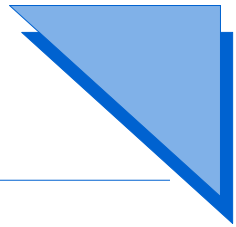
MicroStation MDL™



Programmer's Reference Guide

DAA009540-1/0004

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Unpublished – rights reserved under the copyright laws of the United States and other countries.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

MDL Overview

With MDL, You Can: 1-1

MDL Fundamentals 1-2
Advantages of MDL over MicroCSL. 1-3

MDL Applications 1-5
Performance considerations 1-6

The MDL Runtime Environment 1-7

Application Task Identifier 1-8

MDL Development Utilities 1-9

File Types 1-9
Resource source files 1-10

MDL Application Organization 1-12

MDL Examples 1-14

MDE Workspace

Tools Menu 2-1
Command Table Editor (Ctrl+T) 2-1
Dialog Box Builder (Ctrl+B) 2-2
Icon Editor (Ctrl+I) 2-2
RDE (Ctrl+R) 2-2
String List Editor (Ctrl+S) 2-2
Generate Resource S6ource (CtrlR) 2-2

MDE On-line Help 2-2

Design Methodology

Where Do I Start? 3-1
Functional specification 3-1
High-level MDL application design 3-2
Project task list 3-3

Designing a Dialog Box 3-5

Implementing Command Processing 3-6
Immediate Commands 3-6
Utility Commands 3-6
Primitive Commands 3-6

Basic Application Architecture 3-7

Recommended Directory Structure 3-8

MDE Editing Tools

MicroStation Development Environment 4-1

Command Table Editor 4-2
File menu 4-2
Edit Menu 4-4

Dialog Box Builder 4-4
Dialog Menu 4-5
Load (Ctrl+O) 4-6
Edit Menu 4-7
Options Menu 4-9
The Tools Palette 4-11
Alignment Menu 4-13

Icon Editor 4-14
File menu 4-15
Edit Menu 4-16
Import Menu 4-16

Resource Development Environment (RDE) 4-16
File Menu 4-17
Resource Menu 4-18
Options Menu 4-19
Opening and Closing Files 4-20
Example of creating source code for a new resource 4-20
Example of modifying source code
for an existing resource 4-21

Limitations 4-22
Macro Description Dialog Box 4-23

String List Editor 4-24
File menu 4-25
Edit Menu 4-26

A Comparison of MDL and C

Major Differences Between MDL and ANSI-C 5-1

Pragmas 5-2
Pragma: alias 5-2
pragma: incompatiblePointerParameters, incompatiblePointers, incompatibleReturn, noAnsiDeclaration, noReturnStatement, undeclaredFunction 5-4
pragma: options 5-7
pragma: packedLittleEndianData 5-8
pragma: pointerToNative 5-9
pragma: resourceID 5-9
pragma: ident 5-10
pragma: suppressREQCmds 5-11
pragma: translate 5-12
pragma: Version 5-13

Structure Layout 5-14

Bitfield Handling 5-14

MicroStation Resources

MicroStation Resources 6-1

An Overview of Resources 6-1
Benefits from using resources 6-2
Application Resources 6-3
General Resource Definition Format 6-3
Resource Examples 6-4
Variable sized arrays 6-5

Creating Resources 6-6
Error message translation: an example of static resource use 6-7

User preferences: An example of dynamic resource use . . 6-7
Predefined resources in MicroStation:
Message list and table. 6-8

Modifying Resources. 6-9
Finding your way around a resource 6-9
The Resource Manager Functions 6-11

Managing Resources 6-13
Message Subsystem 6-14
Command Parser 6-14
mdlParse_loadCommandTable 6-15
mdlParse_unloadTable 6-15
mdlParse_loadKeywordTable 6-15
Dialog Box Manager. 6-16

The MicroStation Resource Manager 6-16
Resource File Management 6-16
Loading and querying resources 6-17

A Resource Programming Example 6-22

Binary Portability 6-27
Differing data representations 6-28
MicroStation file types and their portability 6-29
Binary portable files 6-30
How the Binary portability functions work 6-31
Four Recipes for Binary portable data 6-31
A. Type 66 Level 20 Elements (Application Elements) . . . 6-31
B. User Attribute Linkages. 6-32
C. 4.x Resource Files. 6-33
D. 5.0 Resource Files 6-34
Linkage Functions. 6-35

Resource Source Generator 6-35
Generate Resource Source dialog box 6-36

Resource Utility Programs 6-37
cmdcnvert conversion utility for
command table source files 6-37
rdump resource librarian dump utility 6-38

Building Applications

Building Application Process 7-1

Creating a Makefile and Using the bmake Utility 7-2
Makefile format 7-3
Starting the bmake utility 7-9

Compiling an Application Command Table 7-11

Generating Resource Files from C Type Definitions 7-17
Predefined macros 7-18
Syntax 7-19

Compiling Resources 7-19
Command line syntax 7-20

Compiling MDL Applications 7-21

Using the MDL Librarian 7-24

Linking MDL Applications 7-24

Resource Librarian 7-27

Running MDL Applications

Relationships Between Terms 8-1

Loading an MDL Program 8-1

Unloading an MDL Program 8-3

Using Commands in MDL Tasks 8-4

Aborting an MDL Task 8-5

Using MS_INITAPPS Applications 8-5

Debugging MDL Applications

Debugging 9-1

Debugger Input & Output 9-2

How the Debugger Finds Source Code 9-3

Preparing an Application for Use With the Debugger 9-4

Using C Expressions With the Debugger 9-4

Debugger Commands 9-5
Debugger command syntax. 9-5
The ALIAS command 9-5
The BREAKPOINT command 9-6
The CALLS command 9-7
The DISPLAY command 9-8
The DOWN command 9-9
The DUMP command 9-9
The GO command 9-10
The HELP command. 9-10
The IF command 9-10
The MEMORY command. 9-11
The QUIT command. 9-11
The RECORD command 9-11
The SCOPE command. 9-12
The SET command 9-13
The STEP command 9-13
The SYMBOLS command 9-13
The TYPE command. 9-14
The UP command. 9-14
The WATCH command. 9-14

Memory Debugging 9-15

Automatic Fault Reporting. 9-16
Debugger Initialization File 9-16

Documentation

Useful Documentation 10-1

Development Methods 10-1
Tools 10-2
Converting to Help. 10-5
Example. 10-6
Documentation Principles. 10-7
Relevant sections of MicroStation's documentation. 10-8
Why your application should have on-line documentation 10-9
Suggested Application Document Organization 10-9

Element Descriptors

Element Descriptors 11-1
Recursive programming 11-4
Functions that use element descriptors 11-4
Element descriptor validation 11-4
Cautions about memory use and element descriptor size . .11-5

Standard C Functions

ANSI Standard Functions 12-1

ANSI-Compliant Function Definition Files (.fdf) 12-3

MDL Built-In Variables

Variables 13-1

Sample MDL Applications

Examples 14-1
Building sample MDL applications 14-2
Running MDL applications 14-2

dlogdemo MDL Example 14-3

chngtxt MDL Example 14-4

Dialog Box Manager Overview

Overview 15-1

Features 15-2
Application aware dialog items 15-2
Advanced functionality 15-3
Reusable dialog items 15-3
Flexible architecture 15-3
No complicated input loop 15-4
Fast prototyping 15-4
Portable across multiple hardware platforms 15-5
User interface independent 15-5
Designed for internationalization 15-5

Dialog Box Manager Basic Concepts 15-6
MicroStation windows 15-6
Resources 15-7

Dialog item resource specifications
and item list specifications 15-8
Dialog item state: internal value versus external state 15-9

Referencing Application Variables from Resource Files 15-10
Modeless and modal dialog boxes 15-10
Item synchronization 15-12
Synonym resources 15-12
Keyboard focus 15-13
Sinking dialog boxes 15-13
Specifying coordinates 15-14
Text font 15-16
Color 15-16
The dialog box manager's internal architecture 15-16
Hook function IDs 15-18

Dialog Box Manager Header Files 15-19

The Dialog Box Manager Library 15-19

Creating A Dialog Box 15-20

Dialog Box Manager Sample Programs 15-21

Recommended Reading 15-22
OSF/Motif Style Guide 15-22
Microsoft Guidelines 15-22
Apple Computer User Interface Guidelines 15-22
General Interface Guidelines 15-22

Standard Dialog Box Items

Structured Items 16-1

DialogBoxRsc Structure 16-2

DialogItemRsc Structure 16-6

Common Item Resource Fields 16-8

Label Item 16-9
Item list specification 16-9
Item resource specification 16-10
Item hook function messages 16-10
Label item functions 16-10

Group Box Item 16-11
Item List Specification 16-11
Item resource specification 16-11
Item hook function messages 16-12
Group box item functions 16-12

Separator Item 16-12
Item List Specification 16-12
Item resource specification 16-13
Item hook function messages 16-13
Separator item functions 16-13

Toggle Button Item 16-13
Item list specification 16-13
Item resource specification 16-14
Item hook function messages 16-15
Toggle button item functions 16-16

Push Button Item 16-16
Item list specification 16-16
Item resource specification 16-17
Item hook function messages 16-19
Push button item functions 16-19

Option Button Item 16-21
Item list specification 16-21
Item resource specification 16-22
Item hook function messages 16-25
Option button item functions 16-25

Scroll Bar Item 16-26
Item list specification 16-26
Item resource specification 16-27
Item hook function messages 16-28
Scroll bar item functions 16-28
View window icons and scroll bars 16-28

Text Item 16-28
Item list specification 16-29
Item resource specification 16-29
Item hook function messages 16-33
Text item functions 16-33

Multi-line Text Item 16-33
Item list specification 16-34
Item resource specification 16-35
Item hook function messages 16-37
Multi-line Text item functions 16-38

Color Picker Item 16-38
Item list specification 16-39
Item resource specification 16-39
Item hook function messages 16-40
Color picker item functions 16-41

Level Map Item 16-41
Item list specification 16-42
Item resource specification 16-42
Item hook function messages 16-43
Level map item functions 16-43

Menu Bar Item 16-43
Item list specification 16-44
Item resource specification 16-44
Item hook function messages 16-45
Menu bar item functions 16-45

Text Pull-down Menu 16-47
Item resource specification 16-47
Item hook function messages 16-51
Text pull-down menu specific functions 16-51

Option Pull-down Menu 16-52
Item resource specification 16-52
Item hook function messages 16-54
Option pull-down menu functions 16-55

Color Picker Pull-down Menu 16-55
Item resource specification 16-55
Item hook function messages 16-57
Color picker pull-down menu functions 16-57

Tool Palettes 16-57
Icon command frame 16-58
Icon command palette 16-60
Icon command 16-62
Palettes vs. tool boxes 16-66
Icon Command Resources 16-75

List Box Item 16-77
Item list specification 16-78
Item Resource Specification 16-79
Item hook function messages 16-83
List box item functions 16-83
List boxes 16-84

Generic Item 16-90
Item list specification 16-90
Item resource specification 16-91
Item hook function messages 16-91
Generic item functions 16-92

Button Group Items 16-93
Item list specification 16-93
Item resource specification 16-94
Item hook function messages 16-97

Sash Item 16-98
Item list specification 16-98
Item resource specification 16-99
Item hook function messages 16-100
Sash item functions 16-100

Scale Item 16-100
Item list specification 16-100
Item resource specification 16-101
Item hook function messages 16-103
Scale item functions 16-104

Popup Menu Item 16-104
Item list specification 16-104
Item resource specification 16-105
Item hook function messages 16-106
Popup menu item functions 16-106

Radio Button Item 16-106
Item list specification 16-106
Item resource specification 16-107
Item hook function messages 16-109
Radio button item functions 16-109

Tab Page Item 16-110

Tab Page List 16-110
Item List Specification 16-110
Item Resource Specification 16-111
Item Hook Function Messages. 16-113

Tab Page 16-113
Item List Specification 16-113
Item Resource Specification 16-114
Item Hook Function Messages. 16-115

Combo Box Item 16-116
Item List Specification 16-116
Item Resource Specification 16-116
Item Hook Function Messages. 16-119
ComboBox Item Functions 16-119

SpinBox Item 16-119
Item List Specification 16-119
Item Resource Specification 16-120
Item Hook Function Messages. 16-121

Dialog Box Manager Hook Functions

Hook Functions 17-1

Dialog Hook Functions 17-2
DialogMessage structure 17-2

Dialog Hook Function Messages 17-4
Non-requested Messages. 17-6
Focus Messages 17-10
Size Messages. 17-16
Button Messages. 17-20
Open & Close Messages 17-24
Miscellaneous Messages 17-26

Item Hook Functions 17-28
DialogItemMessage structure. 17-28
DialogItem structure 17-29
RawItemHdr structure 17-32

Item Hook Function Messages. 17-34
General item hook function messages 17-36
Input focusable item hook function messages 17-46
Generic item hook function messages 17-50

A Complete Example

MDL Applications 18-1
The basic.ma application 18-1
The application header file: basic.h 18-3
The resource file: basic.r 18-5
The source code file: basic.mc 18-10
The command table file: basiccmd.r 18-16
The command number header file: basiccmd.h 18-17
The type definition file: basictyp.mt 18-18
The message file: basicmsg.r 18-18
The text file: basictxt.h 18-20
The application makefile: basic.mke 18-20
The makefile include file: basicrsc.mki 18-24

Dialog Box Style Guidelines

Design Small Dialog Boxes 19-1
Emphasize modeless rather than modal dialog boxes 19-2
Limit the number of fonts 19-2
Limit the number of colors 19-2
Use the standard order for push buttons 19-3
Match the dialog item type to the intended operation 19-3
Use MicroStation as an example 19-3

Debugging Hook Functions 19-3
The Dialog Box Manager Message dialog 19-3
Displaying strings 19-4
Tracking hook function messages 19-4

Dynamic Link Modules

Overview 20-1

Changes to MDL Source 20-3

Access to MicroStation's Built-ins From DLMs 20-3

Linking an MDL Program With a DLM 20-4

Runtime Concerns. 20-4

Function Pointers as Parameters to Built-in Functions 20-5

Identifying MDL Applications 20-6

Calling Custom MDL Functions 20-6

Determining When an MDL Program is Unloaded 20-7

Application-Specific Resources versus System Resources. 20-7

Dynamic Link Specification Source Files 20-8

Additional Include Files 20-9

A Sample DLM 20-10
- Dynamic Link Specification Source File 20-10
- Managing Application Resources 20-11
- Emulation of an MDL Asynchronous Function 20-12
- DLM Hook Functions 20-12
- Initialization Function; initialize 20-12
- Error Reporting Function; dlmSystem_displayError 20-12
- C++ DLMS 20-13

Platform Specifics 20-13
- Clipper-specific notes 20-13
- SPARCstation-specific Notes 20-16
- HP700-Specific Notes 20-18
- PC Protected Mode Notes 20-19
- Windows NT Notes. 20-20
- Silicon Graphics Notes 20-22
- IBM RS/6000 Notes 20-23

Calling Across DLMs 20-27

Debugging DLMs 20-28

DLM Functions 20-29

dlmSystem_getDLMPath 20-30
- DLM Memory Functions 20-30
- DLM File Functions. 20-30

dlmSystem_callMdlFunction 20-31

dlmSystem_setFunction 20-31

initialize 20-32
userHook_mdlUnload, userHook_dlmUnload 20-33

MDL Shared Libraries

Difference 21-1
Overview 21-1
Creating an MDL Shared Library 21-3
Linking an MDL Program with an MDL Shared Library 21-3
Loading an MDL Shared Library 21-6
Unloading an MDL Shared Library 21-9
Determining When an MDL Program is Unloaded 21-10
Defining the Shared Functions and Variables 21-12
Runtime Concerns 21-13
initialize 21-13
userHook_mdlUnload, userHook_sharedLibNoMoreClients, userHook_allMdlUnload 21-14

Database Manipulation

SQL Database Interface Toolkit 22-1
Architecture. 22-1
SQL Requests 22-2
DatabaseService Structure 22-2
Communications 22-9
Session Debug 22-10
Environment Variables. 22-10
DBTYPE Resource. 22-11
Using MicroStation Database Servers from Initapps 22-12
Overview 22-12
Sample Database Initapp 22-13

Low Level DOS Interface Functions (PC only)

MDL and DOS 23-1
Example 23-2
int86, intdos 23-2
int86PassThrough 23-3

bdos 23-3
segread 23-4
copyToReal, copyFromReal 23-4

Internationalization

Overview of Internationalization Issues A-2
Target Languages A-2
Recommendation A-2

Character Encoding Methods A-3
DEC Multinational ASCII Character Set A-3
Character Translation Table A-4
Asian Character Set A-5
Codeset Handler A-6
Unicode A-6

Data Input A-7
Keyboard Input A-7
File Input A-10

String Processing A-11
Coding Practices to Avoid A-11
Wide Character Processing A-12
mbstowcs A-14
wcstombs A-14
mbtowc A-14
wctomb A-14
mblen A-15
setlocale A-15
wcscoll A-17
wcsxfrm A-19
wcsftime A-19

Display and Output A-29
Fonts A-29
Text Placement in Design Files A-36
Plotting and Printing A-37

Messages and Resource Files A-37
Untranslatable Strings A-38
Dialog Box Design A-38
Source File Organization A-42
Strings in Program Source (.mc) Files A-44
Strings in Dialog Box Resource (.r) Files A-45
Compiling Translated Resource Files A-47
Numbered Argument Lists A-48

Tools for Development and Testing A-49
Hardware A-49
Software A-49

Recommended Reading A-49

Settings Manager

Creating Settings Manager Files B-1
Creating a Settings Group B-2
Writing a Settings Group B-3

Adding Functionality to ascgroup.ma B-5
Adding Keywords to the Keyword Table B-5
Adding Keywords to the List B-6
Coding the Functionality for new Keywords B-6
Formatting the ASCII Input File B-6
Example, Creating a Settings Manager Resource B-10

MicroStation 95

Platform Changes. C-1

MDL Compiler C-2
Additional type checking C-2
Resource Files C-3
Resource Editing Tools C-3

Other Development Facilities C-4
Compatibility with macro customization C-4
File Open Dialogs C-5
MicroStation vs. PowerDraft C-6
Dialog Manager C-6
Message Boxes C-10

1 MDL Overview

MicroStation Development Language (MDL) is a complete development environment that lets applications take full advantage of the power of the MicroStation CAD engine. MDL can be used to develop simple utilities, customized commands or sophisticated commercial applications for vertical markets.

With MDL, You Can:

- Develop tightly integrated interactive applications that run inside MicroStation and "look and feel" like a part of MicroStation. In fact, many features of MicroStation are implemented as MDL applications.
- Develop applications that can be easily ported between hardware platforms. MDL removes, to the extent possible, the typical concerns with the graphics subsystem, operating system and CPU. Porting an MDL application typically involves just transferring source files to the new platform and rebuilding.
- Enable user interface-related data to be organized for efficient translation for foreign language markets.
- Have MicroStation serve as the "CAD operating system," providing the same type of support to CAD applications as the operating system does to non-graphic applications. MicroStation loads and executes MDL applications.

