlDialog_itemDraw (db, NAMEDLEVELS_LevelsList);
        freeLevelList (db);
    }

    *listP = calloc (sizeof (LevelListData), 1);
    if (!(*listP))
return;

mdlLevel_getParentGroup (&parentID, parentName, groupID, NULL);
(**listP).parentID = parentID;
(**listP).groupID = groupID;
(**listP).rowSelected = -1;
for (index = 0; index < designLevels.numLevels; index++)
{
    if (designLevels.levels[index].groupID == groupID)
        addLevelToList (listP, &designLevels.levels[index]);
}

if ((**listP).numRows > 1)

```

```

mdlUtil_quickSort ((**listP).listPtrs, (**listP).numRows,
                  sizeof (char *) * LEVEL_COLUMNS, levelSort);

mdlDialog_listSetContents (itemP->rawItemP, (**listP).numRows,
                          LEVEL_COLUMNS, (**listP).listPtrs);
mdlDialog_itemDraw (db, NAMEDLEVELS_LevelsList);
}

/*-----+
| name      addGroupToList                               |
| author    slk                                           9/90   |
+-----*/
Private void addGroupToList
(
  GroupListData    **listP, /* <=> list of level groups */
  LevelGroup       *groupP /* => group to add */
)
{
  int allocSize, listIndex;
allocSize = (**listP).numRows + 1) * sizeof (char *);
  if (!(**listP).listPtrs)
    (**listP).listPtrs = malloc (allocSize);
  else
    (**listP).listPtrs = realloc ((**listP).listPtrs, allocSize);
  if (!(**listP).listPtrs)
return;
  listIndex = (**listP).numRows;
  strspc (groupP->name);
  (**listP).listPtrs[listIndex] = malloc (strlen (groupP->name) + 1);
  if (!(**listP).listPtrs[listIndex]) return;
  strcpy ((**listP).listPtrs[listIndex], groupP->name);

(**listP).numRows++;
  (**listP).parentID = groupP->parentID;
}

/*-----+
| name      addLevelToList                               |
| author    slk                                           9/90   |
+-----*/
Private void addLevelToList
(
  LevelListData    **listP, /* <=> list of names levels */
  NamedLevel       *levelP /* => level to add */
)
{
  char    parentName[64], numberString[32];
  int     allocSize, listIndex, parentID;

```

```

allocSize = (((**listP).numRows + 1) * LEVEL_COLUMNS) * sizeof (char *);
    if (!(**listP).listPtrs)
        (**listP).listPtrs = malloc (allocSize);
    else
        (**listP).listPtrs = realloc ((**listP).listPtrs, allocSize);
    if (!(**listP).listPtrs)
return;

listIndex = (**listP).numRows * LEVEL_COLUMNS;
    sprintf (numberString, "%d", levelP->level);
    (**listP).listPtrs[listIndex] = malloc (strlen (numberString) + 1);
    if (!(**listP).listPtrs[listIndex]) return;
    strcpy ((**listP).listPtrs[listIndex], numberString);

listIndex++;
    strspc (levelP->name);
    (**listP).listPtrs[listIndex] = malloc (strlen (levelP->name) + 1);
    if (!(**listP).listPtrs[listIndex]) return;
    strcpy ((**listP).listPtrs[listIndex], levelP->name);

listIndex++;
    strspc (levelP->comment);
    (**listP).listPtrs[listIndex] = malloc (strlen (levelP->comment) +
1);
    if (!(**listP).listPtrs[listIndex]) return;
    strcpy ((**listP).listPtrs[listIndex], levelP->comment);

(**listP).numRows++;
    mdlLevel_getParentGroup (&parentID, parentName, levelP->groupID,
NULL);
    (**listP).parentID = parentID;
    (**listP).groupID = levelP->groupID;
}

/*-----+
| name          freeGroupList          |
| author        slk                    9/90      |
+-----*/
Private void freeGroupList(DialogBox *db)
{
    int      index;
    DialogItem *itemP;
    GroupListData **listP;

    itemP = mdlDialog_itemGetByIndex (db, NAMEDLEVELS_GroupList);
    listP = &itemP->rawItemP->userDataP;

    if (!(*listP))
return;

    if ((*listP).listPtrs)

```

```

    {
        for (index = 0; index < (**listP).numRows; index++)
            if ((**listP).listPtrs[index]) free ((**listP).listPtrs[index]);

        free ((**listP).listPtrs);
    }

    free (*listP);
    *listP = NULL;
}

/*-----+
| name          freeLevelList                               |
| author        slk                                         9/90   |
+-----*/
Private void freeLevelList(DialogBox *db)
{
    int      index;
    DialogItem *itemP;
    LevelListData **listP;

    itemP = mdlDialog_itemGetByIndex (db, NAMEDLEVELS_LevelsList);
    listP = &itemP->rawItemP->userDataP;

    if (!(*listP))
        return;

    if ((**listP).listPtrs)
    {
        for (index = 0; index < ((**listP).numRows * LEVEL_COLUMNS); index++)
            if ((**listP).listPtrs[index]) free ((**listP).listPtrs[index]);

        free ((**listP).listPtrs);
    }

    free (*listP);
    *listP = NULL;
}

/*-----+
| name          doLevelListPick                             |
| author        slk                                         9/90   |
+-----*/
Private void doLevelListPick(DialogItemMessage *dimP)
{
    int      rowSelected;
    LevelListData*levelListP;
    DialogItem*diP  = dimP->dialogItemP;
    RawItemHdr*rihP = diP->rawItemP;

    if ((levelListP = rihP->userDataP))

```



```

    {
        mdlDialog_listGetSelection (&rowSelected, NULL, rihP);
        if (rowSelected != levelListP->rowSelected)
        {
            levelListP->rowSelected = rowSelected;
            if (levelListP->rowSelected >= 0)
                processLevelListPick (rowSelected, levelListP, dimP);
        }
    }
}

/*-----+
| name          doGroupListPick                               |
| author        slk                                           9/90   |
+-----*/
Private void doGroupListPick(DialogItemMessage *dimP,
boolean isDoubleClick)
{
    int          rowSelected;
    GroupListData*groupListP;
    DialogItem*diP = dimP->dialogItemP;
    RawItemHdr*rihP = diP->rawItemP;

    if ((groupListP = rihP->userDataP))
    {
        mdlDialog_listGetSelection (&rowSelected, NULL, rihP);
        if (rowSelected != groupListP->rowSelected)
        {
            groupListP->rowSelected = rowSelected;
            if (groupListP->rowSelected >= 0)
            {
                if (isDoubleClick)
                    doOpenGroup (dimP);
                else
                    processGroupListPick (rowSelected, groupListP, dimP);
            }
        }
    }
}

/*-----+
| name          processGroupListPick                           |
| author        slk                                           9/90   |
+-----*/
Private void processGroupListPick(int rowSelected,
GroupListData *groupListP, DialogItemMessage *dimP)
{
    char    parentName[64], *groupName;
    int     parentID;

```

```

        GroupListData *newGroupListP;

/* parent group selected ? */
    if (!rowSelected && groupListP->parentID)
    {
        if (!mdlLevel_getParentGroup (&parentID, parentName,
                                       groupListP->parentID, NULL))
        {
            loadGroupList (dimP->db, parentID);
            /* refresh our pointer to the group list */
            getGroupList (&newGroupListP, dimP->db);
            loadLevelsList (dimP->db, newGroupListP->parentID);
        }
    }
    else
        loadLevelsList (dimP->db, groupListP->parentID);
}

/*-----+
| name          processLevelListPick          |
| author        slk                          9/90 |
+-----*/
Private void processLevelListPick(int rowSelected,
LevelListData *levelListP, DialogItemMessage *dimP)
{
    /* clear the selection in the group list */
    loadGroupList (dimP->db, levelListP->groupID);
}

/*-----+
| name          doDisplayObject                |
| author        slk                          9/90|
+-----*/
Private void doDisplayObject(DialogItemMessage *dimP)
{
    /* first check for level selected, then group */
    if (doLevelDisplay (dimP))
        doGroupDisplay (dimP);
}

/*-----+
| name          doLevelDisplay                 |
| author        slk                          9/90|
+-----*/
Private int doLevelDisplay(DialogItemMessage *dimP)
{
    int          rowSelected, index;
    LevelListData*levelListP;
    DialogItem*diP;

```

```

    RawItemHdr*rihP;

    diP = mdlDialog_itemGetByIndex (dimP->db, NAMEDLEVELS_LevelsList);
    rihP = diP->rawItemP;

    if ((levelListP = rihP->userDataP))
    {
        mdlDialog_listGetSelection (&rowSelected, NULL, rihP);
        if (rowSelected >= 0)
        {
            index = (rowSelected * LEVEL_COLUMNS) + 1;
            memset (levelMask, 0, sizeof (levelMask));
            mdlLevel_getLevelMaskFromLevel (levelMask,
                levelListP->listPtrs[index], levelListP->groupID);
            doDisplayCommand();
            return (SUCCESS);
        }
    }
    return (ERROR);
}

/*-----+
| name          doGroupDisplay                               |
| author        slk                                         9/90|
+-----*/
Private int doGroupDisplay(DialogItemMessage *dimP)
{
    int      rowSelected, groupID;
    GroupListData*groupListP;
    DialogItem*diP;
    RawItemHdr*rihP;

    diP = mdlDialog_itemGetByIndex (dimP->db, NAMEDLEVELS_GroupList);
    rihP = diP->rawItemP;

    if ((groupListP = rihP->userDataP))
    {
        mdlDialog_listGetSelection (&rowSelected, NULL, rihP);
        if (!rowSelected && groupListP->parentID) return;
        if (rowSelected >= 0)
        {
            mdlLevel_getGroupIDFromNameAndParent (&groupID,
                groupListP->listPtrs[rowSelected], groupListP->parentID);
            memset (levelMask, 0, sizeof (levelMask));
            mdlLevel_getLevelMaskFromGroup (levelMask, groupID);
            doDisplayCommand ();
            return (SUCCESS);
        }
    }
    return (ERROR);
}

```

```

    }

/*-----+
| name      dodisplayCommand      |
| author    slk                    9/90      |
+-----*/
Private void doDisplayCommand()
{
    int commandIndex;

    commandIndex = turnLevelsOn ? LEVELS_DISPLAY_ON : LEVELS_DISPLAY_OFF;
    mdlState_startViewCommand(changeViewLevels, -commandIndex, -
    SELECT_VIEW);
}

/*-----+
| name      changeViewLevels      |
| author    slk                    9/90      |
+-----*/
Private void changeViewLevels()
{
    mdlView_setLevels (turnLevelsOn, levelMask, statedata.viewflags,
    TRUE);
}

/*-----+
| name      getLevelList|
| author    slk                    9/90      |
+-----*/
Private void getLevelList
(
    LevelListData **listP,/* <= levels list */
    DialogBox *db/* => dialog box */
{
    DialogItem *itemP;

    itemP = mdlDialog_itemGetByIndex (db, NAMEDLEVELS_LevelsList);
    *listP = itemP->rawItemP->userDataP;
}

/*-----+
| name      getGroupList|
| author    slk                    9/90      |
+-----*/
Private void getGroupList
(
    GroupListData **listP,/* <= group list */
    DialogBox *db/* => dialog box */
)
{

```

```

    DialogItem    *itemP;

    itemP = mdlDialog_itemGetByIndex (db, NAMEDLEVELS_GroupList);
    *listP = itemP->rawItemP->userDataP;
}

/*-----+
| name          handleLevelEdit|
| author        ezs              11/90      |
+-----*/
Private void handleLevelEdit(DialogBox *dbP)
{
    LevelListData **listP;
    DialogItem    *diP;
    char    levelName[20], levelComment[40], levelNumberStr[4];
    int     levelNumber;

listP = (LevelListData **) editListP;

/* Get level name */
    diP = mdlDialog_itemGetByTypeAndId (dbP, RTYPE_Text,
                                         TEXTID_LevelEditName, 0);
    mdlDialog_itemGetValue (NULL, NULL, levelName, dbP,
                           diP->itemIndex, sizeof (levelName));

/* Get level comment */
    diP = mdlDialog_itemGetByTypeAndId (dbP, RTYPE_Text,
                                         TEXTID_LevelEditComment, 0);
    mdlDialog_itemGetValue (NULL, NULL, levelComment, dbP,
                           diP->itemIndex, sizeof (levelComment));

/* Get level number */
    diP = mdlDialog_itemGetByTypeAndId (dbP, RTYPE_Text,
                                         TEXTID_LevelEditNumber, 0);
    mdlDialog_itemGetValue (NULL, NULL, levelNumberStr, dbP,
                           diP->itemIndex, sizeof (levelNumberStr));
    sscanf (levelNumberStr, "%d", &levelNumber);

if ((**listP).rowSelected == -1)
{
    mdlLevel_addLevel (levelName, levelComment, (**listP).groupID,
                      levelNumber);
}
else
{
    int listIndex = (**listP).rowSelected * LEVEL_COLUMNS;
    mdlLevel_editLevel ((**listP).listPtrs[listIndex + 1],
                       (**listP).groupID, levelName, levelComment,
                       levelNumber);
}

```

```

    }
    }

/*-----+
| name      levels_levelEditDialogHook          |
| author    ezs                                11/90 |
+-----*/
Public void levels_levelEditDialogHook(DialogMessage *dmP)
{
    dmP->msgUnderstood = TRUE;
switch (dmP->messageType)
{
    case DIALOG_MESSAGE_CREATE:
        isDialogClosing = FALSE;
        dmP->u.create.interests.keystrokes = TRUE;
        dmP->u.create.interests.mouses = TRUE;
        break;

    case DIALOG_MESSAGE_ACTIONBUTTON:
        if (dmP->u.actionButton.actionType == ACTIONBUTTON_OK)
            handleLevelEdit (dmP->db);
        isDialogClosing = TRUE;
        break;

    default:
        dmP->msgUnderstood = FALSE;
        break;
}
}

/*-----+
| name      setupEditText|
| author    ezs                                11/90 |
+-----*/
Private void setupEditText(DialogItemMessage *dimP)
{
    LevelListData **listP;
    DialogItem *diP;
    int listIndex;

listP = (LevelListData **) editListP;
if ((*listP).rowSelected != -1)
{
    listIndex = ((*listP).rowSelected * LEVEL_COLUMNS;
    if (dimP->itemIndex == LEVELEDIT_LevelNumberField)
    {
        /* Set the original level number */
        diP = mdlDialog_itemGetByTypeAndId (dimP->db, RTYPE_Text,

```

```

        TEXTID_LevelEditNumber, 0);
mdlDialog_itemSetValue (NULL, 0, NULL,
        (**listP).listPtrs[listIndex],
        dimP->db, diP->itemIndex);
    }

if (dimP->itemIndex == LEVELEDIT_LevelNameField)
{
    /* Set the original level name */
    diP = mdlDialog_itemGetByTypeAndId (dimP->db, RTYPE_Text,
        TEXTID_LevelEditName, 0);
    mdlDialog_itemSetValue (NULL, 0, NULL,
        (**listP).listPtrs[listIndex + 1],
        dimP->db, diP->itemIndex);
}

if (dimP->itemIndex == LEVELEDIT_LevelCommentField)
{
    /* Set the original level comment */
    diP = mdlDialog_itemGetByTypeAndId (dimP->db, RTYPE_Text,
        TEXTID_LevelEditComment, 0);
    mdlDialog_itemSetValue (NULL, 0, NULL,
        (**listP).listPtrs[listIndex + 2],
        dimP->db, diP->itemIndex);
}
}
}

/*-----+
| name      verifyLevelEditName|
| author    ezs                11/90      |
+-----*/
Private void verifyLevelEditName(DialogItemMessage *dimP)
{
    LevelListData **listP;
    DialogItem *diP;
    char fieldValue[20];

    listP = (LevelListData **) editListP;
    diP = mdlDialog_itemGetByTypeAndId (dimP->db, RTYPE_Text,
        TEXTID_LevelEditName, 0);

    mdlDialog_itemGetValue (NULL, NULL, fieldValue, dimP->db,
        diP->itemIndex, sizeof (fieldValue));

    /* If editing and names are the same, this is ok */
    if ((**listP).rowSelected != -1)
    {

```

```

    int listIndex = (**listP).rowSelected * LEVEL_COLUMNS;

    if (!strcmpi ((**listP).listPtrs[listIndex + 1], fieldValue))
        return;
    }

    if (!mdlLevel_isLevelUnique ((**listP).parentID, fieldValue))
    {
        char errorMessage[256];

        mdlRsrc_fetchString (errorMessage, DUPLICATE_LEVEL);
        strcat (errorMessage, fieldValue);
        mdlOutput_error (errorMessage);
        dimP->u.focusOut.outOfRange = TRUE;
        dimP->u.focusOut.hookHandled = TRUE;
    }
    }

/*-----+
| name          verifyLevelEditNumber|
| author        ezs                  11/90      |
+-----*/
Private void verifyLevelEditNumber(DialogItemMessage *dimP)
{
    DialogItem    *diP;
    char          fieldValue[20];
    int           levelNumber;

    diP = mdlDialog_itemGetByTypeAndId (dimP->db, RTYPE_Text,
                                         TEXTID_LevelEditNumber, 0);

    mdlDialog_itemGetValue (NULL, NULL, fieldValue, dimP->db,
                           diP->itemIndex, sizeof (fieldValue));

    sscanf (fieldValue, "%d", &levelNumber);
    if (levelNumber < 1 || levelNumber > 63)
    {
        dimP->u.focusOut.outOfRange = TRUE;
        dimP->u.focusOut.hookHandled = TRUE;
    }
    }

/*-----+
| name          levels_levelEditTextHook|
| author        ezs                  11/90      |
+-----*/
Public void levels_levelEditTextHook(DialogItemMessage *dimP)
{
    dimP->msgUnderstood = TRUE;

```



```

switch (dimP->messageType)
{
    case DITEM_MESSAGE_CREATE:
        setupEditText (dimP);
        break;

    case DITEM_MESSAGE_FOCUSOUT:
        if (!isDialogClosing)
        {
            if (dimP->itemIndex == LEVELEDIT_LevelNameField)
                verifyLevelEditName (dimP);
            else if (dimP->itemIndex == LEVELEDIT_LevelNumberField)
                verifyLevelEditNumber (dimP);
        }
        break;

    default:
        dimP->msgUnderstood = FALSE;
        break;
}

}

/*-----+
| name      handleGroupEdit|
| author    ezs              11/90      |
+-----*/
Private void handleGroupEdit(DialogBox *dbP)
{
    GroupListData **listP;
    DialogItem *diP;
    char    fieldValue[20];
    int     groupID;

listP = (GroupListData **) editListP;
diP = mdlDialog_itemGetByTypeAndId (dbP, RTYPE_Text,
                                     TEXTID_GroupEditName, 0);

    mdlDialog_itemGetValue (NULL, NULL, fieldValue, dbP,
                           diP->itemIndex, sizeof (fieldValue));

    if ((*listP).rowSelected == -1)
    {
        mdlLevel_getNextGroupID (&groupID);
        mdlLevel_addGroup (fieldValue, groupID, (*listP).parentID);
    }
    else
    {
        mdlLevel_getGroupIDFromNameAndParent (&groupID,
                                               (*listP).listPtrs[(*listP).rowSelected],

```

```

        (**listP).parentID);
    mdlLevel_editGroup (groupID, fieldValue);
}
}

/*-----+
| name      levels_groupEditDialogHook      |
| author    ezs                             11/90      |
+-----*/
Public void levels_groupEditDialogHook(DialogMessage *dmP)
{
    dmP->msgUnderstood = TRUE;
switch (dmP->messageType)
{
    case DIALOG_MESSAGE_CREATE:
        isDialogClosing = FALSE;
        dmP->u.create.interests.keystrokes = TRUE;
        dmP->u.create.interests.mouses = TRUE;
        break;

    case DIALOG_MESSAGE_ACTIONBUTTON:
        if (dmP->u.actionButton.actionType == ACTIONBUTTON_OK)
            handleGroupEdit (dmP->db);
        isDialogClosing = TRUE;
        break;

    default:
        dmP->msgUnderstood = FALSE;
        break;
}
}

/*-----+
| name      levels_groupEditTextHook      |
| author    slk                             8/90      |
+-----*/
Public void levels_groupEditTextHook(DialogItemMessage *dimP)
{
    dimP->msgUnderstood = TRUE;
switch (dimP->messageType)
{
    case DITEM_MESSAGE_CREATE:
        {
            GroupListData **listP;
            DialogItem *diP;

            listP = (GroupListData **) editListP;
            if ((**listP).rowSelected != -1)

```

```

    {
        diP = mdlDialog_itemGetByTypeAndId (dimP->db, RTYPE_Text,
                                            TEXTID_GroupEditName, 0);

        /* Set the original name */
        mdlDialog_itemSetValue (NULL, 0, NULL,
                                (**listP).listPtrs[(**listP).rowSelected],
                                dimP->db, diP->itemIndex);
    }
}
break;

case DITEM_MESSAGE_FOCUSOUT:
{
    GroupListData **listP;
    DialogItem *diP;
    char fieldValue[20];

    if (isDialogClosing)
        break;

    listP = (GroupListData **) editListP;

    diP = mdlDialog_itemGetByTypeAndId (dimP->db, RTYPE_Text,
                                        TEXTID_GroupEditName, 0);

    mdlDialog_itemGetValue (NULL, NULL, fieldValue,
                            dimP->db, diP->itemIndex, sizeof (fieldValue));

    /* If editing and names are the same, this is ok */
    if ((**listP).rowSelected != -1)
    {
        if (!strcmpi ((**listP).listPtrs[(**listP).rowSelected],
                     fieldValue))
            break;
    }

    if (!mdlLevel_isGroupUnique ((**listP).parentID, fieldValue))
    {
        char errorMessage[256];

        mdlRsrc_fetchString (errorMessage, DUPLICATE_GROUP);
        strcat (errorMessage, fieldValue);
        mdlOutput_error (errorMessage);
        dimP->u.focusOut.outOfRange = TRUE;
        dimP->u.focusOut.hookHandled = TRUE;
    }
}
break;

```

```

default:
    dimP->msgUnderstood = FALSE;
    break;
}

```

dynamic.mc

```

/*-----+
| Copyright (1995) Bentley Systems, Inc., All rights reserved.|
| "MicroStation" is a registered trademark and "MDL" and "MicroCSL"|
| are trademarks of Bentley Systems, Inc.      |
| Limited permission is hereby granted to reproduce and modify this|
| copyrighted material provided that the resulting code is used only |
| in conjunction with Bentley Systems products under the terms of the|
| license agreement provided therein, and that this notice is retained|
|                                     |
| in its entirety in any such reproduction or modification.|
+-----*/
/*-----+
|   $Logfile:   J:/mdl/examples/doc/dynamic.mcv  $
|   $Workfile:   dynamic.mc  $
|   $Revision:   5.3  $
|   $Date:      20 Jun 1995 08:49:56  $
+-----*/
/*-----+
| dynamic.mc                                     |
| Example program that contains a few MDL commands that illustrate|
| various techniques to use the concept of "simple dynamics".|
| These are not intended to be true application commands (since they|
| have no resource files associated with them and they really do not|
| do sufficient prompting or error checking and reporting). Rather,|
| they are intended to show various aspects of MicroStation's|
| dynamics and how MDL primitive functions interact with same.|
+-----*/
/*-----+
| Include Files                                     |
+-----*/
#include    <mselems.h>
#include    <global.h>
#include    <mdl.h>
#include    < tcb.h>
#include    <userfnc.h>

/*-----+
| Private function declarations|
+-----*/

```

```
void moveAnElement(), placeACell();

/*-----+
|  MOVE EXISTING ELEMENT example|
+-----*/
/*-----+
|  name  getMoveDistance- get distance from anchor point to      |
|          current point.      |
|  authorBSI          10/90      |
+-----*/
Private voidgetMoveDistance
(
Dpoint3d    *distance,/* <= distance from anchor */
Dpoint3d    *pt/* => current point */
)
{
    Dpoint3danchor;
/*-----+
The element location logic stores the point where the user
located the element in the global variable
"statedata.pointstack[0]". We use that point for our anchor point.
Since it is stored in integer world coordinates we need to convert
it to Dpoint3d first.
-----*/
    mdlCnv_IPointToPoint(&anchor, statedata.pointstack);
/* subtract anchor point from current point to get the distance */
    mdlVec_subtractPoint (distance, pt, &anchor);
}

/*-----+
|  name  offsetElement - dynamic function for "moveAnElement".  |
|  authorBSI          10/90      |
+-----*/
Private voidoffsetElement
(
Dpoint3d    *pt/* => current location of cursor */
)
{
    Dpoint3ddistance;

    getMoveDistance (&distance, pt);
    mdlElement_offset (dgnBuf, dgnBuf, &distance);
}

/*-----+
|  name  elementModify_move - called indirectly for each element|
|          to be moved by mdlModify_elementMulti.|
|  authorBSI          10/90      |
+-----*/
```

```

+-----*/
Private intelementModify_move
(
MSElementUnion*el,/* <> element to be modified */
Dpoint3d*distance/* => from params in mdlModify_element... */
)
{
    /* offset the element by specified distance */
    if (mdlElement_offset (el, el, distance))
        return MODIFY_STATUS_ERROR;

    /* indicate that we did something to the element */
    return MODIFY_STATUS_REPLACE;
}

/*-----+
| name moveElement_accept - data point function for|
|      "moveAnElement" example.|
| authorBSI      10/90      |
+-----*/
Private voidmoveElement_accept
(
Dpoint3d      *pt/* => final point for move element */
)
{
    Dpoint3ddistance;
    int fileNum, modifyFlags = MODIFY_ORIG;
    ULongfilePos;
    /* Get the distance by which to move each element. */
    getMoveDistance (&distance, pt);
    /* Get the file position and file number of the element to move. */
    filePos = mdlElement_getFilePos (FILEPOS_CURRENT, &fileNum);
    /*-----
        If we are using a selection set we want to draw the element(s) in
        their new location in their normal color. If we are moving a single
        element, we want to draw them in the hilite color.
    -----*/
    if (!mdlSelect_isActive())
        modifyFlags |= MODIFY_DRAWINHILITE;
    /* Now move each element the user accepted. */
    mdlModify_elementMulti (fileNum, filePos, MODIFY_REQUEST_HEADERS,
        modifyFlags, elementModify_move, &distance, 1);
    /* Save new anchor point (the current acceptance point) */
    mdlCnv_DPointToIPoint (statedata.pointstack, pt);
    /* Reload the dynamic buffer with the new element. */
    mdlDynamic_loadElement (NULL, fileNum, filePos);

    /* Change the reset function to restart the whole command rather than to

```

```

        continue to look for additional element about the last data
point. */
    mdlState_setFunction (STATE_RESET, moveAnElement);

/* See if we were single-shotted, or if we acted upon a selection set
(in which case we don't want to keep performing the same action). */
    mdlState_checkSingleShot();
}

/*-----+
| name moveAnElement - emulate (almost) the MicroStation|
|           "MOVE ELEMENT" command.|
| authorBSI           10/90           |
+-----*/
cmdName void moveAnElement(void)
{
    /*-----
        Start a "modification" command. This command will use groups
        (either selection sets or graphic groups), and requires 2
        data points to complete.
    -----*/
    mdlState_startModifyCommand (moveAnElement, moveElement_accept,
        offsetElement, NULL, NULL, 0, 0, TRUE, 2);
    /* initialize the element location logic so that it starts at the
        beginning of the file finding all displayable element types */
    mdlLocate_init();
}

/*-----+
| CELL PLACEMENT example |
+-----*/
Dpoint3dcell0Origin; /* origin of cell to be placed,
                        set by first data point. */
int    cellView; /* view of cell to be placed,
                  set by first data point. */

/*-----+
| name getCellTransform - return transformation matrix based |
|           on the current view and active scales.|
| authorBSI           10/90           |
+-----*/
Private void getCellTransform
(
    Transform    *cellTrans, /* <= transformation to be set */
    int    view /* => view in which to draw cell */
)
{
    RotMatrix rMatrix;
    /* get the inverse of the rotation matrix for the given view */
    mdlRMMatrix_fromView (&rMatrix, view, FALSE);

```

```

        mdlRMatrix_invert (&rMatrix, &rMatrix);

/* convert to a transformation matrix and scale the columns */
        mdlTMatrix_fromRMatrix (cellTrans, &rMatrix);
        mdlTMatrix_scale (cellTrans, cellTrans, tcb->xactscl, tcb-
>yactscl,
            tcb->zactscl);
    }

/*-----+
| name getCursorAngle - find the angle to rotate the cell|
|           for the "placeACell" command.|
| authorBSI          8/91          |
+-----*/
Private doublegetCursorAngle
(
Dpoint3d    *pt/* => current location of cursor */
)
{
    double  zRotation;
if (pt->x - cell0origin.x == 0)
    zRotation = fc_piover2;
    else
        zRotation = atan ((pt->y - cell0origin.y) / (pt->x - cell0origin.x));

        if (pt->x - cell0origin.x < 0)
            zRotation += fc_pi;

        return (zRotation);
    }

/*-----+
| name drawRotatingCell - dynamic function for rotating the|
|           cell for the "placeACell" command.|
| authorBSI          8/91          |
+-----*/
Private int drawRotatingCell
(
Dpoint3d    *pt,/* => current location of cursor */
int         view /* => current view */
)
{
    TransformtMatrix;
    getCellTransform (&tMatrix, cellView);
    /* rotate the transformation matrix to current cursor position. */
    mdlTMatrix_rotateByAngles (&tMatrix, &tMatrix, fc_zero, fc_zero,
        getCursorAngle (pt));
    mdlTMatrix_setTranslation (&tMatrix, &cell0origin);

```



```

        mdlElement_transform (dgnBuf, dgnBuf, &tMatrix);
    }

/*-----+
| name drawCell - dynamic function for the "placeACell"|
|           command.                                |
| authorBSI          10/90          |
+-----*/
Private int drawCell
(
Dpoint3d    *pt, /* => current location of cursor */
int         view /* => current view */
)
{
    TransformtMatrix;
    getCellTransform (&tMatrix, view);
    /* rotate the transformation matrix by the active angle */
    mdlTMatrix_rotateByAngles (&tMatrix, &tMatrix, fc_zero, fc_zero,
        tcb->actangle * fc_piover180);
    mdlTMatrix_setTranslation (&tMatrix, pt);
    mdlElement_transform (dgnBuf, dgnBuf, &tMatrix);
}

/*-----+
| name acceptPlaceCell - second data point function for|
|           "placeACell".                                |
| authorBSI          10/90          |
+-----*/
Private void acceptPlaceCell
(
Dpoint3d    *pt, /* => final data point */
int         view /* => view for same */
)
{
    char activeCell[10];
    TransformtMatrix;
    RotMatrixrMatrix;

    getCellTransform (&tMatrix, cellView);
    /* rotate the transformation matrix to current cursor position. */
    mdlTMatrix_rotateByAngles (&tMatrix, &tMatrix, fc_zero, fc_zero,
        getCursorAngle (pt));
    mdlRMatrix_fromTMatrix (&rMatrix, &tMatrix);

    mdlParams_getActive (activeCell, ACTIVEPARAM_CELLNAME);
    mdlCell_placeCell(OL, &cellOrigin, NULL, &rMatrix, NULL, 0, FALSE, 0,
        tcb->ext_locks.sharedCells, activeCell);

    mdlState_restartCurrentCommand();
}

```

```

/*-----+
| name placeACellPt1 - first data point function for|
|           "placeACell".                          |
| authorBSI      8/91      |
+-----*/
Private void placeACellPt1
(
Dpoint3d    *pt,    /* => first data point */
int         view    /* => view for same */
)
{
    /* Set the cell origin and view */
    cellOrigin = *pt;
    cellView = view;

    /* Set function to use second data point */
    mdlState_setFunction (STATE_DATAPOINT, acceptPlaceCell);

    /* establish a new "simple dynamics" function to rotate the cell
during
    cursormovement */
    mdlState_dynamicUpdate (drawRotatingCell, TRUE);
}

/*-----+
| name restartDefault          |
| authorBSI      10/90      |
+-----*/
Private void restartDefault(void)
{
    mdlState_startDefaultCommand();
}

/*-----+
| name      placeACell - simple example of how an MDL application can|
|      emulate the MicroStation "PLACE CELL ABSOLUTE"|
|      command.          |
| author    BSI      10/90      |
+-----*/
cmdName void placeACell(void)
{
    MSElementDescr *cellDscrP;
    char activeCell[10];
    mdlState_startPrimitive (placeACellPt1, placeACell, 0, 0);
    mdlState_setFunction (STATE_RESET, restartDefault);

    /* first get the name of the active cell */
    mdlParams_getActive (activeCell, ACTIVEPARAM_CELLNAME);

    /* get an element descriptor to hold the active cell (for use with

```

```

        the dynamic function) */
mdlCell_getElmDscr (&cellDscrP, NULL, 0L, NULL, NULL, NULL, NULL, 0,
                    tcb->ext_locks.sharedCells, activeCell);

/*-----
Load the element descriptor into MicroStation's dynamic buffer.
This is the only place in MDL where an application creates an element
descriptor and does not have to free it. Once an element descriptor
is passed to mdlDynamic_setElmDscr, MicroStation owns the descriptor
and will free it when finished with it.
-----*/
mdlDynamic_setElmDscr (cellDscrP);

/* establish a "simple dynamics" function to draw the cell during cursor
movement */
mdlState_dynamicUpdate (drawCell, TRUE);
}

```

math.mc

trumpet

database.mc

```

/*-----+
| Copyright (1995) Bentley Systems, Inc., All rights reserved.|
| "MicroStation" is a registered trademark and "MDL" and "MicroCSL"|
| are trademarks of Bentley Systems, Inc.      |
| Limited permission is hereby granted to reproduce and modify this|
| copyrighted material provided that the resulting code is used only |
| in conjunction with Bentley Systems products under the terms of the|
| license agreement provided therein, and that this notice is retained|
| in its entirety in any such reproduction or modification.|
+-----*/
/*-----+
| $Logfile:   J:/mdl/examples/doc/database.mcv  $
| $Workfile:  database.mc  $
| $Revision:  5.5  $
| $Date:      20 Jun 1995 08:49:42  $
+-----*/
/*-----+
| database.mc - examples for the mdlDB_ functions.|
| This file is intended as an adjunct to the MDL manual to|

```

```

| illustrate MDL built-in function calling conventions and parameter|
| values. While it can be compiled, it does NOT, on its own,|
| constitute a workable MDL example.|
| The MDL example "gis", located in the \ustn40\mdl\examples\misc|
| directory, is a complete database interface example.|
+-----*/
#include <mdl.h> /* system include files */
#include <global.h>
#include <mselems.h>
#include <dbdefs.h>
#include <dberrs.h>
#include <dbserver.h>

/*-----+
| name          submitQuery                                     |
| author          BSI                                           1/91                                     |
+-----*/
Public void submitQuery()
{
    char    sqlStatement[128], surface[16];
    int     status;
    double  grade;
    grade = .01;
    strcpy (surface, "asphalt");
    sprintf (sqlStatement,
        "select * from roadways where grade > %lf and surface = '%s'",
        grade, surface);
    status = mdlDB_processSQL (sqlStatement);
    if (status != SUCCESS)
        mdlOutput_error ("Error processing SELECT statement");
}

/*-----+
| name          attachElement                                   |
| author          BSI                                           1/91                                     |
+-----*/
Public int attachElement()
{
    MSElementUnion u;
    ULong    filePosition;
    /* an element has been located and is present in DGNBUF */
    mdlDB_attachActiveEntityElement (&u, dgnBuf);
    filePosition = mdlElement_getFilePos (FILEPOS_CURRENT, 0);
    mdlElement_rewrite (&u, dgnBuf, filePosition);
    return (SUCCESS);
}

/*-----+
| name          attachCellDescriptor|

```

```
| Illustrates the placement of a cell with a database linkage. The|
| active entity is defined through the use of an SQL SELECT statement.|
| author          BSI                      6/91          |
+-----*/
cmdName void attachCellDescriptor
(
char*cellName    /* => cell name to place */
)
{
char selectStatement[256], updateStatement[256], description[64];
short  attributeData[MAX_ATTRIBSIZE];
int    length;
DatabaseLink  databaseLink;
MSElementDescr *cellDescrP;
/* set the linkage mode; FILE DESIGN will save */
mdlDB_activeLinkageMode ("new");

/* create an SQL SELECT statement to locate the target row;
   our table has a primary key 'cellname' which matches the name
   of the graphics cell name */
sprintf (selectStatement,
        "select * from furniture where cellname = '%s'", cellName);
/* establish the active entity */
mdlDB_defineAEBBySQLSelect (selectStatement);
/* retrieve the element descriptor from the cell library */
mdlCell_getElmDscr (&cellDescrP, NULL, 0L, NULL, NULL, NULL, NULL,
        0, FALSE, cellName);
/* attach our element descriptor to the active entity */
mdlDB_attachActiveEntityDscr (&cellDescrP);

/* add the cell to the design file */
mdlElmdscr_add (cellDescrP);
/* display the new cell */
mdlElmdscr_display (cellDescrP, MASTERFILE, NORMALDRAW);
/* now extract the linkage from the cell */
mdlElement_extractAttributes(&length, attributeData, &cellDescrP-
>el);
/* translate linkage to internal format */
mdlDB_decodeLink (&databaseLink, attributeData);

/* now build an SQL UPDATE statement based on the MSLINK from the
link */
sprintf (updateStatement,
        "update furniture set description = 'desk' where mslink = %lu",
        databaseLink.mslink);
/* submit the UPDATE statement */
mdlDB_processSQL (updateStatement);
/* now retrieve a column from the attached row */
```

```

    sprintf (selectStatement,
        "select description from furniture where mslink = %lu",
        databaseLink.mslink);
    /* now query the database */
    mdlDB_sqlQuery (description, selectStatement);
    mdlOutput_printf (MSG_MESSAGE, "Description = %s",
description);

/* free the element descriptor now that we are done */
    mdlElmdscr_freeAll (&cellDescrP);
}

/*-----+
| name           placeCellWithLinkage|
|   Illustrates the placement of a cell with a database|
|   linkage. The linkage is constructed using|
|   mdlDB_buildLink () and then passed as the attribute|
|   argument to mdlCell_placeCell.|
| author          BSI                      6/91          |
+-----*/
void placeCellWithLinkage
(
    DPoint3d      *pointP,      /* => cell position */
    char          *cellName     /* => cell name */
)
{
    charcolumnValue[64];
    UShortattributeBuffer[(sizeof (OracleLink) / sizeof (short)) + 1];
    int status, dasType, length;
    ULongmslinkKey;
    LinkPropsproperties;
    /* determine MSLINK key of target row */
    status = mdlDB_sqlQuery (columnValue,
        "select mslink from parcel where owner = 'John Smith'");
    mslinkKey = atol (columnValue);

/* first word of attribute buffer is length (words) of attribute data */
    attributeBuffer[0] = sizeof (OracleLink) / sizeof (short);
    /* build the database linkage */
    memset (&properties, 0, sizeof (LinkProps));
    dasType = 0;
    mdlDB_buildLink (attributeBuffer + 1, &length, ORACLE_LINKAGE,
        &properties, "parcel", mslinkKey, dasType);
    status = mdlCell_placeCell (OL, pointP, NULL, NULL, attributeBuffer,
        0, FALSE, 0, 2, cellName);
}

/*-----+
| name           reportElement          |
|

```

```

| author          BSI                               1/91          |
+-----+-----+-----+-----+-----+-----+-----+-----+
Public int reportElement()
{
    /* clear the report tables */
    mdlDB_openReport ();
    /* add only the element in DGNBUF to the report tables
    mdlDB_elementReport (dgnBuf);
    /* close report tables and terminate reporting operation */
    mdlDB_closeReport ();
    return (SUCCESS);
}

/*-----+
| name          showLinkages                        |
| author        BSI                               1/91          |
+-----+-----+-----+-----+-----+-----+-----+
Public int showLinkages(MSElementUnion *p )
{
    /* review the attributes of the element */
    mdlDB_reviewAttributes (p);
    return (SUCCESS);
}
-----+

| name          displayOrderForm                    |
| author        BSI                               1/91          |
+-----+-----+-----+-----+-----+-----+-----+
Public int displayOrderForm ()
{
    short    status;
    status = mdlDB_executeScreenForm ("inventory", NULL);

    return (status);
}

/*-----+
| name          analyzeLink                         |
| author        BSI                               1/91          |
+-----+-----+-----+-----+-----+-----+-----+
Public int analyzeLink ()
{
    char    message[128];
    short    *start;
    int    status;
    Header    *p;
    DatabaseLink    link;
    /* element located is in DGNBUF */
    p = (Header *) dgnBuf;

```

```

/* first check for attribute data */
    if (mdlDB_elementFilter(p) == FALSE)
    {
        mdlOutput_error ("Element has no attribute linkages");
        return (ERROR);
    }
    /* first word of the attribute data */
    start = (short *) (&p->dhdr.props.s + p->dhdr.attindx);
    /* extract the information from the linkage */
    status = mdlDB_decodeLink (&link, start);
    if (status == SUCCESS)
    {
        sprintf (message, "Tablename: %s, Entity: %d, Mslink: %ld\n",
            link.tablename, link.entity, link.mslink);
        mdlOutput_message (message);
    }

return (SUCCESS);
}

/*-----+
| name      stripElement      |
| author    BSI                1/91      |
+-----*/
Public int stripElement()
{
    MSElementUnion  u;
    ULong    filePosition;

    /* element has been located and is in DGNBUF */
    mdlDB_detachAttributesElement (&u, dgnBuf);

    /* retrieve current file position and rewrite to the design file */
    filePosition = mdlElement_getFilePos (FILEPOS_CURRENT, 0);
    mdlElement_rewrite (&u, dgnBuf, filePosition);
return (SUCCESS);
}

*-----+
+
| name      createLink      |
| author    BSI                1/91      |
+-----*/
Public int createLink()
{
    char    message[128], tableName[64];
    short    linkage[8];
    int    status, dasType, linkLength;
    ULong    mslink, filePosition;
    LinkProps    props;

```



```

    Header *p;
    MSElementUnion u;

/* element has been located and is present in DGNBUF */
    p = (Header *) dgnBuf;

props.information = FALSE;
    props.remote      =FALSE;
    props.modified    = FALSE;
    props.user        = TRUE;
    strcpy (tableName, "roadways");
    mslink = 10;
    dasType = 1;
    status = mdlDB_buildLink (linkage, &linkLength, ORACLE_LINKAGE,
                             &props, tableName, mslink, dasType);
    if (status == SUCCESS)
    {
        status = mdlElement_appendAttributes (p, linkLength, linkage);
        filePosition = mdlElement_getFilePos (FILEPOS_CURRENT, 0);
        status = mdlElement_rewrite (p, NULL, filePosition);
        return (SUCCESS);
    }
    return (status);
}

/*-----+
| name          processLinksForDelete          |
| author        BSI                            1/91      |
+-----*/
Public int processLinksForDelete(MSElementUnion *p)
{
    intstatus = SUCCESS;
    /* check for attribute linkages */
    if (mdlDB_elementFilter(p) == TRUE)
        status = mdlDB_deleteElement (p);
    return (status);
}

/*-----+
| name          processLinksForCopy            |
| author        BSI                            1/91      |
+-----*/
Public int processLinksForCopy(MSElementUnion *p)
{
    int status = SUCCESS;
    /* check for attribute linkages */
    if (mdlDB_elementFilter(p) == TRUE)
        status = mdlDB_copyElement (p);
    return (status);
}

```

```

    }

/*-----+
| name      testElement                               |
| author    BSI                                     1/91 |
+-----*/
Public int testElement (MSElementUnion *p)
{
    int status;
    /* check for attribute linkages */
    if (p->hdr.dhdr.props.b.a)
    {
        status = mdlDB_elementFilter (p);
        return (status);
    }
    return (FALSE);
}

/*-----+
| name      testFenceFilter                           |
| author    BSI                                     1/91 |
+-----*/
Public int testFenceFilter(MSElementUnion *p)
{
    char    keyin[256];
    int     status;
    /* check for attribute linkages */
    if (mdlDB_elementFilter(p) == TRUE)
    {
        /* establish a database search criteria */
        strcpy (keyin, "define search select * from offices");
        mdlInput_sendKeyin (keyin, TRUE, 0, NULL);
        /* test our element for inclusion in the filter */
        status = mdlDB_fenceFilter (p);
        return (status);
    }
    return (FALSE);
}

/*-----+
| name      writeDataPoint                             |
| author    BSI                                     1/91 |
+-----*/
Public int writeDataPoint ()
{
    char     message[128], xposition[64], yposition[64];
    short    *start;
    int      status;
    Header   *p;

```

```

        DatabaseLink    link;
        /* element has been located and is in DGNBUF */
        p = (Header *) dgnBuf;

/* first check for attribute data */
    if (mdlDB_elementFilter(p) == FALSE)
    {
        mdlOutput_error ("Element has no attribute linkages");
        return (ERROR);
    }

/* first word of the attribute data */
    start = (short *) (&p->dhdr.props.s + p->dhdr.attindx);
    /* extract the information from the linkage */
    status = mdlDB_decodeLink (&link, start);
    if (status == SUCCESS)
    {
        /* save the position of the last datapoint */
        sprintf (xposition, "%ld", statedata.inPoint.uors.x);
        sprintf (yposition, "%ld", statedata.inPoint.uors.y);
        status = mdlDB_writeColumn (link.tablename, link.mslink,
                                    "XORIGIN", xposition);
        status = mdlDB_writeColumn (link.tablename, link.mslink,
                                    "YORIGIN", yposition);

        return (status);
    }
mdlOutput_message ("Error decoding linkage");
return (ERROR);
}

/*-----+
| name          readCustomerName          |
| author        BSI                      1/91|
+-----*/
Public int readCustomerName()
{
    char    message[128], customerName[128];
    short    *start;
    int      status;
    Header    *p;
    DatabaseLinklink;
    /* element has been located and is in DGNBUF */
    p = (Header *) dgnBuf;

/* first check for attribute data */
    if (mdlDB_elementFilter(p) == FALSE)
    {
        mdlOutput_error ("Element has no attribute linkages");
        return (ERROR);
    }

```

```

    }

    /* first word of the attribute data */
    start = (short *) (&p->dhdr.props.s + p->dhdr.attindx);
    /* extract the information from the linkage */
    status = mdlDB_decodeLink (&link, start);
    if (status == SUCCESS)
    {
        status = mdlDB_readColumn (customerName, link.tablename,
                                   link.mslink, "NAME");
        sprintf (message, "Customer is %s\n", customerName);
        mdlOutput_message (message);
        return (status);
    }

    mdlOutput_message ("Error decoding linkage");
    return (ERROR);
}

/*-----+
| name      deleteRow      |
| author    BSI             1/91      |
+-----*/
Public int deleteRow()
{
    char    message[128];
    short   *start;
    int     status;
    Header   *p;
    DatabaseLink link;
    /* element has been located and is in DGNBUF */
    p = (Header *) dgnBuf;

    /* first check for attribute data */
    if (mdlDB_elementFilter(p) == FALSE)
    {
        mdlOutput_error ("Element has no attribute linkages");
        return (ERROR);
    }

    /* first word of the attribute data */
    start = (short *) (&p->dhdr.props.s + p->dhdr.attindx);
    /* extract the information from the linkage */
    status = mdlDB_decodeLink (&link, start);
    if (status == SUCCESS)
    {
        status = mdlDB_deleteRow (link.mslink, link.tablename);
        if (status == SUCCESS)
            mdlOutput_message ("Row deleted");
        return (status);
    }

```

```

    }

    mdlOutput_message ("Error decoding linkage");
    return (ERROR);
}

/*-----+
| name          biggestKey                                |
| author        BSI                                     1/91 |
+-----*/
Public int biggestKey()
{
    char    message[128];
    short   *start;
    int     status;
    unsigned long    maxMslink;
    Header   *p;
    DatabaseLink    link;
    /* element located is in DGNBUF */
    p = (Header *) dgnBuf;

    /* first check for attribute data */
    if (mdlDB_elementFilter(p) == FALSE)
    {
        mdlOutput_error ("Element has no attribute linkages");
        return (ERROR);
    }

    /* first word of the attribute data */
    start = (short *) (&p->dhdr.props.s + p->dhdr.attindx);
    /* extract the information from the linkage */
    status = mdlDB_decodeLink (&link, start);
    if (status == SUCCESS)
    {
        mdlDB_largestMslink (&maxMslink, link.tablename);
        sprintf (message, "Biggest Key: %ld", maxMslink);
        mdlOutput_message (message);
        return (SUCCESS);
    }

    mdlOutput_message ("Error decoding linkage");
    return (ERROR);
}

/*-----+
| name          displayTableStructure                    |
| author        BSI                                     1/91 |
+-----*/
Public int displayTableStructure(char *table_name)
{

```

```

MS_sqldasqla;
shortstatus, i;
status = mdlDB_describeTable (&sqla, table_name);
if (status != SUCCESS)
return (ERROR);
for (i = 0; i < sqla.numColumns; i++)
printf ("%20s %d\n", sqla.name[i], sqla.type[i]);
return (SUCCESS);
}

/*-----+
| name          printEmployeeTable          |
| author        BSI                        1/91 |
+-----*/
Public int printEmployeeTable()
{
    charquery[128];
    shorti;
    MS_sqldasqla;
    strcpy (query, "select * from employees");
    mdlDB_openCursor (query);
    while (mdlDB_fetchRow (&sqla) != QUERY_FINISHED)
    for (i = 0; i < sqla.numColumns; i++)
    printf ("%20s, %s\n", sqla.name[i], sqla.value[i]);
    mdlDB_closeCursor ();
    mdlDB_freeSQLDADescriptor (&sqla);
    return (SUCCESS);
}

```

file.mc

```

/*-----+
--+
|                                     |
|   $Workfile:   file.mc   $         |
|   $Revision:   5.1   $   $Date:    21 Aug 1991   6:19:58   $   |
|                                     |
+-----*/
/*
/*-----+
--+
|                                     |
| Copyright (C) 1991   Bentley Systems, Inc., All rights reserved. |
|                                     |
| "MicroStation", "MDL", and "MicroCSL" are trademarks of Bentley |
| Systems, Inc.                                     |

```

```

|                                     |
| Limited permission is hereby granted to reproduce and modify this|
| copyrighted material provided that the resulting code is used only in
|
| conjunction with Bentley Systems products under the terms of the|
| license agreement provided therein, and that this notice is retained|
| in its entirety in any such reproduction or modification.|
|                                     |
+-----+
*/
/*-----+
|
| file.mc - examples for the mdlFile_ functions.|
|                                     |
| This file is intended as an adjunct to the MDL manual to|
| illustrate MDL built-in function calling conventions and parameter|
| values. While it can be compiled, it does NOT, on its own,|
| constitute a workable MDL example.|
|                                     |
+-----+
*/
/*-----+
|
| Include Files
|                                     |
+-----+
*/
#include <mdl.h>
#include <msdefs.h> /* For MAXFILELENGTH, MAXDIRLENGTH, etc. */
#include <basedefs.h> /* For SUCCESS */
#include <system.h> /* For FindFileInfo */
#include <stdio.h>

/*-----+
|
| name main
|                                     |
| authorBSI 6/91
|                                     |
+-----+
*/
main
(
int argc,

```

```

char**argv
)
{
    char      outputFileName [MAXFILELENGTH+1];
    char      deviceName [MAXDEVICELENGTH+1];
    char      dirName [MAXDIRLENGTH+1];
    char      rootName [MAXNAMELENGTH+1];
    char      extName [MAXEXTENSIONLENGTH+1];
    int       logicalDrives, defaultDrive, diskFree;
    int       fileCount, attributes;
    FindFileInfo *fileInfoP;

    /* Look for the file "file.ma" in the list of directories specified
    via the BSI environment variable MS_MDL.    */
    if (mdlFile_find (outputFileName, "file", "MS_MDL", ".ma") ==
SUCCESS)
        mdlOutput_printf (MSG_MESSAGE, "%s found", outputFileName);
    else
        mdlOutput_error ("file.ma not found");

    /* Parse a file name into its components. We initialize the arrays
    receiving the of name. This is necessary since mdlFile_parseName
    is not guaranteed to change any of them.    */
    deviceName [0] = dirName [0] = rootName [0] = extName [0] = '\0';
    mdlFile_parseName (*argv, deviceName, dirName, rootName, extName);

    mdlOutput_printf (MSG_MESSAGE, "%s %s %s", deviceName, rootName,
extName);
    mdlOutput_printf (MSG_STATUS, "%s", dirName);

    /* mdlFile_buildName assembles a file name from the components */
    mdlFile_buildName (outputFileName, deviceName, dirName,
        "*", "ma");

    mdlOutput_printf (MSG_MESSAGE, "Finding %s", outputFileName);

    /* Find a list of files satisfying the requirements */
    if (mdlFile_findFiles (&fileInfoP, &fileCount, outputFileName,
FF_NORMAL)

                                == SUCCESS)
    {
        if (fileInfoP != NULL)
            free (fileInfoP);
    }

    mdlFile_getcwd (dirName, sizeof (dirName));
    mdlOutput_printf (MSG_STATUS, "Current dir is %s", dirName);

```



```

#if defined (pm386)
    /* mdlFile_setDrive is ignored and returns an error on platforms
       other than the PC. The following example sets the drive to C:
       on the PC. */
    mdlFile_setDrive (&logicalDrives, 3);

    /* Now find out what the system thinks is the current default. */
    if (mdlFile_getDrive (&defaultDrive) == SUCCESS)
        mdlOutput_printf (MSG_MESSAGE, "Default drive is %d", defaultDrive);

    /* Find out how much disk space is available on the C drive. */
    if (mdlFile_getDiskFree (&diskFree, 3) == SUCCESS)
        mdlOutput_printf (MSG_STATUS, "C: has %d bytes free", diskFree);
#endif

#if defined (pm386)
    /* Specify that for all calls to fopen by this MDL application,
       MicroStation should try to open the file with MDL_SHARE_DENY_NONE
       sharing mode.
       */
    mdlFile_setDefaultShare (MDL_SHARE_DENY_NONE);
#endif

if (mdlFile_getFileAttributes (&attributes, dirName) == SUCCESS)
{
    char message [80];

    message [0] = '\0';

    if (attributes & FF_SUBDIR)
        strcat (message, "SUBDIR ");
    if (attributes & FF_READONLY)
        strcat (message, "READONLY ");

    mdlOutput_printf (MSG_STATUS, message);
}
}

/*-----
-+
|
| name misc
|
| authorBSI          6/91
|
| This function illustrates the calling sequence of some of
| the mdlFile_ functions.
|

```

```

|                                     |
+-----+
*/
cmdName misc (void)
{
    FILE    *fileP;
    char     outputFileName [MAXFILELENGTH];

    /* Create a file using "junkfile" as the root name and a path from
    the list of names specified via the environment variable MS_MDL.
    This function generates a complete file path that can be used to
    create the file. It also deletes the existing file with the same
    name if it already exists.
    */
    if (mdlFile_create (outputFileName, "junkfile", "MS_MDL", "") ==
    SUCCESS)
    {
        mdlOutput_printf (MSG_MESSAGE, "%s created", outputFileName);
        if ((fileP = fopen (outputFileName, "w")) != NULL)
        {
            fprintf (fileP, "Created by the file example program.\n");
            fclose (fileP);
        }
    }
    else
        mdlOutput_error ("junkfile not created");

    /* Copy file "source" to file "dest" */
    if (mdlFile_copy ("dest", "source") != SUCCESS)
    {
        mdlOutput_printf (MSG_ERROR, "Unable to copy %s", "source");
    }
}

```

tutorial.mc

```

/*-----+
-+
|                                     |
| Copyright (c) 1985-91; Bentley Systems, Inc., All rights reserved.|
|                                     |
| "MicroStation", "MDL", and "MicroCSL" are trademarks of Bentley|
| Systems, Inc. and/or Intergraph Corporation. |
|                                     |
| This program is proprietary and unpublished property of Bentley |

```

```
| Systems Inc. It may NOT be copied in part or in whole on any medium,
|
| either electronic or printed, without the express written consent|
| of Bentley Systems, Inc. |
|
+-----+
*/
/*-----+
+
|
| $Workfile: scanexec.tmp $
| $Revision: 5.1 $ $Date: 30 Jul 1991 14:23:20 $
|
+-----+
*/
/*-----+
+
|
| tutorial.mc - examples for the mdltutorial_ functions.|
|
| This file is intended as an adjunct to the MDL manual to|
| illustrate MDL built-in function calling conventions and parameter|
| values. While it can be compiled, it does NOT, on its own,|
| constitute a workable MDL example.|
|
+-----+
*/
/*-----+
+
|
| Include Files
|
+-----+
*/
#include <mdl.h> /* system include files */
#include <userfnc.h>
#include <msinputq.h>

/*-----+
+
|
| Function declarations
|
+-----+
*/

void receiveMessage (), monitorQueue ();
```

```

/*-----
-+
|
| name main
|
| authorBSI          9/90
|
+-----
*/
main ()
{
    if (mdlTutorial_load ("sample") != SUCCESS)
    /* mdlTutorial_load displays an error message */
    exit (1);

    /* Install some MDL input user functions. */
    mdlInput_setMonitorFunction (MONITOR_ALL, monitorQueue);
    mdlInput_setFunction (INPUT_MESSAGE_RECEIVED, receiveMessage);
}

/*-----
-+
|
| name monitorQueue
|
| authorBSI          9/90
|
| monitorQueue is an example of a input-monitor user function.
| See userInput_monitor in the MDL documentation for more
| information.
|
+-----
*/
Private int monitorQueue
(
    Inputq_element*queueElementP
)
{
    /* If this is a tutorial keyin, redirect it to this application
       instead of letting MicroStation handle it. */
    if (queueElementP->hdr.cmdtype == TUTKEYIN)
        strcpy (queueElementP->hdr.taskId, mdlSystem_getCurrTaskID ());

    return INPUT_ACCEPT;
}

```

```

/*-----
-+
|
| name receiveMessage
|
| authorBSI          9/90
|
| This is a input-receive user function. See userInput_receive
| in the MDL manual for more information.
|
| This function receives the messages that monitorQueue|
| redirects to this application.
|
| This functions justs echos the tutorial keyin to an output
| field. It is assumed that fields 1-3 are input fields and
| field 4-6 are output fields.
|
+-----
*/
Private int receiveMessage
(
Inputq_element*queueElementP
)
{
    char    buffer [80];

    strcpy (buffer, "OUTPUT ");
    strcat (buffer, queueElementP->u.tutkeyin.keyin);
    mdlTutorial_output (queueElementP->hdr.uc_fno_value+3, buffer,
                        strlen (buffer));
}

/*-----
-+
|
| name chooseField
|
| authorBSI          9/90
|
+-----
*/
cmdName chooseField
(
char*fieldP
)
{
    intfieldNumber;

```

```

        if ((sscanf (fieldP, "%d", &fieldNumber) != 1) ||
            (fieldNumber < -2 || fieldNumber > 3))
        {
            mdlOutput_error ("Expected a field number between -1 and 3.");
            return;
        }

        /* Select the desited field and draw the box. */
        mdlTutorial_positionInputField (fieldNumber, 1);
    }

```

create.mc

```

/*-----+
|
| Copyright (1995) Bentley Systems, Inc., All rights reserved.|
|
| "MicroStation" is a registered trademark and "MDL" and|
| "MicroCSL" are trademarks of Bentley Systems, Inc.    |
|
| Limited permission is hereby granted to reproduce and modify |
| this copyrighted material provided that the resulting code is |
| used only in conjunction with Bentley Systems products under |
| the terms of the license agreement provided therein, and that |
| this notice is retained in its entirety in any such reproduction |
| or modification.                                           |
+-----*/
/*-----+
|
| $Logfile:   J:/mdl/examples/doc/create.mcv  $
| $Workfile:  create.mc  $
| $Revision:  5.5  $
| $Date:      20 Jun 1995 08:49:38  $
+-----*/
/*-----+
|
| create.mc - examples for the mdlXXXX_create functions.|
| This file is intended as an adjunct to the MDL manual to|
| illustrate MDL built-in function calling conventions and |
| parameter values. While it can be compiled, it does NOT, on |
| its own, constitute a workable MDL example.|
+-----*/
#include    <mdl.h>    /* system include files */
#include    <global.h>
#include    <mselems.h>

```

```

#include    <tcb.h>
#include    <userpref.h>
#include    <wchar.h>
#include    <mselemen.fdf>
/*-----+
| name placeAShape                                     |
| authorBSI      8/90                                |
+-----*/
Private void placeAShape(void)
{
    Dpoint3dshapePts[3];
    MSElementUnionshape;
    shapePts[0].x = fc_zero;
    shapePts[0].y = 1.0;
    shapePts[0].z = fc_zero;
    shapePts[1].x = 100.0;
    shapePts[1].y = 200.0;
    shapePts[1].z = fc_zero;
    shapePts[2].x = 300.0;
    shapePts[2].y = 400.0;
    shapePts[2].z = fc_zero;
    if (mdlShape_create (&shape, NULL, shapePts, 3, -1) == SUCCESS)
    {
        mdlElement_display (&shape, NORMALDRAW);
        mdlElement_add (&shape);
    }
}

/*-----+
| name placeALineString                               |
| authorBSI      8/90                                |
+-----*/
Private void placeALineString(Dpoint3d *lsPoints, int numVerts)
{
    MSElementUnionlineString;
    if (mdlLineString_create(&lineString, NULL, lsPoints,
        numVerts)== SUCCESS)
    {
        mdlElement_display(&lineString, NORMALDRAW);
        mdlElement_add(&lineString);
    }
}

/*-----+
| name placeAnArc                                     |
| authorBSI      8/90                                |
+-----*/
Private void placeAnArc(MSElementUnion *existingArc)

```

```

    {
        Dpoint3d center;
        MSElementUnion arc;

        center.x = 100.0;
        center.y = 100.0;
        center.z = 100.0;

        if (mdlArc_create (&arc, existingArc, &center, 100.0, 200.0,
            NULL, fc_zero, fc_pi) == SUCCESS)
        {
            mdlElement_display (&arc, NORMALDRAW);
            mdlElement_add (&arc);
        }
    }

/*-----+
| name placeAnArcByPoints          |
| authorBSI            8/90       |
+-----*/
Private void placeAnArcByPoints(void)
{
    MSElementUnion arc;
    Dpoint3d arcPt[3];

    arcPt[0].x = 20.;    /* start point */
    arcPt[0].y = 30.;
    arcPt[0].z = fc_zero;
    arcPt[1].x = 50.;    /* mid point */
    arcPt[1].y = 60.;
    arcPt[1].z = fc_zero;
    arcPt[2].x = 20.;    /* end point */
    arcPt[2].y = 80.;
    arcPt[2].z = fc_zero;
    if (mdlArc_createByPoints(&arc, NULL, arcPt)==SUCCESS)
    {
        mdlElement_display (&arc, NORMALDRAW);
        mdlElement_add (&arc);
    }
}

/*-----+
| name placeAnArcByCenter          |
| authorBSI            8/90       |
+-----*/
Private void placeAnArcByCenter(int view)
{
    MSElementUnion arc;
    Dpoint3d arcPt[3];

```



```

    arcPt[0].x = 20.;    /* start point */
    arcPt[0].y = 30.;
    arcPt[0].z = fc_zero;
    arcPt[1].x = 50.;    /* center */
    arcPt[1].y = 30.;
    arcPt[1].z = fc_zero;
    arcPt[2].x = 80.;    /* end point */
    arcPt[2].y = 30.;
    arcPt[2].z = fc_zero;
    /* create an arc with a radius of 44.0 and center at [50,30,0] */
    if (mdlArc_createByCenter(&arc, NULL, arcPt, TRUE,
        44.0, view)== SUCCESS)
    {
        mdlElement_display(&arc, NORMALDRAW);
        mdlElement_add(&arc);
    }
}

/*-----+
| name placeEllipse                                |
| authorBSI            8/90                        |
+-----*/
Private void placeEllipse(void)
{
    MSElementUnion ellipse;
    Dpoint3d center;
    center.x = 100.0;
    center.y = 100.0;
    center.z = 100.0;
    if (mdlEllipse_create(&ellipse, NULL, &center, 10.0, 15.0,
        NULL, -1)==SUCCESS)
    {
        mdlElement_display (&ellipse, NORMALDRAW);
        mdlElement_add (&ellipse);
    }
}

/*-----+
| name placeCircleByPoints                          |
| authorBSI            8/90                        |
+-----*/
Private void placeCircleByPoints(void)
{
    MSElementUnion circle;
    Dpoint3d circlePt[3];
    circlePt[0].x = 200.;
    circlePt[0].y = 300.;
    circlePt[0].z = 000.;

```

```

        circlePt[1].x = 500.;
        circlePt[1].y = 600.;
        circlePt[1].z = 000.;
        circlePt[2].x = 200.;
        circlePt[2].y = 800.;
        circlePt[2].z = 000.;
    if (mdlCircle_createBy3Pts (&circle, NULL, circlePt, -1)==SUCCESS)
    {
        mdlElement_display (&circle, NORMALDRAW);
        mdlElement_add (&circle);
    }
}