This chapter discusses:

- MDL fundamentals
- MDL applications
- MDL runtime environment
- Application task identifier
- MDL development utilities
- File types
- MDL application organization

MDL Fundamentals

MDL applications are tightly integrated with MicroStation. To the user, an MDL command or operation cannot be distinguished from a core MicroStation command. In fact, many features of the base MicroStation product are implemented as MDL extensions. For the commercial developer, a family of cooperative programs can be built in a modular fashion to provide the same level of integration among all MDL programs.

MDL uses the standard C programming language. Using C has several advantages. For instance, the programmer who already knows C is prepared to start immediately. On the other hand, the programmer who does not have C experience will find a wealth of learning resources, including books, computer-aided instructional material, and professionally taught C programming courses. The investment in learning C will pay off handsomely in the long run, since a large portion of the world's professional software (and virtually all CAD software) is developed in C.

MDL supplies the following:

- A complete implementation of C. The MDL compiler compiles C source files into **pseudocodes** understood by MicroStation.
- A pseudocode interpreter inside MicroStation that executes MDL programs.
- A complete development environment, including the following tools:
  - C compiler, librarian and linker
  - Resource compiler and librarian
  - Source level debugger
  - Make program

  No additional tools are required, except for a text editor.
- A runtime library of over 1750 **built-in functions** — functions that are part of the MicroStation executable. Built-in functions run at full compiled speed and are the same functions used internally by MicroStation's own commands. These built-in functions are essential for applications to work seamlessly with MicroStation and other applications. MDL programmers use the same tools as MicroStation developers to create commands.
- Access to **built-in variables**. Much of MicroStation's global data is available to MDL programs as built-in variables. Header files define the data structures of built-in variables.
- **Hooks** (pointers to functions) to modify MicroStation's behavior. There are many places where applications can use MDL to modify MicroStation's behavior. An application can designate one of its functions as a hook and cause MicroStation to call that function when certain events occur.

- A runtime environment that provides standard operating system functions such as file access, memory management, and memory protection.
- Tools for developing a platform-independent graphical user interface including dialog boxes, pull-down menus and palette menus. The user interface tool box insures that the same dialog box code will work on all platforms.
- The **resource manager** lets you separate your application's code and data. The Resource Manager controls data that is not actually included in the C program. The Resource Manager includes a C-like language used for defining data, a Resource Compiler for compiling definitions, and a library of MDL built-in functions used to access data contained in resource files. The type of data managed by the resource manager includes dialog box definitions, messages, and prompts.

MDL has several other advantages. For example, it is designed to make applications easy to port from one MicroStation platform to another. In many cases, porting involves only transferring source files to the target platform and rebuilding. MDL also enables user-interface-related data to be organized so the application developer can efficiently set up applications to translate for non-English-speaking markets.

Advantages of MDL over MicroCSL

MDL provides much better *integration* with MicroStation.

- MDL applications use the same address space as MicroStation — this makes sharing data between MDL applications trivial. MicroCSL programs run as a separate image (in UNIX, they are a separate process), which makes sharing data very difficult.
- MDL applications have better access to MicroStation's internal data structures. Since MDL applications run in the same address space as MicroStation, many of the internal structures used by MicroStation are available. MDL also provides many built-in functions to access these structures.
- An MDL application does not have to be loaded and unloaded each time it is called, but may remain loaded throughout the session. This is similar to a TSR (Terminate and Stay Resident) program in DOS. Subsequent calls to the application are faster since the function need not be reloaded each time.
- MDL commands can look and act just like MicroStation primitive commands and can be called by another application or user command. Since an MDL command acts as a MicroStation primitive there is no extra code needed to handle a viewing command that may interrupt the MDL command.

- Resources (such as dialog box definitions) can be contained in an MDL executable.
- MicroStation contains a built-in MDL debugger. This is a large advantage over MicroCSL as a familiar debugger is always available no matter which hardware platform the development is being done on. The MDL debugger also has direct access to all published structures in MicroStation.

MDL provides a much better *function* library.

- There are far more functions available in MDL than in MicroCSL, making the scope of MDL greater than that of MicroCSL.
- MDL functions are organized according to specific operations. They are not a random collection of functions used to simply modify a design file. Similar functions have similar prefixes (for example, `mdlSystem_…` for system functions and `mdlFence_…` for fence operations).
- Function arguments in MDL are defined in a logical way for the C language. Function arguments in MicroCSL are based on PDP-11 FORTRAN and are not particularly logical to a C programmer.

MDL applications are much more *portable*.

- MDL applications are source code compatible on all hardware platforms supported by MicroStation (version 4.0 and later). Applications built for the PC work on a UNIX-based system or a Macintosh with little (or no) conversion.
- Since MDL has its own compiler which creates its own p-code, it is very compatible between releases of MicroStation.

MDL applications use a “function call” instead of a “command” level interface.

- MDL is much more flexible than MicroCSL because it only has to create functions to handle the specific events of an application. MDL applications run within MicroStation’s main loop and can modify MicroStation’s behavior. MicroCSL applications can only sequence MicroStation commands.
- MDL applications run in the main loop of MicroStation and can alter the events of that design session. The MDL application is, in effect, driving MicroStation. In contrast, MicroStation controls a MicroCSL application.

MDL supports *event driven programming*.

- In event driven programming, applications respond to the user rather than using the "prompt and wait" technique.
- Internally, MicroStation is event driven. Therefore, all MDL applications, which run inside MicroStation, must also use this technique.
- Any environment that has a graphical user interface will require that the applications which support it use the event driven programming techniques.

MDL is memory-based, making MDL applications *faster* and *more reliable* than MicroCSL applications for most operations.

- A MicroCSL application must perform all operations directly in the design file. In contrast, an MDL application can perform most operations in memory. For example, an MDL application performs element manipulations in memory — changes are not written to the design file unless the MDL application requests it. Since reading and writing to disk is time-intensive, MDL applications can be much faster than those written with MicroCSL.
- When an MDL application writes elements to the design file, it can use the same code MicroStation uses to verify the elements. This makes MDL applications more reliable than those written with MicroCSL.
- An application that handles events and is memory-based is usually more modular.

MicroCSL programs are faster for some processor-intensive calculations. See "When to use external programs" on page 1-7 for more information.

MDL Applications

This manual uses the terms **MDL program**, **MDL application** and **MDL task**. An **MDL program** resembles programs in other systems. The user creates an MDL program by compiling the source modules and then linking the object modules. An MDL application consists of an MDL program, plus all other associated resources. The program is the only resource required in an application. An application can contain more than one program. An MDL task is an MDL program loaded by MDL. A task is either active (when MicroStation's pseudocode interpreter is executing its instructions) or waiting to be activated by some event occurring within MicroStation.

An MDL program consists of a code segment containing instructions understood by the MDL pseudocode interpreter and a data segment that can contain some initialized data.

The pseudocodes are not the same as the native computer's instructions. Consequently, an object library cannot be linked to an MDL application unless the MDL compiler created the object library.

The program is the only resource required in an MDL application. Other resources frequently used in MDL applications include the following:

- A **command table** describes the syntax associated with commands available with the MDL application.
- **Dialog boxes** make a platform independent graphical user interface possible.
- **Message lists** are resources where MDL programs can store infrequently used messages. Message lists allow these messages to be stored in data files rather than as part of the resident application. Thus, you can create foreign-language versions of an application by translating messages in the external file. At run time, these messages are accessed through built-in functions.
- **Help text**.

Performance considerations

In most cases, the performance of a well-written MDL application is excellent, for the following reasons:

- MDL source code is compiled into "P-Code" (pseudocode). The MDL interpreter is not a source level interpreter that uses tokens, such as the user command interpreter.
- The MDL interpreter is highly optimized for MDL P-Code.
- MDL applications use the same address space as MicroStation and should use built-in functions, which run in native mode with no context switching (the need to switch from one application to another) required for most processing.

When to use dynamic link modules (DLMs)

Dynamic Link Modules (DLMs) are portions of code that can be loaded and linked by MicroStation at runtime. These modules are compiled into native machine code. Dynamic Link Modules are created using the host operating system's tools such as the compiler and linker. The MDL tools are not used to create Dynamic Link Modules.

Dynamic linking is appropriate when:

- MDL's performance is insufficient.
- Shared libraries are required so that applications can use common code.
- Access to object-code libraries is needed.

When to use external programs

MDL applications can communicate with MicroCSL programs using shared memory and message queues. The interaction between MDL and MicroCSL on all platforms is similar to the interaction between MicroStation and MicroCSL that previously existed only in MicroStation 32.

MDL's external program interface should be used if:

- The application requires library functions that are not shipped in source code (for example, it may have to link with an object library supplied by another vendor). An MDL application cannot call a function that is not an MDL function or built-in function. The external program can be linked with this library, and the functionality of the library is now available to the MDL application through the external program.
- There are processor-intensive operations, such as intense floating point calculations, that must be run at true machine speed.

External programs can and should be used to evolve existing MicroCSL applications. Initially, MDL may be used to create a user interface that is fully integrated with MicroStation while continuing to use MicroCSL code to do most of the actual processing.

The MDL Runtime Environment

This section describes the features provided by the MDL runtime environment. This environment provides capabilities that an operating system generally provides to the program.

MDL applications can access files through standard C file-handling. MDL applications can access dynamic memory allocation through standard C memory management functions such as `malloc` and `free`.

Data pointers are real pointers to MicroStation's data address space. These data pointers can be shared with MicroStation and other MDL applications.

In contrast, MDL **function pointers** are offsets in the MDL program's address space; they are not real pointers. Thus, they must point to MDL functions (a sequence of pseudocodes produced by the MDL compiler) and cannot point to native compiled functions (native machine code instructions produced by the C compiler and specific to the machine and operating system). An MDL function pointer can be used only within an MDL application and cannot be shared with other MDL applications. MicroStation provides built-in functions that let MDL programs designate user hook addresses to MicroStation. This capability lets MicroStation give function pointers the special handling they need so they can be called from the native compiled code in MicroStation.

The MDL runtime environment provides **exception** handling. An exception is a serious error than can be detected at runtime. When an exception is detected, MDL reports the exception type and tries to abort the application. An application can prevent the abort by specifying a user hook using the `mdlSystem_setFunction` built-in function.

Exceptions fall in two categories:

- Faults detected by the native machine and operating system. For example, if an MDL program causes a floating point exception or tries to use a `NULL` pointer, a hardware exception occurs. MicroStation intercepts the fault and performs its error recovery.
- Faults detected by MDL. Faults in this category include stack overflow and divide-by-zero.

Application Task Identifier

Every active MDL task has a **task identifier** (**task ID**). The task ID must be unique, since it is frequently used in MicroStation to unambiguously identify a task. Also, when you enter the name of an MDL task, you specify the task ID. All messages pertaining to an MDL application's status include the task ID.

Each MDL task has an associated task ID. By default, the task ID is the base name of the file from which the program is loaded. To override the default, specify the task ID to mlink using the -t link option.

You must use the -t link option to associate a task ID with a program under the following circumstances:

- When another program or resource file needs to know the task ID, the task ID must be constant and therefore needs to be specified. If you do not use the link option to specify the task ID and the file containing the program is renamed, the task ID will also change.

- When the application resource file contains more than one MDL program, you must use the link option to specify a task ID. If you do not, MDL cannot determine which program to load. See "Running MDL applications" on page 14-2 for more information.

An MDL program that does not communicate with other MDL programs and does not combine programs into the same resource file does not need to use the task ID.

MDL Development Utilities

This section briefly describes development utilities provided with MDL. More detailed information on all of the utilities is included later in "Building Applications" on page 7-1.

Utility name	Description
bmake	Make utility used to automate compiling, linking, and resource building for complicated applications.
mcomp	Compiler used to compile MDL source files.
mlib	Librarian used to manage file images in a library file.
mlink	Linker used to combine object files into a program.
rcomp	Resource compiler used to compile resource source files.
rlib	Resource librarian used to merge multiple resource files into one resource file.
rsctype	Type generator used to generate type descriptions for a group of built-in functions that can be used to evaluate C expressions at runtime.

File Types

There are two major types of source files for an MDL application — C source (usually with the suffix **.mc**) and resource source (usually with the suffix **.r**).

The `main` function is optional. In MDL, it is not mandatory to have a `main` function. If one is present it is usually just initializes the application variables and performs any initial setup. Since MDL is a set of functions to handle events, an MDL application does most of its work in command functions, state functions and dialog manager hooks.

MicroStation calls a **command function** to start a command. A command function has the keyword `cmdName` or `cmdNumber` as part of its declaration. MicroStation calls a command function when:

- The user keys in a command or selects it from a tool palette or other menu.
- A program sends a key-in to MicroStation (for example, a key statement in a user command).

A command function typically displays prompts and initializes MicroStation's state machine, specifying a set of state functions.

MicroStation calls **state functions** to handle user events while a command is processed — most commonly data points, resets and key-ins. MicroStation manages two sets of state functions:

- One set is for **primitive commands** — commands that can create, delete or modify elements.
- The other is for **viewing commands** — such as ZOOM and WINDOW AREA. Viewing commands cannot create, delete or modify elements.

MicroStation allows a viewing command to temporarily "take over" the state machine from a primitive command. This allows MDL developers to create primitive commands with absolutely no concern about how they interact with viewing commands.

Dialog manager hooks are functions that process the events generated by the dialog box manager and the interaction between MicroStation, dialog boxes and the user.

Functions may also be written to handle other events in MicroStation such as update, undo or command parsing.

Resource source files

MicroStation uses resources to manage most data other than design files, cell libraries, and font libraries. Most MDL applications contain resource source code. The most common uses of resource source code are to define command tables, dialog boxes, and message lists.

- A **command table** defines an application's command syntax. When an application is loaded, it tells MicroStation to add its command table to the group of tables used by MicroStation to parse key-ins.
- A **dialog box** is a rectangular window on the screen that accepts and processes user input.
- A **message list** is a list of messages used by an application.

An MDL application can include some or all of the following file types:

Extension	File description
.h	Referred to as a **header file** or **include file**, which contains definitions shared by multiple source files. Header files are included in .mc files, .mt files and .r files.
.ma	Referred to as an **application file**, a resource file that contains all resources associated with one application. An application file can be created with mlink or rlib.
.mc	Referred to as an MDL **source file**.
.mke	Referred to as a **makefile**, read by bmake to learn how to build an application.
.ml	Referred to as a **library file**, created and updated by mlib. Library files typically contain MDL object files, but can contain any kind of file.
.mm	Referred to as a **map file**, generated by mlink.
.mo	Referred to as an **object file**, created by mcomp. These files are used as input to mlink.
.mp	Referred to as a **program file**, a special type of resource file that can be merged with other resource files using rlib. A program file, created by mlink, contains pseudocodes that MDL can interpret. A program file is the simplest kind of application file.
.mt	Referred to as a **type file**, a source file used to generate type definitions to be used by the MDL C expression built-in function. rsctype reads a type file and generates a resource source file.
.r	Referred to as a **resource source file**. Some examples of data in resource source files are dialog box descriptions, command tables and messages. rcomp reads a resource source file and generates a resource file.
.rsc	Referred to as a **resource file**. rcomp generates most resource files. MDL program and application files are also resource files.

The reader who is familiar with a typical C development environment will notice many parallels. For example, the .mc source files are analogous to .c source files. Object .mo files resemble .obj or .o object files. The .mp files that mlink produces are similar to the executable files (.exe) produced by other linkers.

However, concepts related to resources may not be as familiar. To an MDL program, resources consist of data that the program will require at runtime. However, this data does not replace all of a program's data. This data is either the same every time the program runs (as are the messages or dialog boxes that a program displays) or are modified during execution and saved for the next execution (as are user preferences). In MDL, resource data is defined in resource source.r files so that it is consistent with C structure definitions. The resource compiler rcomp is provided to produce resource

.rsc files from these source files. Multiple resource files can be merged using the resource librarian, rlib. Since keeping an application together in one file is convenient, program (.mp) files can be treated exactly like other resource files. Many MDL built-in functions let you access the data stored by the resource manager.

Many MDL applications use only MDL source files, object files, and program files. For these applications, the program file is the application file. For example, if example.mc contains all source code for an application, enter the following commands to build the application:

```
MCOMP EXAMPLE
MLINK EXAMPLE
```

More complicated applications generally merge resource and program files into an application file. For these applications, mlink creates a program file that has the .mp extension. rlib creates the application file by merging the program file with other resource files.

MDL Application Organization

There are two basic approaches to developing applications using MDL. The first is to use the input handling functions, activate MicroStation commands, and simulate operator inputs, sequencing them as appropriate. The second approach is to call the equivalent functions in the MDL built-in library, accomplishing the same tasks without activating MicroStation commands. Of these two approaches, the second is distinctly superior. Applications developed with built-in library functions are more reliable, more predictable, and more efficient than those that sequence MicroStation commands.

Applications that sequence MicroStation commands have the following problems:

- They require a complete knowledge of inputs to the MicroStation commands used.
- Making them work exactly like other MicroStation commands is difficult. For example, they must manually allow windowing and intermediate commands.
- Multiple applications cannot be active at the same time since one application must drive MicroStation.

Therefore, when possible, MDL applications should not use the input handling functions to sequence MicroStation commands.

All MicroStation primitive commands operate in a similar manner; they accept input as data points, resets and key-ins. They also allow windowing and immediate commands to operate transparently and work with selection sets if applicable. Good MDL applications should operate similarly and, in effect, become part of MicroStation. Users

should not be able to distinguish a properly constructed MDL program from an internal MicroStation command.

The key to accomplishing this transparency is creating MDL applications that are **event-driven**. Event-driven applications do not become suspended while awaiting a specific type of input (such as a key-in or a data point). Rather, they establish functions, or **state handlers**, that are called when various events occur. These handlers process the input and establish new handlers when appropriate.