/*-----+
| name  placeText                                     |
| authorBSI                8/90                      |
+-----*/
Private void placeText(MSElementUnion *currentText, long textPos)
{
    MSElementUniontext;
    Dpoint3dpt[3];
    TextSizeParamtxtSize;
    TextParamtxtParam;
    TextEDFieldedFields[2];
    TextEDParamedParam;
    pt[0].x = 285.;/* origin of text element */
    pt[0].y = 450.;
    pt[0].z = fc_zero;
    txtSize.mode= TXT_BY_TEXT_SIZE;
    txtSize.size.width= fc_2;
    txtSize.size.height= 3.;
    txtParam.font= 2;
    txtParam.just= -1;/* use active justification */
    txtParam.style= -1;/* use active style */
    txtParam.viewIndependent = FALSE;
    edFields[0].start= 1;
    edFields[0].len= 2;
    edFields[0].just= 2;
    edFields[1].start= 5;
    edFields[1].len= 1;
    edFields[1].just= 3;
    edParam.numEDFields= 2;
    edParam.edField= edFields;
    /* create a new text element */
    if (mdlText_create (&text, NULL, "this text placed in MDL", pt,
        &txtSize, NULL, &txtParam, &edParam) == SUCCESS)
    {
        mdlElement_display (&text, NORMALDRAW);
    }
}

```

```

mdlElement_add (&text);
}

/* overwrite an existing text element without changing parameters */
if (mdlText_create (&text, currentText, "overwritten text",
                    NULL, NULL, NULL, NULL, NULL) == SUCCESS)
{
    mdlElement_rewrite (&text, currentText, textPos);
    mdlElement_display (&text, NORMALDRAW);
}
}

/*-----+
| name placeTextNode |
| authorBSI          8/90 |
+-----*/
Private void placeTextNode(void)
{
    MSElementUnion textNode;
    Dpoint3d origin;
    MStextSize nodeSize;
    TextParam txtParam;
    double lineSpacing;

    origin.x= 100.;
    origin.y= 200.;
    origin.z = 0.;
    nodeSize.width = 10.;
    nodeSize.height= 13.;
    lineSpacing = 1.1;
    txtParam.font= 2;
    txtParam.just= -1;/* use active justification */
    txtParam.style= -1;/* use active style */
    txtParam.viewIndependent = FALSE;
    if (mdlTextNode_create (&textNode, NULL, &origin, NULL, &lineSpacing,
                           &nodeSize, &txtParam)==SUCCESS)
    {
        mdlElement_display(&textNode, NORMALDRAW);
        mdlElement_add(&textNode);
    }
}

/*-----+
| name text_getExtendedParam |
| sets flags and values for extended text attributes|
| (slant, underline, character spacing, and direction)|
| according to tcb and userpref |
| author BSI 4/93 |
+-----*/
Public void text_setExtendedParam(

```

```

/*-----+
| name  placeTextWide                                |
| authorBSI          7/93                            |

```

```

|-----|
+-----*/
Private void placeTextWide(MSElementUnion *currentText, long textPos)
{
    MSElementUnion text;
    Dpoint3d pt[3];
    TextSizeParam txtSize;
    TextParamWide txtParam;
    TextEDField edFields[2];
    TextEDParam edParam;
    char *string1 = "this text placed in MDL";
    char *string2 = "overwritten text";
    MSWideChar wString1[50], wString2[50];
    /* convert to wide-character string - the third argument is the size
       of wString, see "stdlib.h" for details */
    mbstowcs (wString1, string1, 50);
    mbstowcs (wString2, string2, 50);
    pt[0].x = 285.; /* origin of text element */
    pt[0].y = 450.;
    pt[0].z = fc_zero;
    txtSize.mode = TXT_BY_TEXT_SIZE;
    txtSize.size.width = fc_2;
    txtSize.size.height = 3.;
    txtParam.font = 2;
    txtParam.just = -1; /* use active justification */
    txtParam.style = -1; /* use active style */
    txtParam.viewIndependent = FALSE;
    /* set extended params (slant, underline, char spacing, and
    direction) */
    text_setExtendedParam (&txtParam);
    edFields[0].start = 1;
    edFields[0].len = 2;
    edFields[0].just = 2;
    edFields[1].start = 5;
    edFields[1].len = 1;
    edFields[1].just = 3;
    edParam.numEDFields = 2;
    edParam.edField = edFields;
    /* create a new text element */
    if (mdlText_createWide (&text, NULL, wString1, pt, NULL,
        &txtSize, &txtParam, &edParam) == SUCCESS)
    {
        mdlElement_display (&text, NORMALDRAW);
        mdlElement_add (&text);
    }
    /* overwrite an existing text element without changing parameters */
    if (mdlText_createWide (&text, currentText, wString2,

```

```

        NULL, NULL, NULL, NULL, NULL) == SUCCESS)
    {
        mdlElement_rewrite (&text, currentText, textPos);
        mdlElement_display (&text, NORMALDRAW);
    }
}

/*-----+
|
| name placeTextNodeWide
|
| authorBSI          7/93
|
|-----*/
Private void placeTextNodeWide(void)
{
    MSElementUnion  textNode;
    Dpoint3d         origin;
    TextSizeParam    nodeSize;
    TextParamWide    txtParam;
    origin.x= 100.;
    origin.y= 200.;
    origin.z = 0.;
    nodeSize.mode= TXT_BY_TEXT_SIZE;
    nodeSize.size.width= 10.;
    nodeSize.size.height= 13.;
    txtParam.font= 2;
    txtParam.just= -1; /* use active justification */
    txtParam.style= -1; /* use active style */
    txtParam.viewIndependent = FALSE;
    txtParam.lineSpacing = tcb->nodespace;
    /* set extended params (slant, underline, char spacing, and
direction) */
    text_setExtendedParam (&txtParam);
    if (mdlTextNode_createWide (&textNode, NULL, &origin, NULL,
        &nodeSize, &txtParam) == SUCCESS)
    {
        mdlElement_display (&textNode, NORMALDRAW);
        mdlElement_add (&textNode);
    }
}

/*-----+
|
| name placeCone
|
| authorBSI          8/90
|
|-----*/

```

```

Private void placeCone(RotMatrix *rMatrix)
{
    MSElementUnion cone;
    Dpoint3d conePts[2];
    conePts[0].x = 100.;
    conePts[0].y = 200.;
    conePts[0].z = 0.;
    conePts[1].x = 100.;
    conePts[1].y = 200.;
    conePts[1].z = 10.;
    if (mdlCone_create (&cone, NULL, 2.0, 1.0, &conePts[0], &conePts[1],
        rMatrix) == SUCCESS)
    {
        mdlElement_display (&cone, NORMALDRAW);
        mdlElement_add (&cone);
    }
}

/*-----+
| name . |
| authorBSI 8/90 |
|-----*/
Private void placeCylinder(void)
{
    MSElementUnion cone;
    Dpoint3d conePts[2];
    conePts[0].x = 100.;
    conePts[0].y = 200.;
    conePts[0].z = 0.;
    conePts[1].x = 100.;
    conePts[1].y = 200.;
    conePts[1].z = 10.;
    if (mdlCone_createRightCylinder (&cone, NULL, 2.0, &conePts[0],
        &conePts[1]) == SUCCESS)
    {
        mdlElement_display (&cone, NORMALDRAW);
        mdlElement_add (&cone);
    }
}

```

extract.mc

```

/*-----+
-+

```

```

|                                     |
| Copyright (1995) Bentley Systems, Inc., All rights reserved.|
|                                     |
| "MicroStation" is a registered trademark and "MDL" and "MicroCSL"|
| are trademarks of Bentley Systems, Inc.      |
|                                     |
| Limited permission is hereby granted to reproduce and modify this|
| copyrighted material provided that the resulting code is used only |
| in conjunction with Bentley Systems products under the terms of the|
| license agreement provided therein, and that this notice is retained|
|                                     |
| in its entirety in any such reproduction or modification.|
|                                     |
+-----+
*/
/*-----+
+
|                                     |
| $Logfile:   J:/mdl/examples/doc/extract.mcv $
| $Workfile:  extract.mc $
| $Revision:  5.4 $
| $Date:      20 Jun 1995 08:49:40 $
|                                     |
+-----+
*/
/*-----+
+
|                                     |
| extract.mc - examples for the mdlXXXX_extract functions.|
|                                     |
| This file is intended as an adjunct to the MDL manual to|
| illustrate MDL built-in function calling conventions and parameter|
| values. While it can be compiled, it does NOT, on its own,|
| constitute a workable MDL example.|
|                                     |
+-----+
*/
/*-----+
+
|                                     |
| Include Files                    |
|                                     |
+-----+
*/
#include <mdl.h>    /* system include files */
#include <global.h>
#include <mselems.h>

```



```

#include    <wchar.h>
#include    <mselemen.fdf>
/*-----
-+
|
| name dumpLinear                                |
|
| authorBSI            8/90            |
|
|-----
*/
Private void dumpLinear(MSElementUnion *linearElem, int fileNum)
{
    Dpoint3dpt[MAX_VERTICES];
    int i, numVertices;
    if (mdlLinear_extract (pt, &numVertices, linearElem, fileNum) ==
SUCCESS)
    {
        printf ("element contains [%s] vertices\n", numVertices);
        for (i=0; i<numVertices; i++)
        {
            printf ("vert[%d]=[%f,%f,%f]\n", i, pt[i].x, pt[i].y, pt[i].z);
        }
    }
}

/*-----
-+
|
| name dumpArc                                |
|
| authorBSI            8/90            |
|
|-----
*/
Private void dumpArc(MSElementUnion *arcElem)
{
    Dpoint3dendPts[2], center;
    doublestart, sweep, primary, secondary;
    RotMatrixrMatrix;
    if (mdlArc_extract (endPts, &start, &sweep, &primary, &secondary,
&rMatrix, &center, arcElem) == SUCCESS)
    {
        printf ("startPt=[%f,%f,%f], endPt=[%f,%f,%f]\n",
            endPts[0].x, endPts[0].y, endPts[0].z,
            endPts[1].x, endPts[1].y, endPts[1].z);
        printf ("start=%f, sweep=%f, primary=%f, secondary=%f\n",
            start, sweep, primary, secondary);
    }
}

```

```

        printf ("center=[%f,%f,%f]\n", center.x, center.y, center.z);
    }
}

/*-----
-+
|
| name dumpText                                |
|
| authorBSI          8/90          |
|
|-----
*/
Private void dumpText(MSElementUnion *textElem)
{
    int      i, numEdfields, just;
    Dpoint3d  origin, userOrigin;
    TextEDField  edFields[MAX_EDFIELDS];
    MSTextSize  textSize;
    char  textString[256];
    TextStyleInfo  tsi;
    if (mdlText_extract (&origin, &userOrigin, &numEdfields,
        edFields, textString, NULL, &tsi, &just, NULL,
        &textSize, textElem) == SUCCESS)
    {
        printf ("text string=[%s]\n", textString);
        printf ("origin=[%f,%f,%f], userOrigin=[%f,%f,%f]\n",
            origin.x, origin.y, origin.z,
            userOrigin.x, userOrigin.y, userOrigin.z);
        printf ("text size=[%f,%f]\n", textSize.height, textSize.width);
        printf ("text font=%d, just=%d\n", tsi.font, just);
        printf ("text contains %d enter-data fields\n", numEdfields);
        for (i=0; i<numEdfields; i++)
        {
            printf ("field[%d], start=%d, length=%d, just=%d\n",
                i+1, edFields[i].start, edFields[i].len, edFields[i].just);
        }
    }
}

/*-----
-+
|
| name dumpTextWide                                |
|
| authorBSI          7/93          |
|
|-----

```

```

+-----
*/
Private void dumpTextWide(MSElementUnion *textElem)
{
    int      i;
    Dpoint3d  origin, userOrigin;
    TextEDField  edFields[MAX_EDFIELDS];
    TextEDParam  edParam;
    TextSizeParam  textSizeParam;
    TextParamWide  textParam;
    char    textString[256];
    MSWideChar  wideString[256];
    edParam.edField = edFields;
    memset (&textParam, 0, sizeof(textParam));
    textSizeParam.mode = TXT_BY_TILE_SIZE;
    if (mdlText_extractWide (wideString, &origin, &userOrigin, NULL,
                            &textSizeParam, &textParam, &edParam,
                            textElem) == SUCCESS)
    {
        wcstombs (textString, wideString, 256);
        printf ("text string=[%s]\n", textString);
        printf ("origin=[%f,%f,%f], userOrigin=[%f,%f,%f]\n",
                origin.x, origin.y, origin.z,
                userOrigin.x, userOrigin.y, userOrigin.z);
        printf ("text size=[%f,%f]\n",
                textSizeParam.size.height, textSizeParam.size.width);
        printf ("text font=%d, just=%d\n", textParam.font, textParam.just);
        printf ("text contains %d enter-data fields\n", edParam.numEDFields);
        for (i=0; i<edParam.numEDFields; i++)
        {
            printf ("field[%d], start=%d, length=%d, just=%d\n",
                    i+1, edFields[i].start, edFields[i].len, edFields[i].just);
        }
        if (textParam.flags.slant)
            printf ("text slanted by %f degrees\n",
                    textParam.slant * fc_180overpi);
        if (textParam.flags.vertical)
            printf ("vertical text\n");
        if (textParam.flags.underline)
            printf ("underlined text\n");
        if (textParam.flags.fixedWidthSpacing)
            printf ("fixed-width spacing=%f\n", textParam.characterSpacing);
        else if (textParam.flags.interCharSpacing)
            printf ("inter-char spacing=%f\n", textParam.characterSpacing);
    }
}

```

```

/*-----
-+
|                                     |
| name  showTextShape                |
|                                     |
| authorBSI          8/90           |
|                                     |
+-----
*/
Private void showTextShape(MSElementUnion *textElem, int view)
{
    Dpoint3dpoints[5];
    int i;
    if (mdlText_extractShape (points, NULL, textElem, FALSE, view) ==
SUCCESS)
    {
        for (i=0; i<5; i++)
        {
            printf ("textshape[%d] = [%f,%f,%f]\n", i, points[i].x,
                    points[i].y, points[i].z);
        }
    }
}

/*-----
-+
|                                     |
| name  showTextString                |
|                                     |
| authorBSI          8/90           |
|                                     |
+-----
*/
Private void showTextString(MSElementUnion *textElem)
{
    char    textString[256];
    if (mdlText_extractString (textString, textElem) == SUCCESS)
    {
        printf ("string=[%s]\n", textString);
    }
}

/*-----
-+
|                                     |
| name  showTextStringWide            |
|                                     |
| authorBSI          7/93           |
|                                     |

```

```

| |
+-----+
*/
Private void showTextStringWide(MSElementUnion *textElem)
{
    chartextString[256];
    MSWideCharWideString[256];
    if (mdlText_extractStringWide (wideString, textElem) == SUCCESS)
    {
        wcstombs (textString, wideString, 256);
        printf ("string=[%s]\n", textString);
    }
}

/*-----+
-+
| |
| name dumpTextNode |
| |
| authorBSI 8/90 |
| |
+-----+
*/
Private void dumpTextNode(MSElementUnion *nodeElem)
{
    Dpoint3d origin;
    MTextSize tileSize;
    TextParam textParam;
    RotMatrix rMatrix;
    double lineSpacing;
    int nodeNum;

    if (mdlTextNode_extract (&origin, &rMatrix, &tileSize, &lineSpacing,
        &textParam, &nodeNum, nodeElem) == SUCCESS)
    {
        printf ("Node origin=[%f,%f,%f]\n", origin.x, origin.y, origin.z);
        printf ("character size=[%f,%f]\n", tileSize.height, tileSize.width);
        printf ("font=%d\n", textParam.font);
        printf ("node number = %d\n", nodeNum);
        printf ("lineSpacing=%f\n", lineSpacing);
    }
}

/*-----+
-+
| |
| name dumpTextNodeWide |
| |
| authorBSI 7/93 |
| |

```

```

|
+-----|
*/
Private void dumpTextNodeWide(MSElementUnion *nodeElem)
{
    Dpoint3d lowerLeft, snapPoint;
    TextSizeParam textSizeParam;
    TextParamWide textParam;

memset (&textParam, 0, sizeof(textParam));
    textSizeParam.mode = TXT_BY_TILE_SIZE;
    if (mdlTextNode_extractWide (&lowerLeft, &snapPoint, NULL,
        &textSizeParam, &textParam, nodeElem) == SUCCESS)
    {
        printf ("Lower-left corner=[%f,%f,%f]\n",
            lowerLeft.x, lowerLeft.y, lowerLeft.z);
        printf ("Snap point=[%f,%f,%f]\n",
            snapPoint.x, snapPoint.y, snapPoint.z);
        printf ("character size=[%f,%f]\n",
            textSizeParam.size.height, textSizeParam.size.width);
        printf ("font=%d\n", textParam.font);
        printf ("node number = %d\n", textParam.nodeNumber);
        printf ("lineSpacing=%f\n", textParam.lineSpacing);
        if (textParam.flags.slant)
            printf ("text node slanted by %f degrees\n",
                textParam.slant * fc_180overpi);
        if (textParam.flags.vertical)
            printf ("vertical text node\n");
        if (textParam.flags.underline)
            printf ("underlined text node\n");
        if (textParam.flags.fixedWidthSpacing)
            printf ("fixed-width spacing=%f\n", textParam.characterSpacing);
        else if (textParam.flags.interCharSpacing)
            printf ("inter-char spacing=%f\n", textParam.characterSpacing);
    }
}

/*-----
-+
|
| name showTextNodeShape |
|
| authorBSI 8/90 |
|
+-----
*/
Private void showTextNodeShape(MSElementUnion *nodeElem, int view)
{

```

```

Dpoint3dpoints[5];
int i;
if (mdlTextNode_extractShape (points, NULL, nodeElem, FALSE, view)
    == SUCCESS)
{
for (i=0; i<5; i++)
{
printf ("nodeShape[%d] = [%f,%f,%f]\n", i, points[i].x,
points[i].y, points[i].z);
}
}
}

```

misc.mc

```

/*-----
-+
|
|      Copyright (c) 1985-91;  Bentley Systems, Inc., All rights reserved.|
|
|      "MicroStation", "MDL", and "MicroCSL" are trademarks of Bentley|
|      Systems, Inc. and/or Intergraph Corporation. |
|
|      This program is proprietary and unpublished property of Bentley  |
|      Systems Inc. It may NOT be copied in part or in whole on any medium,
|
|      either electronic or printed, without the express written consent|
|      of Bentley Systems, Inc. |
|
|-----
*/
/*-----
-+
|
|      $Workfile:  scanexec.tmp  $
|      $Revision:  5.1  $  $Date:  30 Jul 1991 14:23:46  $
|
|-----
*/
/*-----
-+
|
|      misc.mc - examples of functions documented in the Miscellaneous
|
|      Functions section of the manual. The categories included are|

```

```

|   mdlUtil... functions, mdlString... functions, and mdlUserPrefs_.
|
|
|   This file is intended as an adjunct to the MDL manual to
|   illustrate MDL built-in function calling conventions and parameter
|   values. While it can be compiled, it does NOT, on its own,
|   constitute a workable MDL example.
|
+-----+
*/
/*-----+
|
|   Include Files
|
+-----+
*/

#include    <mdl.h>      /* system include files */
#include    <userpref.h> /* Needed for the mdlUserPrefs_ functions */

/*-----+
|
|   Private Global variables
|
+-----+
*/

/* The following arrays are used in the examples using the
   utilities. */
longlongArray [] = {1, 75, -100, 40000, 0, 1, 1, -90000};
doubledoubleArray [] = {7.9, 1.5, 17e22, -1100, .000123};
char*stringArray [] =
{
    "Test",
    "this",
    "and",
    "print",
    "the",
    "result",
    "."
};

/*-----+
|
|

```



```

|   Function declarations   |
|                           |
+-----+
*/

Private int compareStrings (char **, char **) ;

/*-----+
|
| name sortLongs           |
|                           |
| authorBSI               9/90   |
|                           |
+-----+
*/
cmdName void sortLongs
(
char*argP/* => request for ASCENDING or DESCENDING. No argument
           is interpreted as ASCENDING. */
)
{
    int    ascending, i;
    int    numEntries = sizeof (longArray)/ sizeof (long);
    long    localCopy [sizeof (longArray)/ sizeof (long)];

    if ((ascending = checkForAscending (argP)) < 0)
return;

    /* Sort a local copy */
    memcpy (localCopy, longArray, sizeof (longArray));
    mdlUtil_sortLongs (localCopy, numEntries, ascending);

    /* Print the sorted array */
    printf ("Sort Longs Example.\n");
    for (i = 0; i < numEntries; i++)
printf ("%d\n", localCopy [i]);
}

/*-----+
|
| name sortDoubles         |
|                           |
| authorBSI               9/90   |
|                           |
+-----+

```

```

+-----
*/
cmdName void sortDoubles
(
char*argP/* => request for ASCENDING or DESCENDING. No argument
           is interpreted as ASCENDING. */
)
{
    int    ascending, i;
    int    numEntries = sizeof (doubleArray)/ sizeof (double);
    double localCopy [sizeof (doubleArray)/ sizeof (double)];

    if ((ascending = checkForAscending (argP)) < 0)
return;

    /* Sort a local copy */
    memcpy (localCopy, doubleArray, sizeof (doubleArray));
    mdlUtil_sortDoubles (localCopy, numEntries, ascending);

    /* Print the sorted array */
    printf ("Sort Doubles Example.\n");
    for (i = 0; i < numEntries; i++)
        printf ("%g\n", localCopy [i]);
}

/*-----
-+
|                                     |
| name sortStrings                   |
|                                     |
| authorBSI          9/90           |
|                                     |
+-----
*/
cmdName void sortStrings
(
char*argP
)
{
    int    ascending, i;
    int    numEntries = sizeof (stringArray)/ sizeof (char *);
    char    *localCopy [sizeof (stringArray)/ sizeof (char *)];

    if ((ascending = checkForAscending (argP)) < 0)
return;

    /* Sort a local copy */

```

```

memcpy (localCopy, stringArray, sizeof (stringArray));
mdlUtil_sortStrings (localCopy, numEntries, ascending);

/* Print the sorted array */
printf ("Sort Strings Example.\n");
for (i = 0; i < numEntries; i++)
printf ("%s\n", localCopy [i]);
}

/*-----
-+
|                                     |
| name quickSort                     |
|                                     |
| authorBSI          9/90            |
|                                     |
+-----
*/
cmdName void quickSort
(
void
)
{
    int    i;
    int    numEntries = sizeof (stringArray)/ sizeof (char *);
    char    *localCopy [sizeof (stringArray)/ sizeof (char *)];

    /* Sort a local copy */
    memcpy (localCopy, stringArray, sizeof (stringArray));

    /* Sort the strings into ascending order. */
    mdlUtil_quickSort (localCopy, numEntries, sizeof (char *),
        compareStrings);

    /* Print the sorted array */
    printf ("Quick Sort Example.\n");
    for (i = 0; i < numEntries; i++)
    printf ("%s\n", localCopy [i]);
}

/*-----
-+
|                                     |
| name stringTo                       |
|                                     |
| authorBSI          9/90            |
|                                     |
|                                     |
+-----

```

```

| This function illustrates the use of mdlString_to... |
| functions. |
| |
+-----+
*/
cmdName void stringTo
(
void
)
{
    Dpoint3ddPoint3d;
    doubleangle;
    doubleuors;

    mdlString_toPoint (&dPoint3d, NULL, "1:20:, ::1000", 0);

    mdlOutput_printf (MSG_MESSAGE, "Point: %f, %f",
dPoint3d.x, dPoint3d.y);

    mdlString_toAngle (&angle, "20^10'13'");
    mdlOutput_printf (MSG_STATUS, "Angle: %g", angle);

    mdlString_toUors (&uors, ":10");
    mdlOutput_printf (MSG_PROMPT, "Uors: %g", uors);
}

/*-----+
--
| |
| name stringFrom |
| |
| authorBSI 9/90 |
| |
| This function illustrates the use of mdlString_from... |
| functions. |
| |
+-----+
*/
cmdName void stringFrom
(
void
)
{
    charbuffer [80];
    Dpoint3ddPoint3d;

```

```

dPoint3d.x = 1.0;
dPoint3d.y = 7.5;
dPoint3d.z = 0.;

mdlString_fromPoint (buffer, &dPoint3d, FALSE);
mdlOutput_printf (MSG_MESSAGE, "Point: %s", buffer);

/* dms=YES, igdsChar=NO, decimal=2, trailingZeros=NO */
mdlString_fromAngle (buffer, 45.0, TRUE, FALSE, 2, 0);
mdlOutput_printf (MSG_STATUS, "Angle: %s", buffer);

mdlString_fromDirection (buffer, 27.45, TRUE, FALSE, TRUE, -1, 2,
0);
mdlOutput_printf (MSG_COMMAND, "Direction: %s", buffer);

mdlString_fromUors (buffer, 1000.0);
mdlOutput_printf (MSG_PROMPT, "Uors: %s", buffer);
}

/*-----
-+
|
| name  setTaskSize          |
|
| authorBSI          9/90    |
|
|-----
*/
cmdName setTaskSize
(
char*argP
)
{
    longtaskSize;
    UserPrefsuPref;

    if ((taskSize = atoi (argP)) != 0)
    {
        mdlUserPrefs_get (&uPref);
        uPref.taskSize = taskSize;
        mdlUserPrefs_save (&uPref);
    }
    else
        mdlOutput_error ("Tasksize not set.");
}

```

```

/*-----
-+
|
| name regExpression          |
|
| authorBSI          9/90    |
|
+-----
*/
cmdName void regExpression
(
char*argP
)
{
    static char testString [] =
    "This is the test string. Use mdlString_matchRe to find patterns"
    " in it. [Try to find this]. 111111222344455 Run the tests on it.";
    /* Find "test" in the test string. */
    static char test1 [] = "test";
    /* Find the longest string beginning with "in" and ending with "it"
    */
    static char test2 [] = "in.*it";
    /* Find "it" followed by any other character and the end of the line
    */
    static char test3 [] = "it.$";
    /* Find the longest string starting with "t", ending with "st"
       with no intervening spaces. */
    static char test4 [] = "t[^ ]st";
    /* Find the longest string starting with "m" and followed by one
       or more characters from the [cta]. */
    static char test5 [] = "m[cta]+";
    /* Shows the use of '\ ' to tell the regular expression compiler to
       treat "[" as a normal character. */
    static char test6 [] = "\\[Try to find this\\]\\.";
    /* Find 2 consecutive digits. */
    static char test7 [] = ":d:d";
    /* Find the longest string that starts with mdl and does not
    contain
    a blank, comma, or period. */
    static char test8 [] = "mdl[^,\\. ]*";

    char      *startMatchP, *endMatchP;

    if (*argP != '\0')
    {
        /* A string was provided as part of the command. Use it
           as the search pattern. */

```

```

runMatchRETest (testString, argP);
}
else
{
/* A string was not provided. Run the standard tests. */
runMatchRETest (testString, test1);
runMatchRETest (testString, test2);
runMatchRETest (testString, test3);
runMatchRETest (testString, test4);
runMatchRETest (testString, test5);
runMatchRETest (testString, test6);
runMatchRETest (testString, test7);
runMatchRETest (testString, test8);
}
}

/*-----
-+
|                                     |
|   Private Utility Functions       |
|                                     |
+-----
*/
/*-----
-+
|                                     |
| name runMatchRETest               |
|                                     |
| authorBSI          9/90          |
|                                     |
+-----
*/
Private void runMatchRETest
(
char*searchString,
char*pattern
)
{
char*startMatch, *endMatch;
charbuffer [300];
int length, retValue;

retValue = mdlString_matchRE (searchString, pattern,
                             &startMatch, &endMatch);

if (retValue != SUCCESS)
{

```

```

    mdlUtil_beep (3);
    printf ("mdlString_matchRE returned %d.\n", retValue);
    return;
}

    length = endMatch - startMatch;
    strncpy (buffer, startMatch, length);
    buffer [length] = '\0';
    printf ("%s\n", buffer);
    return;
}

/*-----
-+
|
| name  checkForAscending      |
|
| authorBSI          9/90      |
|
+-----
*/
Private int checkForAscending /* <= -1 for error, 0 for descending,
                             1 for ascending. */
(
    char*argP
)
{
    int    ascending;

    strupr (argP);
    /* If no argument was entered, it is a substring of ASCENDING */
    if ((ascending = substring (argP, "ASCENDING")) == 0)
    {
        if (substring (argP, "DESCENDING") == 0)
        {
            mdlOutput_error ("Invalid argument");
            return -1;
        }
    }

    return ascending;
}

/*-----
-+
|
|

```



```

| name substring          |
|
| authorBSI              9/90    |
|
+-----+
*/
Private int substring
(
char*subP,
char*stringP
)
{
    while (*subP)
    if (*subP++ != *stringP++)
        return 0;

    /* The end of *subP was encountered without finding a difference. */
    return 1;
}