The following diagram of MicroStation's input loop illustrates the concept of event-driven applications:

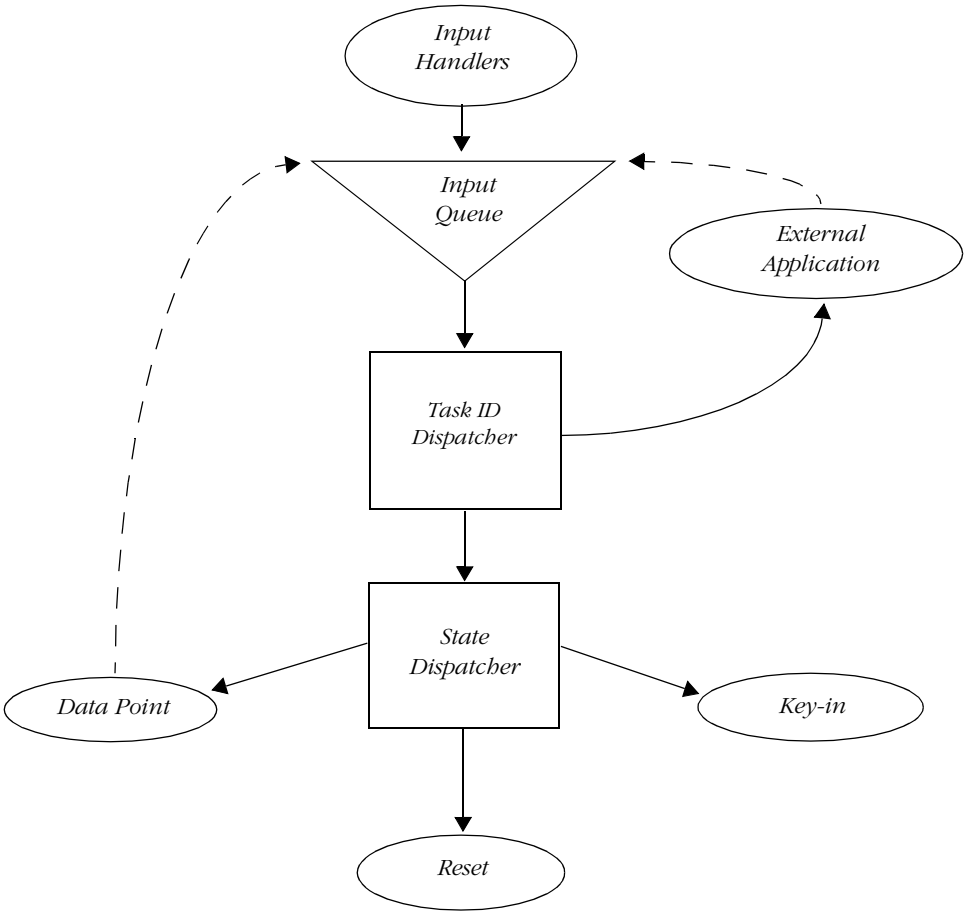


Figure 1.1 MicroStation Input Loop

The **input queue** is at the heart of MicroStation's input loop. All events input to MicroStation are gathered by the input handlers (such as keyboard, mouse, tablet and external programs) and put in the input queue.

When the event reaches the head of the queue, the task ID dispatcher processes it. The dispatcher sends the event to the appropriate application. If your MDL application is sequencing MicroStation commands, the application becomes responsible for fully processing the event.

If no application is designated for the event, the default processor, the **state dispatcher**, assumes control. The state dispatcher preprocesses the event and then calls the appropriate state handler that you designated in your MDL application.

See "Input Handling Functions" in the MicroStation MDL Function Reference Manual for more information on using the input handling functions. See "Command Organization with State Control Functions" in the MicroStation MDL Function Reference Manual for more information on using the state functions.

MDL Examples

MicroStation provides a number of MDL examples. The examples are all in subdirectories of the mdl\examples directory.

The mdl\examples\doc directory illustrates calling sequences of the built-in functions. The examples in the other directories illustrate fundamental concepts in the context of applications. For more information on these examples and how to build and run them, see "Sample MDL Applications" on page 14-1.

2 MDE Workspace

The MDE workspace contains a visual environment that provides tools to assist in the development and testing of MicroStation applications. From the MDE command window, you can access the various editors, help and the resource source generator and switch to other user interfaces or workspaces.

To exit MDE and return to the system prompt, choose Exit from the File menu.

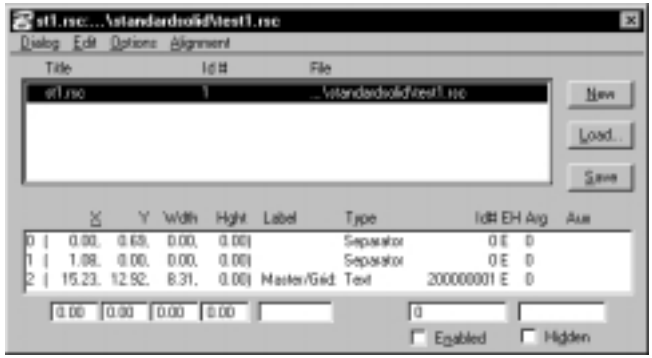


Figure 2.1 The MDE Workspace

Tools Menu

The Tools menu in the Workbench command window is used to access the Workbench tools. Instructions for using the tools can be found in “MDE Editing Tools” on page 4-1.

Command Table Editor (Ctrl+T)

The Command Table Editor tool is used to create and modify application command tables, which define the commands used to operate an application. Command word values and command word attributes can be changed and resaved in the binary resource file.

Dialog Box Builder (Ctrl+B)

The Dialog Box Builder tool is used to create new dialog boxes or edit existing dialog boxes. Builder's visual environment simplifies layout of items in a dialog box and testing of the dialog box's functionality. Using Builder, you can build a new dialog box, associate access strings with dialog items, edit item-specific attributes, and save the results in a binary resource file. Builder will also let you test the dialog as you add items instead of making you wait until the application is compiled and loaded.

Icon Editor (Ctrl+I)

The Icon Editor tool is used to create and modify icons. You can create and modify icons in either the standard MicroStation icon sizes (23 and 31 pixels square) or in any size between 5 and 100 pixels on an axis. Brush, line, rectangle, and circle drawing tools are provided.

RDE (Ctrl+R)

The RDE (Resource Development Environment) is a tool used to create and modify resource files and header source files.

String List Editor (Ctrl+S)

The String List Editor tool is used to create and modify string list resources, which define an application's prompts and error messages. Strings can be altered and additional strings can be added to existing string lists.

Generate Resource S6ource (CtrlR)

The Resource Source Generator tool is used to decompile binary resource files (both .ma and .rsc files) into resource source file entries with resource item definitions and all needed defines.

In addition to the development and testing tools, MDE includes the on-line MDL reference, which has helpful information about MDL functions, such as argument lists and return values, and related topics.

MDE On-line Help

MDE provides the MDL programmer with an on-line, slightly abridged version of both the MDL Programmer's Guide (this volume) and the MDL Function Reference Manual. Every function that is listed in the Function Reference Manual is included in the On-Line Help, complete with cross-references, for the convenience of the MDE developer.

On-line help is an MDL application called USTNHELP. MDE loads USTNHELP if it is not already loaded. The default operation for USTNHELP is to display the on-line user documentation. To be presented with a list of all help files, choose Available Files from the Help menu.

When the Help dialog is opened, you will be presented with either a list of available help files or the top-level topics for the file you have selected. To select a help file or view documentation on a particular topic or function, click your choice a single time.

As you traverse the help hierarchy, you can view your location in the tree by using the Topic option button. Selecting higher levels will move you back up the tree.

If you are searching for a particular function or subject, click the Search button and use the Search dialog to find the documentation you require. This method is especially fast if you ensure that the Topics Only toggle button is highlighted.

Function descriptions in MDL On-line Help mirror those listed in the MDL Function Reference Manual, though long functions are sometimes abridged for space considerations. Information listed in the On-Line Help system includes:

- #include files required (if any)
- Arguments of the function
- A description of the function's purpose and use
- Return values of the function
- Related functions

Any related functions are displayed as hypertext and can be instantly referenced by double-clicking their names.

While the On-Line Help window is open, the help article for each selected topic or function is displayed. Previous topics or functions can be recalled by clicking the History button to open the History dialog.

3 Design Methodology

This chapter will help you plan the overall design of your MDL application. It is strongly recommended that you spend some time using the tips included in this chapter before you start writing code or designing resources. The more planning you do before starting work, the less time you will spend later redesigning and recoding should you realize your efforts were misguided.

Where Do I Start?

This is a common question when creating a new application in a new development environment, especially in a graphical user interface (GUI) environment where a program's structure tends to be more complex than in traditional top-down programs.

With over 1750 functions to choose from in MDL, the prospect of building an application may seem daunting. A developer needs a "road map" for guidance through the design and implementation of an MDL application.

Functional specification

Before you go into the details of your application's design, you must first have a functional specification that describes what the application is to perform. This description should be from the user's viewpoint. Few or no details specific to the development environment (MDL in our case) should be present. If done well, a functional specification of this sort leads directly to a high level design.

We recommended using the following approach to create a functional specification that translates easily into a high level design.

Define the objects

Define the types of objects that users of the product will manipulate in terminology that is familiar to the users. For example, GIS users work with parcels, so definitions of these are in the functional specification. GIS users are also accustomed to thinking about their work in terms of projects.

List the attributes of the objects

What shapes, sizes, colors, etc., are valid for each type of object? Furthermore, what attributes are mutually exclusive? The rasticon MDL example allows icons to be either 23x23 or 31x31, or a user-defined size.

Define the object placement and manipulations

Once you have fully defined the objects and their attributes, define the ways in which the objects are placed and manipulated. rasticon allows icons to be built in specific ways and supplies specific drawing operations, such as lines, circles, points and rectangles.

Other required functionality

Other items in the functional specification might be the types of database information to maintain, report formats, and any other processing that does not involve object manipulation.

High-level MDL application design

Determine MicroStation elements to use to represent application objects

The objects that are defined in the functional specification must be represented by elements in the design file. For example, GIS represents parcel markers as centroids (small rectangles).

Determine what settings must be collected from the user

The attributes of the objects listed in the functional specification help us determine the form of our data collection and organization. We refer to this information in our MDL application as settings. Once we know the settings to collect and how to organize them, we can design dialog boxes that allow the user to set them.

Design element placement and manipulation tools

The part of the functional specification that defines object placement and manipulation is the basis for a list of the application's commands. Knowing this, we can build the command tables and tool palettes, and start designing the element placement and manipulation routines.

Design tools and settings boxes for other functionality

The other items in the functional specification also require user interface and processing routines.

Project task list

When the high level design is complete, a project task list emerges. Although the actual list is different for each application, we can identify a superset of the items most likely to appear on such a list.

Application initialization logic (main() processing)

This is the logic that sets up the application for all future processing. Among the tasks to accomplish are:

1. Open resource files for dialog boxes, command tables, and message lists.
2. Register message list IDs to be used for prompts or errors.
3. Load command tables.
4. Set user hooks for system events (For example, UNLOAD or RELOAD.)
5. Register dialog hook functions with MicroStation's Dialog Box Manager.
6. Open the application's initial dialog box.
   After the initialization is complete, the `main` function returns and the application waits for one of its dialog box hooks, user hooks, or command functions to be invoked.
7. Build a cell library:
   An application that requires a set of pre-defined graphic symbols to represent the objects of the system should have its own pre-built cell library. The cell library is generated by the developer and then packaged with the application. For information on cells and how to create them, see "Cell Placement and Manipulation" (General procedures for working with cells) and "Menu Reference" (Settings menu) in the MicroStation On-line Reference Guide. An example of cell creation is in "Learning MicroStation" (Lesson 1) in the *MicroStation User's Guide*. For information about using cells in your MDL applications, see "Element Creation" (`mdlCell_create`), and "Cells" in the *MicroStation MDL Function Reference Manual*.
8. Create command tables:
   A **command table** defines the syntax of an application's commands. Command tables are hierarchical with one main table branching downward into a tree of subcommands. An application's command table is defined in a resource source file that is compiled to produce two output files Ñ one is for the parsing and validation of key-ins, and the other is a list of unsigned integers that uniquely identifies each command.

For more information on the creation and use of command tables, refer to "Building Applications" on page 7-1.

9. Build dialog boxes:
   Dialog boxes present and collect data. See "Creating A Dialog Box" on page 15-20 for a general procedure for building dialog boxes that makes them easier to debug.

10. Write primitive command logic:
    Write all of the application's primitive command routines that handle element placement and manipulation. Each primitive command requires setting up state functions for the handling of events such as data points, resets and to display dynamics - that is display an element as it is dragged across the screen, sometimes referred to as "rubberbanding."

11. Write logic for immediate and utility commands:
    An **immediate command** is a command that sets the value of an internal settings variable. In MicroStation versions 4.0 and later, dialog boxes are a much more elegant way to set the value of internal variables, and as a result there is less need for immediate commands. However, there are times when it may still be necessary to define immediate commands. For instance, if you foresee the need to set an MDL application variable from a MicroCSL program or user command, an immediate command must be defined because MicroCSL programs and user commands cannot display dialog boxes.
    A **utility command** is a command that does not require interactive input from the user and does not merely set application variables.

12. Build tool palettes:
    Tools are graphic icons that start commands. When the user clicks on a tool, the appropriate command is started. An application with MicroStation's "look and feel" has at least one tool palette if it provides commands to the user.
    In the initial stages of development, it is not absolutely necessary to create a tool palette since commands can be keyed in. However, testing is much easier when you can click on an icon to start a command instead of keying it in.

13. External processing:
    The degree of external processing used by an application varies from none to all depending on the purpose, and sometimes the history, of the application. A developer with an existing MicroCSL application may only want to exploit parts of MDL. For example, an MDL "front-end" could be written with a message queue or shared memory interface to the MicroCSL program. On the other hand, a developer may wish to totally re-implement an existing external program in MDL. Somewhere in between the two extremes is a MicroCSL program that has part of its functionality moved into MDL or vice versa.
    An external program does not have to make MicroCSL calls.

14. Create on-line help:
    Like documentation, on-line help usually begins to take shape later in the development cycle of a product. Implementing on-line help is a matter of creating help resources containing textual information and associating them with certain commands, dialog boxes, or dialog items. Help resources can also be "stand-alone;" that is they can only be referenced by manually traversing the help topics in MicroStation's Help dialog box.

At this point, we will focus on two tasks that require most of the work done in writing an MDL application

- Building dialog boxes
- Writing the logic to implement commands.

Designing a Dialog Box

There are several steps required to build a dialog box.

1. Implement "dumb" dialog boxes, concentrating on the appearance of the dialog box and its items. These are dialog boxes and items with no commands, hooks, or access strings.
2. Add **access strings** where appropriate. An access string defines a program variable to associate with a dialog item. This allows you to change program variables (called settings) from a dialog box without using a hook function. Element placement and manipulation commands will reference these settings. The MDL debugger is ideal for inspecting access strings while the application is running.
   For information on access strings, see "Dialog Box Manager Overview" on page 15-1. For more information on using the MDL debugger, see "Debugging MDL Applications" on page 9-1.
3. You may need to extend the basic functionality of the dialog box using **dialog hooks**. Hook functions establish relationships between dialog items, initiate processing beyond what the dialog manager does, or define the behavior of generic items (user-defined dialog box items).
   The rasticon example application uses a hook function to control the display of the icon image as it is being built by tracking button and drawing events.

For more information about dialog hooks, see "Dialog Box Manager Hook Functions" on page 17-1.

Implementing Command Processing

There are two steps required to implement command processing:

1. Generate command tables and "stub" command functions. A stub function outputs a message that says the command has been invoked and then returns, allowing you to verify that all commands are defined with correct syntax and that all command functions are associated with the correct command.
2. The next step varies slightly, depending on the type of command.

Immediate Commands

The main command function needs to be changed to update the appropriate application variable and return.

Utility Commands

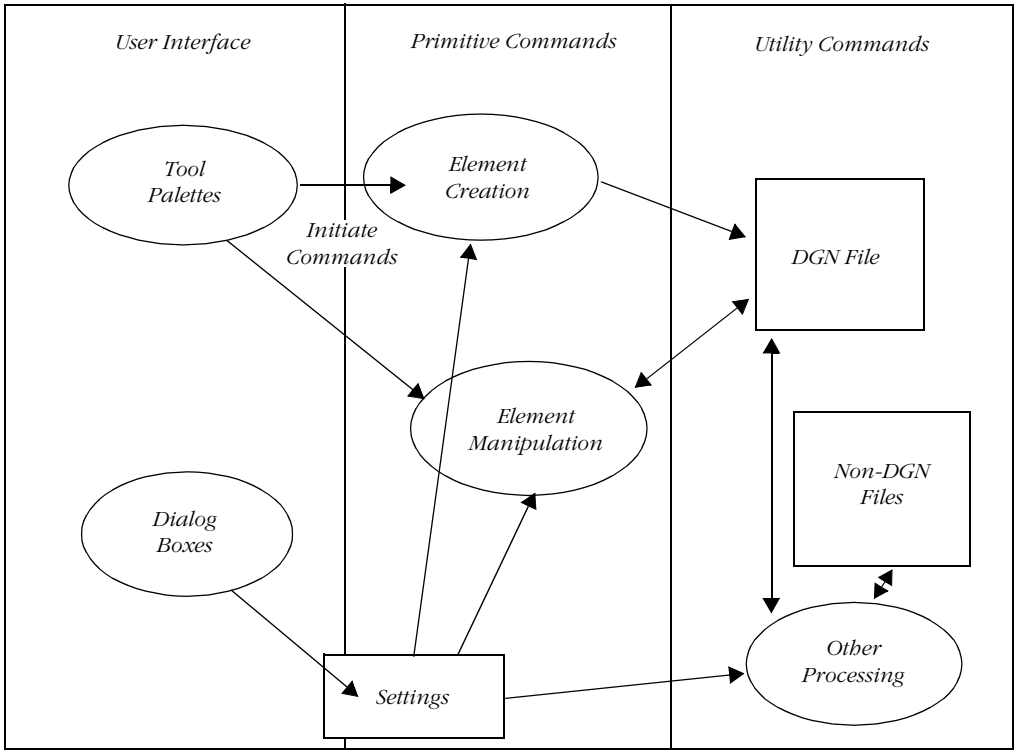
The main command function may make one or more function calls beyond setting application variables, but does not require user interaction. Setting up state functions is not required. Utility commands may do database access such as scanning the design file for specified elements. The stub function would typically have one top-level function call which in turn breaks down into several other compact, modular routines.

Primitive Commands

The main command function for primitive commands must set up the state machine to handle events such as data points, resets, and element selection. Do this by calling one or more of the `mdlState_...` or `mdlDynamic_...` routines. This implies that routines to handle the "state transitions" need to be coded.

Basic Application Architecture

The following diagram shows the relationships between the different parts of an MDL application.



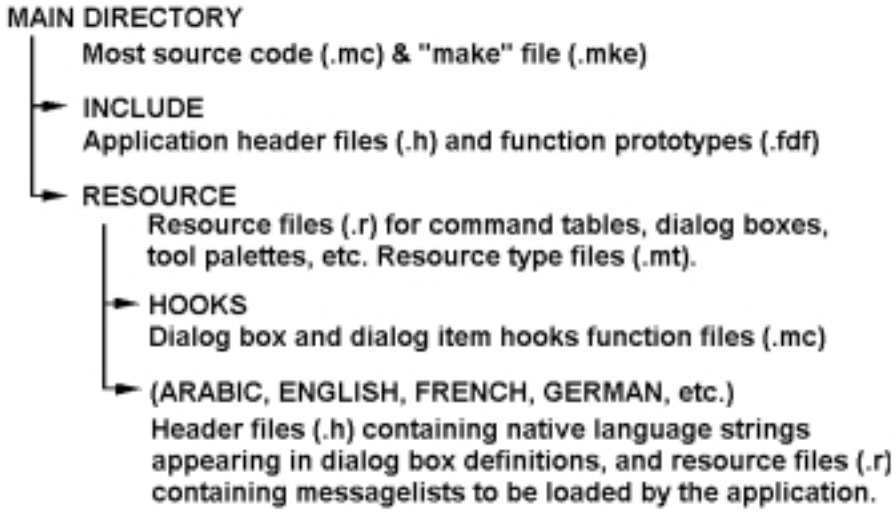
Dialog boxes and tool palettes comprise the user interface. Dialog boxes are generally used to collect settings used by command processing routines. Tool palettes are used to start commands graphically.