/*-----+
-+
|
| name compareStrings    |
|
| authorBSI              9/90    |
|
+-----+
*/
Private int compareStrings
(
char**string1,
char**string2
)
{
    return -strcmp (*string1, *string2);
}

```


Index

A

- ACS (Auxiliary Coordinate System) 5-152
- ANSI C 1-3
- application 4-1
- assemblies 5-155
- association point 13-1
- attachment functions 26-32
 - defining an attachment 26-32
 - overview 26-32
 - table of functions 26-32
- auxiliary coordinate system functions 5-152
 - attaching specified coordinate system 5-152
 - defining the ACS 5-154
 - deleting specified coordinate system 5-152
 - origin of ACS 5-153
 - rotation matrix of ACS 5-153
 - saving current coordinate system 5-153
 - table of functions 5-152

B

- BASIC interface functions 28-56
 - mdlBasic_getPublicVariable 28-57
 - mdlBasic_setPublicVariable 28-59
 - table of functions 28-56
- Bézier curve 9-60
- Bézier surface 9-60
- binary portability 24-8
- binary portability functions
 - data definition resource to conversion rules 24-9
 - data definition resources 24-9
 - determining proper buffer size 24-14
 - file format to native format data 24-12
 - native format to file format data 24-11
 - overview 24-8
 - table of functions 24-9
- bitmapped color images 18-1
 - mapped images 18-1
 - monochrome images 18-1
 - unmapped images 18-1

- boolean functions 5-81
 - complex shape from difference of input elements 5-82
 - complex shape from intersection of input elements 5-82
 - complex shape from union of input elements 5-81
 - computing region bounded by elements 5-83
 - table of functions 5-81
 - uses of 5-81
- B-spline Construction Functions
 - mdlBspline_computeEqualChordByLength 9-48
 - mdlBspline_cubicInterpolationExt 9-48
 - mdlBspline_parameterFromArcLength 9-49
- B-spline construction functions 9-5, 9-6
 - appending of two B-spline curves 9-14
 - appending of two B-spline surfaces 9-15
 - blending between curves 9-31
 - blending between rail curves 9-33
 - blending between surfaces 9-32
 - boundaries of surface 9-24
- B-splines as end caps of element 9-12
- Coon's patch surface creation 9-19
- copying B-spline curves 9-10
- copying B-spline surfaces 9-11
- creating a curve from poles 9-37
- creating a fillet between surfaces 9-34
- creating a fillet between two surfaces 9-34, 9-35
- creating a hole in surface 9-26
- creating a planar B-spline surface 9-36
- creating an offset curve 9-38
- creating an offset surface 9-39
- creating B-spline curve from element 9-11
- creating B-spline curve helix 9-26
- creating B-spline curve spiral 9-25
- creating B-spline surface from element 9-12
- creating curve from a mathematical expression 9-27
- creating element descriptor containing B-spline curve 9-9

- creating element descriptor containing B-spline surface 9-10
- creating element descriptor from curve 9-21
- creating element descriptor from surface 9-22
- creating skinned surfaces 9-20
- creating surface from a mathematical expression 9-29
- creating surface of projection 9-18
- creating surface of revolution 9-18
- curve chain from element descriptor 9-45
- curve definition function 9-28
- curve from interpolation of points 9-40, 9-42
- curve from set of points 9-15
- curves to generate surface of projection 9-24
- curves to generate surface of revolution 9-24
- element descriptor from curve chain 9-44
- element descriptor from surface chain 9-44
- element descriptor from surface edges 9-43
- intersection points of surfaces 9-46
- portions of curves 9-13
- returning portion of curve 9-13
- silhouette curves 9-23
- surface chain from element descriptor 9-45
- surface definition function 9-30
- surface from interpolation of curves 9-41, 9-42
- surface from interpolation of points 9-40
- surface from set of points 9-16
- surface from two B-spline curves 9-19
- table of functions 9-6
- B-spline functions 9-1
 - uses of 9-1
- B-spline knot functions 9-5, 9-67
 - adding knot value to B-spline curve 9-71
 - adding knot value to B-spline surface 9-72
 - calculating knot vector of B-spline 9-67
 - distinct knots of knot vector 9-69
 - Greville abscissa of knot vector 9-70
 - interval of knot vector 9-71
 - knot tolerance 9-68
 - normalizing knot vectors 9-68
 - number of knots in knot vector 9-73
 - table of functions 9-67
- B-spline modification functions 9-5, 9-50
 - adding curve to existing 9-56
 - adding surface element to existing 9-56
 - closed B-spline curves 9-58
 - closed B-spline surfaces 9-59
 - compatible curves 9-63
 - compatible surfaces 9-64
 - copying boundary elements 9-66
 - curve element information 9-52
 - direction of B-spline curve 9-62
 - direction of B-spline surfaces 9-62
 - increasing order of B-spline curve 9-59
 - increasing order of B-spline surface 9-59
 - memory allocation for B-spline curves 9-53
 - memory allocation for B-spline surfaces 9-54
 - memory deallocation 9-54, 9-55
 - minimizing poles of curve 9-66
 - obtaining Bézier curves 9-60
 - obtaining Bézier surfaces 9-60
 - open B-spline curves 9-58
 - open B-spline surfaces 9-59
 - pole coordinates based on weights 9-57
 - rational B-spline curves 9-61
 - rational B-spline surfaces 9-61
 - surface - ray intersection 9-82
 - surface element information 9-53
 - switching parameter directions of surface 9-64
 - table of functions 9-50
 - weight of poles 9-57
- B-spline query functions 9-5, 9-74
 - boundary points of parameter square 9-84
 - B-spline surface element descriptor 9-86
 - calculating point and partial derivatives on surface 9-81
 - calculating point and tangent on curve 9-80
 - calculating points on B-spline curve 9-78
 - calculating points on B-spline surface 9-79
 - checking for curve closure 9-87
 - checking for surface closure 9-88
 - checking space enclosed by B-spline surface 9-98
 - cuspid points of B-spline curve 9-92
 - degenerate edges 9-98
 - derivatives of surface points 9-82
 - determining if point lies on surface edge 9-97
 - finding closest point on B-spline curve 9-77
 - finding closest point on B-spline surface 9-78
 - Frenet frame calculation 9-76
 - inflection points of B-spline curve 9-93
 - intersection curves of B-spline surfaces 9-91
 - intersection of B-spline curve and segment 9-89
 - intersection points of B-spline curves 9-88
 - intersection points of B-spline surfaces 9-90
 - knot vector check 9-86

- mesh approximating B-spline surface 9-95
 - normal of a plane containing curve 9-83
 - number of derivatives of curve 9-76
 - rule lines 9-94
 - surface - ray intersection 9-81
 - table of functions 9-74
 - trim boundaries of surface 9-85
 - trim boundaries, checking for 9-85
 - trimmed B-spline surfaces 9-94
 - B-splines 9-1
 - BsplineDisplay 9-3
 - BsplineParam 9-3
 - curve types 9-2
 - error codes 9-5
 - manipulation of 9-1
 - MSBsplineCurve structure 9-1
 - MSBsplineSurface structure 9-1
 - rail curves 9-33
 - silhouette curves 9-23
 - storage of 9-1
 - surface types 9-2
 - busy bar window functions 3-140
 - mdlDialog_busyBarStopProcessing 3-142
 - mdlDialog_busyBarUpdateControlParms 3-142
 - mdlDialog_busyBarUpdateMessage 3-143
 - table of functions 3-140
- ## C
- C expression handling functions 2-102
 - creating array type definition 2-107
 - creating pointer type definition 2-106
 - creating structure/union from resource file definition 2-106
 - determining arrays 2-109
 - determining char pointer type 2-110
 - determining structure types 2-109
 - determining union types 2-109
 - evaluating expressions 2-107
 - freeing symbol sets 2-105
 - generating error messages 2-109
 - initializing symbol sets 2-105
 - integer division 2-103
 - publishing symbols 2-104
 - storing specified values 2-108
 - table of functions 2-103
 - uses of 2-102
 - CAD engine 5-1
 - auxiliary coordinate system functions 5-1
 - boolean functions 5-1
 - current transformation functions 5-1
 - element creation functions 5-1
 - element descriptor functions 5-1
 - element information extraction functions 5-1
 - element intersection functions 5-1
 - measurement functions 5-1
 - miscellaneous element functions 5-1
 - Sectioning and hidden line viewing functions 5-1
 - transient element functions 5-1
 - view functions 5-1
 - camera settings 5-106
 - clipping planes 5-107
 - field of vision 5-106
 - lens types 5-106
 - mdlView_getCamera 5-106
 - position 5-106
 - target 5-106
 - caps 12-1
 - cell functions
 - attaching cell libraries 15-2
 - cell definition 15-12
 - cell header 15-9
 - cell header in design file 15-11
 - creating cells in memory 15-10
 - creating new cell in library 15-4
 - element levels 15-9
 - file position cell 15-2
 - index files 15-11
 - indication if cell exists in library 15-3
 - placing cells in design file 15-6
 - point cells 15-9
 - removing cells from library 15-5
 - renaming cells in library 15-5
 - returning cell's element descriptor 15-7
 - table of functions 15-1
 - writing cells to library 15-5
 - cells 15-1
 - cell functions 15-1
 - shared cell functions 15-1
 - clipboard functions 27-45
 - adding data to clipboard 27-49
 - alerting MDL functions of clipboard activity 27-54
 - emptying the clipboard 27-48
 - example application 27-45

- opening Windows clipboard 27-47
- overview 27-45
- programmer's view 27-45
- registering new clipboard format names 27-50
- registering user functions for clipboard
 - events 27-53
- retrieving clipboard data 27-48
- retrieving registered format name 27-48
- table of functions 27-46
- table of user functions 27-47
- user's view 27-45
- clipping boundaries 14-2
- clipping descriptor 7-42
- clipping region 7-42
- cmdName *see command functions reserved words*
- cmdNumber *see command functions reserved words*
- coincident edges 5-119
- color codes 17-1
- color configuration functions 17-28
 - color manager configuration 17-28
 - colors from video hardware 17-29
 - positions of colors in colortable 17-29, 17-30
- color descriptor functions 17-8
 - changing color palette status 17-20
 - color name 17-11
 - draw values 17-12, 17-13
 - menu color 17-15
 - querying color descriptors 17-9, 17-13
 - querying HSV values 17-14
 - RGB triad 17-17
 - RGB value of color descriptor 17-10, 17-16
 - setting from HSV color structure 17-15
 - setting RGB values 17-11, 17-14
- color descriptors 17-3
- color indexes 17-3
- color management functions 17-1
 - color configuration functions 17-1
 - color descriptor functions 17-1
 - color map functions 17-1
 - color palette functions 17-1
 - color table functions 17-1
 - RGB conversion functions 17-1
- color map functions 17-31
 - associating colormaps with colortable 17-31
 - CmapCtbl structure 17-32
 - generating colormaps 17-33
 - obsolescence of color maps 17-31
 - obtaining existing colormap 17-32
 - uses of 17-31
- color maps 17-1
- color palette functions 17-17
 - color descriptor arrays 17-21
 - color palette from colortables 17-22
 - creating color palettes 17-18
 - draw values of 17-20
 - freeing color palettes 17-19
 - number of entries in palette 17-23
 - obtaining color descriptor pointer 17-19
 - status of color palettes 17-20
 - uses of 17-17
- color palettes 17-17
- color picker item functions 3-72
 - color picker information 3-72
 - table of functions 3-72
 - uses of 3-72
- color table functions 17-5
 - active color table 17-6
 - active color table name 17-6
 - attaching color tables 17-5
 - default color table 17-7
 - finding closest RGB 17-7
 - retrieving color table 17-7
- color tables 17-1
- command classifications 1-1
 - immediate commands 1-1
 - primitive commands 1-1
 - utility commands 1-1
 - view commands 1-1
- command functions 1-3
 - command tables 1-3
 - example 1-3
 - reserved words 1-3
- command queueing functions
 - task ID 3-132
- command queuing functions 3-130
 - closing dialog boxes 3-130
 - mdlDialog_cmdNumberQueueByDb 3-134
 - mdlDialog_cmdNumberQueueByTaskId 3-135
 - mdlDialog_cmdNumQByDbExt 3-134
 - mdlDialog_cmdNumQByTaskIdExt 3-135
 - putting commands onto the input queue 3-131
 - table of functions 3-130
- command tables 1-3
- commands 1-1
 - suspension 1-1
 - termination 1-1

- completion bar functions 3-73
 - closing completion bar window 3-74
 - displaying text 3-74
 - mdlDialog_completionBarDisplayMessage 3-76
 - mdlDialog_completionBarOpen 3-76
 - mdlDialog_completionBarUpdate 3-77
 - message box of completion bar 3-75
 - table of functions 3-73
 - updating completion bar 3-75
 - uses of 3-73
- complex chain creation functions 7-40
 - creating complex chain header element 7-40
 - end of complex chain 7-41
 - file position of complex chain header 7-41
 - table of functions 7-40
 - uses of 7-40
- complex elements 5-45
 - examples of 5-45
- configuration variable files 4-38
 - creation of 4-38
- configuration variable functions 4-38
 - uses of 4-38
- configuration variables
 - obtaining values of 4-38
- configuration variables dialog 4-38
- ConsEquation structure 26-2
- ConsModel structure 26-2
- constraint
 - constraint data structure 26-2
- constraint equation functions
 - adding objects to model 26-37
- constraint functions 26-1, 26-11
 - fixing angle between line-like construction frames 26-19
 - fixing angle of construction frame line 26-20
 - fixing diameter of circle 26-14
 - fixing diameters of ellipse 26-14
 - fixing distance between points 26-17
 - fixing location of target 26-13
 - fixing normal distance between point and curve 26-17
 - fixing parallel distance between constraints 26-18
 - fixing radius of circle 26-14
 - fixing radius of ellipse 26-14
 - fixing value of a solver variable 26-25
 - forcing a point to lie on curve 26-15
 - forcing coincident target points 26-15
 - forcing constraint parameter to equal dimensional constraint 26-12
 - forcing curve to intersect point 26-24
 - forcing curves to be tangent 26-23
 - forcing geometric attributes of frames to match 26-25
 - forcing parallel line-like construction frames 26-21
 - forcing perpendicular line-like construction frames 26-21
 - forcing point to lie at curves intersection 26-16
 - forcing target circle to be tangent to curve 26-22
 - freezing geometry attributes of construction frame 26-24
 - inverting sense of IOTangent offset 26-26
 - inverting sense of LLDistance constraint 26-26
 - modifying constraint's sense 26-27
 - overview 26-1, 26-11
 - redundant constraints 26-25
 - retrieving specified target 26-26
 - setting post-check attribute of constraint 26-28
 - setting weak attribute of constraint 26-27
 - specific terms 26-11
 - table of functions 26-11
 - uses of 26-1, 26-11
- constraint model functions 26-33
 - adding constraint to model 26-36
 - applying function to constraint model object 26-40
 - defining empty constraint 26-35
 - destroying constraints 26-35
 - determining degrees of freedom 26-38
 - identifying constraint problem solutions 26-40
 - memory deallocation 26-35
 - modifying attribute variables 26-37
 - modifying object geometry 26-42
 - overview 26-33
 - redundant constraints 26-38
 - removing object from constraint model 26-36
 - restoring values of attribute variables 26-39
 - saving attribute variables 26-39
 - searching for constraint model object 26-41
 - table of functions 26-34
 - transforming geometry of constraint model objects 26-41
 - translating constraint model objects 26-42
 - under-determined variables 26-38
- constraint object functions 26-43

- coordinates of object location 26-44
 - determining fully defined objects 26-46
 - freeing memory of object 26-45
 - overview 26-43
 - querying object attribute variables 26-43
 - retrieving attribute variable of object 26-46
 - rotation matrix of object geometry 26-44
 - table of functions 26-43
 - constraint parameter functions 26-9
 - algebraic constraints 26-9
 - changing units of constraint parameter 26-10
 - defining a constraint parameter 26-10
 - overview 26-9
 - table of functions 26-9
 - constraint problems 26-1
 - constraints 26-1
 - analyzing constraint problems 26-3
 - constraint model data structure 26-2
 - defined 26-1
 - defining constraint problems 26-3
 - equation constraint 26-2
 - relationship to MicroStation elements 26-1
 - solver variables 26-2
 - solving constraint problems 26-3
 - construction frame functions
 - defining a B-spline cell construction frame 26-8
 - defining a B-spline construction frame 26-7
 - defining a construction circle 26-7
 - defining a construction ellipse 26-6
 - defining a construction line 26-5
 - defining construction frame point 26-5
 - overview 26-4
 - querying geometric attributes 26-9
 - table of functions 26-5
 - content rectangle 2-12
 - conversion functions 24-1
 - ASCII to Radix50 string conversion 24-3
 - determining rational fractions 24-4
 - double to float conversion 24-7
 - double to long conversion 24-7
 - double to unsigned long conversion 24-7
 - Dpoint3d to Point3d conversion 24-2
 - float to double conversion 24-7
 - internal to file format conversion 24-5
 - internal to scan format conversion 24-4
 - master unit to UOR conversion 24-8
 - MIDDLE_ENDIAN long int to native long int conversion 24-6
 - overview 24-1
 - Point3d to Dpoint3d conversion 24-2
 - Radix50 to ASCII string conversion 24-3
 - scan to internal format conversion 24-4
 - table of functions 24-1
 - UOR to master units conversion 24-8
 - Coon's patch 9-43
 - coordinate system conversions 20-1
 - creating curve from surface edge 9-37
 - current transformation 5-155
 - current transformation functions 5-155
 - converting from current coordinates to .dgn coordinates 5-160
 - pushing current transformation matrix 5-157
 - replacing current transformation matrix with identity matrix 5-157
 - scaling current matrix 5-160
 - table of functions 5-155
 - translating origin of current matrix 5-159
 - current transformation functions
 - rotating current transformation matrix 5-158
 - cursor 7-19
 - element location cursor 7-19
 - MicroStation cursor 7-19
 - curve *see constraint functions, specific terms*
- ## D
- data conversion 24-1
 - binary portability functions 24-1
 - conversion functions 24-1
 - data definition resource 6-1
 - automatic data conversion 6-1
 - generation of 6-1
 - data definition resources
 - see binary portability functions*
 - Database Interface Functions 21-17
 - mdlDB_findLinks 21-17
 - mdlDB_freeSimpleSelectResultXbase 21-18
 - mdlDB_getMscatalogName 21-18
 - mdlDB_setLinkCacheInfo 21-19
 - mdlDB_simpleSelectXbase 21-19
 - database interface functions 21-2
 - building displayable attribute linkages 21-14
 - closing report tables 21-6
 - constructing user data linkage 21-9
 - converting linkage from internal format 21-8
 - defining new active entities 21-15

- displaying active entity 21-16
- editing active entity fields 21-16
- establishing new fence filters 21-12
- extracting linkage parameters from database linkages 21-8
- initializing report tables in database 21-6
- linking complex elements with active entity 21-4
- linking graphic elements with active entity 21-4
- loading displayable attributes 21-13
- processing database linkages 21-7
- removing attribute linkages from complex elements 21-5
- removing attribute linkages from input element 21-5
- returning highest MSLINK of table 21-10
- reviewing attribute linkages on an element 21-7
- running database screen form 21-9
- testing for attribute data linkages 21-12
- database settings functions 21-32
 - attaching database to DGN file 21-34
 - controlling transaction capable databases 21-33
 - defining report tables 21-36
 - deleting database rows 21-34
 - setting database linkage mode 21-35
 - setting displayable attribute type 21-33
 - specifying SQL statements 21-37
 - table of functions 21-32
 - using screen forms for editing 21-35
- dBase database 21-1
- DBForms Functions
 - mdlDBDialog_attachCurrentRow 21-38
 - mdlDBDialog_clearValues 21-38
 - mdlDBDialog_deleteCurrentRow 21-38
 - mdlDBDialog_detachCurrentRow 21-38
 - mdlDBDialog_firstRow 21-38
 - mdlDBDialog_generateRscFile 21-38
 - mdlDBDialog_insertRow 21-38
 - mdlDBDialog_itemGetState 21-38
 - mdlDBDialog_itemSetState 21-38
 - mdlDBDialog_lastRow 21-38
 - mdlDBDialog_locateCurrentRow 21-38
 - mdlDBDialog_nextPage 21-38
 - mdlDBDialog_nextRow 21-38
 - mdlDBDialog_openForm 21-38
 - mdlDBDialog_openFormFromElement 21-39
 - mdlDBDialog_prevPage 21-39
 - mdlDBDialog_prevRow 21-39
 - mdlDBDialog_processUserQuery 21-39
 - mdlDBDialog_publishExtraHooks 21-39
 - mdlDBDialog_queryRscFile 21-39
 - mdlDBDialog_review 21-39
 - mdlDBDialog_setPage 21-39
 - mdlDBDialog_startQuery 21-39
 - mdlDBDialog_updateCurrentRow 21-39
 - mdlDBDialog_useFenceIfActive 21-39
 - mdlDialog_openWithDBQuery 21-39
- DDE (Dynamic Data Exchange)
 - conversation 27-1
 - data handles 27-1
 - string handles 27-1
- DDEML (Dynamic Data Exchange Management Library)
 - ddeclkcnc.mc 27-2
 - example programs 27-2
 - ntcbtext.mc 27-2
- design files
 - rendering 18-1
- dialog box drawing functions 3-112
 - beveled edge diamond, drawing 3-113
 - characteristics of graphics, setting 3-115
 - converting coordinate units to pixel value 3-112
 - converting pixel value to coordinate units 3-113
 - ellipses, drawing 3-114
 - scroll bar-like arrows, drawing 3-116
 - strings, determining pixel width 3-116
 - table of functions 3-112
 - text, specifying colors 3-118
 - underlining text 3-119, 3-120
- dialog box font functions 3-127
 - font height, determining 3-128
 - font index 3-129
 - font information, retrieving 3-128
 - table of functions 3-127
- dialog box general functions 3-78, 3-112
 - action type, setting 3-90
 - alert dialog boxes, opening 3-82
 - automatic opening of dialogs upon startup 3-85
 - bringing dialog box into view 3-86
 - determining input focus 3-87
 - dialog box pointer from resourceclass 3-87
 - file filter string, setting 3-94
 - hook functions, ID numbers and addresses 3-90
 - hook functions, sending user messages to 3-89
 - modal dialog boxes, opening 3-79, 3-80, 3-81, 3-82
 - modeless dialog boxes, opening 3-79, 3-81

- obtaining pointer to command window 3-88
- obtaining resource ID of parent dialog box 3-88
- obtaining task descriptor 3-87
- obtaining user data pointer 3-89
- opening dialog box without displaying 3-85
- pointer to open dialog box 3-86
- publishing variables 3-91
- synchronization 3-94
- synonym resources 3-94
- table of functions 3-78
- uses of 3-78, 3-112
- Dialog Box Interface Functions
 - mdlDialog_trackBarStopProcessing 3-138
 - mdlDialog_trackBarUpdateControlParms 3-138
 - mdlDialog_trackBarUpdateDisplayInfo 3-139
- dialog box item functions 3-95
 - color of dialog box item 3-111
 - dialog box item, hiding 3-100
 - dialog box item, loading 3-101
 - dialog box item, redisplaying 3-101
 - dialog box item, redrawing 3-99
 - extent of item, setting 3-104
 - external state of dialog box, retrieving 3-96
 - external state, determining item
 - appearance 3-106, 3-107
 - external state, setting 3-105
 - freeing dialog box items 3-109
 - index of dialog box item, retrieving 3-110
 - internal value of dialog box, retrieving 3-96
 - internal value, setting external state from 3-105
 - label of item, setting 3-104
 - loading dialog box items 3-109
 - moving 3-102
 - number of items in dialog box,
 - determining 3-108
 - pointer to dialog item, retrieving 3-97
 - setting state, enable or disable 3-103
 - swapping position of items with dialog
 - box 3-103
 - table of functions 3-95
 - uses of 3-95
- dialog box manager
 - busy bar window functions 3-2
 - color picker item functions 3-1
 - command queuing functions 3-1
 - completion bar item functions 3-1
 - dialog box drawing functions 3-1
 - dialog box font functions 3-1
 - dialog box general functions 3-1
 - dialog box item functions 3-1
 - icon functions 3-1
 - level map item functions 3-1
 - list box item functions 3-1
 - menu bar item functions 3-1
 - miscellaneous dialog box functions 3-2
 - option button item functions 3-1
 - option pull-down menu functions 3-1
 - push button item functions 3-1
 - rectangle drawing functions 3-1
 - scroll bar item functions 3-1
 - text item functions 3-1
 - text pull-down menu functions 3-1
 - toggle button item functions 3-1
 - track bar window functions 3-1
- dialog box manager functions
 - command number functions 3-2
- dialog font functions
 - uses of 3-127
- digitizer functions 23-1
 - calling userDigitize_getTabletPoint 23-3
 - interpreting input coordinates 23-2
 - table of functions 23-1
- dimension element 10-1
 - options section 10-1
 - parameter section 10-1
- dimension elements
 - text 10-1
- dimension functions 10-1
 - converting to IGDS 8.8 primitives 10-12
 - creating dimension elements 10-4
 - defining dimension rotation matrix 10-5
 - deleting points 10-9
 - dimension element 10-1
 - dimension types and requirements 10-2
 - extracting height value from dimension 10-7
 - extracting points 10-10
 - extracting rotation matrix 10-6
 - inserting points 10-9
 - modifying dimension text location 10-8
 - replacing dimension strings 10-11
 - retrieving values from dimension elements 10-4
 - returning dimension text strings 10-10
 - setting dimension range block 10-12
 - setting height of dimension 10-7
 - table of functions 10-3
 - uses of 10-1

- DOS specifics, external programs 22-4
 - extprg_messageReceive 22-4
 - extprg_messageSend 22-4
 - mdlExternal_messageSend 22-4
 - mdlExternal_startMCSLProgram 22-4
 - mdlExternal_startProgram 22-4
 - message queues 22-4
 - protected mode 22-4
 - shared memory 22-4
 - dynamic array functions 28-36
 - copies a dynamic array 28-37
 - creating a dynamic array 28-37
 - deallocating memory of a dynamic array 28-38
 - dynamic array indexing 28-38
 - inserting members in a dynamic array 28-39
 - moving members in dynamic array 28-40
 - number of members in dynamic array 28-40
 - overview 28-36
 - pointer passing 28-40
 - removing members from a dynamic array 28-37, 28-41
 - dynamic buffer functions 7-45
 - loading dynamic buffer with element 7-46
 - loading dynamic buffer with element descriptor 7-45
 - table of functions 7-45
 - uses of 7-45
 - dynamics 1-2
 - complex dynamics 1-2
 - defined 1-2
 - simple dynamics 1-2
- ## E
- element association functions 13-1
 - 3D data points 13-8
 - adding tag to elements 13-9
 - arc elements 13-2
 - AssocPoint 13-1
 - B-spline curves 13-7
 - checking elements for tags 13-8
 - complex elements 13-9
 - creating associations 13-1, 13-2, 13-3
 - current associative point 13-10
 - ellipse elements 13-2
 - intersecting elements 13-3
 - linear elements 13-3
 - locating elements by ID 13-7
 - multi-line elements 13-4, 13-5
 - origin of base elements 13-6
 - table of functions 13-1
 - uses of 13-1
 - element clipping functions 7-42
 - clipping descriptor 7-42
 - clipping elements 7-42
 - deallocating memory 7-43
 - descriptor of current active fence 7-43
 - descriptor of reference file boundary 7-44
 - determining if element is within clipping descriptor 7-44
 - table of functions 7-42
 - uses of 7-42
 - element creation functions 5-2
 - arguments 5-2
 - creating arc elements 5-7
 - creating cell header element 5-16
 - creating circular ellipse elements 5-9
 - creating cone elements 5-14, 5-15
 - creating curve elements 5-4
 - creating cylinder element 5-15
 - creating ellipse elements 5-8
 - creating line string element 5-4
 - creating MicroStation line element 5-3, 5-5
 - creating point string element 5-16
 - creating shape elements 5-4
 - creating text elements 5-10
 - creating text node element with text strings 5-13
 - creating text node elements 5-12
 - in parameter 5-3
 - out parameter 5-3
 - table of functions 5-2
 - uses of 5-2
 - element descriptor functions 5-45
 - adding element descriptor chain 5-65
 - adding element descriptors to .dgn file 5-50
 - allocating new element descriptor 5-63
 - attribute information, appending 5-74
 - attribute information, extracting 5-75
 - attribute information, removing 5-74
 - compared with element functions 5-45
 - converting 2D to 3D element descriptor 5-70
 - converting 3D to 2D element descriptor 5-69
 - converting closed to open element 5-73
 - converting compound to simple elements 5-71
 - copying element descriptors 5-59
 - creating element descriptors 5-49

- creating line string 5-66
- deallocating memory 5-69
- deleting elements 5-51
- determining closed elements 5-72
- determining open elements 5-72
- displaying element descriptor 5-53, 5-54
- displaying graphics in windows 5-77
- endpoints of open elements 5-62
- file size of elements in element descriptor 5-57
- file size of single element descriptor 5-58
- fill attribute linkage 5-68
- generating portion of element descriptor 5-67
- hidden lines 5-68
- hole elements 5-67
- marking element in element descriptor 5-58
- memory considerations 5-45
- normal vector of planar elements 5-76
- overwriting elements 5-52
- removing element from element descriptor chain 5-64
- replacing element descriptor 5-65
- reversing normal direction of surface 5-61
- stroking element descriptor 5-55
- table of functions 5-46
- transforming element descriptors 5-51
- uses of 5-45
- element information extraction functions 5-17
 - ASCII strings of text elements 5-21
 - bounding boxes 5-21, 5-23
 - extracting an array of coordinates 5-18
 - extracting information from arc element 5-19
 - extracting information from cell header elements 5-24
 - extracting information from cone elements 5-23
 - extracting information from ellipse 5-19
 - extracting information from shared cell definition element 5-25
 - extracting information from shared cell instance element 5-25
 - extracting information from text elements 5-20
 - extracting information from text node elements 5-22
 - finding line segment of linear element 5-18
 - table of functions 5-17
 - uses of 5-17
- element intersection functions 5-93
 - intersections between two elements 5-94
 - routines 5-93
- table of functions 5-93
- uses of 5-93
- element linkage functions 6-1
 - appending attribute linkage(s) to element(s) in descriptor 6-9
 - appending user attribute linkage to element 6-5
 - deleting linkage(s) from element 6-4
 - deleting linkage(s) from element(s) in descriptor 6-8
 - extracting linkage(s) from element 6-2
 - extracting linkage(s) from element(s) in descriptor 6-7
 - table of functions 6-2
 - uses of 6-1
- element location and manipulation
 - complex chain creation 7-1
 - dynamic buffer functions 7-1
 - element clipping functions 7-1
 - element location functions 7-1
 - element modification functions 7-1
 - fence processing functions 7-1
 - scan functions 7-1
 - selection set processing 7-1
 - surface creation functions 7-1
- element location cursor 7-19
- element location functions 7-11
 - designating user functions 7-19
 - element location process 7-20
 - locating additional elements 7-16
 - locating elements for modification 7-15
 - low-level element searching 7-17
 - modifying location search masks 7-13
 - return information from mdlLocate_findElement 7-18
 - setting bits in element type search mask 7-14
 - setting MicroStation cursor to element location cursor 7-19
 - starting search at beginning of design file 7-16
 - starting search at end of design file 7-16
 - table of functions 7-12
 - use with mdlState_startModifyCommand 7-12
 - uses of 7-11
- element modification functions
 - deallocating memory 7-29
 - modifying element descriptors 7-28
 - processing graphic groups 7-27
 - processing selection sets 7-27
 - reading element from design file 7-24

- reading elements from design file 7-24
 - table of functions 7-23
- element modification functions 7-23
 - uses of 7-23
- equation constraint 26-2
- equation constraint functions 26-29
 - activating equation constraints 26-31
 - constraint parameter objects 26-31
 - creating an equation 26-29
 - detecting syntax errors 26-30
 - initializing constraint equation data structure 26-30
 - overview 26-29
 - table of functions 26-30
- Exact hidden line removal enhancements
 - advantages of 5-119
 - mdlElmdscr_hiddenLineRemoval2 5-119
 - mdlView_hiddenLineRemoval2 5-119
- example MDL applications 1-2, 2-96
 - bspline.mc 9-9, 9-52, 9-67, 9-75
 - cell.mc 15-2
 - cexpr.mc 2-103
 - CHNGTXT 1-2
 - create.mc 5-3, 28-61
 - database.mc 21-4, 21-21, 21-33
 - ddeclkcن.mc 27-2
 - dimline.mc 10-4
 - dynamic.mc 7-45, 20-3
 - elemdscr.mc 5-46, 5-48
 - element.mc 5-28
 - excfgvar.mc 4-39
 - extract.mc 28-61
 - fence.mc 7-34
 - file.mc 25-2
 - grphtest.mc 2-3, 2-21
 - input.mc 2-75
 - locate.mc 7-13
 - lvlnames.mc 19-10
 - math.mc 20-13, 20-20
 - misc.mc 28-73
 - mline.mc 12-2
 - ntcbtext.mc 27-2, 27-45
 - params.mc 19-1
 - parse.mc 2-68
 - patnmdl.mc 16-1
 - plaimage 18-25
 - PLASHAPE 1-3
 - resmover.mc 2-41
 - select.mc 7-30
 - system.mc 4-5
 - tangcons 26-4
 - trumpet 5-157, 7-37, 20-13
 - tutorial.mc 28-52
 - view.mc 5-100
- external program communication functions 22-1
 - accessing shared memory 22-21
 - attaching message queues 22-18
 - creating MicroCSL sessions 22-13
 - detaching from message queue 22-19
 - detaching shared memory 22-21
 - examining message queues 22-21
 - initializing for external programs 22-22
 - obtaining message queue 22-14
 - obtaining shared memory region 22-16
 - overview 22-1
 - platform shutdown for external programs 22-22
 - receiving messages 22-15, 22-20, 22-23
 - removing message queues 22-16
 - removing shared memory region 22-17
 - returning external program arguments 22-22
 - sending messages 22-14, 22-19
 - sending signals 22-17
 - starting external programs 22-11, 22-13
 - starting real mode programs 22-12
 - table of functions 22-1
 - terminating external programs 22-10
 - user functions 22-18
 - waiting for external programs to terminate 22-17
- external programs 22-1
 - DOS specific concerns
 - see DOS specifics, external programs*
 - table of functions 22-1, 22-2
 - UNIX concerns
 - see UNIX, external program concerns*
 - Windows NT concerns *see Windows NT, external program concerns*
- external state of dialog box item 3-96
- extprg_getCommandLineArguments 22-22
- extprg_initializePlatform 22-22
- extprg_messageReceive 22-4, 22-20
- extprg_messageSend 22-4, 22-19
- extprg_queueAttach 22-18
- extprg_queueDetach 22-19
- extprg_shmemAttach 22-21
- extprg_shmemDetach 22-21
- extprg_shutdownPlatform 22-22

extprg_signalUstn 22-21

F

fast font 8-1

Fence Processing Functions 7-37

mdlFence_clear 7-37

fence processing functions 7-34

table of functions 7-34

uses of 7-34

fences 7-34

file drag and drop utility functions 27-43

designating drag and drop functions 27-44

overview 27-43

registering user functions 27-43

table of functions 27-43

file list functions 25-18

choosing a file from dialog box 25-25

choosing seed files 25-24

creating a file 25-22

creating list of files 25-20

opening a file 25-22

overview 25-18

searching for files 25-27

table of functions 25-19

file manipulation 25-1

file list functions 25-1

file utility functions 25-1

low level I/O support 25-1

text file functions 25-1

work file functions 25-1

File Utility Functions 25-12

mdlFile_isUntitledDesign 25-12

file utility functions 25-1

abbreviating filename 25-12

checking for 2D design files 25-10

checking for 3D design files 25-10

checking for design files 25-10

copying contents of file 25-11

creating a directory 25-5

creating filename 25-4

creating files 25-2

current default drive number 25-7

determining free space 25-6

determining if drive exists 25-6

establishing default drive for file system 25-6

finding file based on name 25-2

obtaining attributes of file 25-9

overview 25-1

parsing filenames 25-4

setting file type 25-10

storing name of working directory 25-5

table of functions 25-1

float data type, MDL support of 24-7

Floyd-Steinberg filter 18-18

fonts

loading 8-1

FoxPro database 21-1

function key functions 28-44

adding to function key definition list 28-46

attaching function key menus 28-44

file specification of attached function key
menu 28-45

number of function keys available 28-45

obtaining button definition 28-45, 28-46

obtaining button name 28-45, 28-46

obtaining function key index 28-46

obtaining virtual key code 28-45

overview 28-44

removing from function key definition list 28-47

saving function key menus 28-44

H

hatching 16-3

help functions 28-42

designating MDL function calls 28-43

opening MicroStation help window 28-42

overview 28-42

hidden line removal functions 5-117

invisibleFunction 5-117

preFunction 5-117

visibleFunction 5-117

HSV (Hue-Saturation Value) 17-26

HTML Authoring Tool Library 29-1

mdlHTMMLib_addCaption 29-1

mdlHTMMLib_addCitationText 29-1

mdlHTMMLib_addDefinitionListItem 29-1

mdlHTMMLib_addEmphasizedText 29-1

mdlHTMMLib_addFormItemItem 29-1

mdlHTMMLib_addFormCheckBoxRadio
ButtonItem 29-1

mdlHTMMLib_addFormHiddenItem 29-1

mdlHTMMLib_addFormImageItem 29-1

mdlHTMMLib_addFormSelectOptionItem 29-1

mdlHTMMLib_addFormTextItem 29-1

mdlHTMMLib_addFrame 29-1	mdlHTMMLib_endTableCell 29-3
mdlHTMMLib_addFrameset 29-1	mdlHTMMLib_endTableRow 29-3
mdlHTMMLib_addHeading 29-1	mdlHTMMLib_endUnorderedList 29-3
mdlHTMMLib_addHorizontalRuledLine 29-1	mdlHTMMLib_hrefListFromStringList 29-3
mdlHTMMLib_addHREFAnchor 29-1	mdlHTMMLib_htmlListFromStringList 29-4
mdlHTMMLib_addImage 29-2	mdlHTMMLib_imageTableFromList 29-4
mdlHTMMLib_addKeyboardSampleText 29-2	mdlHTMMLib_openFile 29-4
mdlHTMMLib_addLineBreak 29-2	mdlHTMMLib_setDocTypeToHTML 29-4
mdlHTMMLib_addListItem 29-2	mdlHTMMLib_setMetaNameGenerator 29-4
mdlHTMMLib_addListItemOpt 29-2	mdlHTMMLib setPageTitle 29-4
mdlHTMMLib_addMapArea 29-2	mdlHTMMLib_startBody 29-4
mdlHTMMLib_addNameAnchor 29-2	mdlHTMMLib_startCenter 29-4
mdlHTMMLib_addNewLine 29-2	mdlHTMMLib_startDefinitionList 29-4
mdlHTMMLib_addNoFrameMessage 29-2	mdlHTMMLib_startDirectoryList 29-4
mdlHTMMLib_addPlainText 29-2	mdlHTMMLib_startFileHeader 29-4
mdlHTMMLib_addProtocolToString 29-2	mdlHTMMLib_startForm 29-4
mdlHTMMLib_addProtocolToStringPath 29-2	mdlHTMMLib_startHREFAnchor 29-4
mdlHTMMLib_addSourceCodeText 29-2	mdlHTMMLib_startHTML 29-4
mdlHTMMLib_addStartFormSelectItem 29-2	mdlHTMMLib_startMap 29-4
mdlHTMMLib_addStrongText 29-2	mdlHTMMLib_startMenuList 29-4
mdlHTMMLib_addTableCell 29-2	mdlHTMMLib_startNameAnchor 29-4
mdlHTMMLib_addTableHeaderRow 29-2	mdlHTMMLib_startNoFrame 29-4
mdlHTMMLib_addTextAreaItem 29-2	mdlHTMMLib_startOrderedList 29-4
mdlHTMMLib_addTextString 29-2	mdlHTMMLib_startParagraph 29-4
mdlHTMMLib_addVariableName 29-2	mdlHTMMLib_startStandardTable 29-5
mdlHTMMLib_beginFrameDocument 29-2	mdlHTMMLib_startTable 29-5
mdlHTMMLib_beginHTMLDocument 29-2	mdlHTMMLib_startTableCell 29-5
mdlHTMMLib_closeFile 29-2	mdlHTMMLib_startTableRow 29-5
mdlHTMMLib_closeFrame 29-2	mdlHTMMLib_startTableRowAlign 29-5
mdlHTMMLib_createFile 29-2	mdlHTMMLib_startUnorderedList 29-5
mdlHTMMLib_endAnchor 29-2	mdlHTMMLib_textHREFTableFromList 29-5
mdlHTMMLib_endBody 29-2	mdlHTMMLib_textTableFromList 29-5
mdlHTMMLib_endCenter 29-3	mdlTag_attachURL 29-5
mdlHTMMLib_endDefinitionList 29-3	mdlTag_createInternetTagSet 29-5
mdlHTMMLib_endDirectoryList 29-3	mdlTag_extractURL 29-5
mdlHTMMLib_endFileHeader 29-3	mdlTag_hasInternetTagSet 29-5
mdlHTMMLib_endForm 29-3	mdlWeb_getLastModified 29-5
mdlHTMMLib_endFormSelectItem 29-3	mdlWeb_getUrlFinish 29-5
mdlHTMMLib_endFrameDocument 29-3	mdlWeb_getUrlProgress 29-5
mdlHTMMLib_endFrameset 29-3	mdlWeb_getUrlToFileBegin 29-5
mdlHTMMLib_endHTML 29-3	mdlWeb_stopBrowser 29-5
mdlHTMMLib_endHTMLDocument 29-3	
mdlHTMMLib_endMap 29-3	
mdlHTMMLib_endMenuList 29-3	
mdlHTMMLib_endNoFrame 29-3	
mdlHTMMLib_endOrderedList 29-3	
mdlHTMMLib_endParagraph 29-3	
mdlHTMMLib_endTable 29-3	

I

- icon frame functions 3-35
 - uses of 3-35
- icon frams 3-35
- icon functions 3-35

- number of icons in icon command palette 3-41
- number of items in icon command frame 3-36
- number of sub-items in icon palette 3-37
- return information 3-36
- selection state of icon 3-37, 3-38, 3-39
- selection status of icon 3-39
- setting icon command resource 3-40
- table of functions 3-35
- uses of 3-35
- icon palettes 3-35
- image file formats 18-6
 - example applications 18-9
 - internal image formats
 - see internal image formats*
- Imaging and Rendering Functions
 - mdlImage_runLengthDecodeBitMapRow 18-69
- immediate commands 1-1
 - examples of 1-1
 - uses of 1-1
- input handling functions 2-73
 - command filter user function 2-94
 - copying last queue element 2-87
 - creating CMDNUM queue elements 2-81
 - creating data point queue element 2-87
 - creating KEYIN queue elements 2-82
 - creating NULL queue elements 2-88
 - creating RESET queue elements 2-86
 - designating user functions 2-92
 - determining if mouse is being used 2-90
 - determining if tablet is being used 2-90
 - determining primitive commands 2-89
 - determining view commands 2-89
 - disabling command class 2-90
 - enabling command class 2-90
 - input monitor user function 2-93
 - pausing MicroStation tasks 2-89
 - placing elements at head of queue 2-86
 - placing keystrokes into input queue 2-85
 - preprocessing key-ins 2-92
 - receive user function 2-94
 - relinquishing active command status 2-89
 - requesting active command status 2-89
 - saving contents of jump buffer 2-91
 - sending last queue element to MicroStation 2-81
 - suspending MDL tasks 2-88
 - table of functions 2-73
 - uses of 2-73
- input processing and sequencing 2-75

- MicroStation input loop 2-75
- internal image formats
 - IMAGEFORMAT_BitMap 18-9
 - IMAGEFORMAT_ByteMap 18-9
 - IMAGEFORMAT_GreyScale 18-9
 - IMAGEFORMAT_PackByte 18-10
 - IMAGEFORMAT_RGB 18-10
 - IMAGEFORMAT_RGBA 18-10
 - IMAGEFORMAT_RGBSeparate 18-10
 - IMAGEFORMAT_RLEBitMap 18-9
 - IMAGEFORMAT_RLEByteMap 18-10
- internal value of dialog box item 3-96
- internationalization 28-61
- invisibleFunction 5-117

J

- jiffies 18-29
- JPEG (Joint Photographic Experts Group) 18-6

L

- Level Functions
 - mdlLevelSymbology_extract 19-29
- level functions
 - adding level groups to level structure 19-15
 - adding levels to level structure 19-13
 - allocating memory 19-16, 19-18
 - deleting levels from level structure 19-13
 - external resource files 19-25
 - freeing memory 19-18, 19-19, 19-23, 19-24
 - identifying level groups by group ID 19-14
 - level path specification 19-19
 - modifying existing level group names 19-16
 - obtaining level group 19-21
 - obtaining level group index 19-21
 - obtaining level index 19-20
 - obtaining level masks 19-10, 19-11
 - removing level groups from level
 - structure 19-15
 - retrieving attributes of level name 19-14
 - saving level structure 19-26
- level map item functions 3-71
 - level access strings, retrieving 3-71
 - level access strings, setting 3-71
 - table of functions 3-71
 - uses of 3-71
- license management functions 28-32

- mdlLicense_getHardwareKeyPermission 28-33
 - mdlLicense_getInitialHardwareKeySerial 28-33
 - mdlLicense_getStrings 28-33
 - mdlLicense_getVarietyName 28-34
 - mdlLicense_isAcademicProduct 28-34
 - mdlLicense_isHardwareKeyRequired 28-35
 - mdlLicense_isHardwareKeySupported 28-35
 - mdlLicense_isRegisteredProduct 28-35
 - mdlLicense_isSerializedProduct 28-35
 - table of functions 28-32
 - line definition 12-1
 - line style definition 11-1
 - application interface 11-3
 - line style engine 11-3
 - line style functions 11-1, 11-5
 - converting element to individual primitives 11-11
 - creating string list from resource name map 11-10
 - freeing memory 11-13
 - inserting entries in line style name 11-6
 - inserting line style component into cache 11-11
 - loading line style name maps 11-9
 - modifying information in design file name maps 11-7
 - modifying information in resource file maps 11-7
 - removing entries from line style name 11-6
 - removing line style component from cache 11-12
 - returning resource info line style name map 11-8
 - storing line style name maps 11-9
 - table of functions 11-5
 - line style names 11-2
 - line style resource definitions 11-4
 - line style resource files 11-2
 - line-like *see constraint functions, specific terms*
 - list box item functions 3-41
 - array of selected cells 3-47
 - cells in list box, determining selection 3-47
 - column parameters, retrieving 3-43
 - column parameters, setting 3-43
 - currently selected cells in list box 3-54
 - deleting columns from list box 3-43
 - disabling range of cells in list box 3-53
 - drawing contents of list box item 3-52
 - enabling range of cells in list box 3-53
 - inserting columns in list box 3-48
 - list box parameters, retrieving 3-44
 - list box parameters, setting 3-44
 - list box, number of columns 3-46
 - range of displayable cells 3-51
 - searching for next selected cell in list box 3-54
 - selecting range of cells in list box 3-53
 - selection cursor location 3-45
 - selection cursor location, setting 3-46
 - setting first row in list box 3-49
 - string lists 3-41, 3-49, 3-51
 - table of functions 3-41
 - uses of 3-41
- M**
- mapped images 18-1
 - material table functions 18-82
 - allocating space 18-83
 - freeing memory 18-83
 - reading material tables 18-84
 - rendering 3D MicroStation files 18-84
 - uses of 18-82
 - material tables 18-1, 18-82
 - math functions 20-1
 - rotation matrix functions 20-1
 - transformation matrix functions 20-1
 - vector functions 20-1
 - matrices 20-1
 - MDL application 4-1
 - MDL clipboard operations 27-2
 - clipboard programming guide 27-4
 - newClipboardData structure 27-4
 - pasting 27-2
 - processing example 27-3
 - MDL program 4-1
 - MDL runtime environment 4-1
 - MDL task 4-1
 - mdlACS_attachNamed 5-152
 - mdlACS_createElement 5-154
 - mdlACS_deleteNamed 5-152
 - mdlACS_extractElement 5-154
 - mdlACS_getCurrent 5-153
 - mdlACS_saveNamed 5-153
 - mdlACS_setCurrent 5-154
 - mdlArc_create 5-7
 - mdlArc_createByCenter 5-7
 - mdlArc_createByPoints 5-7

mdlArc_extract 5-19
 mdlAssoc_createArc 13-2
 mdlAssoc_createBCurve 13-7
 mdlAssoc_createIntersection 13-3
 mdlAssoc_createKeypoint 13-3
 mdlAssoc_createLinear 13-4
 mdlAssoc_createMline 13-5
 mdlAssoc_createOrigin 13-6
 mdlAssoc_getCurrent 13-10
 mdlAssoc_getElement 13-7
 mdlAssoc_getPoint 13-8
 mdlAssoc_isTagged 13-8
 mdlAssoc_tagElement 13-9
 mdlAssoc_tagElementComplex 13-9
 mdlBasic_getMacroInfo 28-56
 mdlBasic_getPublicVariable 28-57
 mdlBasic_setPublicVariable 28-59
 mdlBspline_addCurveLink 9-56
 mdlBspline_addKnot 9-71
 mdlBspline_addKnotSurface 9-72
 mdlBspline_addSurfaceLink 9-56
 mdlBspline_allBoresiteToSurface 9-82
 mdlBspline_allocateCurve 9-53
 mdlBspline_allocateSurface 9-54
 mdlBspline_appendCurves 9-14
 mdlBspline_appendSurfaces 9-15
 mdlBspline_approximateCurve 9-27
 mdlBspline_approximateSurface 9-29
 mdlBspline_arcLengthFromParameters 9-47
 mdlBspline_blendCurve 9-31
 mdlBspline_blendRails 9-33
 mdlBspline_blendSurface 9-32
 mdlBspline_boresiteToSurface 9-81
 mdlBspline_catmullRomCurve 9-40
 mdlBspline_catmullRomSurface 9-40
 mdlBspline_closeCurve 9-58
 mdlBspline_closestEdge 9-84
 mdlBspline_closeSurface 9-59
 mdlBspline_computeDerivatives 9-76
 mdlBspline_computeDerivativesSurface 9-82
 mdlBspline_computeEqualChordByLength 9-48
 mdlBspline_computeGrevilleAbscissa 9-70
 mdlBspline_computeKnotVector 9-67
 mdlBspline_constantRadiusFillet 9-34
 mdlBspline_convertToCurve 9-11
 mdlBspline_convertToCurveChain 9-45
 mdlBspline_convertToEndcaps 9-12
 mdlBspline_convertToPlanarSurface 9-36
 mdlBspline_convertToSurface 9-12
 mdlBspline_convertToSurfaceChain 9-45
 mdlBspline_coonsPatch 9-19
 mdlBspline_copyBoundaries 9-66
 mdlBspline_copyCurve 9-10
 mdlBspline_copySurface 9-11
 mdlBspline_createCurve 9-9
 mdlBspline_createCurvesFromChain 9-44
 mdlBspline_createSurface 9-10
 mdlBspline_createSurfacesFromChain 9-44
 mdlBspline_crossSectionSurface 9-42
 mdlBspline_cubicInterpolation 9-42
 mdlBspline_cubicInterpolationExt 9-48
 mdlBspline_curveDataReduction 9-66
 mdlBspline_cuspPoints 9-92
 mdlBspline_edgeCode 9-97
 mdlBspline_elevateDegree 9-59
 mdlBspline_elevateDegreeSurface 9-59
 mdlBspline_evaluateCurve 9-78
 mdlBspline_evaluateCurvePoint 9-80
 mdlBspline_evaluateSurface 9-79
 mdlBspline_evaluateSurfacePoint 9-81
 mdlBspline_extractBoundary 9-24
 mdlBspline_extractCapAsCurve 9-94
 mdlBspline_extractCapAsCurves 9-94
 mdlBspline_extractCapAsSurface 9-94
 mdlBspline_extractCurve 9-52
 mdlBspline_extractCurveNormal 9-83
 mdlBspline_extractFromCurve 9-21
 mdlBspline_extractFromSurface 9-22
 mdlBspline_extractIsoCurve 9-22
 mdlBspline_extractProfile 9-24
 mdlBspline_extractSilhouette 9-23
 mdlBspline_extractSurface 9-53
 mdlBspline_findSpan 9-71
 mdlBspline_freeCurve 9-54
 mdlBspline_freeCurveChain 9-55
 mdlBspline_freeSurface 9-55
 mdlBspline_freeSurfaceChain 9-55
 mdlBspline_frenetFrame 9-76
 mdlBspline_getCurveFromSurface 9-37
 mdlBspline_getEdgeFromSurface 9-37
 mdlBspline_getKnotMultiplicity 9-69
 mdlBspline_gordonSurface 9-41
 mdlBspline_helix 9-26
 mdlBspline_imposeBoundary 9-26
 mdlBspline_inBounds 9-85
 mdlBspline_inflectionPoints 9-93

mdlBspline_intersectCurves 9-88
 mdlBspline_intersectOffsetSurfaces 9-91
 mdlBspline_intersectSegment 9-89
 mdlBspline_intersectSurfaceChains 9-46
 mdlBspline_intersectSurfaces 9-90
 mdlBspline_isDegenerateEdge 9-98
 mdlBspline_isPhysicallyClosed 9-87
 mdlBspline_isPhysicallyClosedCurve 9-87
 mdlBspline_isPhysicallyClosedSurface 9-88
 mdlBspline_isSolid 9-98
 mdlBspline_isValidBoundary 9-85
 mdlBspline_isValidKnotVector 9-86
 mdlBspline_isValidSurface 9-86
 mdlBspline_knotTolerance 9-68
 mdlBspline_leastSquaresToCurve 9-15
 mdlBspline_leastSquaresToSurface 9-16
 mdlBspline_make2CurvesCompatible 9-63
 mdlBspline_make2SurfacesCompatible 9-64
 mdlBspline_makeBezier 9-60
 mdlBspline_makeBezierSurface 9-60
 mdlBspline_makeCurvesCompatible 9-63
 mdlBspline_makeRational 9-61
 mdlBspline_makeRationalSurface 9-61
 mdlBspline_meshSurface 9-95
 mdlBspline_minimumDistanceToCurve 9-77
 mdlBspline_minimumDistanceToSurface 9-78
 mdlBspline_normalizeKnotVector 9-68
 mdlBspline_nSidedPatch 9-43
 mdlBspline_numberKnots 9-73
 mdlBspline_offset 9-38
 mdlBspline_offsetSurface 9-39
 mdlBspline_openCurve 9-58
 mdlBspline_openSurface 9-59
 mdlBspline_parameterFromArcLength 9-49
 mdlBspline_reverseCurve 9-62
 mdlBspline_reverseSurface 9-62
 mdlBspline_ruledSurface 9-19
 mdlBspline_segmentCurve 9-13
 mdlBspline_segmentDisjointCurve 9-65
 mdlBspline_segmentDisjointSurface 9-65
 mdlBspline_segmentSurface 9-13
 mdlBspline_skinPatch 9-20
 mdlBspline_spiral 9-25
 mdlBspline_surfaceOfProjection 9-18
 mdlBspline_surfaceOfRevolution 9-18
 mdlBspline_swapUV 9-64
 mdlBspline_trimmedPlaneFromCurves 9-36
 mdlBspline_unWeightPoles 9-57
 mdlBspline_variableRadiusFillet 9-35
 mdlBspline_weightPoles 9-57
 mdlCell_addLibDescr 15-4
 mdlCell_addLibElement 15-5
 mdlCell_attachLibrary 15-2
 mdlCell_begin 15-11
 mdlCell_create 5-16
 mdlCell_createFromFence 15-4
 mdlCell_createLibraryHeader 15-10
 mdlCell_deleteInLibrary 15-5
 mdlCell_end 15-12
 mdlCell_existsInLibrary 15-3
 mdlCell_extract 5-24
 mdlCell_fixLevels 15-9
 mdlCell_generateLibIndex 15-11
 mdlCell_getElmDscr 15-7
 mdlCell_getFilePosInLibrary 15-2
 mdlCell_getFilePosition 15-2
 mdlCell_isPointCell 15-9
 mdlCell_placeCell 15-6
 mdlCell_rename 15-5
 mdlCell_rewriteLibElement 15-5
 mdlCell_setRange 15-10
 mdlCExpression_freeSet 2-105
 mdlCExpression_generateMessage 2-109
 mdlCExpression_getValue 2-107
 mdlCExpression_initializeSet 2-105
 mdlCExpression_isArray 2-109
 mdlCExpression_isCharPointer 2-110
 mdlCExpression_isStructUnion 2-109
 mdlCExpression_setValue 2-108
 mdlCExpression_symbolPublish 2-102, 2-104
 mdlCExpression_typeArray 2-107
 mdlCExpression_typeFromRsc 2-106
 mdlCExpression_typePointer 2-106
 mdlChain_begin 7-41
 mdlChain_end 7-41
 mdlCircle_createBy3Pts 5-9
 mdlClip_element 7-42
 mdlClip_free 7-43
 mdlClip_getFence 7-43
 mdlClip_getRefBoundary 7-44
 mdlClip_isElemInside 7-44
 mdlClipboard_closeClipboard 27-47
 mdlClipboard_emptyClipboard 27-48
 mdlClipboard_getClipboardData 27-48
 mdlClipboard_getClipboardFormatName 27-48
 mdlClipboard_openClipboard 27-47

mdlClipboard_registerClipboardFormat 27-50
 mdlClipboard_setClipboardData 27-49
 mdlClipboard_setFunction 27-2, 27-53
 mdlCnv_bufferFromFileFormat 24-12
 mdlCnv_bufferToFileFormat 24-11
 mdlCnv_calcFileSizeFromDataDef 24-14
 mdlCnv_calcMemSizeFromDataDef 24-14
 mdlCnv_compileDataDefBlock 24-9
 mdlCnv_doubleFromFileFormat 24-5
 mdlCnv_doubleToFileFormat 24-5
 mdlCnv_doubleToNativeFloat 24-7
 mdlCnv_DPointToIPoint 24-2
 mdlCnv_DPointToIPointArray 24-2
 mdlCnv_fromAsciiToR50 24-3
 mdlCnv_fromR50ToAscii 24-3
 mdlCnv_fromScanFormat 24-4
 mdlCnv_fromScanFormatArray 24-4
 mdlCnv_IPointToDPoint 24-2
 mdlCnv_IPointToDPointArray 24-2
 mdlCnv_masterToUOR 24-8
 mdlCnv_nativeFloatToDouble 24-7
 mdlCnv_roundDoubleToLong 24-7
 mdlCnv_roundDoubleToULong 24-7
 mdlCnv_swapWord 24-6
 mdlCnv_swapWordArray 24-6
 mdlCnv_toRational 24-4
 mdlCnv_toScanFormat 24-4
 mdlCnv_toScanFormatArray 24-4
 mdlCnv_UORToMaster 24-8
 mdlColor_assignColorMap 17-31
 mdlColor_attachColorTable 17-5
 mdlColor_convertRGBtoIndex 17-24
 mdlColor_elementColorFromRGB 17-25
 mdlColor_elementColorToRGB 17-24
 mdlColor_getColorConfig 17-28
 mdlColor_getColorMap 17-32
 mdlColor_getColorTable 17-6
 mdlColor_getColorTableByFileNumber 17-7
 mdlColor_getDefaultColorTable 17-7
 mdlColor_getExactColorPositions 17-29
 mdlColor_getName 17-6
 mdlColor_hsvToRgb 17-27
 mdlColor_interpolateColors 17-27
 mdlColor_luminosityOf 17-25
 mdlColor_matchColorMap 17-33
 mdlColor_matchLongColorMap 17-33
 mdlColor_reallocateColors 17-29
 mdlColor_rgbToHsv 17-26
 mdlColor_searchRGBArray 17-7
 mdlColor_setExactColorPositions 17-30
 mdlColorDescr_getBestBWContrast 17-11
 mdlColorDescr_getColorId 17-9
 mdlColorDescr_getColorName 17-11
 mdlColorDescr_getDrawIndex 17-12
 mdlColorDescr_getElemColorNumber 17-13
 mdlColorDescr_getHsv 17-14
 mdlColorDescr_getMenuColor 17-15
 mdlColorDescr_getRgb 17-16
 mdlColorDescr_setByColorId 17-10
 mdlColorDescr_setByColorName 17-11
 mdlColorDescr_setByElemColorNumber 17-14
 mdlColorDescr_setByHsv 17-15
 mdlColorDescr_setByMenuColor 17-16
 mdlColorDescr_setByRgb 17-17
 mdlColorDescr_setDrawIndex 17-13
 mdlColorPal_create 17-18
 mdlColorPal_destroy 17-19
 mdlColorPal_getColorDescr 17-19
 mdlColorPal_getDrawValue 17-20
 mdlColorPal_getNumColors 17-23
 mdlColorPal_modifyStatus 17-20
 mdlColorPal_setEntries 17-21
 mdlColorPal_setPaletteFromColortable 17-22
 mdlComplexChain_createHeader 7-40
 mdlCone_create 5-14
 mdlCone_createRightCylinder 5-15
 mdlCone_createWithRotMatrix 5-15
 mdlCone_extract 5-23
 mdlCons_destroy 26-45
 mdlCons_getPoint 26-44