Primitive commands reference settings variables to create or modify elements. Primitive commands can also access a database or resource file to obtain parameters for a specific type of element. Although primitive commands are mainly used for interactive placement and manipulation of elements, they are not limited in the number or type of MDL functions that they can call.

Commands can create and display elements without writing them to the design file. Utility commands are can make references to settings variables and design files. In fact, a utility command could access any type of file.

Recommended Directory Structure

Although you can use any directory structure, we offer one here to use as a guideline:



There are several reasons why we recommend this structure or one like it:

- The more you break down your files, the fewer file contention problems you will have among your developers.
- Source code files, header files, and resource files are segregated to make it easier to navigate through the files.
- (This is more of a file naming convention suggestion.) In the **resource/hooks** directory, there is one **.mc** file (with the same root filename) for each **.r** file that defines a dialog box in the **resource** directory. This makes it easy to locate the hook functions for a dialog box.
- The resource directory's native language sub-directory contains resource **.r** files and header **.h** files that store all of the text string constants used by the application.
  Any language-dependent strings in **.mc** files are removed from the code and placed in a message list in one or more **.r** files in the native language sub-directory. The source code references the string's index in the message list to obtain the string at execution time.
  Any language-dependent strings in **.r** files are replaced with macros (`#define`) in a header **.h** file(s) in the native language sub-directory. They are defined as macros in a header file because the strings used in dialog boxes are needed at compile time by the resource compiler. There are at least two reasons why it is a good idea to separate language-dependent text strings this way:

- Messages can be globally changed without recompiling the source code - only resource source files need to be recompiled. This means the executable does not change, but rather just opens an updated resource file. This is useful if an application is used in countries that speak different languages. This is one part of a process called **internationalization**.
- The program's memory requirement is reduced since messages are only loaded as needed. Also, MicroStation has logic that minimizes the memory required by messages in message lists.

4 MDE Editing Tools

The following sections describe how to use the MDE editing tools to work with MDL program resources. These tools prevent you from having to create resource source files from scratch with an ASCII editor, and instead let you concentrate on the task of designing your application.

MicroStation Development Environment

The following tools are discussed in this chapter:

- Command Table Editor
- Dialog Box Builder
- Icon Editor
- RDE (Resource Development Environment)
- String List Editor

Although you can also use these tools to edit MicroStation's resources, you cannot save changes directly to MicroStation's resource file (**ustation.rsc**); modifications to MicroStation resources must be written to another file (**.ma** or **.rsc**). To make changes to the MicroStation resource file, you must:

1. Make a copy of MicroStation's resource file.
2. Use the resource editors to change the desired resources and save the changes back to the copy.
3. Replace MicroStation's resource file with the modified copy.

When saving a resource (command table, icon or string list), the resource editor will overwrite existing resources with new resource information when the resource IDs match. That is, if resource ID 15 in file A is edited and saved back to file A or saved to an existing file B containing another matching resource with an ID of 15, then the original resource information in the file will be overwritten.

Command Table Editor

The Command Table Editor is used to create and edit compiled (binary) command table resources in a **.ma** or **.rsc** file. Support is provided for both the generic (resource class `CmdT`) and MicroStation (resource class `Cmds`) command table resources, although only generic command table resources can be created. For more information on the creation and structure of command tables, see "Compiling an Application Command Table" on page 7-11.

➤ **To edit an existing command table resource**

1. From the Tools menu, choose Command Table Editor.
   The Command Table Editor dialog box opens.
2. From the File menu in the Command Table Editor dialog box, choose Open File.
   The Resource File to Open dialog box opens.
3. Select the file containing the command table resource you wish to edit and click OK.
   The Open Command Table Resource dialog box opens and displays the list of command table resources that can be opened and edited.
4. Select the command table resource you wish to edit and click the OK button.
   The command table resource is then displayed for editing in the Command Table Editor dialog box.
5. At this point, you can edit a command word entry in the command table resource by double-clicking on the desired command word. This causes the Command Options dialog box to open and display the attributes of the selected command word. The command word can now be modified. To save your changes, click the OK button and then choose Save or Save To from the File menu in the Command Table Editor dialog box.

File menu

New File (Ctrl+N)

Closes any open command table resources without saving and initializes a new command table resource for editing.

Open File... (Ctrl+O)

Opens the Resource File to Open dialog box, from which a **.ma** or **.rsc** file can be opened.

➤ **To open a Command Table**

1. Select the MDL application or Binary (compiled) resource file that contains the command table resource you wish to edit and click the OK button.
   The Open Command Table Resource dialog box opens. The Resource column lists the command table hierarchy (1 for the root table and so on).
2. Select the number of the command table resource to edit and click OK.
   The command numbers and strings for the command table resource are displayed in the Command Table Editor dialog box.

New Command Table

Closes the open command table resource and creates a new command table resource for editing in the open resource/application file if one exists.

Open Command Table... (Ctrl+T)

Opens a different command table resource from the open resource/application file.

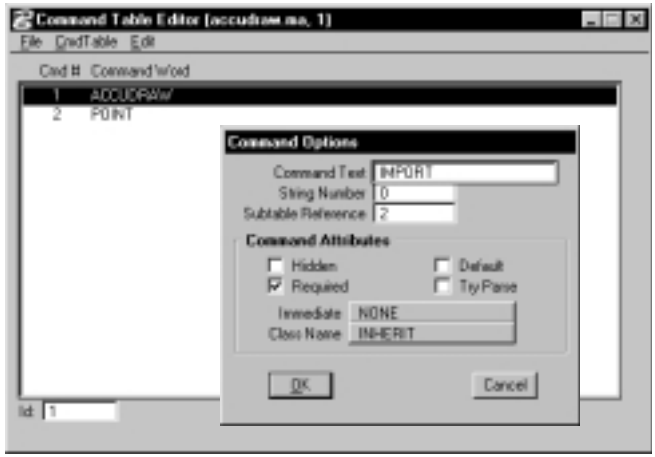


Figure 4.1 Command Table Editor dialog box and Command Options dialog box.

Save (Ctrl+S)

Saves the open command table resource to the original file containing the resource. If no file was opened, the Save to File dialog box is opened for you to select an already existing binary (.rsc or .ma) file in which to save the command table resource.

Save To...

Opens the Save to New Resource File dialog box, which is used to save the open command table resource to a new or existing binary (.rsc or .ma) resource file.

Exit

Exits the Command Table Editor.

Edit Menu

Delete (Del)

Deletes the selected command table entry.

Append (Ctrl+A)

Opens the Command Options dialog box, which is used to create a new command word and append it to the end of the open command table resource. The default attributes are those of the selected command. After you have completed your edits, click OK to change the command word.

Dialog Box Builder

Builder allows the MDL developer to create and modify dialog resources and save the changes or new resource items to a binary (.rsc or .ma) file. Any item in a standard dialog can be placed and manipulated in any configuration, and the dialog itself can be resized. These modifications can be made by someone other than the developer so that the tasks of user interface design and underlying product development can be separated. For more information on the creation and structure of dialog resources, see "Standard Dialog Box Items" on page 16-1.

When “Dialog Box Builder” is selected from the MDE Tools menu, the following dialog box and tool palette appear:

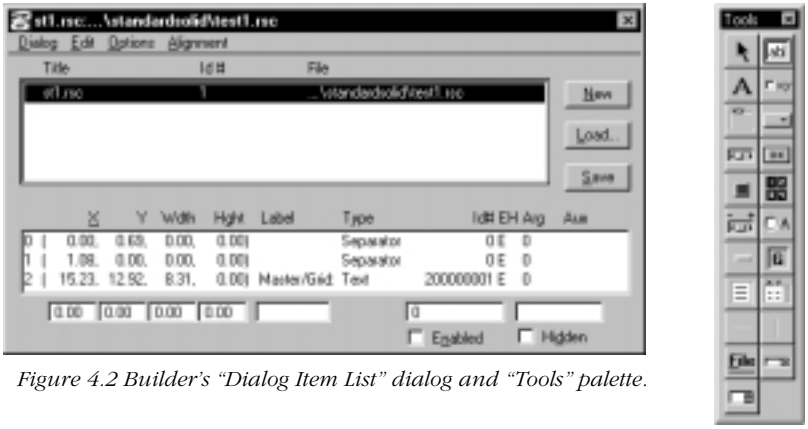


Figure 4.2 Builder’s “Dialog Item List” dialog and “Tools” palette.

Dialog Menu

The Dialog menu contains selections for creating, selecting and deleting resource files, and creating and selecting dialogs boxes to be edited. During an editing session, use Load to open as many dialogs as you like at one time. This allows you to compare dialog layouts for consistency, and to cut and paste items between dialogs to ensure that they are identical.

To	Use
Create a new output resource file	New
Open an existing output resource file	Load
Close selected dialog box	Close
Save changes to selected dialog box	Save
Save changes to exiting resource file	Save To Existing...
Save dialog to new resource file	Save To New...
Delete selected dialog box	Delete

New (Ctrl+N)

Creates a new, untitled dialog box. The new dialog is not associated with any resource file until you select one of Save, Save To Existing or Save To New.

Load (Ctrl+O)

Opens the Load Dialog dialog box. This dialog lets you select the resource file from which to open the dialog box, which dialog within the resource file to open, and the mode in which the dialog will be opened.

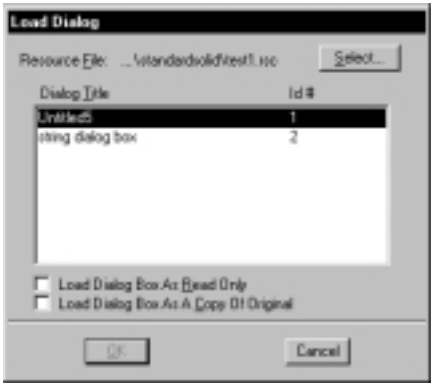


Figure 4.3
Load Dialog dialog box.

The dialog box you open will be listed in the Dialog field of the Dialog Item List dialog.

Load Dialog Box As Read Only

A dialog box can be opened for editing or merely for review. If this check button is highlighted, the dialog box may not be modified by the user. Items cannot be dragged, the OK button in change dialogs is grayed out, and none of the save options can be chosen from the Dialog menu.

Load Dialog Box As A Copy Of Original

If this check button is highlighted, a copy of the selected dialog box is made with no associated resource file. Any changes made are not applied to the original dialog and the dialog box is saved to the default Resource File provided that the Save To Existing option has been selected.

Close

Closes the dialog box that is currently selected. If unsaved changes exist in the selected dialog box, an Alert dialog box will appear, asking whether the original dialog box should be overwritten.

Save (Ctrl+S)

Saves any changes made to the selected dialog. An additional Alert dialog box will appear, asking whether the original dialog should be overwritten.

Save To Existing... (Ctrl+E)

Saves the selected dialog to an existing resource file. Saves are not allowed to be made to new resource files.

Save To New... (Ctrl+W)

Saves the selected dialog to a new resource file. Saves are not allowed to be made to existing resource files.

Delete (Ctrl+D)

Deletes the dialog box that is currently selected. An Alert dialog box will appear, asking whether the selected dialog should be removed.

Edit Menu

The Edit menu contains selections for moving, copying, deleting, selecting and arranging dialog box items. Cut, Copy and Paste can be used between multiple dialog boxes for accelerated dialog box creation.

To	Use
Cut the selected dialog item(s) to the clipboard	Cut
Copy the selected dialog item(s) to the clipboard	Copy
Paste dialog item(s) from the clipboard into the current dialog	Paste
Delete selected dialog item(s)	Delete
Select all the items in the current dialog	Select All
Bring the selected dialog item(s) to the front	Bring To Front
Send the selected dialog item(s) behind any other dialog items in the same area	Send To Back

Cut (Ctrl+X)

Cuts the selected item(s) from the dialog. The items are placed in the clipboard and can be pasted back into any open dialog. Any attributes associated with the items are cut with them. The cut items are removed from the dialog item list.

Copy (Ctrl+C)

Copies the selected item(s) to the clipboard without affecting the original items themselves. The copies of the items in the clipboard can be pasted into any open dialog. Any attributes associated with the items are copied with them.

Paste (Ctrl+V)

Pastes the item(s) in the clipboard into the dialog that has the input focus. Any attributes associated with the items are pasted with them. The pasted items are added to the end of the dialog item list.

Delete (Del)

Deletes the currently selected dialog box item(s).

Select All

Selects all the items in the current dialog. This function is equivalent to individually selecting each item while holding down the shift key or drawing a selection box around all the items.

Bring To Front

Brings the selected item(s) in front of all other items in the dialog. Using this command, you can control which items are selected first when you click an area that is occupied by more than one item. An item's front or back location affects its selectability both when the dialog is being edited and when the dialog is being used.

Send To Back

Sends the selected item(s) in back of all other items in the dialog. Using this command, you can control which items are selected first when you click an area that is occupied by more than one item. An item's front or back location affects its selectability both when the dialog is being edited and when the dialog is being used.

Options Menu

To	Use
toggle the dialog between edit mode and test mode.	Test Dialog
open the Preferences dialog box.	Preferences
open the Tools palette.	Tools
open the Dialog Box Attributes dialog box.	Dialog Attributes...
open the appropriate dialog item editor.	Item Attributes

Test Dialog

Toggles the dialog between edit mode and test mode. Edit mode is Builder's normal mode, in which dialogs and the items in them can be added, deleted, and modified. In test mode, the dialog behaves as it would in a normal MDL application; option buttons can be changed, check buttons toggled, text in text items can be edited, scroll bars moved, list boxes scrolled, color pickers selected, and level maps changed. Test mode allows the dialog box designer to test the "usability" of the dialog box's layout.

All linkages to the application code (such as dialog item hooks and access strings) are not connected and do not function in Builder's test mode.

Preferences

Opens the Preferences dialog box. A value having units of Average Character Height is provided in the Vertical Spacing text item. Multiple dialog box items which are selected can then be separated by this value when the Alignment->Space Vertically menu item is chosen.

Figure 4.4
Preferences dialog box.

Tools

Opens the Tools palette. The Tools palette is used to create new dialog items.

Dialog Attributes...

Opens the Dialog Box Attributes dialog box. In this dialog you can change the dialog's many attributes, such as its name, modality, help Id#, type, width, height and hook Id#.

The Dialog Box Attributes dialog can also be opened by double-clicking in an unoccupied area of the dialog box whose attributes are to be changed.

Item Attributes... (Ctrl+A)

Opens the attributes dialog box that is associated with the type of item that is selected. Fifteen different attribute dialogs exist, one each for text, label, checkbutton, groupbox, optionbutton, scrollbar, pushbutton, multiline text, listbox, colorpicker, levelmap, scale, radiobutton, sash and generic items.

Item attribute dialogs can also be opened by double-clicking the item whose attributes are to be changed. Item attribute dialogs do not exist for the horizontal and vertical separator items.

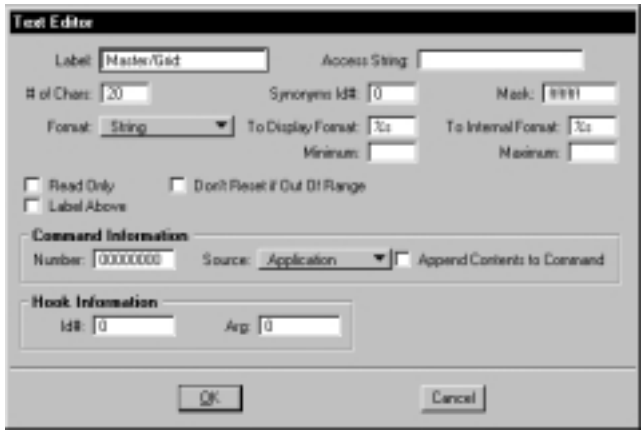


Figure 4.5
Text item attributes dialog box, can be opened by double-clicking on any text item.

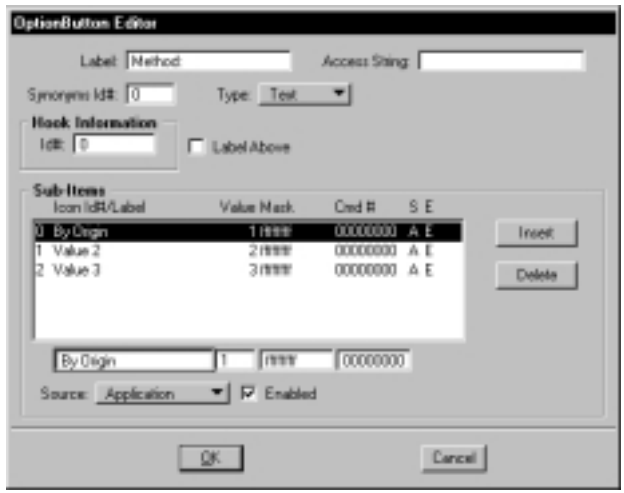


Figure 4.6
Option Button item attributes dialog box can be opened by double-clicking on any option button item.

Figure 4.7
Push Button item attributes dialog box can be opened by double-clicking on any push button item.

Figure 4.8
Label item attributes dialog box can be opened by double-clicking on any label item.

The Tools Palette

The tool palette has many more types of dialog items when used in the mdl environment than when used in the MicroStation Basic environment.

To	Use	
Select one or more items to be modified. .	Item Selection Tool	
Add a single-line, user-editable, text box.	Place Text Item	
Add static text label.	Place Label Item	
Add a togglebutton.	Place ToggleButton Item	
Add a labeled box to group related items.	Place GroupBox Item	
Add a user-selectable, option button.	Place OptionButton Item	

To	Use	
Add a scroll bar.	Place ScrollBar Item	
Add a push button.	Place PushButton Item	
Add a color-picker item.	Place ColorPicker Item	
Add a level map item.	Place LevelMap Item	
Add a scale item.	Place Scale Item	
Add a radio button.	Place RadioButton Item	
Add a sash item.	Place Sash Item	
Add a generic item.	Place Generic Item	
Add a multi-line, user-editable, text box.	Place Multiline Text Item	
Add a user-scrollable list box item.	Place ListBox Item	
Add a horizontal separator.	Place Horizontal Separator	
Add a vertical separator.	Place Vertical Separator	
Add a menu bar item.	Place MenuBar Item	
Add a combo box item.	Place ComboBox Item	
Add a spin box item.	Place SpinBox Item	

Dialog Item Placement

Dialog items are placed using their respective tools listed on the previous page. Items are placed with generic names and sizes. Names and other item attributes can be changed using the methods described in the “Item Attributes...” section above. Item sizes and positions can be manipulated in the Dialog Item List dialog.

The default item placement mode is “single shot;” that is, if you single-click on the item placement tool, only one occurrence of the item can be

placed before Builder reverts to the item selection tool. However, if you double-click on an item placement tool, Builder enters a mode where every data point placed in the dialog creates another occurrence of the item. Reset returns to the item selection tool.

Alignment Menu

The Alignment menu contains selections that will help you create dialog boxes with aligned items for consistency. Many of the selections, when used in conjunction with one another, can save you large amounts of time from manually aligning dialog items.

Align [Left | Right | Top | Bottom]

Aligns all the selected items by their left, right, top or bottom edges. Left and right alignment does not affect vertical position, and top and bottom alignment does not affect horizontal position.

Center on Horizontal, Center on Vertical

Aligns all the selected items about a common horizontal or vertical axis. Horizontal centering does not affect vertical position, and vertical centering does not affect horizontal position.

Spread Horizontally, Spread Vertically

Spreads all the selected item(s) out an equal distance from one another and from the edges of the dialog box. Horizontal spreading does not affect vertical position, and vertical spreading does not affect horizontal position.

Space Vertically

Aligns all the selected items so that they adjoin one-another vertically. Horizontal position is not affected. This menu bar item can be used in conjunction with the Preferences dialog box discussed earlier.

Make Same Size

Resizes all selected items to the same size. The size to which the items are changed is determined by the first item selected. Vertical and Horizontal position are not affected.

Icon Editor

The Icon Editor is used to create and edit compiled (binary) icon resources in a **.ma** or **.rsc** file. You can create and edit icons in either the standard MicroStation icon sizes (23 or 31 pixels square) or of any size between 5 and 100 pixels on an axis.

➤ To edit an existing icon resource

1. From the Tools menu, choose Icon Editor.
   The Icon Editor dialog box opens.
2. From the File menu in the Icon Editor dialog box, choose Open File.
   The Resource File to Open dialog box opens.
3. Select the file containing the icon resource you wish to edit and click OK.
   The Open Icon Resource dialog box opens and displays the list of icon resources that can be opened and edited.
4. Select the icon resource you wish to edit and click the OK button.
   The icon resource is then displayed for editing in the Icon Editor dialog box.
5. At this point, you can edit the icon by using the Brush, Line, Rectangle and Circle drawing tools to set, clear and toggle pixels. The tool, pixel setting mode (Set, Clear or Toggle), and icon size are set with option menus in the Icon Editor dialog box. To shift the position of the entire icon one pixel at a time, use the buttons below the pixel map.

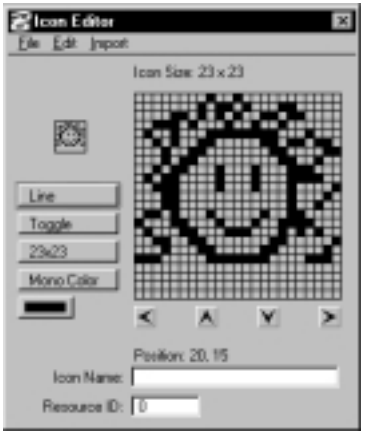


Figure 4.9
Icon Editor dialog box.

File menu

New File (Ctrl+N)

Closes open icon resources without saving and initializes a new icon resource for editing. To set the icon size, use the bottom option menu. To specify a non-standard size, choose User from the option menu and use the User Size dialog box.

Open File... (Ctrl+O)

Opens the Resource File to Open dialog box, which is used to open a .ma or .rsc file.

➤ **To open a .ma or .rsc file**

1. Select the MDL application or binary (compiled) resource file that contains the icon resource you want to edit and click the OK button. The Open Icon Resource dialog box opens.
2. Select the number of the icon to edit and click OK. The icon is displayed for editing in the Icon Editor dialog box.

New Icon

Closes the open icon resource and creates a new icon resource for editing in the open resource/application file if one exists. To set the icon size, use the bottom option menu. To specify a non-standard size, choose User from the option menu and use the User Size dialog box.

Open Icon... (Ctrl+I)

Opens a different icon resource from the open resource/application file.

Save (Ctrl+S)

Saves the open icon resource to the original file containing the resource. If no file was opened, the Save to File dialog box is opened for you to select an already existing binary (.rsc or .ma) file in which to save the icon resource.

Save To...

Opens the Save to New Resource File dialog box, which is used to save the open icon resource to a new or existing binary (.rsc or .ma) resource file.

Exit

Exits the Icon Editor.

Edit Menu

Fill Shapes

If Fill Shapes is on, the current pixel operation mode (Set, Clear or Toggle) is applied to all pixels inside the boundaries of a shape placed with the Circle or Rectangle tool.

Clear

Clears the current icon map and name information (that is, clears all pixels and the icon name in the current icon definition).

Shift [Left | Right | Up | Down]

Shifts the icon one pixel left, right, up, or down. Any icon pixels that are shifted off the side of the editing grid are lost.

Import Menu

Graphics

Imports graphics with the current fence content. A fence must be place prior to executing the command.

Cell

Imports the graphic representation of a cell from the current cell library.

Resource Development Environment (RDE)

The Resource Development Environment is a tool for creating and modifying resource source files and header source files. Resource source files, .r files, contain a description of resources such as dialog box item resources. Header files contain the definitions of the macros used in the description of resources. The application is started with the MicroStation command, `mdl load rde`, (or via the MDE user interface).

Macros refer to the `#define` preprocessor declarations in header files. A binary resource file (.rsc file) or resource source file (.r file) can be loaded into RDE. When you choose to edit a resource, the appropriate graphical resource editor, such as the Dialog Builder, is launched.

In the case of the Dialog Builder resource editor, it is possible to modify the names and contents of macros. See “Dialog Box Builder” on page 4-4 for further details.

The graphical resource editors manipulate the binary form of the resource. However, any changed or new resources are saved into the resource text file and its appropriate header files (.h files).

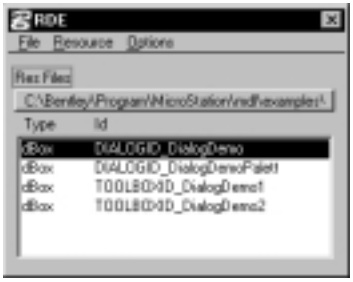


Figure 4.10
Resource Development Environment dialog box.

File Menu

The File menu contains selections for creating and selecting resource files. Multiple files may be open simultaneously.

To	Use
Create a new resource file	New…
Open an existing resource file	Open…
Close an opened or newly created resource file	Close
Generate source code for a binary resource file	Save
Exit	Exit

New… (Ctrl+N)

Opens the "New Resource File" dialog box for the user to create a resource source file. Typically filenames have a .r extension. The file only gets created when the user saves newly added resources to the resource file. The Default header files for all language dependent text and for the header file containing the id values of the resources are named. The binary resource file that is used by the graphical resource editor is also automatically given a default name. The header files have the same path as the resource source file but have a .h extension. The binary resource file has got the extension .tmp which indicates that this is a temporary file that will be destroyed when you have finished with the resource file.

Open… (Ctrl+0)

Opens the "Select Resource File" dialog box. The file can be opened as either a binary resource file (.rsc) or a text resource file (.r). If the file is opened as a source file, it is compiled and a binary resource file with a .tmp extension is generated. If the file is

opened as a binary resource file, the source is generated in exactly the same way as a new resource source file. A list of all the resources within the opened resource file is displayed in the list box of the RDE dialog box.

Close (Ctrl+C)

Closes the currently selected resource file and deletes the binary resource file if it is a temporary file, i.e., has a .tmp extension. Multiple resource source files can be open simultaneously. The currently selected source file is displayed in the "Res Files" option button in the RDE dialog box. All editable resources are displayed within the RDE dialog's listbox.

Save (Ctrl+S)

Generates source code for the currently selected resource file. The resource source file and its appropriate header files are updated. Read only files and files in the system include directory are not updated. A backup copy of each saved file in the form <filename.ext> is created, where .ext is the original extension with a "2" appended on the end of it. To modify existing source code, it is necessary to save the changes from the graphical resource editor. Presently, the Builder application is the only resource editor which saves both the binary and text formats of the resource. The save option is equivalent to selecting the save option from the Dialog Builder. Save is only enabled for new resource files.

Exit (Ctrl+X)

Exits RDE.

Resource Menu

The Resource menu contains selections for adding a new resource or editing an existing resource.

To	Use
Add a new resource	Add…
Edit an existing resource	Edit…

Add… (Ctrl+A)

Adds a new dialog box resource to the current resource file. If the Dialog box builder application is not loaded, it is launched and a new dialog box resource is displayed. You can edit the new resource and save it from the builder application.

Edit... (Ctrl+E)

Launches the dialog box builder application if it is not already loaded and displays the dialog box that you select to edit. This operation can also be accomplished by double clicking on the resources list box in the RDE dialog box.

Options Menu

The Options menu contains selections for displaying the default files and the include paths used to search for header files of a resource source file.

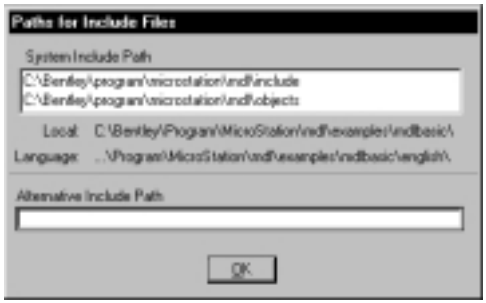


Figure 4.11
Paths for Include Files dialog box.

To	Use
Show the include paths for the runtime compiler	Include Dir...
Show the default resource and header files	Default File...

Include Dir...

Displays the "Paths for Include Files" dialog box. The system, local, and language include directories are read-only items that cannot be modified. Only the alternative include directories can be modified. The format for the alternative include directories is:

<directory[PATH SEPARATOR[directory]]>

where the [ ] indicate optional parameters. Note the last character of this field must NOT be a path separator.

The system include directories default can be overwritten by setting the `MS_RDE_SYSINC` configuration variable. The local include path contains the directory where the

resource file was opened. The language include path contains the language subdirectory off the local include path directory.

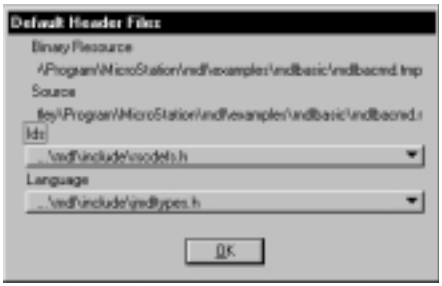


Figure 4.12
Default Header Files dialog box.

Default File...

Displays the default resource source file, binary resource file, default ids file and language header file. Only the ids and language header files may be modified. The grayed out option items in these option buttons indicate read only files.

Opening and Closing Files

When a resource source file is opened, it is compiled and a temporary binary resource file is created. The File > Close menu option destroys this temporary file and removes the resource file name from the Res File option button.

If the resource source file cannot be opened, the "Compiler Error Report" dialog box displays a report of the reasons for not being able to compile the resource file. The most common reason is that an include path has been omitted so the compiler couldn't find a header file. To fix the problem add the appropriate header file by pressing the "Inc Files..." pushbutton. This opens the "Paths for Include Files" dialog box. This dialog box can also be selected from the Options > Inc Dir... menu option. The omitted directory can be typed into the alternative include paths text item.

Example of creating source code for a new resource

The process of creating new resources is simple and straightforward.

➤ **To create a dialog box with a few items**

1. From the RDE File menu choose New...
   The New Resource File dialog box is displayed.
2. In the New Resource File dialog, key-in the name of the resource file you want to create with a **.r** extension. Click the OK button.
   The new resource filename is displayed in the Res File option button in the RDE dialog box. The default header files for the IDs and the language-dependent portion are given default names.

3. From the RDE Resource menu choose Add…
   The graphical dialog box builder is loaded and a new, editable Untitled dialog box is displayed.
4. Using the dialog builder, add a text item and a toggle button to the dialog box. Refer to the chapter on "Adding dialog boxes to Macros" for more details about using the Dialog Builder graphical editor.
5. Save the dialog box from the Dialog Box Builder by pressing the Save push button or by selecting the Fil e >Save menu item.
   The resource file, the IDs header file, and the language header file are generated in the same directory. Once the resource has been saved the dialog box is displayed in the Resource list box of the RDE dialog box.
6. From the RDE File menu choose Close.
   The resource source file is closed and removed from the "Res Files" option button in the RDE dialog box. Check that the resources were created properly by reopening the resource source (**.r**) file that you created in RDE.

It is also possible to create a new resource by loading a .rsc file and choosing the File > Save menu option in RDE.

Example of modifying source code for an existing resource

Modifying a resource involves a similar workflow to creating a new resource.

➤ **To create a new resource**

1. Choose RDE File > Open…
   The "Select Resource File" dialog box displays.
2. In the Select Resource File dialog, key-in the name of the resource file you want to create with a **.r** extension. Click the OK button.
   The new resource filename is displayed in the "Res File" option button in the RDE dialog box. The default header files for the IDs and the language-dependent portion are given default names.
3. From the list box in the RDE dialog box, double click on the dialog box that you want to edit.
   or

   Choose Resource > Edit… from the RDE dialog box menu.

   The graphical dialog box builder is loaded and the selected dialog box is displayed.
4. Using the dialog builder, move the dialog items around and/or change the attributes of the dialog box or the items.
5. Save the dialog box from the Dialog Builder.

An alert dialog box displaying the message "Update Source Code for dialog id #NUM" is shown (where `#NUM` is the number of the dialog box that you are saving.

6. Press the OK push button to generate new source for the dialog box resource that you created in builder.
   The resource file and all the header files in which macros have changed are updated.

7. Choose File > Close from the RDE dialog box menu.
   The resource source file is closed and removed from the "Res Files" option button in the RDE dialog box and the temporary binary resource file is deleted.

Limitations

Complex macros are defined in terms of other macros, e.g. the complex macro is made up of sub-macros. If the value of a complex macro changes, it is very difficult to determine which sub-macro has changed. RDE does not modify these complex macros but instead replaces the field of a resource with a value. For example, suppose we have the following macro and resource definition. Before the value of the `WIDTH` macro changes:

```
#define WIDTH 20*XC+OFFSET /* XC and OFFSET are submacros */
DialogBoxRsc DIALOGID_Elements
{
   DIALOGATTR_DEFAULT, WIDTH, HEIGHT
   ...
}
```

After the width field is modified:

```
#define WIDTH 20*XC+OFFSET
DialogBoxRsc DIALOGID_Elements
{
   DIALOGATTR_DEFAULT, 25*XC, HEIGHT
   ...
}
```

If the macro `WIDTH` is the width specification field of a dialog box resource that changes, it will be replaced with an expression in terms of dialog coordinate units such as `25*XC`. The definition of the `WIDTH` macro is not modified.

RDE does delete resources but comments out the resource with an `#ifdef (HAS BEEN DELETED) ...#endif` statement. Macros are not deleted but a comment is displayed indicating that the macro is possibly not being used. When the user deletes a dialog box resource from builder, the row containing the dialog box resource is removed

from the resources list box. However, no changes are made to the source code until you perform the save option.

RDE treats subresources, enclosed by an `#ifdef ... #endif` statement, as being commented out. However, the resource compiler may compile this same code. This causes the indices of the subresources in the binary format to not correspond to those of the source representation. When this resource is modified, the source is not updated correctly. In order to correctly update this resource, temporarily remove all `#ifdef ... #endif` comments.

Individual resources cannot be saved. The entire resource file with all the changes is updated.

Any modifications to the database cannot be undone. If a macro name or value is changed, it will be updated even if the resource was not saved. For example, if the macro values in resource 1 are modified but not saved and resource 2 is saved, all the changes in resource 1 will automatically be saved as well. If the user makes changes to a dialog box and then closes the dialog box without saving, the dialog box resource is dimmed out in the resources listbox of the RDE dialog box. This resource may not be edited. To edit the resource, the file containing the resource must be closed and reopened.

Macro Description Dialog Box

Although Builder works with resources in binary format, you can also add and change the macro of a field in a resource.

The Macro Description dialog box opens when the user double-clicks in a text field in the dialog attributes dialog box or one of the dialog item editors dialog boxes. The Macro Description dialog is opened only if the dialog box has source code associated with it, i.e., the dialog box has been opened within RDE. The word macro is used in a different context here as opposed to BASIC macros. Here, macros refer to `#define` preprocessor declarations used in header files.

Use the dialog box to modify values of macros and its contents. The macro refers to the text field you double-clicked on to bring up this dialog. You can change an existing macro by changing the New value text field or you can create a new macro by typing in a new name in the macro field. If you type in an already existing macro, the contents of the macro are automatically displayed.

Language specific macros are enclosed in quotes. When editing the value of this type of macro, you must ensure that the enclosing quotes are retained.

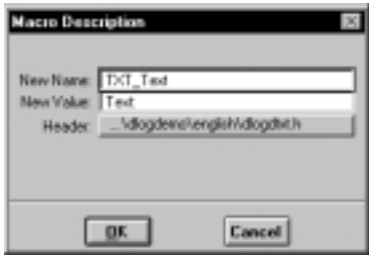


Figure 4.13
Macro Description
dialog box.

String List Editor

The String List Editor is used to create and edit compiled (binary) message list resources in a **.ma** or **.rsc** file. It is not yet possible to edit other types of string lists (that is, string lists with more than one information field).

➤ To edit an existing message list resource

1. From the Tools menu, choose String List Editor.
   The String List Editor dialog box opens.
2. From the File menu in the String List Editor dialog box, choose Open File.
   The Resource File to Open dialog box opens.
3. Select the file containing the message list resource you wish to edit and click OK.
   The Open Message List Resource dialog box opens and displays the list of message list resources that can be opened and edited.
4. Select the message list resource you wish to edit and click the OK button.
   The message list resource is then displayed for editing in the String List Editor dialog box.

   At this point, you can edit a message list entry in the message list resource by selecting the list entry and then either pressing <Tab> or <Return> or clicking in the Edit: field of the dialog box. This causes the string for the message to be displayed for editing in the Edit: field. After you edit the string, press <Tab> or <Return> to accept the string modifications. To save the changes, choose Save or Save To from the File menu.

File menu

New File (Ctrl+N)

Closes any open message list resources without saving and initializes a new message list resource for editing.

Open File... (Ctrl+O)

Opens the Resource File to Open dialog box, which is used to open a .ma or .rsc file.

➤ **To open a .ma or .rsc file**

1. Select the MDL application or binary (compiled) resource file that contains the message list resource you want to edit and click the OK button.
   The Open String List Resource dialog box opens.
2. Select the number of the message list resource to edit and click OK.
   The messages are displayed for editing in the String List Editor dialog box.

New Message List

Closes the open message list resource and creates a new message list resource for editing in the open resource/application file if one exists.

Open Message List...

Opens a different message list resource from the open resource/application file.

Figure 4.14 String List Editor dialog box.

Save (Ctrl+S)

Saves the open message list resource to the original file containing the resource. If no file was opened, the Save to File dialog box is opened for you to select an already existing binary (.rsc or .ma) file in which to save the message list resource.

Save To...

Opens the Save to New Resource File dialog box, which is used to save the open message list resource to a new or existing binary (.rsc or .ma) resource file.

Exit

Exits the String List Editor.

Edit Menu

Delete Member (Del)

Deletes the selected message list string (the text is removed from the entry but the entry remains).

Append Members (Ctrl+A)

Appends 10 empty list entries to the end of the currently open message list resource.

5 A Comparison of MDL and C

Closely resembling ANSI C, the MDL language is a dialect of the C programming language. This chapter describes the MDL language and compares it to ANSI C.