 mdlCons_getRMatrix 26-44
 mdlCons_getVar 26-46
 mdlCons_isChanged 26-43
 mdlCons_isOperational 26-46
 mdlConsAttachment_create 26-32
 mdlConsEquation_create 26-30
 mdlConsEquation_gen 26-31
 mdlConsEquation_parse 26-30
 mdlConsEquation_resolve 26-31
 mdlConsFrame_createBspline 26-7
 mdlConsFrame_createBsplineCell 26-8
 mdlConsFrame_createCircle 26-7
 mdlConsFrame_createEllipse 26-6
 mdlConsFrame_createLine 26-5
 mdlConsFrame_createPoint 26-5
 mdlConsFrame_isUnderDetermined 26-9

mdlConsMod_add 26-36
 mdlConsMod_apply 26-40
 mdlConsMod_backupVars 26-39
 mdlConsMod_chooseSolution 26-40
 mdlConsMod_concat 26-37
 mdlConsMod_create 26-35
 mdlConsMod_destroy 26-35
 mdlConsMod_destroyNodes 26-35
 mdlConsMod_dof 26-38
 mdlConsMod_drop 26-36
 mdlConsMod_isFound 26-41
 mdlConsMod_mirror 26-42
 mdlConsMod_offset 26-42
 mdlConsMod_restoreVars 26-39
 mdlConsMod_solve 26-37
 mdlConsMod_transform 26-41
 mdlConsMod_validate 26-38
 mdlConsParm_create 26-10
 mdlConsParm_setUnits 26-10
 mdlConstraint_chooseSolution 26-27
 mdlConstraint_createAngle 26-19
 mdlConstraint_createAngleX 26-20
 mdlConstraint_createAssignment 26-12
 mdlConstraint_createConcentric 26-15
 mdlConstraint_createDiameter 26-14
 mdlConstraint_createDistance 26-17
 mdlConstraint_createIntersection 26-16
 mdlConstraint_createIOTangent 26-22
 mdlConstraint_createLLDistance 26-18
 mdlConstraint_createLockFrame 26-24
 mdlConstraint_createLockVar 26-25
 mdlConstraint_createMatch 26-25
 mdlConstraint_createOffset 26-19
 mdlConstraint_createParallel 26-21
 mdlConstraint_createPCDistance 26-17
 mdlConstraint_createPerpendicular 26-21
 mdlConstraint_createPinPoint 26-24
 mdlConstraint_createPointLocation 26-13
 mdlConstraint_createPointOn 26-15
 mdlConstraint_createRadius 26-14
 mdlConstraint_createTangent 26-23
 mdlConstraint_flipInsideOutside 26-26
 mdlConstraint_getTarget 26-26
 mdlConstraint_rGroup 26-25
 mdlConstraint_setPostCheck 26-28
 mdlConstraint_setWeak 26-27
 mdlCurrTrans_begin 5-157
 mdlCurrTrans_clear 5-157
 mdlCurrTrans_end 5-157
 mdlCurrTrans_fromACS 5-162
 mdlCurrTrans_getAddresses 5-162
 mdlCurrTrans_identity 5-157
 mdlCurrTrans_invScaleDoubleArray 5-160
 mdlCurrTrans_invtransPointArray 5-161
 mdlCurrTrans_invtransRMatrix 5-161
 mdlCurrTrans_masterUnitsIdentity 5-157
 mdlCurrTrans_rotateByAngles 5-158
 mdlCurrTrans_rotateByRMatrix 5-158
 mdlCurrTrans_rotateByView 5-158
 mdlCurrTrans_scale 5-160
 mdlCurrTrans_scaleDoubleArray 5-160
 mdlCurrTrans_toACS 5-162
 mdlCurrTrans_transformPointArray 5-161
 mdlCurrTrans_transformRMatrix 5-161
 mdlCurrTrans_translateOrigin 5-159
 mdlCurrTrans_translateOriginWorld 5-159
 mdlCurve_create 5-4
 mdlCurve_createI 5-5
 mdlDArray_clear 28-37
 mdlDArray_copy 28-37
 mdlDArray_create 28-37
 mdlDArray_destroy 28-38
 mdlDArray_findMember 28-38
 mdlDArray_getIndex 28-38
 mdlDArray_getMemberP 28-39
 mdlDArray_insertMembers 28-39
 mdlDArray_moveMembers 28-40
 mdlDArray_nMembers 28-40
 mdlDArray_processAll 28-40
 mdlDArray_removeMembers 28-41
 mdlDArray_setMember 28-41
 mdlDB_activeAutoCommitMode 21-33
 mdlDB_activeDatabase 21-34
 mdlDB_activeDAType 21-33
 mdlDB_activeDeleteMode 21-34
 mdlDB_activeFormsMode 21-35
 mdlDB_activeLinkageMode 21-35
 mdlDB_activeReportFile 21-36
 mdlDB_activeReviewTable 21-37
 mdlDB_activeRowConfirmMode 21-37
 mdlDB_additionalRequest 21-32
 mdlDB_addRowWithMslink 21-28
 mdlDB_attachActiveEntityDscr 21-4
 mdlDB_attachActiveEntityElement 21-4
 mdlDB_buildDALinkFromLink 21-14
 mdlDB_buildLink 21-9

mdlDB_closeCursor 21-26
 mdlDB_closeCursorByID 21-29
 mdlDB_closeReport 21-6
 mdlDB_copyElement 21-11
 mdlDB_copyTable 21-30
 mdlDB_databaseProfile 21-29
 mdlDB_decodeLink 21-8
 mdlDB_defineAEBByLink 21-15
 mdlDB_defineAEBBySQLInsert 21-15
 mdlDB_defineAEBBySQLSelect 21-16
 mdlDB_defineFenceFilter 21-12
 mdlDB_deleteElement 21-11
 mdlDB_deleteRow 21-23
 mdlDB_describeColumn 21-31
 mdlDB_describeDatabase 21-31
 mdlDB_describeTable 21-25
 mdlDB_detachAttributesDscr 21-5
 mdlDB_detachAttributesElement 21-5
 mdlDB_displayActiveEntity 21-16
 mdlDB_editActiveEntity 21-16
 mdlDB_elementFilter 21-12
 mdlDB_elementReport 21-7
 mdlDB_encodeLink 21-8
 mdlDB_executeScreenForm 21-9
 mdlDB_extractLinkages 21-30
 mdlDB_fenceFilter 21-13
 mdlDB_fetchRow 21-26
 mdlDB_fetchRowByID 21-29
 mdlDB_findLinks 21-17
 mdlDB_freeSimpleSelectResultXbase 21-18
 mdlDB_freeSQLDADescriptor 21-27
 mdlDB_getErrorText 21-24
 mdlDB_getMscatalogName 21-18
 mdlDB_insertRowByForm 21-28
 mdlDB_largestMslink 21-10
 mdlDB_loadDADscr 21-13
 mdlDB_openCursor 21-26
 mdlDB_openCursorWithID 21-29
 mdlDB_openReport 21-6
 mdlDB_processSQL 21-24
 mdlDB_readColumn 21-22
 mdlDB_reviewAttributes 21-7
 mdlDB_setLinkCacheInfo 21-19
 mdlDB_simpleSelectXbase 21-19
 mdlDB_sqlQuery 21-23
 mdlDB_writeColumn 21-22
 mdlDBDialog_attachCurrentRow 21-39
 mdlDBDialog_clearValues 21-40
 mdlDBDialog_deleteCurrentRow 21-40
 mdlDBDialog_detachCurrentRow 21-40
 mdlDBDialog_firstRow 21-41
 mdlDBDialog_generateRscFile 21-41
 mdlDBDialog_insertRow 21-42
 mdlDBDialog_itemGetState 21-43
 mdlDBDialog_itemSetState 21-43
 mdlDBDialog_lastRow 21-41
 mdlDBDialog_locateCurrentRow 21-43
 mdlDBDialog_nextPage 21-44
 mdlDBDialog_nextRow 21-41
 mdlDBDialog_openForm 21-44
 mdlDBDialog_openFormFromElement 21-46
 mdlDBDialog_prevPage 21-44
 mdlDBDialog_prevRow 21-41
 mdlDBDialog_processUserQuery 21-48
 mdlDBDialog_publishExtraHooks 21-49
 mdlDBDialog_queryRscFile 21-49
 mdlDBDialog_review 21-49
 mdlDBDialog_setPage 21-44
 mdlDBDialog_startQuery 21-50
 mdlDBDialog_updateCurrentRow 21-51
 mdlDBDialog_useFenceIfActive 21-51
 mdlDialog_autoOpen 3-85
 mdlDialog_busyBarStartProcessing 3-141
 mdlDialog_busyBarStopProcessing 3-142
 mdlDialog_busyBarUpdateControlParms 3-142
 mdlDialog_busyBarUpdateMessage 3-143
 mdlDialog_callFunction 3-148
 mdlDialog_closeCommandQueue 3-130
 mdlDialog_closeCompletionBar 3-74
 mdlDialog_cmdNumberQueue 3-131, 3-133
 mdlDialog_cmdNumberQueueByDb 3-131, 3-134
 mdlDialog_cmdNumberQueueByTaskId
 3-132, 3-135
 mdlDialog_cmdNumQByDbExt 3-134
 mdlDialog_cmdNumQByTaskIdExt 3-135
 mdlDialog_cmdNumQueueExt 3-133
 mdlDialog_colorPickerGetInfo 3-72
 mdlDialog_colorPickerSetInfo 3-72
 mdlDialog_commandWindowGet 3-88, 3-153
 mdlDialog_completionBarClose 3-75
 mdlDialog_completionBarDisplayMessage 3-76
 mdlDialog_completionBarOpen 3-76
 mdlDialog_completionBarUpdate 3-77
 mdlDialog_create 3-85
 mdlDialog_defFileCreate 25-25
 mdlDialog_defFileOpen 25-25

mdlDialog_diamondDrawBeveled 3-113
 mdlDialog_diamondFill 3-113
 mdlDialog_displayCompletionBarMessage 3-75
 mdlDialog_dmsgsPrint 3-147
 mdlDialog_dump 3-148
 mdlDialog_ellipseFill 3-114
 mdlDialog_fileCreate 25-22
 mdlDialog_fileCreateFromSeed 25-24
 mdlDialog_fileOpen 25-22
 mdlDialog_fileOpenExt 3-153
 mdlDialog_find 3-86
 mdlDialog_findByTypeAndId 3-87
 mdlDialog_fixResourceAddress 3-149
 mdlDialog_focusItemIndexGet 3-110
 mdlDialog_focusItemIndexSet 3-110
 mdlDialog_fontGetCurHeight 3-128
 mdlDialog_fontGetHeight 3-128
 mdlDialog_fontGetInfo 3-128
 mdlDialog_fontIndexGet 3-129
 mdlDialog_fontIndexSet 3-129
 mdlDialog_fontNameGet 3-128
 mdlDialog_hasFocus 3-87
 mdlDialog_hookDialogSendUserMsg 3-89
 mdlDialog_hookItemSendUserMsg 3-89
 mdlDialog_hookPublish 3-90
 mdlDialog_icFrameGetItemInfo 3-36
 mdlDialog_icFrameGetNItems 3-36
 mdlDialog_icFrameGetNSubItems 3-37
 mdlDialog_icFrameSelectIcon 3-37
 mdlDialog_icFrameSetItemInfo 3-36
 mdlDialog_icPaletteGetItemInfo 3-40
 mdlDialog_icPaletteGetNItems 3-41
 mdlDialog_icPaletteSelectIcon 3-39
 mdlDialog_icPaletteSetItemInfo 3-40
 mdlDialog_itemDraw 3-99
 mdlDialog_itemGetByIndex 3-97
 mdlDialog_itemGetByTypeAndId 3-97
 mdlDialog_itemGetState 3-96
 mdlDialog_itemGetValue 3-96
 mdlDialog_itemHide 3-100
 mdlDialog_itemLoad 3-101
 mdlDialog_itemMove 3-102
 mdlDialog_itemsApply 3-108
 mdlDialog_itemSetEnabledState 3-103
 mdlDialog_itemSetExtent 3-104
 mdlDialog_itemSetLabel 3-104
 mdlDialog_itemSetState 3-105
 mdlDialog_itemSetValue 3-105
 mdlDialog_itemsFree 3-109
 mdlDialog_itemsGetNumberOf 3-108
 mdlDialog_itemShow 3-101
 mdlDialog_itemsLoad 3-109
 mdlDialog_itemsSwapOrder 3-103
 mdlDialog_itemsSynch 3-107
 mdlDialog_itemSynch 3-106
 mdlDialog_keyinWindowGet 3-156
 mdlDialog_labelSetAttributes 3-147
 mdlDialog_lastActionTypeSet 3-90
 mdlDialog_levelMapGetInfo 3-71
 mdlDialog_levelMapSetInfo 3-71
 mdlDialog_lineStyleSet 3-115
 mdlDialog_listBoxDeleteAll 3-43
 mdlDialog_listBoxDeleteColumn 3-43
 mdlDialog_listBoxDrawContents 3-52
 mdlDialog_listBoxEnableCells 3-53
 mdlDialog_listBoxGetColInfo 3-43
 mdlDialog_listBoxGetDisplayRange 3-51
 mdlDialog_listBoxGetInfo 3-44
 mdlDialog_listBoxGetLocationCursor 3-45
 mdlDialog_listBoxGetNColumns 3-46
 mdlDialog_listBoxGetNextSelection 3-54
 mdlDialog_listBoxGetSelections 3-47
 mdlDialog_listBoxGetSelectRange 3-54
 mdlDialog_listBoxGetStrListP 3-49
 mdlDialog_listBoxInsertColumn 3-48
 mdlDialog_listBoxIsCellEnabled 3-55
 mdlDialog_listBoxIsCellSelected 3-55
 mdlDialog_listBoxLastCellClicked 3-56
 mdlDialog_listBoxNRowsChanged 3-51
 mdlDialog_listBoxSelectCells 3-53
 mdlDialog_listBoxSetColInfo 3-43
 mdlDialog_listBoxSetInfo 3-44
 mdlDialog_listBoxSetLocationCursor 3-46
 mdlDialog_listBoxSetSelections 3-47
 mdlDialog_listBoxSetStrListP 3-49
 mdlDialog_listBoxSetTopRow 3-51
 mdlDialog_listBoxSetTopRowRedraw 3-49
 mdlDialog_menuBarActivate 3-3
 mdlDialog_menuBarAddAppMenu 3-4
 mdlDialog_menuBarAddCmdWinMenu 3-5
 mdlDialog_menuBarAttachMenu 3-5
 mdlDialog_menuBarDeleteAllItems 3-16
 mdlDialog_menuBarDeleteAllMenus 3-13
 mdlDialog_menuBarDeleteCmdWinMenu 3-5
 mdlDialog_menuBarDeleteItem 3-16
 mdlDialog_menuBarDeleteMenu 3-14

mdlDialog_menuBarDetachMenu 3-6
 mdlDialog_menuBarFind 3-6
 mdlDialog_menuBarFindAppMenu 3-7
 mdlDialog_menuBarFindItem 3-8
 mdlDialog_menuBarFindMenu 3-10
 mdlDialog_menuBarGetCmdWinP 3-7
 mdlDialog_menuBarGetItem 3-9
 mdlDialog_menuBarGetMenu 3-11
 mdlDialog_menuBarGetNItems 3-12
 mdlDialog_menuBarGetNMenus 3-11
 mdlDialog_menuBarGetSelection 3-12
 mdlDialog_menuBarInsertMenu 3-15
 mdlDialog_menuBarInsMenu 3-15
 mdlDialog_menuBarMenuGetEnabled 3-17
 mdlDialog_menuBarMenuGetTitle 3-16
 mdlDialog_menuBarMenuSetEnabled 3-17
 mdlDialog_menuBarMenuSetTitle 3-16
 mdlDialog_menuBarRegister 3-4
 mdlDialog_menuBarSetDefault 3-7
 mdlDialog_menuBarUnloadApp 3-7
 mdlDialog_mlTextGetCursor 3-60
 mdlDialog_mlTextGetInfo 3-59
 mdlDialog_mlTextGetLineCoords 3-61
 mdlDialog_mlTextGetLineRange 3-62
 mdlDialog_mlTextInsertString 3-63
 mdlDialog_mlTextSetCursor 3-60
 mdlDialog_mlTextSetInfo 3-59
 mdlDialog_mlTextTopRowNumber 3-63
 mdlDialog_open 3-79
 mdlDialog_openAdvisoryBox 3-150
 mdlDialog_openAlert 3-81
 mdlDialog_openAlertById 3-82
 mdlDialog_openCompletionBar 3-74
 mdlDialog_openInfoBox 3-82
 mdlDialog_openMessageBox 3-83, 3-151
 mdlDialog_openModal 3-80
 mdlDialog_openPalette 3-81
 mdlDialog_openWithDBQuery 21-47
 mdlDialog_optionButtonDeleteAll 3-34
 mdlDialog_optionButtonDeleteItem 3-34
 mdlDialog_optionButtonGetItemInfo 3-29
 mdlDialog_optionButtonGetNItems 3-28
 mdlDialog_optionButtonGetSubInfo 3-31
 mdlDialog_optionButtonInsertItem 3-32
 mdlDialog_optionButtonInsSubItem 3-32
 mdlDialog_optionButtonSetEnabled 3-28
 mdlDialog_optionButtonSetExtent 3-29
 mdlDialog_optionButtonSetItemInfo 3-29
 mdlDialog_optionButtonSetSubInfo 3-31
 mdlDialog_optionPDMItemGetInfo 3-25
 mdlDialog_optionPDMItemInsert 3-26
 mdlDialog_optionPDMItemSetEnabled 3-24
 mdlDialog_optionPDMItemSetInfo 3-25
 mdlDialog_overallTitleBarGet 3-152
 mdlDialog_ownerMDGet 3-87
 mdlDialog_parentIdGet 3-88
 mdlDialog_parentIdSet 3-88
 mdlDialog_publishBasicPtr 3-91
 mdlDialog_publishBasicVariable 3-91
 mdlDialog_publishComplexPtr 3-91
 mdlDialog_publishComplexVariable 3-91
 mdlDialog_publishStructure 3-91
 mdlDialog_pushButtonActivate 3-66
 mdlDialog_pushButtonGetInfo 3-68
 mdlDialog_pushButtonSetCancel 3-67
 mdlDialog_pushButtonSetDefault 3-66
 mdlDialog_pushButtonSetInfo 3-68
 mdlDialog_rectClearBevel 3-124
 mdlDialog_rectDraw 3-122
 mdlDialog_rectDrawBeveled 3-124
 mdlDialog_rectDrawCD 3-122
 mdlDialog_rectDrawEdge 3-124
 mdlDialog_rectEqual 3-126
 mdlDialog_rectFill 3-123
 mdlDialog_rectFillCD 3-123
 mdlDialog_rectHeight 3-126
 mdlDialog_rectInset 3-125
 mdlDialog_rectInvert 3-124
 mdlDialog_rectOffset 3-125
 mdlDialog_rectOverlap 3-127
 mdlDialog_rectPointInside 3-126
 mdlDialog_rectSet 3-125
 mdlDialog_rectWidth 3-126
 mdlDialog_scrollArrowDraw 3-116
 mdlDialog_scrollBarGetInfo 3-64
 mdlDialog_scrollBarSetInfo 3-64
 mdlDialog_scrollBarSetRange 3-65
 mdlDialog_selectIconsByCmd 3-39
 mdlDialog_selectIconsByCmdNoMsg 3-39
 mdlDialog_selectIconsById 3-38
 mdlDialog_selectIconsByIdNoMsg 3-38
 mdlDialog_setFilterString 3-94
 mdlDialog_show 3-86
 mdlDialog_statusAreaGet 3-152
 mdlDialog_stringnWidth 3-116
 mdlDialog_stringWidth 3-116

mdlDialog_synonymsSynch 3-94
 mdlDialog_tabPageFreeItems 3-163
 mdlDialog_tabPageGetInfo 3-161
 mdlDialog_tabPageGetItemByIndex 3-163
 mdlDialog_tabPageGetItemByTypeAndId 3-162
 mdlDialog_tabPageListFreePages 3-160
 mdlDialog_tabPageListGetInfo 3-158
 mdlDialog_tabPageListGetPageById 3-159
 mdlDialog_tabPageListGetPageByIndex 3-159
 mdlDialog_tabPageListLoadPages 3-160
 mdlDialog_tabPageListSetInfo 3-158
 mdlDialog_tabPageLoadItems 3-164
 mdlDialog_tabPageSetInfo 3-162
 mdlDialog_textDraw 3-117
 mdlDialog_textDrawCD 3-118
 mdlDialog_textDrawN 3-117
 mdlDialog_textDrawNCD 3-118
 mdlDialog_textGetInfo 3-57
 mdlDialog_textGetRange 3-57
 mdlDialog_textPDMItemGetInfo 3-20
 mdlDialog_textPDMItemIns 3-21
 mdlDialog_textPDMItemInsert 3-21
 mdlDialog_textPDMItemSetEnabled 3-19
 mdlDialog_textPDMItemSetInfo 3-20
 mdlDialog_textPDMItemSetLabel 3-19
 mdlDialog_textPDMItemSetMark 3-19
 mdlDialog_textSetInfo 3-57
 mdlDialog_textSetRange 3-57
 mdlDialog_toDCoord 3-113
 mdlDialog_toggleButtonGetInfo 3-70
 mdlDialog_toggleButtonSetInfo 3-70
 mdlDialog_toPixels 3-112
 mdlDialog_trackBarStartProcessing 3-136
 mdlDialog_trackBarStopProcessing 3-138
 mdlDialog_trackBarUpdateControlParms 3-138
 mdlDialog_trackBarUpdateDisplayInfo 3-139
 mdlDialog_underlineDraw 3-119
 mdlDialog_underlineDrawCD 3-120
 mdlDialog_updateCompletionBar 3-75
 mdlDialog_userDataPtrGet 3-89
 mdlDialog_userDataPtrSet 3-89
 mdlDialog_userPrefFileOpen 3-148
 mdlDigitize_getTabletPoint 23-3
 mdlDigitize_setFunction 23-2
 mdlDim_create 10-4
 mdlDim_defineRotMatrix 10-5
 mdlDim_deletePoint 10-9
 mdlDim_extractPoints 10-10
 mdlDim_getElementDescr 10-12
 mdlDim_getHeight 10-7
 mdlDim_getParam 10-4
 mdlDim_getRotMatrix 10-6
 mdlDim_getStrings 10-10
 mdlDim_insertPoint 10-9
 mdlDim_setHeight 10-7
 mdlDim_setParam 10-4
 mdlDim_setRotMatrix 10-6
 mdlDim_setStrings 10-11
 mdlDim_setTextOffset 10-8
 mdlDim_validate 10-12
 mdlDither_drawRow 18-19
 mdlDynamic_loadElement 7-46
 mdlDynamic_setElmDescr 7-45
 mdlElement_add 5-29
 mdlElement_append 5-29
 mdlElement_appendAttributes 5-32
 mdlElement_cnvFromFileFormat 5-40
 mdlElement_cnvToFileFormat 5-40
 mdlElement_createColorTable 5-37
 mdlElement_createDigitizerSettings 5-44
 mdlElement_createDimensionSettings 5-44
 mdlElement_createExtendedTCB 5-44
 mdlElement_createViewSettings 5-44
 mdlElement_display 5-30
 mdlElement_displayInSelectedViews 5-30
 mdlElement_extractAttributes 5-32
 mdlElement_getFilePos 5-34
 mdlElement_getFillColor 5-38
 mdlElement_getLineStyle 5-42
 mdlElement_getProperties 5-38
 mdlElement_getSymbolology 5-36
 mdlElement_getType 5-36
 mdlElement_igdsSize 5-33
 mdlElement_isFilled 5-39
 mdlElement_offset 5-41
 mdlElement_read 5-34
 mdlElement_rewrite 5-29
 mdlElement_setFilePos 5-35
 mdlElement_setFillColor 5-38
 mdlElement_setLineStyle 5-43
 mdlElement_setProperties 5-38
 mdlElement_setSymbolology 5-36
 mdlElement_size 5-33
 mdlElement_stripAttributes 5-33
 mdlElement_stroke 5-41
 mdlElement_transform 5-41

- mdlElement_undoableDelete 5-31
- mdlEllipse_create 5-8
- mdlElmdscr_add 5-50
- mdlElmdscr_addFill 5-68
- mdlElmdscr_addToChain 5-65
- mdlElmdscr_append 5-50
- mdlElmdscr_appendAttributes 5-74
- mdlElmdscr_appendDscr 5-65
- mdlElmdscr_appendElement 5-63
- mdlElmdscr_close 5-73
- mdlElmdscr_convertTo2D 5-69
- mdlElmdscr_convertTo3D 5-70
- mdlElmdscr_copyParallel 5-79
- mdlElmdscr_createFromVertices 5-66
- mdlElmdscr_createShapeWithHoles 5-67
- mdlElmdscr_differenceShapes 5-82
- mdlElmdscr_display 5-53
- mdlElmdscr_displayFromFile 5-53
- mdlElmdscr_displayFromFileViews 5-54
- mdlElmdscr_displayInSelectedViews 5-54
- mdlElmdscr_displaySingle 5-54
- mdlElmdscr_displayToWindow 5-77
- mdlElmdscr_distanceAtPoint 5-73
- mdlElmdscr_duplicate 5-59
- mdlElmdscr_duplicateSingle 5-59
- mdlElmdscr_extendedDisplayToWindow 5-78
- mdlElmdscr_extractAttributes 5-75
- mdlElmdscr_extractEndPoints 5-62
- mdlElmdscr_extractNormal 5-76
- mdlElmdscr_freeAll 5-69
- mdlElmdscr_fromCompoundElement 5-71
- mdlElmdscr_generatePartial 5-67
- mdlElmdscr_hiddenLineRemoval 5-68, 5-119
- mdlElmdscr_hiddenLineRemoval2 5-119
- mdlElmdscr_igdsSize 5-57
- mdlElmdscr_insertElement 5-63
- mdlElmdscr_intersectShapes 5-82
- mdlElmdscr_isClosed 5-72
- mdlElmdscr_isGroupedHole 5-72
- mdlElmdscr_isOpen 5-72
- mdlElmdscr_markElement 5-58
- mdlElmdscr_new 5-63
- mdlElmdscr_open 5-73
- mdlElmdscr_operation 5-55
- mdlElmdscr_partialDelete 5-62
- mdlElmdscr_pointAtDistance 5-73
- mdlElmdscr_read 5-49
- mdlElmdscr_readToMaster 5-49
- mdlElmdscr_removeElement 5-64
- mdlElmdscr_replaceDscr 5-65
- mdlElmdscr_replaceElement 5-64
- mdlElmdscr_reverse 5-61
- mdlElmdscr_reverseNormal 5-61
- mdlElmdscr_rewrite 5-52
- mdlElmdscr_setProperties 5-76
- mdlElmdscr_singleIgdsSize 5-58
- mdlElmdscr_stripAttributes 5-74
- mdlElmdscr_stripFill 5-68
- mdlElmdscr_stroke 5-55
- mdlElmdscr_transform 5-51
- mdlElmdscr_undoableDelete 5-51
- mdlElmdscr_unionShapes 5-81
- mdlElmdscr_validate 5-60
- mdlExternal_messageReceive 22-3, 22-9, 22-15
- mdlExternal_messageSend 22-4, 22-14
- mdlExternal_queueGet 22-14
- mdlExternal_queueRemove 22-16
- mdlExternal_sendSignal 22-17
- mdlExternal_setFunction 22-18
- mdlExternal_shmemGet 22-16
- mdlExternal_shmemRemove 22-17
- mdlExternal_startMCSLProgram 22-4, 22-13
- mdlExternal_startProgram 22-4, 22-9, 22-11
- mdlExternal_startRealModeProgram 22-12
- mdlExternal_terminateProgram 22-9, 22-10
- mdlExternal_wait 22-3, 22-9, 22-17
- mdlFence_clear 7-36, 7-37
- mdlFence_fromShape 7-35
- mdlFence_process 7-35
- mdlFence_toShape 7-35
- mdlFile_abbreviateName 25-12
- mdlFile_buildName 25-4
- mdlFile_checkDesignFile 25-10
- mdlFile_copy 25-11
- mdlFile_create 25-2
- mdlFile_find 25-2
- mdlFile_findFiles 25-8
- mdlFile_getcwd 25-5
- mdlFile_getDiskFree 25-6
- mdlFile_getDrive 25-7
- mdlFile_getFileAttributes 25-9
- mdlFile_isUntitledDesign 25-12
- mdlFile_isValidDrive 25-6
- mdlFile_mkdir 25-5
- mdlFile_parseName 25-4
- mdlFile_setDefaultShare 25-9

mdlFile_setDrive 25-6
 mdlFile_setFileType 25-10
 mdlFileDragAndDrop_setFunction 27-43
 mdlFileList_edit 25-20
 mdlFileList_fromString 25-27
 mdlFileList_get 25-19
 mdlFuncKey_attachMenu 28-44
 mdlFuncKey_currentMenu 28-45
 mdlFuncKey_define 28-46
 mdlFuncKey_getKeyByIndex 28-45
 mdlFuncKey_getKeyByVirtualKey 28-46
 mdlFuncKey_getKeyName 28-46
 mdlFuncKey_numKeys 28-45
 mdlFuncKey_remove 28-47
 mdlFuncKey_saveMenu 28-44
 mdlHelp_getHelpTopic 28-42
 mdlHelp_setShowMeFunction 28-43
 mdlHTML_startStandardTable 29-37
 mdlHTMMLib_addCaption 29-5
 mdlHTMMLib_addCitationText 29-6
 mdlHTMMLib_addDefinitionListItem 29-6
 mdlHTMMLib_addEmphasizedText 29-7
 mdlHTMMLib_addFormButtonItem 29-7
 mdlHTMMLib_addFormCheckBoxRadioButtonItem 29-7
 mdlHTMMLib_addFormHiddenItem 29-9
 mdlHTMMLib_addFormImageItem 29-8
 mdlHTMMLib_addFormSelectOptionItem 29-8
 mdlHTMMLib_addFormTextItem 29-9
 mdlHTMMLib_addFrame 29-10
 mdlHTMMLib_addFrameset 29-10
 mdlHTMMLib_addHeading 29-11
 mdlHTMMLib_addHorizontalRuledLine 29-12
 mdlHTMMLib_addHREFAnchor 29-12
 mdlHTMMLib_addImage 29-12
 mdlHTMMLib_addKeyboardSampleText 29-13
 mdlHTMMLib_addLineBreak 29-14
 mdlHTMMLib_addListItem 29-14
 mdlHTMMLib_addListItemOpt 29-14
 mdlHTMMLib_addMapArea 29-15
 mdlHTMMLib_addNameAnchor 29-15
 mdlHTMMLib_addNewLine 29-16
 mdlHTMMLib_addNoFrameMessage 29-16
 mdlHTMMLib_addPlainText 29-16
 mdlHTMMLib_addProtocolToString 29-17
 mdlHTMMLib_addProtocolToStringPath 29-17
 mdlHTMMLib_addSourceCodeText 29-18
 mdlHTMMLib_addStartFormSelection 29-18
 mdlHTMMLib_addStrongText 29-18
 mdlHTMMLib_addTableCell 29-19
 mdlHTMMLib_addTableHeaderRow 29-19
 mdlHTMMLib_addTextAreaItem 29-20
 mdlHTMMLib_addTextString 29-20
 mdlHTMMLib_addVariableName 29-21
 mdlHTMMLib_beginFrameDocument 29-21
 mdlHTMMLib_beginHTMLDocument 29-22
 mdlHTMMLib_closeFile 29-22
 mdlHTMMLib_closeFrame 29-23
 mdlHTMMLib_createFile 29-23
 mdlHTMMLib_endAnchor 29-23
 mdlHTMMLib_endBody 29-23
 mdlHTMMLib_endCenter 29-24
 mdlHTMMLib_endDefinitionList 29-24
 mdlHTMMLib_endDirectoryList 29-24
 mdlHTMMLib_endFileHeader 29-25
 mdlHTMMLib_endForm 29-25
 mdlHTMMLib_endFormSelectItem 29-25
 mdlHTMMLib_endFrameDocument 29-25
 mdlHTMMLib_endFrameset 29-26
 mdlHTMMLib_endHTML 29-26
 mdlHTMMLib_endHTMLDocument 29-26
 mdlHTMMLib_endMap 29-27
 mdlHTMMLib_endMenuList 29-27
 mdlHTMMLib_endNoFrame 29-27
 mdlHTMMLib_endOrderedList 29-27
 mdlHTMMLib_endParagraph 29-28
 mdlHTMMLib_endTable 29-28
 mdlHTMMLib_endTableCell 29-28
 mdlHTMMLib_endTableRow 29-29
 mdlHTMMLib_endUnorderedList 29-29
 mdlHTMMLib_hrefListFromStringList 29-29
 mdlHTMMLib_htmlListFromStringList 29-30
 mdlHTMMLib_imageTableFromList 29-30
 mdlHTMMLib_openFile 29-31
 mdlHTMMLib_setDocTypeToHTML 29-32
 mdlHTMMLib_setMetaNameGenerator 29-32
 mdlHTMMLib setPageTitle 29-32
 mdlHTMMLib_startBody 29-33
 mdlHTMMLib_startCenter 29-33
 mdlHTMMLib_startDefinitionList 29-33
 mdlHTMMLib_startDirectoryList 29-33
 mdlHTMMLib_startFileHeader 29-34
 mdlHTMMLib_startForm 29-34
 mdlHTMMLib_startHREFAnchor 29-34
 mdlHTMMLib_startHTML 29-35
 mdlHTMMLib_startMap 29-35

- mdlHTMMLib_startMenuList 29-35
- mdlHTMMLib_startNameAnchor 29-36
- mdlHTMMLib_startNoFrame 29-36
- mdlHTMMLib_startOrderedList 29-36
- mdlHTMMLib_startParagraph 29-37
- mdlHTMMLib_startTable 29-37
- mdlHTMMLib_startTableCell 29-38
- mdlHTMMLib_startTableRow 29-38
- mdlHTMMLib_startTableRowAlign 29-39
- mdlHTMMLib_startUnorderedList 29-39
- mdlHTMMLib_textHREFTableFromList 29-39
- mdlHTMMLib_textTableFromList 29-40
- mdlHview_clearView 5-142
- mdlHview_closeContext 5-143
- mdlHview_eraseAndRedrawView 5-143
- mdlHview_message 5-144
- mdlHview_openContext 5-149
- mdlHview_openFile 5-150
- mdlHview_processView 5-150
- mdlHview_setSectionPlanes 5-151
- mdlImage_addMovieFrame 18-30
- mdlImage_appendFliFromMap 18-29
- mdlImage_appendFliFromRGB 18-29
- mdlImage_applyGamma 18-25
- mdlImage_applyGammaToPalette 18-20
- mdlImage_byteMapToBitMap 18-31
- mdlImage_byteMapToGreyScale 18-32
- mdlImage_captureScreen 18-33
- mdlImage_captureScreenMap 18-20
- mdlImage_checkStop 18-26
- mdlImage_completeFli 18-29
- mdlImage_completeMovie 18-34
- mdlImage_computeScreenMap 18-35
- mdlImage_convertToUpperLeftHorizontal 18-36
- mdlImage_createFileFromBitMap 18-37
- mdlImage_createFileFromBuffer 18-38
- mdlImage_createFileFromMap 18-15
- mdlImage_createFileFromRGB 18-16
- mdlImage_createFli 18-29
- mdlImage_createMovie 18-41
- mdlImage_createRasterFromRGB 18-25
- mdlImage_deleteMovieFrame 18-28
- mdlImage_displayDescrGet 18-41
- mdlImage_ditherCleanup 18-18
- mdlImage_ditherInitialize 18-18
- mdlImage_ditherRow 18-18
- mdlImage_doubleImage 18-42
- mdlImage_extMapToRGB 18-43
- mdlImage_extractBitMapSubImage 18-44
- mdlImage_extractByteMapSubImage 18-45
- mdlImage_extractIngrAttach 18-46
- mdlImage_extractPackByteSubImage 18-47
- mdlImage_extractSubImage 18-49
- mdlImage_extReadFileToMap 18-50
- mdlImage_extRGBToMap 18-51
- mdlImage_extRGBToScreenMap 18-52
- mdlImage_freeImage 18-53
- mdlImage_freeMovie 18-29
- mdlImage_getBalancedPalette 18-16
- mdlImage_getExportFormat 18-12
- mdlImage_getExportSupport 18-12
- mdlImage_getExtension 18-11
- mdlImage_getImportFormat 18-12
- mdlImage_getMapUsage 18-54
- mdlImage_getOptimizedPalette 18-17
- mdlImage_getOSFileType 18-12
- mdlImage_getScreenPalette 18-20
- mdlImage_greyScaleToBitMap 18-55
- mdlImage_insertMovie 18-28
- mdlImage_insertMovieFrame 18-28
- mdlImage_isAVIAvailable 18-56
- mdlImage_mapToRGB 18-22
- mdlImage_mapToRGBBuffer 18-22
- mdlImage_memorySize 18-56
- mdlImage_mirror 18-57
- mdlImage_negate 18-21
- mdlImage_negatePalette 18-21
- mdlImage_packByteBuffer 18-58
- mdlImage_packByteToBitMap 18-59
- mdlImage_packByteToGreyScale 18-60
- mdlImage_paletteToGreyScale 18-61
- mdlImage_readFileInfo 18-11
- mdlImage_readFileToBitMap 18-61
- mdlImage_readFileToMap 18-13
- mdlImage_readFileToRGB 18-14
- mdlImage_readMovie 18-27
- mdlImage_remapToScreenPalette 18-63
- mdlImage_renderViewToRGB 18-26
- mdlImage_resize 18-24
- mdlImage_rgbToBitMap 18-64
- mdlImage_rgbToGreyScale 18-65
- mdlImage_RGBToMap 18-23
- mdlImage_RGBToMapWithGamma 18-23
- mdlImage_RGBToPackByte 18-65
- mdlImage_RGBToPackByteWithGamma 18-67
- mdlImage_RGBToScreenMap 18-23

mdlImage_rotate 18-68
 mdlImage_runLengthDecodeBitMapRow 18-69
 mdlImage_runLengthEncodeBitMap 18-69
 mdlImage_runLengthEncodeBitMapRow 18-70
 mdlImage_saveMovie 18-27
 mdlImage_setMapIfRGBMatch 18-71
 mdlImage_setMapPolygon 18-71
 mdlImage_setRGBPolygon 18-73
 mdlImage_stretchRLEToBitMap 18-74
 mdlImage_subByteMapFromBitMap 18-75
 mdlImage_subByteMapFromPackByte 18-76
 mdlImage_subByteMapFromRLEBitMap 18-77
 mdlImage_tintImage 18-78
 mdlImage_tintPalette 18-79
 mdlImage_typeFromExtension 18-11
 mdlImage_typeFromFile 18-79
 mdlImage_unpackByteBuffer 18-80
 mdlImage_updateIngrAttach 18-81
 mdlImage_warp 18-81
 mdlInput_commandState 2-89
 mdlInput_disableCommandClass 2-90
 mdlInput_enableCommandClass 2-90
 mdlInput_endCommand 2-89
 mdlInput_getMessage 2-87
 mdlInput_getTabletType 2-90
 mdlInput_longjmp 2-91
 mdlInput_pause 2-89
 mdlInput_requeueLastInput 2-81
 mdlInput_sendCommand 2-81
 mdlInput_sendKeyin 2-82
 mdlInput_sendKeystroke 2-85
 mdlInput_sendMessage 2-86
 mdlInput_sendReset 2-86
 mdlInput_sendResume 2-88
 mdlInput_sendUORPoint 2-87
 mdlInput_setFunction 2-92
 mdlInput_setjmp 2-91
 mdlInput_setMonitorFunction 2-92
 mdlInput_startCommand 2-89
 mdlInput_waitForMessage 2-88
 mdlIntersect_allBetweenElms 5-94
 mdlIntersect_allBetweenExtendedElms 5-94
 mdlIntersect_betweenElmsByIndex 5-95
 mdlIntersect_closestBetweenElms 5-95
 mdlItem_colorChanged 3-111
 mdlLevel_addGroup 19-15
 mdlLevel_addLevel 19-13
 mdlLevel_buildLevelPath 19-19
 mdlLevel_deleteGroup 19-15
 mdlLevel_deleteLevel 19-13
 mdlLevel_editGroup 19-16
 mdlLevel_editLevel 19-14
 mdlLevel_freeAll 19-23
 mdlLevel_freeGroups 19-24
 mdlLevel_freeGroupSet 19-18
 mdlLevel_freeLevels 19-24
 mdlLevel_freeLevelSet 19-19
 mdlLevel_getDescendentGroups 19-17
 mdlLevel_getDescendentLevels 19-16
 mdlLevel_getGroupIDFromNameAndParent 19-23
 mdlLevel_getGroupIndex 19-21
 mdlLevel_getGroupIndexFromName 19-22
 mdlLevel_getGroupLevels 19-18
 mdlLevel_getGroupNameFromID 19-23
 mdlLevel_getLevelIndex 19-20
 mdlLevel_getLevelIndexFromName 19-20
 mdlLevel_getLevelMask 19-10
 mdlLevel_getLevelMaskFromGroup 19-11
 mdlLevel_getLevelMaskFromLevel 19-11
 mdlLevel_getLevelMaskFromPath 19-12
 mdlLevel_getLevelName 19-28
 mdlLevel_getLevelNameByFile 19-28
 mdlLevel_getNextGroupID 19-14
 mdlLevel_getParentGroup 19-21
 mdlLevel_loadAllFromResource 19-25
 mdlLevel_loadGroupsFromResource 19-25
 mdlLevel_loadLevelsFromResource 19-26
 mdlLevel_saveAllToResource 19-26
 mdlLevel_saveGroupsToResource 19-27
 mdlLevel_saveLevelsToResource 19-27
 mdlLevelSymbology_extract 19-29
 mdlLicense_getCurrentHardwareKeySerial 28-32
 mdlLicense_getHardwareKeyPermission 28-33
 mdlLicense_getInitialHardwareKeySerial 28-33
 mdlLicense_getStrings 28-33
 mdlLicense_getVarietyName 28-34
 mdlLicense_isAcademicProduct 28-34
 mdlLicense_isHardwareKeyRequired 28-35
 mdlLicense_isHardwareKeySupported 28-35
 mdlLicense_isRegisteredProduct 28-35
 mdlLicense_isSerializedProduct 28-35
 mdlLine_create 5-3
 mdlLine_createI 5-5
 mdlLinear_extract 5-18
 mdlLinear_getClosestSegment 5-18
 mdlLineString_create 5-4

mdlLineStyle_createI 5-5
 mdlLineStyle_cacheDelete 11-12
 mdlLineStyle_cacheFree 11-13
 mdlLineStyle_cacheInsert 11-11
 mdlLineStyle_getElementDescr 11-11
 mdlLineStyle_nameDelete 11-6
 mdlLineStyle_nameGetStringList 11-10
 mdlLineStyle_nameInsert 11-6
 mdlLineStyle_nameLoadMap 11-9
 mdlLineStyle_nameModify 11-7
 mdlLineStyle_nameQuery 11-8
 mdlLineStyle_nameSaveMap 11-9
 mdlLinkage_appendToElement 6-5
 mdlLinkage_appendUsingDescr 6-9
 mdlLinkage_deleteFromElement 6-4
 mdlLinkage_deleteUsingDescr 6-8
 mdlLinkage_extractFromElement 6-2
 mdlLinkage_extractUsingDescr 6-7
 mdlLocate_allowLocked 7-13
 mdlLocate_clearElemSearchMask 7-14
 mdlLocate_findElement 7-17
 mdlLocate_getProjectedPoint 7-18
 mdlLocate_identifyElement 7-15
 mdlLocate_init 7-16
 mdlLocate_noElemAllowLocked 7-13
 mdlLocate_noElemNoLocked 7-13
 mdlLocate_normal 7-13
 mdlLocate_restart 7-16
 mdlLocate_setCursor 7-19
 mdlLocate_setElemSearchMask 7-14
 mdlLocate_setFunction 7-19
 mdlMaterial_cleanup 18-83
 mdlMaterial_initialize 18-83
 mdlMaterial_load 18-84
 mdlMaterial_useTable 18-84
 mdlMeasure_areaProperties 5-87
 mdlMeasure_elmDscrArea 5-85
 mdlMeasure_linearProperties 5-88
 mdlMeasure_polygonArea 5-85
 mdlMeasure_polygonProperties 5-89
 mdlMeasure_surfaceProperties 5-92
 mdlMeasure_volumeProperties 5-90
 mdlMemory_showHeap 28-75
 mdlMinDist_betweenElms 5-86
 mdlMline_create 12-3
 mdlMline_deleteBreak 12-7
 mdlMline_deletePoint 12-4
 mdlMline_extractPoints 12-5
 mdlMline_getBoundaryDescr 12-13
 mdlMline_getBreak 12-6
 mdlMline_getCapDef 12-9
 mdlMline_getElementDescr 12-12
 mdlMline_getInfo 12-3
 mdlMline_getLineDef 12-8
 mdlMline_insertBreak 12-7
 mdlMline_insertPoint 12-4
 mdlMline_joinCorner 12-15
 mdlMline_joinCross 12-15
 mdlMline_joinTee 12-14
 mdlMline_modifyPoint 12-5
 mdlMline_setCapDef 12-10
 mdlMline_setClosure 12-11
 mdlMline_setLineDef 12-8
 mdlMline_validate 12-13
 mdlModify_elementDescr 7-28
 mdlModify_elementMulti 7-27
 mdlModify_elementSingle 7-24
 mdlModify_freeGMap 7-29
 mdlOutput_command 2-98
 mdlOutput_commandU 2-96
 mdlOutput_error 2-98
 mdlOutput_errorU 2-96
 mdlOutput_keyin 2-98
 mdlOutput_keyinU 2-96
 mdlOutput_message 2-98
 mdlOutput_messageU 2-96
 mdlOutput_printf 2-99
 mdlOutput_prompt 2-98
 mdlOutput_promptU 2-96
 mdlOutput_rscfPrintf 2-101
 mdlOutput_rscPrintf 2-100
 mdlOutput_rscsPrintf 2-101
 mdlOutput_rscvfPrintf 2-101
 mdlOutput_rscvPrintf 2-100
 mdlOutput_rscvsPrintf 2-101
 mdlOutput_status 2-98
 mdlOutput_statusU 2-96
 mdlOutput_vprintf 2-99
 mdlParams_getActive 19-1
 mdlParams_getLock 19-6
 mdlParams_saveMasterLevelSymbology 19-5
 mdlParams_setActive 19-1
 mdlParams_setLock 19-6
 mdlParams_storeType9Variable 19-5
 mdlParse_changeTargetTask 2-70
 mdlParse_findTableByName 2-70

mdlParse_getTableName 2-70
 mdlParse_keyWord 2-69
 mdlParse_loadCommandTable 2-68
 mdlParse_loadCommandTableByNumber 2-70
 mdlParse_loadKeywordTable 2-69
 mdlParse_reconstructFullKeyin 2-71
 mdlParse_setFunction 2-72
 mdlParse_unloadTable 2-68
 mdlPattern_area 16-1
 mdlPattern_hatch 16-3
 mdlPattern_linear 16-4
 mdlPlot_execute 28-55
 mdlPlot_flushBuffer 28-55
 mdlPlot_getInfo 28-54
 mdlPlot_writeCommand 28-55
 mdlPlot_writeString 28-55
 mdlPointString_create 5-16
 mdlProject_perpendicular 5-80
 mdlProject_tangent 5-79
 mdlRastRef_attach 18-87
 mdlRastRef_detach 18-88
 mdlRastRef_extractReferences 18-89
 mdlRastRef_fileNameFromFileSpec 18-89
 mdlRastRef_fileSpecFromFileName 18-90
 mdlRastRef_freeImage 18-91
 mdlRastRef_updateAttachment 18-91
 mdlRefFile_attach 14-4
 mdlRefFile_attachByView 14-9
 mdlRefFile_attachCoincident 14-8
 mdlRefFile_createAttachment 14-10
 mdlRefFile_detach 14-13
 mdlRefFile_extendedAttach 14-6
 mdlRefFile_getParameters 14-15
 mdlRefFile_reload 14-14
 mdlRefFile_rotate 14-13
 mdlRefFile_scale 14-14
 mdlRefFile_setClip 14-12
 mdlRefFile_updateReference 14-15
 mdlRefFile_writeAttachment 14-12
 mdlRegion_floodFill 5-83
 mdlResource_add 2-52
 mdlResource_addByAlias 2-52
 mdlResource_changeAlias 2-66
 mdlResource_closeFile 2-44
 mdlResource_createFile 2-45
 mdlResource_createFileForPlatform 2-45
 mdlResource_delete 2-57
 mdlResource_deleteByAlias 2-57
 mdlResource_directAdd 2-54
 mdlResource_directAddComplete 2-56
 mdlResource_directLoad 2-49
 mdlResource_free 2-50
 mdlResource_getApplLoadFile 2-64
 mdlResource_getRscIdByAlias 2-65
 mdlResource_load 2-46
 mdlResource_loadByAlias 2-46
 mdlResource_loadFromStringList 2-48
 mdlResource_openFile 2-43
 mdlResource_query 2-58
 mdlResource_queryClass 2-59
 mdlResource_queryClassByAlias 2-59
 mdlResource_queryFile 2-61
 mdlResource_queryFileHandle 2-62
 mdlResource_resize 2-52
 mdlResource_write 2-51
 mdlRMatrix_from3Points 20-7
 mdlRMatrix_fromAngle 20-9
 mdlRMatrix_fromColumnVectors 20-8
 mdlRMatrix_fromNormalVector 20-11
 mdlRMatrix_fromQuat 20-9
 mdlRMatrix_fromRowVectors 20-8
 mdlRMatrix_fromTMatrix 20-10
 mdlRMatrix_fromView 20-7
 mdlRMatrix_getColumnVector 20-8
 mdlRMatrix_getIdentity 20-3
 mdlRMatrix_getInverse 20-4
 mdlRMatrix_getRowVector 20-8
 mdlRMatrix_invert 20-4
 mdlRMatrix_isIdentity 20-3
 mdlRMatrix_isOrthonormal 20-4
 mdlRMatrix_multiply 20-4
 mdlRMatrix_multiplyByTMatrix 20-6
 mdlRMatrix_normalize 20-6
 mdlRMatrix_rotate 20-5
 mdlRMatrix_rotatePoint 20-11
 mdlRMatrix_rotatePointArray 20-11
 mdlRMatrix_toAngle 20-10
 mdlRMatrix_toQuat 20-10
 mdlRMatrix_unrotatePoint 20-11
 mdlRMatrix_unrotatePointArray 20-11
 mdlScan_extended 7-9
 mdlScan_file 7-8
 mdlScan_initialize 7-4
 mdlScan_initOpenedFile 7-8
 mdlScan_initScanlist 7-2
 mdlScan_noRangeCheck 7-2

- mdlScan_restoreContext 7-11
- mdlScan_saveContext 7-10
- mdlScan_setDrawnElements 7-2
- mdlScan_singleViewClass 7-2
- mdlScan_viewRange 7-2
- mdlSelect_addElement 7-31
- mdlSelect_allElements 7-32
- mdlSelect_freeAll 7-32
- mdlSelect_handleDrawAll 7-33
- mdlSelect_handleDrawElement 7-33
- mdlSelect_isActive 7-30
- mdlSelect_removeElement 7-31
- mdlSelect_returnPositions 7-31
- mdlShape_create 5-4
- mdlShape_createI 5-5
- mdlSharedCell_addDefinitionElements 15-17
- mdlSharedCell_addToFile 15-16
- mdlSharedCell_create 15-14
- mdlSharedCell_createDefinitionElement 15-16
- mdlSharedCell_deleteDefinition 15-20
- mdlSharedCell_drawAllShares 15-21
- mdlSharedCell_dropOneLevel 15-19
- mdlSharedCell_dropToNormalCell 15-19
- mdlSharedCell_extract 5-25
- mdlSharedCell_find 15-18
- mdlSharedCell_fromCellHeader 15-14
- mdlSharedCell_makeSureDefExists 15-17
- mdlSharedCell_read 15-14
- mdlSharedCell_redefine 15-20
- mdlSharedCell_setRange 15-21
- mdlSharedCell_toNormalCell 15-21
- mdlState_checkSingleShot 1-6
- mdlState_clear 1-5
- mdlState_dynamicUpdate 1-13
- mdlState_exitViewCommand 1-11
- mdlState_registerStringIds 1-2, 1-13
- mdlState_restartCurrentCommand 1-11
- mdlState_setAccudrawContext 1-15
- mdlState_setFunction 1-14
- mdlState_setKeyinPrompt 1-13
- mdlState_startDefaultCommand 1-1
- mdlState_startDefaultCommand 1-5
- mdlState_startFenceCommand 1-1, 1-12
- mdlState_startModifyCommand 1-1, 1-8
- mdlState_startPrimitive 1-1, 1-6
- mdlState_startPrimitiveAndSetPopupMenu 1-7
- mdlState_startViewCommand 1-1, 1-10
- mdlString_fromAngle 28-22
- mdlString_fromDirection 28-22
- mdlString_fromPoint 28-21
- mdlString_fromUors 28-24
- mdlString_matchRE 28-24
- mdlString_setFunction 28-26
- mdlString_toAngle 28-21
- mdlString_toPoint 28-20
- mdlString_toUors 28-23
- mdlStringList_addResource 28-16
- mdlStringList_addResourceWithType 28-16
- mdlStringList_binarySearch 28-15
- mdlStringList_binarySearchByColumn 28-15
- mdlStringList_copy 28-8
- mdlStringList_create 28-4
- mdlStringList_deleteMember 28-8
- mdlStringList_destroy 28-4
- mdlStringList_getInfoField 28-7
- mdlStringList_getMember 28-5
- mdlStringList_insertMember 28-7
- mdlStringList_loadResource 28-18
- mdlStringList_loadResourceWithType 28-18
- mdlStringList_moveMembers 28-10
- mdlStringList_numInfoFields 28-9
- mdlStringList_search 28-13
- mdlStringList_searchByColumn 28-13
- mdlStringList_setInfoField 28-6
- mdlStringList_setMember 28-4
- mdlStringList_size 28-9
- mdlStringList_sort 28-12
- mdlStringList_sortByColumn 28-12
- mdlStringList_sortByIndex 28-10
- mdlStringList_writeResource 28-17
- mdlStringList_writeResourceWithType 28-17
- mdlSurface_createHeader 7-38
- mdlSurface_project 7-38
- mdlSurface_revolve 7-39
- mdlSystem 4-21, 4-22, 4-23, 4-24, 4-25, 4-26, 4-47, 4-48
- mdlSystem_abortRequested 4-11
- mdlSystem_CADInputJournalActive 4-20
- mdlSystem_cancelTimer 4-17
- mdlSystem_closeDesignFile 4-9
- mdlSystem_compressDgnFile 4-18
- mdlSystem_compressLibrary 4-18
- mdlSystem_computeDesignRange 4-13
- mdlSystem_createListFromCfgVarValue 4-38, 4-46
- mdlSystem_createStartupElement 4-7
- mdlSystem_defineCfgVar 4-43

mdlSystem_deleteCfgVar 4-44
 mdlSystem_deleteCfgVarAtLevel 4-44
 mdlSystem_deleteStartupElement 4-7
 mdlSystem_elapsedTime 4-20, 4-47
 mdlSystem_enterDebug 4-13
 mdlSystem_enterGraphics 4-7
 mdlSystem_enterGraphicsExtended 4-8
 mdlSystem_exchangeDesignFile 4-21
 mdlSystem_exit 4-5
 mdlSystem_expandCfgVar 4-42
 mdlSystem_extendedAbortEnable 4-12
 mdlSystem_extendedAbortRequested 4-12
 mdlSystem_fileDesign 4-9
 mdlSystem_findMdlDesc 4-14
 mdlSystem_flushDesignFile 4-11
 mdlSystem_flushWriteCache 4-18
 mdlSystem_getCfgVar 4-41
 mdlSystem_getCfgVarAtLevel 4-47
 mdlSystem_getCfgVarByIndex 4-42
 mdlSystem_getCfgVarLevel 4-43
 mdlSystem_getChar 4-9
 mdlSystem_getCurrMdlDesc 4-14
 mdlSystem_getCurrTaskID 4-9
 mdlSystem_getenv 4-40
 mdlSystem_getExpandedCfgVar 4-47
 mdlSystem_getMdlDescFromProcessNumber 4-16
 mdlSystem_getMdlTaskID 4-14
 mdlSystem_getMdlTaskList 4-10
 mdlSystem_getProcessNumberFromMdlDesc 4-16
 mdlSystem_getTaskStatistics 4-10
 mdlSystem_getTicks 4-10
 mdlSystem_getUstrnOSPid 4-15
 mdlSystem_getWorkspaceList 4-48
 mdlSystem_isCfgVarLocked 4-45
 mdlSystem_journalAccessStrByTaskId 4-22
 mdlSystem_journalAccessString 4-22
 mdlSystem_journalAppMessage 4-23
 mdlSystem_journalCommand 4-24
 mdlSystem_journalCommandString 4-24
 mdlSystem_journalDataPoint 4-25
 mdlSystem_journalKeyin 4-25
 mdlSystem_journalReset 4-26
 mdlSystem_journalTentativePoint 4-26
 mdlSystem_loadMdlProgram 4-6
 mdlSystem_lockCfgVar 4-45
 mdlSystem_newDesignFile 4-8
 mdlSystem_nextMdlApp 4-15
 mdlSystem_pauseTicks 4-10
 mdlSystem_processCfgVarFile 4-45
 mdlSystem_putenv 4-40
 mdlSystem_rewriteCfgVarFile 4-48
 mdlSystem_saveDesignFile 4-38
 mdlSystem_setFunction 4-19
 mdlSystem_setTimerFunction 4-17
 mdlSystem_startBusyCursor 4-18
 mdlSystem_stopBusyCursor 4-19
 mdlSystem_unloadMdlProgram 4-6
 mdlSystem_userAbortEnable 4-11
 mdlTag_addTagDefToSet 21-56
 mdlTag_attachURL 29-41
 mdlTag_create 21-58
 mdlTag_createInternetTagSet 29-41
 mdlTag_createSetDef 21-54
 mdlTag_deleteSetDef 21-54
 mdlTag_deleteTagDef 21-61
 mdlTag_deleteTagInstances 21-60
 mdlTag_extract 21-59
 mdlTag_extractURL 29-41
 mdlTag_freeTagDef 21-61
 mdlTag_freeTagDefArray 21-62
 mdlTag_generateReport 21-63
 mdlTag_getAssocElement 21-63
 mdlTag_getElementTags 21-62
 mdlTag_getSetDef 21-56
 mdlTag_getSetNames 21-55
 mdlTag_getTagDef 21-57
 mdlTag_hasInternetTagSet 29-41
 mdlTag_setSetDef 21-55
 mdlTag_setTagDef 21-58
 mdlText_addStringsToNodeDscr 8-6
 mdlText_addStringsToNodeDscrWide 28-67
 mdlText_changeElementFont 8-2
 mdlText_compressString 8-7
 mdlText_compressStringWide 28-68
 mdlText_countBufferStrings 8-9
 mdlText_create 5-10
 mdlText_createWide 28-61
 mdlText_expandString 8-7
 mdlText_expandStringWide 28-67
 mdlText_expandTabs 8-8
 mdlText_extract 5-20
 mdlText_extractFontStyle 8-3
 mdlText_extractShape 5-21
 mdlText_extractString 5-21
 mdlText_extractStringsFromDscr 8-5
 mdlText_extractStringsFromDscrWide 28-66

mdlText_extractStringWide 28-63
 mdlText_extractWide 28-62
 mdlText_fontExists 8-4
 mdlText_freeStringArray 8-10
 mdlText_getCurrentFont 8-2
 mdlText_getElementDescr 8-11
 mdlText_getFontInfo 8-4
 mdlText_getFontName 8-5
 mdlText_getStringArrayFromBuffer 8-10
 mdlText_loadFontStyle 8-3
 mdlText_nodeFromText 8-9
 mdlText_textFromNode 8-8
 mdlText_wordWrapBuffer 8-11
 mdlTextFile_close 25-29
 mdlTextFile_getString 25-29
 mdlTextFile_open 25-28
 mdlTextFile_putString 25-30
 mdlTextNode_create 5-12
 mdlTextNode_createWide 28-64
 mdlTextNode_createWithStrings 5-13
 mdlTextNode_createWithStringsWide 28-65
 mdlTextNode_extract 5-22
 mdlTextNode_extractShape 5-23
 mdlTextNode_extractWide 28-65
 mdlTMatrix_fromRMatrix 20-17
 mdlTMatrix_getIdentity 20-14
 mdlTMatrix_getTranslation 20-16
 mdlTMatrix_masterToReference 20-18
 mdlTMatrix_multiply 20-14
 mdlTMatrix_referenceToMaster 20-18
 mdlTMatrix_rotateByAngles 20-15
 mdlTMatrix_rotateByRMatrix 20-14
 mdlTMatrix_rotateScalePoint 20-17
 mdlTMatrix_scale 20-15
 mdlTMatrix_setOrigin 20-16
 mdlTMatrix_setTranslation 20-16
 mdlTMatrix_transformPoint 20-17
 mdlTMatrix_transformPointArray 20-17
 mdlTMatrix_translate 20-16
 mdlTMatrix_transpose 20-18
 mdlTransient_addElemDescr 5-164
 mdlTransient_addElement 5-164
 mdlTransient_free 5-167
 mdlTransient_replaceElemDescr 5-165
 mdlTransient_replaceElement 5-165
 mdlTransient_returnElemDescr 5-166
 mdlTutorial_load 28-53
 mdlTutorial_output 28-53
 mdlTutorial_positionInputField 28-53
 mdlTutorial_windowGet 28-54
 mdlUndo_endGroup 28-49
 mdlUndo_isActive 28-48
 mdlUndo_mark 28-48
 mdlUndo_redoActive 28-48
 mdlUndo_reset 28-49
 mdlUndo_setFunction 28-50
 mdlUndo_startGroup 28-49
 mdlUserPrefs_deleteStartupInfo 28-71
 mdlUserPrefs_getFileName 28-70
 mdlUserPrefs_loadStartupInfo 28-71
 mdlUserPrefs_save 28-69
 mdlUserPrefs_saveFileName 28-69
 mdlUserPrefs_saveStartupInfo 28-72
 mdlUtil_beep 28-74
 mdlUtil_quickSort 28-73
 mdlUtil_sortDoubles 28-73
 mdlUtil_sortLongs 28-73
 mdlUtil_sortStrings 28-73
 mdlVar_create 26-48
 mdlVar_destroy 26-51
 mdlVar_getVal 26-50
 mdlVar_hasNoValue 26-50
 mdlVar_isChanged 26-49
 mdlVar_isConstant 26-48
 mdlVar_isUnderDetermined 26-49
 mdlVar_restoreVal 26-51
 mdlVar_saveVal 26-51
 mdlVar_setConstant 26-48
 mdlVar_setUnderDetermined 26-49
 mdlVar_setVal 26-50
 mdlVar_use 26-51
 mdlVec_addPoint 20-22
 mdlVec_addPointArray 20-22
 mdlVec_areParallel 20-20
 mdlVec_arePerpendicular 20-20
 mdlVec_colinear 20-22
 mdlVec_computeNormal 20-23
 mdlVec_crossProduct 20-21
 mdlVec_distance 20-20
 mdlVec_dotProduct 20-21
 mdlVec_extractPolygonNormal 20-24
 mdlVec_forcePlanarity 20-25
 mdlVec_intersect 20-24
 mdlVec_linePlaneIntersect 20-26
 mdlVec_magnitude 20-20
 mdlVec_normalize 20-23

mdlVec_planePlaneIntersect 20-26
 mdlVec_pointEqual 20-22
 mdlVec_pointEqualUOR 20-22
 mdlVec_pointOffDesignPlane 20-25
 mdlVec_projectPoint 20-25
 mdlVec_projectPointToLineInView 20-27
 mdlVec_projectPointToPlaneInView 20-27
 mdlVec_scale 20-24
 mdlVec_subtractPoint 20-22
 mdlVec_subtractPointArray 20-22
 mdlVersion_getPlatform 28-75
 mdlVersion_getVersionNumbers 28-76
 mdlView_applyNamed 5-121
 mdlView_attachNamed 5-122
 mdlView_blitDirtyAreas 5-133
 mdlView_cameraLensAngleFromLength 5-112
 mdlView_cameraLensLengthFromAngle 5-112
 mdlView_clipToContent 5-134
 mdlView_clipToViewRect 5-18, 5-134
 mdlView_createSavedViewElement 5-124
 mdlView_defaultCursor 5-128
 mdlView_deleteNamed 5-123
 mdlView_extractSavedView 5-125
 mdlView_findNamed 5-122
 mdlView_fit 5-101
 mdlView_getCamera 5-106
 mdlView_getCameraParameters 5-110
 mdlView_getContentCorners 5-123
 mdlView_getDisplayControl 5-127
 mdlView_getLevels 5-126
 mdlView_getParameters 5-105
 mdlView_getStandard 5-121
 mdlView_getStandardCameraLens 5-108
 mdlView_getViewRectangle 5-135
 mdlView_hiddenLineRemoval 5-116, 5-119
 mdlView_hiddenLineRemoval2 5-119
 mdlView_indexFromWindow 5-136
 mdlView_isActive 5-129
 mdlView_isStandard 5-121
 mdlView_isVisible 5-129
 mdlView_newWindowCenter 5-131
 mdlView_pointToScreen 5-131
 mdlView_queuePartialUpdate 5-130
 mdlView_refreshGrid 5-129
 mdlView_renderSingle 5-115
 mdlView_rotateToRMatrix 5-105
 mdlView_saveNamed 5-123
 mdlView_screenToPoint 5-132
 mdlView_setActiveDepth 5-104
 mdlView_setActiveDepthPoint 5-104
 mdlView_setArea 5-102
 mdlView_setCameraLens 5-109
 mdlView_setCameraParameters 5-111
 mdlView_setDisplayControl 5-127
 mdlView_setDisplayDepth 5-103
 mdlView_setDisplayDepthPoints 5-103
 mdlView_setFunction 5-133
 mdlView_setLevels 5-126
 mdlView_setPopupMenu 5-133
 mdlView_setRenderMode 5-116
 mdlView_synchWithTCB 5-129
 mdlView_turnOff 5-100
 mdlView_turnOn 5-100
 mdlView_updateMulti 5-113
 mdlView_updateMultiExtended 5-114
 mdlView_updateSingle 5-113
 mdlView_zoom 5-102
 mdlWeb_getLastModified 29-43
 mdlWeb_getUrlFinish 29-43
 mdlWeb_getUrlProgress 29-42
 mdlWeb_getUrlToFileBegin 29-42
 mdlWeb_stopBrowser 29-43
 mdlWin32_createDataHandle 27-8
 mdlWin32_DdeAbandonTransaction 27-12
 mdlWin32_DdeAccessData 27-13
 mdlWin32_DdeAddData 27-14
 mdlWin32_DdeClientTransaction 27-14
 mdlWin32_DdeCmpStringHandles 27-17
 mdlWin32_DdeConnect 27-18
 mdlWin32_DdeConnectList 27-19
 mdlWin32_DdeCreateDataHandle 27-20
 mdlWin32_DdeCreateStringHandle 27-21
 mdlWin32_DdeDisconnect 27-22
 mdlWin32_DdeDisconnectList 27-23
 mdlWin32_DdeEnableCallback 27-23
 mdlWin32_DdeFreeDataHandle 27-24
 mdlWin32_DdeFreeStringHandle 27-25
 mdlWin32_DdeGetData 27-26
 mdlWin32_DdeGetLastError 27-27
 mdlWin32_DdeInitialize 27-29
 mdlWin32_DdeKeepStringHandle 27-32
 mdlWin32_DdeNameService 27-33
 mdlWin32_DdePostAdvise 27-34
 mdlWin32_DdeQueryConvInfo 27-35
 mdlWin32_DdeQueryNextServer 27-36
 mdlWin32_DdeQueryString 27-36

- mdlWin32_DdeReconnect 27-37
- mdlWin32_DdeSetUserHandle 27-37
- mdlWin32_DdeUnaccessData 27-38
- mdlWin32_DdeUninitialize 27-39
- mdlWin32_destroyDataHandle 27-9
- mdlWin32_GetLastError 27-7
- mdlWin32_readDataFromHandle 27-8
- mdlWin32_writeDataToHandle 27-10
- mdlWindow 2-17
- mdlWindow_arcDraw 2-25
- mdlWindow_backgroundCDGet 2-34
- mdlWindow_backgroundColorSet 2-33
- mdlWindow_changeScreen 2-7
- mdlWindow_close 2-6
- mdlWindow_colorIndexGet 2-32
- mdlWindow_contentRectGetGlobal 2-12
- mdlWindow_contentRectGetLocal 2-12
- mdlWindow_cursorTurnOff 2-4
- mdlWindow_displayDescrGet 2-11
- mdlWindow_dockWindow 2-37
- mdlWindow_ellipseDraw 2-25
- mdlWindow_ellipseFill 2-25
- mdlWindow_extentSet 2-6
- mdlWindow_fixedColorIndexGet 2-32
- mdlWindow_float 2-5
- mdlWindow_flush 2-4
- mdlWindow_getDocked 2-38
- mdlWindow_getFirst 2-9
- mdlWindow_getInputFocus 2-17, 2-40
- mdlWindow_getLast 2-9
- mdlWindow_getNext 2-9
- mdlWindow_getPrevious 2-9
- mdlWindow_globalRectGetGlobal 2-12
- mdlWindow_globalRectGetLocal 2-12
- mdlWindow_globalToContentRect 2-14
- mdlWindow_hide 2-6
- mdlWindow_iconDraw 2-29
- mdlWindow_isDialogBox 2-10
- mdlWindow_isDisplayed 2-10
- mdlWindow_isDockable 2-39
- mdlWindow_isPopUp 2-16
- mdlWindow_isTrueColorCapable 2-33
- mdlWindow_isView 2-15
- mdlWindow_isVisible 2-10
- mdlWindow_lineDraw 2-22
- mdlWindow_lineStringDraw 2-22
- mdlWindow_lineStyleSet 2-21
- mdlWindow_lineStyleSetCD 2-22
- mdlWindow_maximize 2-8
- mdlWindow_minimize 2-8
- mdlWindow_nativeWindowHandleGet 2-13
- mdlWindow_organizeApplicationArea 2-39
- mdlWindow_pointDraw 2-22
- mdlWindow_pointToGlobal 2-12
- mdlWindow_pointToLocal 2-12
- mdlWindow_rasterDataDraw 2-30
- mdlWindow_rectClear 2-27
- mdlWindow_rectDraw 2-27
- mdlWindow_rectFill 2-27
- mdlWindow_rectFillByRGB 2-27
- mdlWindow_rectInvert 2-28
- mdlWindow_resize 2-6
- mdlWindow_restore 2-8
- mdlWindow_rgbDataDraw 2-31
- mdlWindow_screenNumGet 2-11
- mdlWindow_setFunction 2-16
- mdlWindow_setInputFocus 2-15, 2-17
- mdlWindow_shapeFill 2-22
- mdlWindow_show 2-6
- mdlWindow_sink 2-5
- mdlWindow_systemColorGet 2-34
- mdlWindow_textDraw 2-24
- mdlWindow_textDrawCD 2-25
- mdlWindow_titleGet 2-4
- mdlWindow_titleSet 2-4
- mdlWindow_toBack 2-5
- mdlWindow_toFront 2-5
- mdlWindow_transparentRasterDataDraw 2-35, 2-40
- mdlWindow_transparentRgbDataDraw 2-36
- mdlWindow_undockWindow 2-39
- mdlWindow_updateAllWindows 2-7
- mdlWindow_viewWindowGet 2-11
- mdlWindow_windowEventsProcessAll 2-9
- mdlWorkDgn_delete 25-16
- mdlWorkDgn_findEof 25-16
- mdlWorkDgn_igdsSize 25-17
- mdlWorkDgn_read 25-14
- mdlWorkDgn_validate 25-17
- mdlWorkDgn_write 25-15
- measurement functions 5-84
 - area of polygon 5-85
 - distance between elements 5-86
 - perimeter of closed element 5-85
 - perimeter of polygon 5-85
 - principal moments and axes 5-87
 - surface properties 5-92

- table of functions 5-84
 - uses of 5-84
 - volume properties 5-90
- memory management functions 27-7
 - error codes 27-7
 - extracting data from memory handle 27-8
 - freeing memory resources 27-9
 - memory allocation 27-8
 - table of functions 27-7
- menu bar item functions 3-2
 - activating menu bar in command window 3-3
 - adding application menu to menu bar 3-4
 - adding menu to Command Window menu bar 3-5
 - adding menu to menu bar 3-5
 - deleting menu items 3-16
 - inserting menus into specified menu bar 3-15
 - last selected menu item 3-12
 - menu bar item creation 3-4
 - pointer to applications menu 3-7
 - pointer to header of Command Window menu bar 3-7
 - pointer to menu bar 3-6
 - pointer to menu items 3-9
 - pointer to pull down menu 3-10, 3-11
 - quantity of menu items within menu 3-12
 - quantity of pull down menus within menu bar 3-11
 - removing applications menu item and menu bar 3-7
 - removing menu from Command Window menu bar 3-5
 - removing menu from menu bar 3-6
 - removing menus from specified menu bar 3-13, 3-14
 - search ID of menu item 3-8
 - setting default Command Window menu bar 3-7
 - state of menus 3-17
 - title of menus 3-16
 - uses of 3-2
- menu bars 3-2
- menus 3-2
- message display functions 2-95
 - displaying in command window fields 2-96
 - displaying MessageList resource strings 2-100
 - displaying printf format strings 2-99
 - table of functions 2-95
 - uses of 2-95
- message lists 1-2
- message queues 22-1
- MicroCSL signal handler 22-9
- MicroStation color manager 17-2
 - displaying colors 17-2
- MicroStation cursor 7-19
- MicroStation elements 5-17
 - calculating intersections between 5-93
 - complex elements 5-45
 - location of 7-12
 - manipulation of 5-45
 - primitive elements 5-45
 - storage of 5-45
 - type 66 5-28
- middle_endian format 24-6
- miscellaneous dialog box functions 3-146
 - calling MDL application functions 3-148
 - displaying diagnostic information 3-148
 - displaying status messages 3-147
 - mdlDialog_keyinWindowGet 3-156
 - mdlDialog_openMessageBox 3-151
 - mdlDialog_overallTitleBarGet 3-152
 - mdlDialog_statusAreaGet 3-152
 - obtaining resource file handle 3-148
 - resource structure members, obtaining 3-149
 - setting label item attributes 3-147
- miscellaneous dialog functions 3-146
- miscellaneous element functions 5-27
 - adding new elements to dgn file 5-29
 - appending attribute information 5-32
 - creating type 5 color elements 5-37
 - deleting elements 5-31
 - design file format of elements 5-40
 - determining element fill status 5-39
 - different element formats 5-33
 - displaying elements 5-30
 - extracting attribute information 5-32
 - extracting color of element 5-36
 - extracting element properties 5-38
 - extracting style of element 5-36
 - extracting weight of element 5-36
 - internal format of elements 5-40
 - line style of element 5-42, 5-43
 - obtaining file position of elements 5-34
 - overwriting existing elements 5-29
 - reading elements from dgn file 5-34
 - removing attribute information 5-33
 - setting file position 5-35

- setting fill colors 5-38
- table of functions 5-27
- transformation matrix 5-41
- transforming elements 5-41
- translating elements 5-41
- type 66 elements 5-44
- uses of 5-27
- vectorizing elements 5-41
- miscellaneous utilities functions 28-73
 - beeps 28-74
 - obtaining ID of platform 28-75
 - obtaining name of platform 28-75
 - obtaining version current executable 28-76
 - quick sort of an array 28-73
- monochrome images 18-1
- multi-line
 - line definition 12-1
 - multi-line profile 12-1
- multi-line element
 - profile 12-1
- multi-line elements
 - breaks 12-1
 - caps 12-1
 - work line 12-1
- multi-line functions 12-1
 - changing definition of line 12-8
 - creating cross joints 12-15
 - creating multi-line elements 12-3
 - creating tee joints 12-14
 - deleting breaks from multi-line 12-7
 - deleting vertex from multi-line element 12-4
 - extracting vertices from multi-line 12-5
 - inserting breaks in multi-line 12-7
 - inserting point in multi-line 12-4
 - multi-line elements 12-1
 - multi-line to IGDS primitive 12-12
 - replacing multi-line vertex 12-5
 - representing boundary of multi-line element 12-13
 - requested break information 12-6
 - retrieving definition of multi-line cap 12-9
 - retrieving information from multi-lines 12-3
 - retrieving multi-line definitions 12-8
 - setting cap definition 12-10
 - setting multi-line range block 12-13
 - setting the closure status 12-11
 - table of functions 12-1
 - uses of 12-1

- multi-line profile 12-1

N

- non-graphical data 21-1
 - database interface functions 21-2
 - database settings functions 21-2
 - SQL functions 21-2
 - tag functions 21-2

O

- opaque structure
 - see window management functions*
- option button item functions 3-27
 - computing extents of option button item 3-29
 - deleting option button sub-items 3-34
 - inserting sub-item into option button item 3-32
 - quantity of sub-items in option button item 3-28
 - retrieving information on option button item 3-29
 - setting state of option button item 3-28
 - table of functions 3-27
 - uses of 3-27
- option pull-down menu functions 3-23
 - inserting menu item into menus 3-26
 - option pull-down menu information 3-25
 - setting state of menu item 3-24
 - table of functions 3-24
 - uses of 3-23
- orthonormal matrices 20-1

P

- parse functions 2-67
 - constructing keyin for specific command 2-71
 - creating MicroStation queue elements 2-72
 - loading command tables 2-68, 2-70
 - loading parse trees from resource files 2-69
 - parsing strings in parse table 2-69
 - setting target task ID 2-70
 - specifying parse user functions 2-72
 - table of functions 2-67
- patterning functions 16-1
 - area patterning 16-1
 - associative patterning functions 16-1
 - basic patterning functions 16-1
 - hatching 16-3

- linear patterning 16-4
- table of functions 16-1
- plotting functions 28-54
 - creating a plotfile 28-55
 - overview 28-54
 - reading plotter configuration file 28-54
 - writing ASCII data to plotfile 28-55
 - writing plot commands to plotfile 28-55
- point-like *see constraint functions, specific terms*
- preFunction 5-117
- primitive commands 1-1
 - examples of 1-1
 - initialization of 1-1
 - suspension of by view commands 1-1
 - termination of 1-1
 - uses of 1-1
- primitive elements 5-45
 - format header 5-45
- program 4-1