This chapter also discusses the pragmas supported by the MDL compiler (**mcomp**) and tells how the MDL compiler handles structure definitions. Understanding how **mcomp** processes structure definitions is important if data will be shared between MDL programs and other C programs. **mcomp** is discussed fully in "Building Applications" on page 7-1.

Major Differences Between MDL and ANSI-C

MDL C has these major differences from ANSI-C:

- In ANSI-style function declarations, **mcomp** does not support types smaller than integer except on the Windows platform. If **mcomp** encounters a `char` or `short` declaration in an ANSI-style function declaration, it produces an error, because the character size is smaller than the integer size. This restriction does not affect pointer declarations such as `char *`, since the pointer size and integer size are the same.
- **mcomp** supports structure assignments. However, structures cannot be passed as function arguments and functions cannot return structures.
- MDL provides many built-in functions and variables that **mcomp** recognizes. These functions and variables cannot be redefined, although they can be redeclared. For example, `malloc` is an MDL built-in function. The `char *malloc();` declaration can appear in the source code. However, if **mcomp** encounters a declaration for `malloc` and the declaration includes a function body, **mcomp** generates an error message saying that built-in functions cannot be replaced.

- Local variable space used for automatic variables is restricted to 32 Kb per function.
- `cmdName` and `cmdNumber` are reserved words. `cmdName` specifies that the function is a command and that its name can be used in a MicroStation MDL COMMAND key-in. For more information on this command, see "Running MDL Applications" on page 8-1. `cmdNumber` followed by a list of command numbers specifies that the function is a command that can be accessed from the list of command numbers. See "Command Parser" on page 6-14 for more information.
- Enumerated types are not supported.
- Bitfields must be unsigned.
- `float` types are treated as `double`. `float` is a synonym for `double`.
- A storage class must be specified before the type. For example, `extern int test;` is accepted, but `int extern test;` is not.
- In each compile, an include file with a given name is read only once, even if several `#include` directives request it.
- Identifier length is unrestricted.
- The **mcomp** preprocessor does not support trigraphs, stringization, or token pasting/concatenation. However, it supports string concatenation and the following preprocessor directives: `#include`, `#define`, `#undef`, `#if`, `#ifdef`, `#ifndef`, `#line`, `#else`, `#elif` and `#endif`. Both `#if` and `#elif` require an expression that results in an integer constant. For example, `#if defined (mdl)` is valid.
- All lines containing preprocessor directives must contain # as the first character. This differs from ANSI C, where the line can contain white space before the #.

Pragmas

Pragmas are preprocessor directives that provide MDL-specific information in C source files. To specify a pragma, enter `#pragma <pragma-name>` followed by the arguments. The following pragmas are available:

Pragma: alias

Syntax

```
#pragma alias "aliasname"
```

Summary

Assign an alias name to a static resource definition in a **.r** file.

Relevant Compilers

rcomp

Detailed Description

Typical resource definitions in a **.r** file only allow for the type and ID of a resource declaration. The type ID, or `resourceClass` of a resource is assigned to the `typedef` used to declare the resource instance.

```
typedef struct mystruct
{
     int member;
} MyStruct;
resourceClass MyStruct RTYPE_MyTypeID;
```

The individual resource ID is assigned in the declaration of a specific resource:

```
MyStruct RSCID_MyID=
{
     5
};
```

Optionally, an alias could be assigned to a specific resource using the `alias` pragma. The pragma must be specified within the opening bracket, '`{`', of the resource initialization stream.

Example

```
MyStruct RSCID_MyId=
{
     #pragma alias "myAlias"
     5
};
```

Such a compiled resource could be loaded at runtime with the `mdlResource_loadByAlias` function call:

```
MyStruct *rscDataP;
rscDataP = mdlResource_loadByAlias(rfHandle, RTYPE_MyTypeId,
                                   RSCID_MyId, "myAlias");
```

pragma: incompatiblePointerParameters, incompatiblePointers, incompatibleReturn, noAnsiDeclaration, noReturnStatement, undeclaredFunction

Syntax

```
#pragma <name> <option>
```

The valid values for `<name>` are `incompatiblePointerParameters`, `incompatiblePointers`, `incompatibleReturn`, `noAnsiDeclaration`, `noReturnStatement` and `undeclaredFunction`.

Summary

The valid values for `<option>` are `push`, `pop`, `warn`, `error` and `ignore`. The option `push` must be followed by `warn`, `error` or `ignore`. If `push` is specified, the previous setting for `<name>` is saved before the new value is used. Each of these pragmas use the following syntax for restoring a previous value:

```
#pragma <name> pop
```

Relevant Compilers

mcomp

General Description

These pragmas were all added to **mcomp** in MicroStation 95. The MicroStation 95 **mcomp** detects many possible problems that previous versions did not detect. These pragmas control whether **mcomp** ignores these conditions, reports the condition as error or reports the condition as a warning.

Detailed Description, incompatiblePointerParameters

The pragma `incompatiblePointerParameters` controls what happens when **mcomp** compares different declarations of the same function and finds that one of the parameters is a pointer in both declarations but they are pointers to different types in the different declarations. If the option is `error`, it reports an error. If the option is `warn`, it issues a warning when pointers do not point to the same type. The default setting of this pragma is `warn`. That is, if the pragma is not used **mcomp** issues a warning whenever it encounters incompatible pointer parameters. If the option is `ignore`, it does not report the difference, **mcomp** considers the two parameters equivalent if both are pointers. It does not matter to what type they point. To illustrate, `incompatiblePointerParameters` controls how the compiler responds to the following condition:

```
int function_1(int *pArg);
int function_1(double *parg);
```

Example, incompatiblePointerParameters

The following statement causes **mcomp** to generate an error if two declarations of a function have conflicting pointer parameters.

```
#pragma incompatiblePointerParameters error
```

To change the handling of incompatible pointer parameters for a section of code, bracket the section of code with:

```
#pragma incompatiblePointerParameters push error
```

and

```
#pragma incompatiblePointerParameters pop
```

Detailed Description, incompatiblePointers

The pragma `incompatiblePointers` is used when comparing types in an assignment statement or passing an argument to a function. `incompatiblePointers` controls how the compiler responds to the following condition:

```
int function_1(int *pArg);
int function_2(double *p)
{
     return function_1(p);    /* function_1 expects "int *" */
}
```

That is, `incompatiblePointers` determines how **mcomp** responds to detecting incompatible pointers. This applies to any place that **mcomp** may compare pointers except for checking multiple declarations. It applies to assigning to a pointer variable and passing parameters. It does not apply to explicit casts, only to implicit. The following example shows how to use the explicit cast to override the error checking:

```
{
     int *pInteger;
     double *pDouble;

     #pragma incompatiblePointers ignore
     pInteger=pDouble;             /* pragma, NO message */

     #pragma incompatiblePointers warn
     pInteger=pDouble;             /* warning is issued */
     pInteger=(int *)pDouble;      /* cast, NO message */
}
```

The default value for `incompatiblePointers` is `warn`.

Detailed Description, incompatibleReturn

The pragma `incompatibleReturn` controls what happens when **mcomp** compares different declarations of the same function and finds that their returns are pointers to different types. If the option is `error`, it reports an error. If the option is `warn`, it issues a warning. If the option is `ignore`, it does not report the difference. If the pragma is not used, a warning is issued.

Detailed Description, noAnsiDeclaration

The pragma `noAnsiDeclaration` controls what happens if **mcomp** detects a function call using the name of a function that has not been declared with a prototype. As with the other pragmas in this section, it can ignore it or can issue an error or warning. Note that a declaration with no argument list and no function body is not considered a prototyped declaration. It tells the compiler that nothing is known about the arguments. To tell the compiler that a function takes no arguments, use (`void`) as the argument list. That is:

```
function_1();
```

says nothing is known about the arguments but:

```
function_1(void);
```

says that the function has no arguments. A declaration with no argument list and a function body tells the compiler that there are no arguments. Therefore, it is considered a prototyped function declaration. The default setting for the pragma `noAnsiDeclaration` is ignore.

Two of the most common conditions caught by this pragma are function declarations of the format `int function();` and built-in functions that are not declared. The declaration `int function();` tells **mcomp** that nothing is known about the parameters. This severely limits the ability of the compiler to detect errors. **mcomp** has built-in knowledge of the built-in functions, but knowledge is limited. More information can be provided by including the appropriate **.fdf** and **.h** files in your program.

Example: The first of the following examples causes **mcomp** to generate an error if it encounters a call to a function that does not have an ANSI-style declaration. The second example pushes the current setting of `noAnsiDeclaration`, and sets it to warn. The final example restores the previous setting for `noAnsiDeclaration`. In this example, it restores it to error.

```
#pragma noAnsiDeclaration error
#pragma noAnsiDeclaration push warn
#pragma noAnsiDeclaration pop
```

Detailed Description, noReturnStatement

The `noReturnStatement` pragma controls what **mcomp** does when it detects a function that is missing a return statement. Although it is valid in C to omit a return statement, it is generally an error. As with the other conditions described in this section, **mcomp** can ignore it or issue an error or warning. By default, **mcomp** issues a warning.

The following example illustrates a coding style that did not generate a warning prior to MicroStation 95:

```
cmdName myCommand()
{
    ...
    /* no return statement */
}
```

Starting with MicroStation 95, **mcomp** would generate a warning for this function. Since the function declaration does not specify a return type, `int` is implied. The compiler issues a warning because this is an int function that does not contain a return statement. To eliminate the warning without using the pragma, change the declaration to:

```
cmdName void myCommand()
{
    ...
    /* no return statement */
}
```

To force **mcomp** to generate an error every time it detects a function missing a return statement, add the following to a standard header file:

```
#pragma noReturnStatement error
```

Detailed Description, undeclaredFunction

The pragma `undeclaredFunction` controls what happens if **mcomp** detects a function call using an undeclared identifier. This is useful because calling undeclared functions limits the compilers effectiveness in detecting some errors. If a function is not declared, **mcomp** does not know anything about the parameters and assumes the return type to be `int`. The default value of `undeclaredFunction` is ignore. As with the other pragmas in this section, it can ignore it or issue an error or warning.

pragma: options

Syntax

```
#pragma options ...
```

Summary

Added to MDL only to eliminate warnings that occur on the `options` pragmas that are in the MDL header files.

Relevant Compilers

mcomp, rcomp, rsctype

Detailed Description

The options pragma is in MDL header files only because it is required for the native C compiler on the RS6000. They are not required for MDL. Since the MDL preprocessor issues a warning whenever it encounters pragma that it does not recognize, these pragmas caused a lot of warnings. Now the preprocessor recognizes these macros and does nothing.

pragma: packedLittleEndianData

Syntax

```
#pragma packedLittleEndianData
```

Summary

Switch the compiler into a mode where all compiled data is generated in packed little endian format.

Relevant Compilers

rsctype

Detailed Description

This pragma is used by the **rsctype** compiler in the context of creating data definitions to be used by the binary portability (`mdlCnv_bufferToFileFormat` or `mdlCnv_bufferFromFileFormat`) or element linkage (`mdlLinkage_*`) MDL functions.

Example

Here is an example of the `packedLittleEndianData` pragma as used in a **.mt** file. The pragma is used to switch the compiler into packed little endian data mode. A data definition will be generated with an identifier of `MYLINKAGE_DATADEF_ID` that describes the format of the data within `mylinkagestruct` as it would be found in a user attribute data linkage on a **.dgn** file element.

For more information on this subject, see "Binary Portability" on page 6-27.

```
#pragma packedLittleEndianData
#include <myfile.h>      /* defines "mylinkagestruct" */
createDataDef(mylinkagestruct, MYLINKAGE_DATADEF_ID);
```

pragma: pointerToNative

Syntax

```
#pragma pointerToNative <typedef_name> [,] ...
```

Summary

Used to specify that a function pointer points to a function in a DLM, not a function in an MDL program.

Relevant Compilers

mcomp

Detailed Description

When the MDL compiler compiles code that uses a function pointer, it must know whether that pointer points to an MDL function or a nativeCode function. By default, function pointers in MDL refer to MDL functions. A `typedef` can be designated as a pointer to a native-code function through the use of the `pointerToNative` pragma.

The `pointerToNative` pragma accepts a comma-separated list of `typedef` names. Each `typedef` must have type pointer-to-function.

Example

To create a function pointer that points to native-code functions, use:

```
typedef int (*FuncP)();
#pragma pointerToNative FuncP
```

pragma: resourceID

Syntax: `#pragma resourceID 'MyId'`

Summary

Used to assign a resource ID to an MDL program. This pragma is useful only when an application file contains more than one program. If the application file contains only one program, specifying `resourceID` is not useful.

Relevant Compilers

mcomp

Detailed Description

The MDL linker always assigns a resource ID to each MDL program it creates. By default, it assigns the resource ID 'main'. It is not possible to store two programs with the same resource ID in the same resource file. Each program must have a unique resource ID. Therefore, if two MDL programs are to be stored in the same MDL file, it is necessary to assign something other than the default resource ID to one of the programs. To do so, use the `resourceID` pragma. If the application contains only one program, specifying the resource ID is not useful.

The `resourceID` pragma requires a constant integer expression as the resource ID. Typically, the integer expression is given as a four character integer expression.

pragma: ident

Summary

Used to brand a resource file with an identifying string.

Relevant Compilers

rcomp

Detailed Description

The first 64 bytes of every binary resource file is a text string used to describe the file. By default, this string will be filled in by the resource compiler with a generic description like:

```
"MicroStation Resource File - Version 5.0"
```

An application can use the Ident pragma to generate it's own identifying string for a .rsc file.

Example

If a .r file contained:

```
#pragma Ident "My resource file version zero"
```

The resulting binary resource file will have that string contained within it, starting at file offset zero.

pragma: suppressREQCmds

Syntax

```
#pragma suppressREQCmds
```

Summary

Used to eliminate unnecessary CMD_ macros from generated command table header files.

Relevant Compilers

rcomp

Detailed Description

In some cases, a third party application may wish to expand on a class of commands that exist in MicroStation. For example, MicroStation has a PLACE LINE command, and a third party developer may wish to expand on PLACE to also recognize a PLACE WIDGET subcommand. The suppressREQCmds pragma suppresses the generation of CMD_ macros in the generated command header files which would otherwise conflict with definitions in the delivered header file cmdlist.h. This is only necessary to prevent subsequent warnings that get generated in .mc files that include both cmdlist.h and local header files with redundant base commands. See Example below for details.

Example

In the example below, the third party application creates a command table for a PLACE WIDGET subcommand off the PLACE base command:

```
#include "rscdefs.h"
#include "cmdclass.h"

#pragma suppressREQCmds
```

```
#define CT_NONE     0
#define CT_MAIN     1
#define CT_PLACE    2
#define CT_WIDGET   3

Table CT_MAIN=
{
     {1, CT_PLACE, 0, REQ, "PLACE"},
};

Table CT_PLACE=
{
     {1, CT_WIDGET, 0, 0, "WIDGET"},
};

Table CT_WIDGET=
{
     {1, CT_NONE, 0, DEF, "OVER"},
     {2, CT_NONE, 0, 0, "UNDER"},
};
```

The header file resulting from the compilation (with `-h` option) looks like this:

```
#define CMD_PLACE_WIDGET      0x01010000
#define CMD_PLACE_WIDGET_OVER      0x01010100
#define CMD_PLACE_WIDGET_UNDER     0x01010200
```

Without the `suppressREQCmds` pragma, an additional entry would have been generated:

```
#define CMD_PLACE        0x01000000
```

The last macro would conflict with the `CMD_PLACE` macro in the delivered header file **cmdlist.h** if both header files were included in the same **.mc** file.

pragma: translate

Syntax

```
#pragma translate "filename.ext"
```

Summary

Used to specify a translation table for compile-time character conversions of characters in string constants. Usually specified at the top of a resource source, **.r**, file.

Relevant Compilers

mcomp, rcomp

Detailed Description

This pragma is used in conjunction with the `-t` option of the **rcomp** and **mcomp** compilers. The `-t` option is used to specify the directory from where translation tables, (**filename.ext**) should be loaded. Every time the compiler encounters a string constant, each character is passed through the translation table along it's way to the compiled output file.

This pragma is most often used within the context of converting **.r** files encoded in one of several DOS Code Pages to the (International Standards Organization) ISO 8859-1 character encoding (sometimes referred to as the Windows ANSI Character Set).

Several character translation tables are delivered with the DOS version of MicroStation.

`$MS`**/chartran/eng1437.rsc** Code Page 437 (English)
`$MS`**/chartran/mult850.rsc** Code Page 850 (Multilingual)
`$MS`**/chartran/port860.rsc** Code Page 860 (Portuguese)
`$MS`**/chartran/cnfr863.rsc** Code Page 863 (Canadian-French)
`$MS`**/chartran/nord865.rsc** Code Page 865 (Nordic)

Example

```
#pragma translate "eng1437.rsc"
```

pragma: Version

Syntax

```
#pragma Version 5:5:0
```

Summary

Used to control the version number that displays in the Version field of the MDL application.

Relevant Compilers

mcomp

Detailed Description

The `Version` pragma specifies a version number to be associated with the MDL program. The version number must be specified as a series of constant integer expressions separated by colons. Up to three levels can be specified. For each unspecified level, 0 is assumed. If the third number is a printable ASCII character, it is

displayed in the Version field of the MDL application Detail dialog box as that ASCII character. For example, specify:

```
#pragma Version 5:0:'e'
```

in the pragma to get `5.0.e` in the Version field. If more than one object file in a program contains a version number specification, all version numbers must be the same.

Typically, programs do not use the `Version` pragma and instead use the version number generated by the MDL linker.

Structure Layout

Structure handling depends on the implementation for C compilers. On any platform, the MDL compiler mimics the C compiler used for compiling MicroStation. The MDL compiler and the host machine's compiler use the same rules for alignment and padding within a structure if all bit field groups start on a word boundary and are padded to a word boundary. For example, if a few contiguous bit fields are defined and the combined length of the bit fields is 22 bits, another 10-bit field should be added to supply the padding needed to pad the group of fields to the size of a `long`.

Because the MDL compiler mimics the host machine's compiler, programs compiled with the MDL compiler and programs compiled with the host machine's compiler can access data with the same structure definitions.

Bitfield Handling

The MDL compiler should allocate bit fields in structures exactly the same as the native C compiler used to compile MicroStation.

A zero-length bitfield tells the compiler that you want to advance to the next alignment boundary.

Special steps are necessary if an MDL application will be using a structure definition to dereference data that is to be platform independent, such as is the case when data is loaded from a resource file and dereferenced with a pointer. In this case, bitfields MUST conform to the following rules:

1. group bitfields together
2. start the groups on word boundaries
3. define all 32 bits of the word, even if that requires defining filler fields

4. define each 32 bit grouping twice, once in reverse order, depending on the existence of the macro `BITFIELDS_REVERSED`. For example,

```
#include <basedefs.h>
struct teststruct
{
     double dblMember;

#if !defined (BITFIELDS_REVERSED)
     unsigned int bf1:1;
     unsigned int bf2:2;
     unsigned int reserved:30;
#else
     unsigned int reserved:30;
     unsigned int bf2:1;
     unsigned int bf1:1;
#endif

     long lngMember;
};
```

If the platform independence is not a concern, but data is to be shared between an MDL application and an external program, the first three steps are still recommended.

6 MicroStation Resources

This chapter deals with how to manage, use, create and modify resources with the MDL application environment. It also introduces the Binary Portability Library functionality to allow for the management of portable information in resources and other structures.

MicroStation Resources

The following sections comprise this chapter:

- An Overview of Resources
- Creating Resources
- Modifying Resources
- Managing Resources
- The MicroStation Resource Manager
- A Resource Programming Example
- Binary Portability
- Resource Source Generator
- Resource Utilities

An Overview of Resources

In its simplest sense, a resource is a variable stored in a file for later use. A resource has three levels of identification:

- File
- `resourceclass` (or type)
- ID. The ID makes a resource unique within a given `resourceclass` and file.

Resources are the entities that MicroStation uses to define the way things are physically presented on the screen, represent static information and provide functional

extensions. MicroStation itself, the core product, is a large, natively compiled and linked program. It contains virtually no static text definitions, no dialog box or item definitions, no information on how the interface is to look visually, and only some additional code beyond the basic CAD functionality provided in versions of MicroStation prior to release 4.0.

All MicroStation versions since 4.0 (3.5 on the Macintosh) store GUI related information in resource files. During execution, MicroStation refers to the resource files for information on user preferences, how and where to display dialog items, and many other aspects of its behavior. Resource files provide an efficient way to extend MicroStation's functionality. The following files are compiled into binary resources and referenced by MicroStation via the MicroStation resource manager:

- Text strings for all error, status, prompt, dialog and command text.
- Command tables.
- User preferences and other static GUI-based control structures.
- The complete GUI visual display base (dialog boxes, settings boxes).
- All of the extensions to MicroStation in the form of MDL applications.

Resource-based subsystems in MicroStation include:

- Command parsing.
- Message lists.
- Dialog box manager.

Benefits from using resources

Several benefits can be gained from using resources:

- Convenience
  Resource manager routines allow a programmer to quickly save or restore data to or from a resource file without having to worry about the details of file formats or file access methods.
- Isolation of data
  By separating data from program code, items like error messages and prompts can be changed without having to recompile program source code.
- Minimize memory requirements
  Using resources means data is loaded into memory only when needed. Also, two applications can load the same resource from the same file, and when this occurs, only one copy of the resource is maintained in memory.

- MicroStation has resource-based subsystems
  MicroStation has several subsystems that use pre-defined resource types. Defining items such as command syntax and messages is more standard and less time consuming.

Application Resources

Like MicroStation, MDL applications use resources to define text strings command tables.-- In addition to these standard resources, applications can define their own application-specific resources for whatever purposes the application needs.

General Resource Definition Format

The syntax for resource source **.r** files is modeled after the syntax for data initialization in C. Specifying resource types and initializing them with data is the same as specifying `typedef` and declaring and initializing variables in C. The differences from standard C are listed below, and examples of some concepts are in the following "Examples" section.

- The resource language is used only for declaring and initializing data. It does not contain any executable statements.
- An MDL-specific statement, `resourceclass`, is used to assign a unique identifier to each `typedef` statement. This `unsigned long` identifier is referred to as a **type ID**, and is associated with each resource declared of that type. Conversely, each resource is considered to be a member of a resource class.

This information is stored in the resource file, and MicroStation's resource manager uses it when locating resources. The syntax for a `resourceclass` statement is as follows:

```
resourceclass <type-name> <resource class>
```

`<type-name>` must have been previously declared in a `typedef` statement.

`<resource class>` is a constant integer expression. The expression is evaluated at compile time and treated as an `unsigned long`.

- A resource declaration is similar to an initialized variable declaration in standard C. In C, a variable name is used to identify a memory address at execution time. A resource is identified with a 32-bit unsigned integer value and the file in which it is stored.
- The resource compiler lets an application define structures with variable-sized members. These structures are called "structures of undefined length," or **variable sized arrays**; arrays without bounds.

When a variable sized array is initialized, **rcomp** generates a hidden `long` value in front of the array data that defines the number of elements in the array. This is why resource structure definitions in header files contain the `#if defined (resource)` preprocessor directives to allow the proper structure definition to be included in the resource or MDL source file.

- Structures can contain **immediate members** - members whose numeric value is hard-coded in the structure and does not need to be specified in resource declarations. These constants are sometimes referred to as "fillers."
- Pointers are not supported.
- String constants in declaration lists are placed directly in the structure as an array of characters. The declared string is truncated or zero-filled to the size of the field as defined in the corresponding `typedef`. The number of characters placed in the array is controlled by the typedef corresponding to that variable. For example, suppose the type is defined as follows:

```
typedef struct
{
     char str[6];
} SimpleString;
resourceclass SimpleString SMPL_STR_TYPE_ID;
```

Also, suppose the declaration of the resource was as follows:

```
SimpleString MY_RES_ID = {"abcdefg"};
```

Thus, the string would get truncated to fit in six characters (including the trailing `NULL` byte) as `abcde\0`.

Resource Examples

The following sub-sections demonstrate the aforementioned differences between resource and C syntax.

Resource classes and resource IDs

Here is an example of how resource classes and resource IDs are used:

```
#define TWO_INT_TYPE     0x5
#define MY_RESOURCE      0x200
#define YOUR_RESOURCE    0x300

typedef struct
{
     int a;
     int b;
} TwoIntStruct;

resourceclass TwoIntStruct TWO_INT_TYPE;
TwoIntStruct MY_RESOURCE = {3, 4},
YOUR_RESOURCE = {6, 5};
```

An MDL application can access the structure at runtime by calling one of the resource management built-in functions specifying `resourceclass TWO_INT_TYPE` and `resourceID MY_RESOURCE` or `YOUR_RESOURCE`.

Variable sized arrays

The `testStruct` structure, shown below, has an undefined length because of its c structure member. This example shows how each instance of `testStruct` will look, but the number of members in the c array will vary with each resource of type `testStruct` declared in the file.

Source file

```
typedef struct testStruct   /* typedef structure containing */
{                           /* an array of undefined size */
   char a[3];
   char b[3];
   char c[];
} testStruct;
resourceclass testStruct 'abcs'; /* Assign 0x61626373 ID to */
                                 /* the "testStruct" structure */
testStruct 1 = {"aaa", "bbb", "ccc"}; /* Declare an instance of a */
                                      /* resource with RSCID of
1 */
```

Dump of compiled resource

```
resourceclass = 0x61626373 ('abcs') -  1 resources
resourceId=(1, 0x00000001) Size/Pos=(14/340)
000000 61 61 61 62 62 62 04 00 00 00 63 63 63 00 | aaabbb.ccc.
                          \         /           |
                         length           NULL terminator
```

The `[ ]` signifies a variable-length array. When a resource of type `TestStruct` is declared later in the file, a hidden long value is inserted after member `b` and before member `c` indicating the number of iterations of member `c` in `TestStruct`.

Immediate members / Fillers

The following example illustrates the use of fillers:

```
typedef struct fourIntStruct
{
     int 0[3];      /* 3 zero-filled ints */
     int b;
} FourInts;
resourceclass FourInts FOUR_INTS_TYPE_ID;
```

A subsequent resource declaration would need only to provide an initializer for the `b` member since the first structure member (array of three `int`s) is already initialized:

```
FourInts MY_RESOURCE_ID = {1};/* 0,0,0,1 */
```

Creating Resources

There are two ways that resources can be created.

➤ **To create resources**

1. The MDL resource compiler **rcomp** generates resource **.rsc** files from resource source **.r** files. Then the MDL linker **mlink** links one or more object **.mo** files into a program **.mp** file. Finally, **rlib** merges the **.mp** file with **.rsc** files to create an executable **.ma** file.
2. An MDL application can use resource manager functions to create a resource.

Resources generated with the resource compiler are generally `static`. Thus, they generally do not change and are just read by MDL applications. The source language for resource source files is an extension of C. It allows the following to be described:

- Variable-length arrays and structures
- Structures with constant fields

Resources generated during MDL application execution are usually more dynamic. They are created, read, written and sometimes deleted. Examples of static and dynamic resource descriptions follow.

Error message translation: an example of static resource use

An example of resource use is an application with error messages that must be translated into a foreign language. If resources are not used, when error messages are hard-coded into the application source code, they are difficult to locate because they can be scattered over many source code modules. The task becomes particularly difficult if the human translator is not the application programmer (usually the case). But that would only be part of the problem. Once the error messages are located and changed, all affected source modules need to be recompiled, a time-consuming process for large applications. If two or more application programs are involved in the system, some error messages can be duplicated, thus increasing the amount of work.

When error messages are extracted from the application source code and placed into a separate source file, they are centralized and therefore easily located. Once translated, they can be compiled into resource file format with the MicroStation Resource Compiler. The compilation process will be much faster than it will when resources are not used, since fewer files and no C instruction code compilation is involved. Error messages such as "`Sorry, that feature is not yet available`" can be common to and shared by several applications.

This last example shows how **static resources** can be used. (A static resource is defined as an ASCII source file that is compiled into resource file format). However, some applications need to collect information and store it in a file during program execution. These resources are more **dynamic**.

User preferences: An example of dynamic resource use

An example of collecting and saving resource information dynamically (during program execution) is found in MicroStation's Preferences dialog box. In this dialog box, you can alter options such as the Undo Buffer size and the Drawing Compatibility mode. This information is then saved in a resource file to be reused during future MicroStation sessions. In this case, MicroStation uses the resource manager functions to create the file where the resource will be stored, to save the resources in the file, and to reload the resources during subsequent sessions.

In general, the resource manager acts as a database interface for storing and retrieving internal program variables (resources) that must be reused across multiple executions of an application, shielding the application from the actual file storage methods used. An application needs only to set up identifiers for its resources.

Predefined resources in MicroStation: Message list and table

The Message List and Table resource classes were predefined for the convenience of applications. These resource classes are defined in the system header file **rscdefs.h**. `MessageLists` are described below. For a description of command tables and the predefined resource class Table, see "Compiling an Application Command Table" on page 7-11.

A `MessageList` contains messages used by the application. Each message in the list has a corresponding message number. These message numbers must be unique and appear in ascending order in the list. The `MessageList`'s resource identifier (referred to as `resourceID`) combined with a message number uniquely identifies a given message in the list. A sample `MessageList` follows:

```
MessageList TEST_LIST_ID=
{
   {
        {4, "Please enter your password."},
        {21, "That function is not available."},
        {9999, "Press any key to continue."};
   }
}
```

An application can obtain a message in a `MessageList` directly using the `mdlResource_loadFromStringList` function or can instruct one of several MDL built-in functions to use its `MessageLists`. Specifying `MessageLists` to MDL built-in functions enables MicroStation to display an appropriate command name and command prompt when application commands are initiated, undone and redone. These built-in functions and their relationships to `MessageLists` are described in the following table:

Function	Used to
`mdlState_registerStringIds`	register an application's Command Prompt MessageList and Command Name MessageList for other built-in functions to use.

`mdlState_startPrimitive`
`mdlState_startFenceCommand`
`mdlState_startModifyCommand`
`mdlState_startViewCommand`
	accept prompt and command name message numbers as parameters. The message numbers are contained in the application's registered Prompt and Command Name MessageLists, respectively.

`mdlOutput_rscPrintf` `mdlOutput_rscvPrintf`	accept a `MessageList ID` and a message number to display an application message using `printf` and `vprintf`.

Modifying Resources

Although in many cases updating resources can be very simple and straightforward, it can also be fraught with many pitfalls for the unwary programmer. Aside from simply creating or modifying a resource and writing that information to a resource file, an application has to be able to reliably traverse any resource in order to access and modify the information in the resource. This section talks about traversal issues as well as creating and modifying resources and resource files.

The following functions are used to access resources. See "Resource Management Functions" in the MicroStation MDL Function Reference Manual for more information.

Function	Used to
`mdlResource_openFile`	open a resource file.
`mdlResource_add`	add a new resource to an open file.
`mdlResource_load`	load a resource.
`mdlResource_write`	update a resource.
`mdlResource_free`	free a resource.
`mdlResource_closeFile`	close a resource file.

These functions will also be discussed later in this section.

Finding your way around a resource

Although pointer variables can't be used in resources, variable array structures can provide the same functionality in a different form. To make this clearer, consider what a C programmer does when a structure is used to keep track of data needed by a program. If the program requires a buffer of unknown size, a pointer variable is stored in the structure and when the memory is allocated for this variable, the address is stored in the pointer field of the structure. But since resources are statically sized storage areas, it must have a way of providing for this circumstance in structure definitions.

A variable sized array is assigned a length at compile time by **rcomp**. As **rcomp** compiles the resource definition, it counts the number of elements in the array. When the resource is fully compiled and all the elements have been counted, **rcomp** writes out the element count followed by the elements to the resource on disk. Later, at execution time, the application will load the resource using the `mdlResource_load` function call. Since the application knows that it is dealing with a variable sized array, it will refer to the `unsigned long` element count in front of the array to determine the array size.

This means that the actual structure of resources in memory containing variable sized arrays differs from what is defined in a header file describing that resource structure.

This is why the structure definitions for the application and the resource manager are different - so that each knows the structure up to the point at which it is statically defined.

A simple example of this is shown below. The structure definition of the resource shows a number of fields followed by an array of additional structures.

```
typedef struct mysubstruct
{
   long subLong1;
   char subChar1;
} MySubStruct;

typedef struct myresource
{
   int int1;
   int int2;
   char char1[4];
   long long1;

#if defined (resource)
   MySubStruct subStructs[];
#else
   long subStructLen;
   MySubStruct subStructs[1];
#endif

} MyResource;
```

Up to the `subStructs` field of the resource, the handling of this resource in memory is straightforward; the resource and MDL structures are identical. The problems occur when trying to reference the information in the `subStructs` field. Here is where most programmers make mistakes - different platforms and compilers have differing alignment restrictions for structures and for basic types.

The first attempt at traversing this structure will typically look something like the following code:

```
int i;
MyResource *rscP;
...
for (i=0; i < rscP->subStructLen; i++)
   myFunc(rscP->subStructs[i].subChar1, rscP->subStructs[i].subLong1);
...
```

The problem with the above code sample is that it is not portable. There are no guarantees that the compiler will generate resource structures that meet requirements

for long and general structure alignment restrictions. The portable way to traverse this array is to use the `mdlDialog_fixResourceAddress` function:

```
void *mdlDialog_fixResourceAddress/* <= pointer to loc. of member */
(
byte *resP,       /* => base of the structure next pointer is in */
byte *memberP,    /* => addr of the member needing alignment fixing */
int sizeMember    /* => alignment (usually sizeof(long)) */
);
```

The function is called with the address of the known point in the structure (typically the base or current traverse point) as the first parameter, the best guess at the address the application is trying to get to as the second parameter, and `sizeof (long)` as the third parameter. Using this function, we can rewrite the above code sample.

```
int i;
MyResource *rscP;
MySubStruct *subP;
...
/* Address first array entry */
subP = &rscP->subStructs[0];
for (i = 0; i < rscP->subStructLen; i++)
{
  myFunc(subP->subChar1, subP->subLong1);
  /* Get address of next array entry */
  subP = mdlDialog_fixResourceAddress((byte *)subP,
                (byte *)subP + sizeof(MySubStruct), sizeof(long));
}
...
```

The above example was very simple compared to the way applications must traverse complicated resources such as Pulldown Menu Items or a Dialog Box Resource. Knowing what to look for simplifies the processing of resources immensely.

Once you know your way around a resource, it is straightforward creating and modifying resources.

The Resource Manager Functions

The Resource Manager provides many functions for updating resource files. These functions can be divided into three main areas:

- Deleting existing resources.
- Creating new resources.
- Updating existing resources.

To delete an existing resource from a resource file, an application simply calls the `mdlResource_delete` function, and indicates the resource file and resource to delete. If no applications have the resource loaded, the resource is deleted from the file, otherwise the request is rejected and the application must try again later.

There are two ways to add new resources to a resource file. One is to allow the Resource Manager to add the resource using the `mdlResource_add` function. The other is for the application to add the resource to the file itself using the `mdlResource_directAdd` function. Both of these functions will reject any attempt to add a resource that matches an existing resource in the file (i.e., the resource ID and class match that of a resource already in the resource file). Also, these are the only Resource Manager functions that deal explicitly with resources that do not require the resource pointer (memory area) to be allocated by the Resource Manager.

The `mdlResource_add` function accepts a pointer to the resource to be added, the resource class, resource ID, resource size and resource alias as input. The `mdlResource_directAdd` function does not require the resource and resource size as input, but does require a pointer to a `RscDirectAccess` structure for returning information about where to add the resource to the file. In addition, after an application has finished writing a resource to a file in conjunction with the `mdlResource_directAdd` function, it must call `mdlResource_directAddComplete` to notify the Resource Manager that the resource has been added to the file. Examples of the use of these functions are shown below:

```
RscDirectAccess direct;
...

/* Add Text resource to the resource file */
mdlResource_add(rscHandle, RTYPE_Text, rscId, textRscP, rscSize,
NULL);

/* Now use regular stream file I/O to write our own user pref rsc */
mdlResource_directAdd(rscHandle, RTYPE_MyPrefs, rscId1, NULL, &direct);
fseek(direct.fileP, direct.filePos, SEEK_SET);
fwrite(prefsRscP, prefsSize, 1, direct.fileP);
fflush(direct.fileP);
mdlResource_directAddComplete();
```

It is important to note that the information in the `RscDirectAccess` structure remains accurate only up to the point that control is given back to MicroStation. This is especially true for the file pointer field. Any time the application needs to perform direct file I/O on a resource file, it must use the proper `mdlResource_direct…` function to load the structure with the most up-to-date and correct information.

Updating resources is done using two functions. `mdlResource_write` performs the actual writing of an existing resource back to the resource file it came from. The resource pointer passed as input to this function must be the same as one originally received from the Resource Manager via `mdlResource_load`. This means that the size of the resource can not have changed without notifying the Resource Manager and an

application cannot copy resources internally and pass the copies to the Resource Manager for processing. To resize a resource, the application needs to use the `mdlResource_resize` function. This function causes the Resource Manager to either allocate a new memory area to contain the resource if it is being enlarged or truncate the original buffer to reflect a smaller resource. It is the responsibility of the application to ensure that the data is properly entered into the new resource area returned before writing the resource and in the case of down-sizing a resource, saving any resource information which could be truncated.

Neither of these functions will succeed if another application has loaded the resource being modified.

```
int localPrefsSize;
MyUserPrefs localPrefs;
MyUserPrefs *currentPrefsP; /* loaded from resource file */

...

/* Truncate preferences by removing the array at end */
localPrefsSize=(byte *)&currentPrefsP->endArrayLen -(byte
*)currentPrefsP;
memcpy(&localPrefs, currentPrefsP, localPrefsSize);
localPrefs.endArrayLen = 0;
localPrefsSize += sizeof(long);

/* Resize preferences rsrc and if successful, write rsrc back out */
currentPrefsP = mdlResource_resize(currentPrefsP, localPrefsSize);
if (currentPrefsP)
{
   memcpy(currentPrefsP, &localPrefs, localPrefsSize);
   mdlResource_write(currentPrefsP);
}
```

In older versions of MicroStation, there is a bug which causes a resized resource not to be updated in memory after being written back to the resource file until the resource file has been closed and reopened. This bug is exhibited if the resized resource was the only one of that type (class) in the resource file.