- publishing symbols 2-102
- publishing variables 3-91
- push button item functions 3-66
 - activating push button 3-66
 - cancel status of push button 3-67
 - default status of push button, changing 3-66
 - push button information 3-68
 - table of functions 3-66
 - uses of 3-66
- push button item information
 - push button information 3-68

R

- Radix50 string, defined 24-3
- rail curves 9-33
- raster image functions 18-1
 - aborting imaging tasks 18-26
 - accurately displaying RGB images 18-17
 - animation files 18-27, 18-29
 - color mode of image file 18-11
 - creating image files from mapped image 18-15
 - creating image files from RGB images 18-16
 - dithering RGB images 18-16
 - flickering problems 18-29
 - gamma values 18-20, 18-25
 - image file extension 18-11
 - import and export concerns 18-12
 - import and exports concerns 18-12

- mapped images of portions of graphics
 - screen 18-20
- mapped images to RGB images 18-22
- mdllImage_byteMapToBitMap 18-31
- mdllImage_byteMapToGreyScale 18-32
- mdllImage_captureScreen 18-33
- mdllImage_completeMovie 18-34
- mdllImage_computeScreenMap 18-35
- mdllImage_convertToUpperLeftHorizontal 18-36
- mdllImage_createFileFromBitMap 18-37
- mdllImage_createFileFromBuffer 18-38
- mdllImage_createMovie 18-41
- mdllImage_displayDescrGet 18-41
- mdllImage_doubleImage 18-42
- mdllImage_extMapToRGB 18-43
- mdllImage_extractBitMapSubImage 18-44
- mdllImage_extractByteMapSubImage 18-45
- mdllImage_extractIngrAttach 18-46
- mdllImage_extractPackByteSubImage 18-47
- mdllImage_extractSubImage 18-49
- mdllImage_extReadFileToMap 18-50
- mdllImage_extRGBToMap 18-51
- mdllImage_extRGBToScreenMap 18-52
- mdllImage_freeImage 18-53
- mdllImage_getMapUsage 18-54
- mdllImage_greyScaleToBitMap 18-55
- mdllImage_isAVIAvailable 18-56
- mdllImage_memorySize 18-56
- mdllImage_mirror 18-57
- mdllImage_packByteBuffer 18-58
- mdllImage_packByteToBitMap 18-59
- mdllImage_packByteToGreyScale 18-60
- mdllImage_paletteToGreyScale 18-61
- mdllImage_readFileToBitMap 18-61
- mdllImage_remapToScreenPalette 18-63
- mdllImage_rgbToBitMap 18-64
- mdllImage_rgbToGreyScale 18-65
- mdllImage_RGBToPackByte 18-65
- mdllImage_RGBToPackByteWithGamma 18-67
- mdllImage_rotate 18-68
- mdllImage_runLengthEncodeBitMap 18-69
- mdllImage_runLengthEncodeBitMapRow 18-70
- mdllImage_setMapIfRGBMatch 18-71
- mdllImage_setMapPolygon 18-71
- mdllImage_setRGBPolygon 18-73
- mdllImage_stretchRLEToBitMap 18-74
- mdllImage_subByteMapFromBitMap 18-75
- mdllImage_subByteMapFromPackByte 18-76

- mdlImage_subByteMapFromRLEBitMap 18-77
- mdlImage_tintImage 18-78
- mdlImage_tintPalette 18-79
- mdlImage_typeFromFile 18-79
- mdlImage_unpackByteBuffer 18-80
- mdlImage_updateIngrAttach 18-81
- mdlImage_warp 18-81
- movies 18-28
- negated color palettes 18-21
- negated RGB images 18-21
- orientation of image file 18-11
- raster images from RGB images 18-25
- reading image files 18-14
- reading mapped image files 18-13
- rendering views to RGB buffer 18-26
- RGB images to mapped images 18-23
- size of image file 18-11
- smoothly shaded images 18-19
- unmapped RGB data to color palettes 18-18
- raster image resizing images 18-24
- Raster Reference Functions
 - mdlRastRef_detach 18-88
 - mdlRastRef_extractReferences 18-89
 - mdlRastRef_fileNameFromFileSpec 18-89
 - mdlRastRef_fileSpecFromFileName 18-90
 - mdlRastRef_freeImage 18-91
 - mdlRastRef_updateAttachment 18-91
- rcomp *see resource compiler*
- rectangle drawing functions 3-121
 - beveled edges 3-124
 - comparing rectangles 3-126
 - determining dimensions 3-126
 - determining extent of 3-126
 - determining intersection 3-127
 - determining offsets 3-125
 - filling rectangles 3-123
 - inverting rectangle color 3-124
 - outlining rectangle 3-122
 - table of functions 3-121
- reference file functions 14-1
 - attaching reference files 14-4
 - changing attachment settings 14-15
 - detaching reference files 14-13
 - master files 14-8
 - modifying reference file attachments 14-12
 - multiple extensions 14-6
 - reloading reference files 14-14
 - retrieving attachment settings 14-15
 - rotating attached reference files 14-13
 - saved views 14-9
 - scaling reference files 14-14
 - setting clipping polygon 14-12
 - table of functions 14-1
 - updating reference files 14-15
- reference files 14-1
 - attachment parameters 14-2
 - coordinates 14-2
 - uses of 14-1
- rendering and imaging 18-1
- rendering modes
 - CONSTANT 5-115
 - PHONG 5-115
 - SMOOTH 5-115
- resource compiler 2-42
- resource files 1-2, 2-41
 - portability 2-41
- resource management
 - alias names 2-43
 - binary portable resource files 2-41
- resource management functions 2-40
 - adding resource to resource file 2-52
 - assigning alias names to resources 2-66
 - closing resource files 2-44
 - creating resource files 2-45
 - deallocating memory for resources 2-50
 - deleting resources by resource ID 2-57
 - loading resources from resource files 2-46
 - obtaining resource file names 2-64
 - obtaining resource ID 2-65
 - opening resource files 2-43
 - querying resource manager 2-58, 2-59, 2-61, 2-62
 - resource loading control 2-49
 - resource manager, informing of added resources 2-56
 - size of resource file, adjusting 2-52
 - strings, copying from string lists 2-48
 - table of functions 2-40
 - uses of 2-40
 - writing resources to resource files 2-54
 - writing to resource files, updating 2-51
- resourceclass statement 2-42
- RGB (Red-Green-Blue) 17-26
- RGB conversion functions 17-23
 - blending RGB values 17-27
 - color manager index 17-24

- color number from RGB value 17-25
 - HSV to RGB 17-27
 - luminosity 17-25
 - RGB values of colortable 17-24
- RGB conversion values
 - HSV color definition 17-26
- RGB images *see unmapped images*
- rotation matrix functions 20-1
 - create from vectors 20-8
 - determining orthonormality 20-4
 - extracting rotation from transformation matrix 20-10
 - generating quaternion matrix 20-10
 - generation from angle 20-9
 - generation from Z-axis direction vector 20-11
 - identity rotation matrix 20-3
 - multiplying matrices 20-4
 - normalizing columns of rotation matrix 20-6
 - obtaining identity matrix 20-3
 - obtaining scaling from transformation matrix 20-10
 - rotating an array of points 20-11
 - rotating array of points by inverse 20-11
 - rotating matrix 20-5
 - rotating points 20-11
 - rotating points by inverse 20-11
 - rotation matrix from three points 20-7
 - transformation matrix multiplication 20-6
 - transpose 20-4
- routines 5-93
- rsctype utility 2-102
- rubber banding *see dynamics*
- runtime environment 4-1

S

- scan functions 7-1
 - establishing scan criteria 7-2
 - initializing scan list 7-2
 - loading scan lists 7-4
 - restoring scanner state 7-11
 - returning control to scanning process 7-10
 - scanning design file 7-8
 - scanning the design file 7-9
 - table of functions 7-1
 - zeroing scan list 7-8
- screen capture utility 18-20
- SCRNCAPT *see screen capture utility*

- scroll bar item functions 3-64
 - range information, setting 3-65
 - scroll bar item information 3-64
 - table of functions 3-64
 - uses of 3-64
- scroll bar items 3-64
- search masks 7-12
- Sectioning and Hidden line viewing functions 5-142
 - mdlHview_clearView 5-142
 - mdlHview_closeContext 5-143
 - mdlHview_eraseAndRedrawView 5-143
 - mdlHview_message 5-144
 - mdlHview_openContext 5-149
 - mdlHview_openFile 5-150
 - mdlHview_processView 5-150
 - mdlHview_setSectionPlanes 5-151
 - table of functions 5-142
- selection set processing functions 7-30
 - adding displayable elements to selection set 7-32
 - adding element to selection set 7-31
 - determining any selected elements 7-30
 - drawing handles on selection set elements 7-33
 - drawing handles on single element 7-33
 - element positions 7-31
 - removing element from selection set 7-31
 - table of functions 7-30
 - uses of 7-30
- selection sets 7-30
- settings functions 19-1
 - level functions 19-1
 - settings manipulation functions 19-1
- settings manipulation functions
 - setting values 19-1
 - storing single Type 9 elements 19-5
 - storing Type 10 elements 19-5
 - uses of 19-1
- shared cell functions 15-13
 - adding shared cell instance 15-16
 - definition elements 15-16
 - deleting shared cell definition 15-20
 - drawing mode 15-21
 - element descriptor 15-14
 - nested shared cells 15-19
 - range block 15-21
 - searching shared cell definitions 15-18
 - shared cell definition 15-17, 15-20
 - shared cell instance element 15-14

- unshared cell equivalents 15-21
- shared memory 22-1
 - DDE (Dynamic Data Exchange) 27-1
- SIGALRM signal 22-10
- SIGQUIT signal 22-9
- SIGUSR1 signal 22-10
- silhouette curves 9-23
- solver variable functions 26-46
 - freeing variable memory 26-51
 - initializing a Var structure 26-48
 - overview 26-46
 - querying constant attribute of Var 26-48
 - querying under-determined attribute of Var 26-49
 - saving value of Var 26-51
 - setting constant attribute of Var 26-48
 - setting under-determined attribute of Var 26-49
 - testing current value of Var 26-49
 - Var structure 26-46
- solver variables 26-47
- SQL 21-1
- SQL functions 21-20
 - adding new rows 21-28
 - adding row to table 21-28
 - creating new table 21-30
 - database server description 21-29
 - defining database requests 21-32
 - deleting rows from table 21-23
 - extracting database linkages 21-30
 - inserting new rows 21-28
 - interface error codes 21-24
 - list of user accessible tables 21-31
 - memory deallocation 21-27
 - opening cursor for SQL queries 21-26
 - opening cursor for SQL query 21-26
 - opening multiple cursors simultaneously 21-29
 - reading columns from table rows 21-22
 - returning column descriptors 21-31
 - row insertion using screen form 21-28
 - SQL queries 21-23
 - submitting non-SELECT SQL statements 21-24
 - table structures 21-25
 - updating columns in table rows 21-22
- stacked edges *see coincident edges*
- state control functions 1-4
 - AccuDraw 1-15
 - clean user function 1-21
 - command cleanup user function 1-18
 - complex dynamic update user function 1-20
 - data point user function 1-19
 - designating user events functions 1-14
 - determining single-shot mode 1-6
 - dynamic user function 1-20
 - exiting view commands 1-11
 - fence content user function 1-22
 - fence outline user function 1-22
 - initializing primitive commands 1-12
 - key-in user function 1-19
 - modifying elements 1-8
 - reset user function 1-19
 - resetting state to "no command active" 1-5
 - restarting primitive commands 1-11
 - setting pop-up menus 1-7
 - show user function 1-21
 - simple dynamics, specifying functions 1-13
 - specifying prompt strings 1-13
 - specifying resource IDs when loading strings 1-13
 - starting default command 1-5
 - starting primitive commands 1-6
 - starting view commands 1-10
 - table of functions 1-4
- string functions
 - convert string to UOR 28-23
 - converting angle from string 28-21
 - converting angle to string 28-22
 - coordinates of string 28-20, 28-21
 - formatting string as a direction 28-29
 - formatting string as an angle 28-30
 - interpretation of user-entered strings 28-26
 - overview 28-19
 - string from UOR conversion 28-28
 - string search within string 28-24
 - string to UOR conversion 28-27
- string list functions
 - adding string lists to resource files 28-16, 28-17
 - copying string list members 28-8
 - creating new string list members 28-7
 - creation of string list 28-4
 - deallocation of memory 28-4
 - deleting string list members 28-8
 - initializing string list members 28-4, 28-6
 - loading string lists 28-18
 - moving string list members 28-10
 - number of information fields 28-9
 - number of string list members 28-9

- obtaining information of string list
 - members 28-5, 28-7
 - searching of string lists 28-13, 28-15
 - sorting string list members 28-10, 28-12
- string list manager 28-1
 - overview 28-1
 - string lists defined 28-1
- string lists 3-41, 28-2
 - example of 28-2
 - resources 28-2
- surface creation functions 7-37
 - creating complex surface header elements 7-38
 - creating surface of projection 7-38
 - creating surface of revolution 7-39
 - table of functions 7-37
 - uses of 7-37
- symbol sets 2-102
 - allocating 2-102
 - removing 2-102
 - visibility flags 2-102
- synchronization 3-94, 3-105
- synonyms 3-94
- System Functions
 - mdlSystem_createListFromCfgVarValue 4-38
 - mdlSystem_getCfgVarAtLevel 4-47
 - mdlSystem_saveDesignFile 4-38
- system functions 4-1
 - configuration variable functions 4-1
 - system functions 4-1

T

- Tab Page List and Tab Page Functions
 - mdlDialog_tabPageFreeItems 3-157
 - mdlDialog_tabPageGetInfo 3-157
 - mdlDialog_tabPageGetItemByIndex 3-157
 - mdlDialog_tabPageGetItemByTypeAndId 3-157
 - mdlDialog_tabPageListFreePages 3-157
 - mdlDialog_tabPageListGetInfo 3-157
 - mdlDialog_tabPageListGetPageById 3-157
 - mdlDialog_tabPageListGetPageByIndex 3-157
 - mdlDialog_tabPageListLoadPages 3-157
 - mdlDialog_tabPageListSetInfo 3-157
 - mdlDialog_tabPageLoadItems 3-157
 - mdlDialog_tabPageSetInfo 3-157
- Tag Functions 21-63
 - mdlTag_getAssocElement 21-63
- tag functions 21-52
 - adding new tag definition 21-56
 - changing tag definitions 21-58
 - creating definition for tag sets 21-54
 - creating tag elements 21-58
 - deleting tag instances 21-60
 - deleting tag set definitions 21-54
 - extracting tag element information 21-59
 - freeing array of tag definitions 21-62
 - generating ASCII report files 21-63
 - obtaining base element from tag 21-63
 - retrieving array of tag definitions 21-56
 - retrieving sorted string lists 21-55
 - retrieving tag definitions 21-57
 - table of functions 21-53
 - updating definitions of tag sets 21-55
 - uses of 21-52
- tag set libraries 21-52
- TagDef structure *see tags, structures*
- tags 21-52
 - attaching tags to elements 21-52
 - naming conventions 21-52
 - structures 21-52
 - tag sets 21-52
 - type 37 MicroStation element 21-52
 - type 66 MicroStation elements 21-52
- TagSetSpec structure *see tags, structures*
- TagSpec structure *see tags, structures*
- TagValue structure *see tags, structures*
- task 4-1
- TCB (Terminal Control Block) 4-8
- text file functions 25-27
 - closing a file 25-29
 - opening text file 25-28
 - overview 25-27
 - reading characters from stream 25-29
 - table of functions 25-28
 - writing characters to stream 25-30
- text item functions 3-56
 - attribute information of text items 3-57
 - byte offset 3-61
 - cursor position of multi-line text elements 3-60
 - first row determination in text item display 3-63
 - maximum value strings 3-57
 - minimum value strings 3-57
 - multi-line text elements, byte indices 3-62
 - scroll bar item information 3-59
 - string insertion in text item 3-63
 - table of functions 3-56

- uses of 3-56
 - text items 3-56
 - text pull-down menu functions 3-18
 - inserting menu item into menu 3-21
 - menu item information 3-20
 - setting label of menu item 3-19
 - table of functions 3-18
 - uses of 3-18
 - text utility functions 8-1
 - adding text strings to text node 8-6
 - current font 8-2
 - descriptive font names 8-5
 - existence of font 8-4
 - font loading 8-1
 - font style of text element 8-3
 - loading library font 8-3
 - memory deallocation 8-10
 - modifying text element font 8-2
 - multi-line text buffer 8-10
 - number of text lines in buffer 8-9
 - properties of font 8-4
 - special characters 8-7
 - string compression with special characters 8-7
 - tab expansion 8-8
 - table of functions 8-1
 - text element from text node element 8-8
 - text node element from text element 8-9
 - text strings 8-5
 - uses of 8-1
 - word wrapping considerations 8-11
 - This 2-1
 - toggle button item functions 3-69
 - table of functions 3-69
 - toggle button item information 3-70
 - track bar window functions 3-136
 - closing track bar window 3-138
 - initializing processing environment 3-136
 - table of functions 3-136
 - updating control information 3-138
 - updating track bar window 3-139
 - transformation matrix 5-155
 - current transformation 5-155
 - rotation 5-155
 - scaling 5-155
 - translation 5-155
 - transformation matrix functions 20-12
 - rotating points 20-17
 - rotating transformation matrix 20-14, 20-15
 - scaling points 20-17
 - scaling transformation matrix 20-15
 - setting to identity matrix 20-14
 - setting to product of matrix mult. 20-14
 - shifting origin of 20-16
 - transforming array of points 20-17
 - transforming point 20-17
 - transpose of 20-18
 - uses of 20-12
 - transient element functions 5-163
 - adding element to transient descriptor 5-164
 - replacing contents of existing transient descriptor 5-165
 - table of functions 5-163
 - uses of 5-163
 - truehide.dll* 5-119
 - tutorial functions 28-52
 - displaying messages 28-53
 - loading designated tutorials 28-53
 - overview 28-52
 - tutorial window pointer 28-54
- ## U
- UDLS (User Defined Line Style) 11-11
 - undo functions 28-47
 - adding elements to undo buffer 28-50
 - clear undo buffer 28-49
 - determine if element resides in redo buffer 28-48
 - determine if element resides in undo buffer 28-48
 - grouping modifications as units 28-49
 - set a mark in undo buffer 28-48
 - undo buffer event processing 28-50
 - UNIX, external program concerns
 - debugging external programs 22-9
 - mdlExternal_terminateProgram 22-9
 - MicroCSL signal handler 22-9
 - portability issues 22-8
 - signal handling 22-9
 - unmapped images 18-1
 - User Functions
 - userShare_sharedLibNoMoreClients 4-36
 - userSystem_fenceChanged 4-37
 - userSystem_save 4-37
 - user interface 2-1
 - C expression handling functions 2-1

- input handling functions 2-1
- message display functions 2-1
- parse functions 2-1
- resource management functions 2-1
- window docking functions 2-1
- window drawing functions 2-1
- window management functions 2-1
- user preference functions 28-68
 - deleting startup information resource 28-71
 - loading startup resource information 28-71
 - obtaining file search parameters 28-70
 - saving file search parameters 28-69
 - saving preferences to disk 28-69
- userBar 3-145
- userBar_cancelFunction 3-144
- userBar_completionFunction 3-145
- userBar_workFunction 3-145
- userBspline_curveFunction 9-28
- userBspline_surfaceFunction 9-30
- userClipboard_copyProcess 27-56
- userClipboard_copyStrategy 27-54
- userClipboard_cutProcess 27-56
- userClipboard_cutStrategy 27-54
- userClipboard_pasteProcess 27-56
- userClipboard_pasteStrategy 27-54
- userDigitize_getTabletPoint 23-3
- userDigitize_setup 23-4
- userDigitize_transform 23-5
- userDigitize_uorInput 23-5
- userExternal_messageReceived 22-23
- userExternal_programTerminated 22-23
- userFileDragAndDrop_handleDrop 27-44
- userHelp_showMeFunction 28-43
- userInput_commandFilter 2-94
- userInput_monitor 2-93
- userInput_preprocessKeyin 2-92
- userInput_receive 2-94
- userLocate_elementFilter 7-20
- userLocate_globalFilter 7-21
- userLocate_selectCmd 7-22
- userParse_handleString 2-72
- userShare_sharedLibNoMoreClients 4-36
- userState_clean 1-21
- userState_commandCleanup 1-18
- userState_complexDynamicUpdate 1-2, 1-20
- userState_datapoint 1-19
- userState_dynamicUpdate 1-2, 1-20
- userState_fenceContent 1-22
- userState_fenceOutline 1-22
- userState_keyin 1-19
- userState_reset 1-19
- userState_show 1-21
- userString_fromAngle 28-30
- userString_fromDirection 28-29
- userString_fromUors 28-28
- userString_toAngle 28-31
- userString_toUors 28-27
- userSystem_allMDLUnloads 4-34
- userSystem_colorMapChange 4-32
- userSystem_elmDscrToFile 4-31
- userSystem_exitDesignFileState 4-35, 22-3
- userSystem_fenceChanged 4-37
- userSystem_mdclChildTerminated 4-27
- userSystem_menuBarChange 4-33
- userSystem_newDesignFile 4-29
- userSystem_referenceAttach 4-33
- userSystem_reloadProgram 4-27
- userSystem_save 4-37
- userSystem_saveAs 4-30
- userSystem_timerExpired 4-29
- userSystem_unloadProgram 4-28
- userSystem_writeToFile 4-30
- userUndo_addToBuffer 28-50
- userUndo_command 28-51
- userView_drawCursor 5-141
- userView_motion 5-138
- userView_noMotion 5-138
- userView_plot 5-140
- userView_update 5-136
- userView_updateEachElement 5-139
- userWin32_DdeUserCallback 27-39
- userWindow_modifyEvents 2-18
- userWindow_scrollEvents 2-19
- utilities
 - BASIC interface functions 28-1
 - dynamic array functions 28-1
 - function key functions 28-1
 - help functions 28-1
 - license management functions 28-1
 - miscellaneous functions 28-1
 - plotting functions 28-1
 - string functions 28-1
 - string list functions 28-1
 - tutorial functions 28-1
 - undo functions 28-1
 - user preferences functions 28-1

- wide text functions 28-1
- utility commands 1-1
 - examples of 1-1
 - termination of 1-1
 - uses of 1-1

V

- Var structure 26-46
 - defined 26-47
- vector manipulation functions 20-19
 - cross products 20-21
 - determining intersection 20-24
 - determining intersection of planes 20-26
 - direction vector 20-20
 - dot products 20-21
 - magnitude of 20-20
 - normalizing direction vector 20-23
 - overview 20-19
 - projection of points onto plane 20-25, 20-27
 - scaling direction vector 20-24
 - testing for colinear points 20-22
 - testing for orthogonality 20-20
 - testing for parallel vectors 20-20
- vectors 20-1
 - cross products 20-1
 - dot products 20-1
- view commands 1-1
 - examples of 1-1
 - initialization of 1-1
 - termination of 1-1
 - uses of 1-1
- View Functions 5-18
 - mdlView_clipToContent 5-18
 - mdlView_clipToViewRect 5-18
 - mdlView_getViewRectangle 5-135
 - mdlView_indexFromWindow 5-136
- view functions
 - affecting drawing appearance 5-139
 - camera lens length 5-112
 - corners of view rectangle 5-123
 - creating type 5 elements 5-124
 - cursor behavior 5-128
 - designating plot functions 5-140
 - designating view cursor drawing functions 5-141
 - designating view update functions 5-136
 - determining camera angle 5-108
 - determining camera focal length 5-108
 - determining camera parameters 5-110
 - determining camera settings 5-106
 - determining view extents 5-101
 - determining view settings 5-105
 - dynamics 5-138
 - erasing grids 5-129
 - hidden line removal 5-116
 - hiding views 5-100
 - level information 5-126
 - modifying view information 5-121
 - reading type 5 elements 5-125
 - redrawing grids 5-129
 - rendering views 5-115
 - repainting view 5-113
 - repainting views 5-113
 - rotating views 5-105
 - saving view information to .dgn file 5-123
 - scaling view extents 5-102
 - searching .dgn file for saved view 5-122
 - setting popup menu 5-133
 - setting active depth of view 5-104
 - setting camera lens angle 5-109
 - setting camera parameters 5-111
 - setting clipping planes of view 5-103
 - setting event handling functions 5-133
 - setting rendering modes 5-116
 - setting rotation matrix 5-121
 - setting view area 5-102
 - setting window center 5-131
 - synchronizing private and public view information 5-129
 - table of functions 5-97
 - table of user functions 5-99
 - turning on views 5-100
 - updating bitmaps 5-133
 - updating portions of MicroStation view 5-130
 - view attribute information 5-127
 - view status 5-129
- views
 - orthographic views 5-107
- visibility flags *see symbol sets*
- visibleFunction 5-117

W

- wide character string functions 28-60
 - overview 28-60

- mdlWindow_dockWindow 2-37
 - mdlWindow_getDocked 2-38
 - mdlWindow_isDockable 2-39
 - mdlWindow_organizeApplicationArea 2-39
 - mdlWindow_undockWindow 2-39
 - table of functions 2-37
 - arc, drawing in window 2-25
 - background color, setting 2-33
 - color descriptor, obtaining pointer to 2-34
 - coordinate system 2-20
 - drawing raster data 2-35
 - drawing RGB data 2-36
 - drawing to designated window 2-22
 - element color, obtaining 2-32
 - ellipses, drawing in window 2-25
 - graphic hardware color index, obtaining 2-32
 - graphics characteristics, setting 2-21
 - icons, drawing in window 2-29
 - line drawing style, setting 2-22
 - raster data, drawing in window 2-30
 - rectangles, color fills 2-27
 - rectangles, drawing in window 2-27
 - rectangles, inverting pixel color 2-28
 - RGB data, drawing to window 2-31
 - table of functions 2-20
 - text, displaying in window 2-24
 - text, drawing in window 2-25
 - true color capabilities, determining 2-33
- Window Functions 2-40
 - mdlWindow_getInputFocus 2-40
 - mdlWindow_transparentRasterDataDraw 2-40
 - mdlWindow_transparentRgbDataDraw 2-36
- buffering graphic output 2-4
 - content rectangle, obtaining 2-12
 - converting from global to local coordinates 2-12
 - cursor, turning off 2-4
 - determining window location 2-7
 - dialog box determination 2-10
 - display descriptor 2-11
 - display descriptor, determining 2-11
 - finding input focus 2-17
 - global rectangle, obtaining 2-12
 - handling scroll events 2-19
 - handling window events 2-18
 - hiding windows 2-6
 - input focus, setting 2-15
 - linked list of windows 2-9
 - moving windows 2-5
 - opaque structure 2-1
 - pop-up windows, determination of 2-16
 - refresh events 2-9
 - screen number, obtaining 2-11
 - setting input focus 2-17
 - table of functions 2-2
 - updating windows 2-7
 - view windows, determination of 2-15
 - visibility, determining 2-10
 - walking through MicroStation windows 2-9
 - window display, determining 2-10
 - window identifier, obtaining 2-13
 - window pointer 2-1
 - window pointers, retrieving 2-11
 - window priority, setting 2-5
 - window size, maximizing 2-8
 - window size, setting 2-6
 - window title, retrieving 2-4
 - window title, setting 2-4
 - windows 2-1
 - Windows NT
 - DDE *see* DDE (Dynamic Data Exchange)
 - Windows NT DDE functions 27-10
 - 32-bit value conversation handles 27-37
 - adding data to DDE object 27-14
 - comparison of string handles 27-17
 - copying DDE data object 27-26
 - creating DDE data object 27-20
 - creating string identifying handle 27-21
 - disabling transactions 27-23
 - enabling transactions 27-23
 - error values 27-27
 - establishing conversation with
 - server 27-18, 27-19
 - freeing DDE data object 27-24
 - freeing string handle 27-25
 - obtaining conversation handle 27-36
 - overview 27-10
 - pointer to data in DDE object 27-13
 - registering DDE functions 27-29
 - registering service names 27-33
 - releasing resources 27-12
 - retrieving information about DDE transactions
 - 27-35
 - starting client/server data transaction 27-14

- table of functions 27-11
- terminating conversation 27-22
- terminating conversation list 27-23
- terminating conversations 27-39
- unaccessing a DDE data object 27-38
- Windows NT, clipboard operations 27-2
 - clipboard strategy functions 27-3
 - format process functions 27-3
 - MDL clipboard operations
 - see MDL clipboard operations*
 - menu of operations 27-2
- Windows NT, external program concerns 22-3
 - debugging external programs 22-4
 - mdlExternal_messageReceive 22-3
 - mdlExternal_wait 22-3
 - terminating external programs 22-3
 - userSystem_exitDesignFileState 22-3
 - Visual C++ tools 22-4
- windows *see window management functions*
- Work file functions
 - overview 25-13
- work file functions 25-13
 - creating element descriptor 25-14
 - deleting elements 25-16
 - returning EOF position 25-16
 - setting complex header 25-17
 - size of elements 25-17
 - table of functions 25-14