Managing Resources

MicroStation provides a complete set of functions which can be used to manage and operate on resources. Resources can be directly accessed using the Resource Manager or accessed indirectly using the various interface functions provided by the Dialog Manager, Command Parser and Message Subsystem. These interfaces are discussed below.

Message Subsystem

The message subsystem provides mechanisms for retrieving text strings from your resources and displaying that text in the MicroStation Command Window or dumping the text to either a file or a buffer. These are the `mdlOutput_rsc…` functions which accept a resource file handle, resource ID, and string (message) list index value of the string to be displayed as input. These functions then retrieve the string from the appropriate resource and resource file and display it in the specified field of the Command Window. No other functions are provided for retrieving and displaying strings from a message list. Only by using the Resource and Dialog Managers directly can you perform any other string/display processing to MicroStation windows.

Command Parser

In addition to performing the actual translation of key-ins into commands, the Command Parser provides various functions for loading and handling the command table resources which provide the foundation of MicroStation.

Many people still have a hard time thinking of the command tables of MicroStation as being simple tables of a parse tree which are statically loaded into MicroStation. But, that is exactly what they are. If the command tables were actually dynamic or compiled directly into the application (including MicroStation), then the Command Parser would be unable to offer the flexibility and extensibility it currently provides.

Command tables are defined in resource files as a series of parse trees which are compiled by the resource compiler. A sample command table is provided below:

```
#include <rscdefs.h>
#include <cmdclass.h>
/*----------------------------------------------------------------------+
|                                                                       |
|  Local Defines                                                        |
|                                                                       |
+----------------------------------------------------------------------*/
#define CT_NONE 0
#define CT_MAIN 1
#define CT_LOAD 2
#define CT_OPEN 3

/* Application Command Tables */
Table CT_MAIN=
{
  {1, CT_LOAD, INPUT, HID, "LOAD"},
};

Table CT_PALLOAD=
{
  {1, CT_OPEN, INPUT, NONE, "OPEN"},
```

```
    {2, CT_NONE, INPUT, NONE, "ADDMENU"},
    {3, CT_NONE, INPUT, CMDSTR(1), "KEYIN"},
};

Table CT_OPEN=
{
    {1, CT_NONE, INPUT, NONE, "PALETTE"},
    {2, CT_NONE, INPUT, NONE, "AUTO"},
};
```

When the command table resource is compiled and a header file generated which contains the command numbers for each command, the application can hook functions to handle each command and the command table resource can be loaded into MicroStation so that the user can access those commands when the application is loaded.

The following sections provide an overview of the general command table resource management functions available.

mdlParse_loadCommandTable

Descr. This function will load the application's command table into memory and "merges" it into MicroStation's parse tree for processing commands keyed-in from the Command Window. This function starts loading command table resources from the specified resource file starting at resource ID 1 and continues loading all tables which are part of the parse tree. This function should be called once from the main entry point of an application. This function returns a pointer to the parse table descriptor created after loading the command table successfully. The pointer can be used to later unload the parse table if needed.

mdlParse_unloadTable

Descr. This function unloads the specified command table from MicroStation and frees the command table resource from memory. Most applications do not need to worry about this call since this operation is performed by MicroStation when an application is unloaded from memory.

mdlParse_loadKeywordTable

Descr. This function is useful for loading an alternate command table resource into memory for use only by the application through using the `mdlParse_keyWord` function. This

function does not provide a table descriptor upon return, and therefore does not allow the unloading of the table from memory until the application exits (i.e., is unloaded).

Dialog Box Manager

The Dialog Box Manager is by far the most common resource interface for the applications programmer. This interface provides the only way for providing a MicroStation compatible user interface to an application or to display dialog boxes on the screen. This interface performs all resource file management for the user as far as dialog boxes and items are concerned. When a single dialog box is opened using `mdlDialog_open` (or one of the other `mdlDialog_open_...` functions), the Dialog Manager goes out to the specified resource file, loads the dialog box resource indicated, and then loads each associated dialog item resource into memory as well. It also manages the display of the resources on the screen and the dynamic modification of the resources in memory via the `mdlDialog_...` functions provided for this purpose.

Functions are provided to allow the label, location, size and attributes of an in-memory resource to be changed dynamically for an occurrence of the dialog box or item. In some cases, this modification is not to the actual copy of the resource itself, but instead to a decoded internal representation of that resource.

The Dialog Manager even controls the freeing of all loaded resources intrinsically as part of the management of the GUI display environment.

The MicroStation Resource Manager

The MicroStation Resource Manager provides the applications programmer with a number of functions that allow the user to read (load), modify, write, create and delete resources to and from resource files. These are the low level functions used by the other subsystems mentioned above and are the only safe direct interface to resources. The following sections discuss various aspects of resource file and resource item management.

Resource File Management

In order to reference any resources, an application must have access to the resource file containing those resources. An application gains access to resources through three methods:

1. Opening resource files using the `mdlResource_openFile` function.
2. By “importing” the resource handle from another task.
3. Inheriting them from parent tasks.

An application can open as many resource files as it needs for its processing. It can also open resource files currently opened by another task within MicroStation. Although an application does not necessarily need to keep track of the resource file handles after opening the files, the handle is required for applications to load a resource from a specific file and to close or add resources to a file. In most cases, an application will simply call `mdlResource_openFile` upon entry to `main` function and `mdlResource_closeFile` prior to exiting the application as shown below:

```
#include <mdl.h>
#include <rscdefs.h>
main(int argc, char *argv[])
{
   RscFileHandle rscHandle;

...
   /* Open the application resource file for READ access. If error
      issue an error message and exit */
   if (mdlResource_openFile(&rscHandle, NULL, FALSE))
   {
        mdlOutput_errorU("Unable to open resource file");
        mdlSystem_exit(-1, 1);
   }

...
   /* Close the resource file before we exit */
   mdlResource_closeFile(rscHandle);
   mdlSystem_exit(0, 1);
}
```

For most applications, no additional `mdlResource_…` function calls are needed to perform any resource references, as this is handled by the various subsystem managers. In cases where the application needs to update resource information in a file, it must open the file for Read/Write access by passing `RSC_READWRITE` as the last parameter to `mdlResource_openFile`.

The `mdlResource_createFile` function allows the user to create new resource files. This function is useful to applications which need to maintain application specific user preferences or other information in a separate file. Remember that this function does not open the file; `mdlResource_openFile` must be called after this function to place resources into the file.

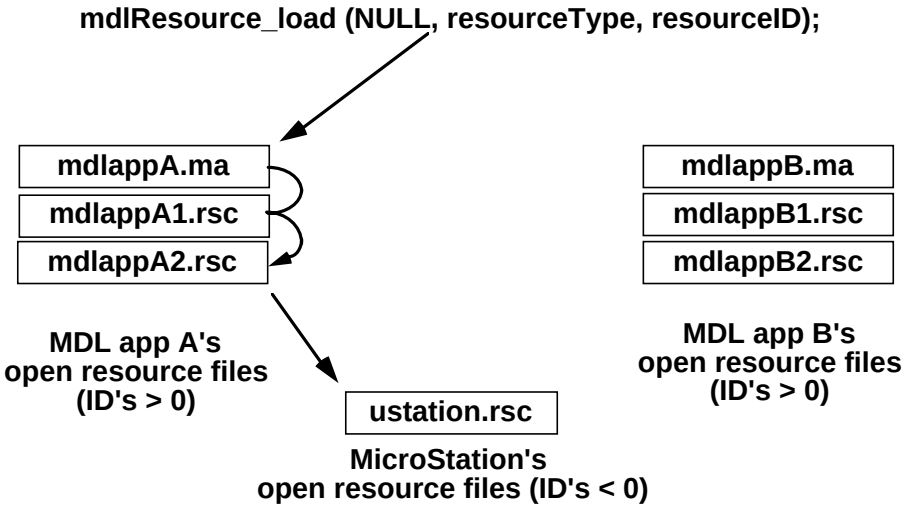
Loading and querying resources

When an application wants to load a resource from a resource file into memory for use, it uses one of the Resource Manager load functions. These functions provide differing levels of access to the resource information in a resource file. On occasion, it may be necessary to retrieve information about resources and resource files. The

mdlResource_query… functions provide this capability. An additional query function is also provided to return information about resources already loaded by the application program.

Loading Resources

Typically, an application doesn't care where a resource is found (i.e., which file it came from) since all of the resources specific to the application are typically placed in their own resource files and any other resources it uses it expects to inherit from its parents. This being the case, an application will typically use the mdlResource_load and mdlResource_loadFromStringList functions specifying NULL as the resource file handle. This has the effect of causing the MicroStation Resource Manager to start looking for the specified resource in the application's resource files and then resource files in the order of most recently opened first until it locates the resource requested. Once the resource is found, it is loaded into memory and the address of the resource is returned to the application. If the resource is not found, NULL is returned to the application. This is shown graphically in the diagram below.



It is important to remember that resources loaded into memory with the mdlResource_load function must be freed using mdlResource_free when the application is done with the resource. If the requested resource was not previously loaded by the Resource Manager, mdlResource_load will load the requested resource into memory allocated by the Resource Manager as it is read from the file and return the address of the resource to the application program. This action is performed only once. If the resource was already loaded, the Resource Manager will simply return the address of the loaded resource to the application.

The mdlResource_loadFromStringList function is a specialized version of the mdlResource_load function which allows the application to extract a single string out

of a message list resource without loading the message list resource and extracting the string itself. Since only the string from the message list is returned, the application does not need to free the resource from memory.

The other function which can be used to load resources into memory is the `mdlResource_directLoad` function. This function does not perform the resource load operation, but instead gives the application program information about the resource so that the application can perform its own resource file loading. The first three parameters of this function are the same as those for the `mdlResource_load` function. A fourth parameter is passed to this function which is a pointer to a structure into which the function returns the necessary information for the application to read the resource. This structure is defined in **rscdefs.h** and shown below.

```
typedef struct rscdirectaccess
{
   RscFileHandle rfHandle;  /* resource file handle */
   FILE *filerP;            /* resource file pointer from fopen */
   unsigned long filePos;   /* position of resource in the file */
   unsigned long rscSize;   /* size of the resource */
} RscDirectAccess;
```

Once the application has this information, it can read the resource file itself instead of relying on the Resource Manager to read the resource. This gives the application complete control of the resource and the memory it occupies. The application does not have to be concerned about more of the resource being read than it needs or keep the resource file open after the resource has been read. The application does need to allocate the memory into which the resource is read.

Querying Resource Information

Four functions are provided for retrieving information about resources and resource files. These functions provide a tiered query structure in which an application can query:

- Resource File Information - Using `mdlResource_queryFile` and `mdlResource_queryFileHandle`, an application can query the number of different resource classes and obtain a list of the resource classes in a file. Additionally, they can query for a version and identity of the resource file.
- Resource Class Information - Using `mdlResource_queryClass`, an application can query the number of resources in a specific class and the resource IDs for each resource instance in a specific class in the resource file. An application can also request the alias of a specific resource instance and the next available resource ID which can be used in the resource file for that class.

- Resource Information - Using `mdlResource_query`, an application can query the size, type (class), resource ID, source resource file handle, alias and source resource file name for the specified resource.

This information can be useful for applications which may need to perform lookups and or processing of multiple resources and files. A sample section of code below shows the use of these functions. This function locates all resources of a specified class in a specified resource file and returns information about both the resource file (the file handle) and the resource located. It returns information about one resource at a time.

```
int findResourceInfo
(
char *rscFile,              /* Name of file resource we are using */
ULong class,                /* Type of resource we are looking for */
RscFileHandle *rscHandle,   /* Resource file containing resources */
Long *rscId,                /* Resource ID of the one we found */
ULong *rscSize,             /* Size of the resource we found */
void *resource              /* The resource we found */
)
{
   static int rscIndex = -1;
   static int rscCount = 0;
   static ULong lastClass = 0L;
   ULong *classes;
   int i, numClasses;

   /* If we have a left over resource - free it now */
   if (resource)
        mdlResource_free(resource);

   /* If don't have any info yet then get initial info
      from the resource file */
   if ((rscId == 0L) || (class != lastClass))
   {
        lastClass = class;
        if (*rscHandle)     /* Close previous rsrc file if open */
             mdlResource_fileClose(*rscHandle);
        *rscHandle = 0;

        /* Get number of resource classes in file */
        if (mdlResource_queryFile(&numClasses, rscFile,
                                  RSC_QRY_NUMTYPES))
             return(ERROR);

        if (!(classes = malloc(sizeof (ULong) * numClasses)))
             return(ERROR);

        /* Now step through classes looking for the one we want */
        for (i = 0; i < numClasses; i++)
             if (classes[i] == class)
```

```
                    break;
          if (i >= numClasses)
               return(ERROR);

          /* Now open the rsrc file & initialize the processing info */
          if (mdlResource_openFile(rscHandle, rscFile, FALSE))
               return(ERROR);
          rscIndex = -1;
          if (mdlResource_queryClass(&rscCount, *rscHandle, class,
                                          RSC_QRY_COUNT, 0))
          {
               mdlResource_closeFile(*rscHandle);
               *rscHandle = 0;
               return(ERROR);
          }
     }

     /* If we are here, we have an open rsrc file and the number of
        resources in the rsrc file for the class we're looking for */
     *rscId = 0L;
     *rscSize = 0L;
     if (++rscIndex >= rscCount)
     {
          /* We are done with the current resource file - clean up */
          mdlResource_closeFile(*rscHandle);
          *rscHandle = 0;
          rscIndex = -1;
          rscCount = 0;
     }
     else
     {

          /* Not done with the current rsrc file - get next resource */
          if (mdlResource_queryClass(rscId, *rscHandle, class,
                                     RSC_QRY_ID, &rscIndex))
               return(ERROR);
          /* Load found resource into memory */
          if (!(resource = mdlResource_load(*rscHandle, class, *rscId)))
               return (ERROR);

          /* Get resource size information */
          if (mdlResource_query(rscSize, resource, RSC_QRY_SIZE))
               return (ERROR);
     }
     return (SUCCESS);
}
```

A Resource Programming Example

In the simplest case, an application can create a resource and add it to a resource file simply by defining the resource as a structure in the application's local memory, initializing it, and finally adding it using `mdlResource_add`. This method (demonstrated below) does not demonstrate how to handle variable sized arrays, which must be buffered before being added to a resource file.

The following function also shows the incoming resource being updated to reflect the date last accessed via the time function and then being rewritten to the resource file.

```
/*----------------------------------------------------------------------+
|  name      initApplicationDefaults                                    |
|                                                                       |
|       Initializes our application data.                               |
|                                                                       |
|  author    BSI       4/91                                             |
+----------------------------------------------------------------------*/
Private void initApplicationDefaults
(
void
)
{
   OurSettings *prefsDataP;
   int prefsRscSize;
   if ((prefsDataP=mdlResource_load(rscHandle, RTYPE_SprinklerDefaults,
                                RSCID_SprinklerDefaults)) == NULL)
   {
        /* resource does not exist, load defaults manually and then
           write the preferences to the file */

        prefsData.sprinklerSettings.shape = 0;
        prefsData.sprinklerSettings.sweep = 360;
        prefsData.sprinklerSettings.spray = 0;
        prefsData.activePipeType = PIPE_MAINLINE;
        prefsData.pipeSettingsP = &mainlineSettings;
        prefsData.mainlineSettings.diameter = 3;
        prefsData.mainlineSettings.color = 7;
        prefsData.mainlineSectionLength = 100.0 * tcb->uorpersub;
        prefsData.lateralSettings.diameter = 1;
        prefsData.lateralSettings.color = 10;
        prefsData.pumpSettings.maxFlowRate = 200;
        prefsData.pumpSettings.goldPlated = FALSE;
        prefsData.lastAccessed = prefsData.created = time(NULL);
        mdlResource_add(rscHandle, RTYPE_OurSettingsPrefs,
                              RSCID_SettingsPrefs, &prefsData,
                              sizeof(OurSettings), NULL);
   }
```

```
    else
    {
        /* load settings from resource */
        mdlResource_query(&prefsRscSize, prefsDataP, RSC_QRY_SIZE);
        mempcy(&prefsData, prefsDataP, prefsRscSize);

        /* Update access time and write back to prefs file */
        prefsDataP->lastAccessed = time(NULL);
        mdlResource_write(prefsDataP);

        /* ALWAYS remember to free the resource.... */
        mdlResource_free(prefsDataP);
    }
    mdlDialog_synonymsSynch(NULL, SYNONYMID_SprinklerSettings, NULL);
    prefsData.simulationSettings.currFlowRate = 50;
    prefsData.simulationSettings.wettedColor = 11;    /* YELLOW */
}
```

Working with resources containing variable sized arrays means that the application is required to know where in the resource structure it is at all times, or needs to calculate the proper position using `mdlDialog_fixResourceAddress` as described above. Once you know how to traverse a structure and how the structure is put together, modifying and creating the structure becomes relatively straightforward.

The key areas to be concerned about when doing this are:

1. Once `mdlDialog_fixResourceAddress` is required, it must continue to be used from that point forward.
2. Remember that variable sized arrays are stored as a long length `N` followed by `N` instances of the array entry. Note that variable sized character arrays are always `NULL` terminated. This means that every variable sized character array has a length of 1 plus the number of characters in the array - an empty array has a length of 1 and a single entry set to 0.
3. When creating new resources, allocate a memory area that will be large enough to contain the whole resource structure.
4. When resizing resources, do the same as above, copy the resource into the new area, resize the resource, and then copy the data back into the resized resource.

The following sample function performs both the modification and creation of a text item.

```
/*----------------------------------------------------------------------+
|                                                                       |
|  name updateAndCreateTextRsc                                          |
|                                                                       |
```

```
|  This function will update a text resource which was passed in    |
|  and also create a new text resource                              |
|                                                                   |
|  author BSI               10/92                                   |
|                                                                 |

|  typedef struct ditem_textrsc                                     |
|  {                                                                |
|       ULong commandNumber;                                        |
|       ULong commandSource;                                        |
|       long synonymsId;                                            |
|       ULong helpInfo;                                             |
|       ULong helpSource;                                           |
|       long itemHookId;                                            |
|       long itemHookArg;                                           |
|       byte maxSize;            // max # of chars in field //      |
|       char formatToDisplay[16];// format string to convert        |
|                                   from internal //                |
|       char formatToInternal[16];// convert to internal from       |
|                                    display str //                 |
|       char minimum[16];        // minimum value //               |
|       char maximum[16];        // maximum value //               |
|       ULong mask;              // only used with integers //     |
|       UShort attributes;       // other attributes //        |

|  #if defined (resource)                                           |
|       char label[];                                               |
|       char accessStr[];                                           |
|  #else                                                            |
|       long labelLength;                                           |
|       char label[1];                                              |
|  #endif                                                           |
|       } DItem_TextRsc;                                            |
|                                                                   |
+-------------------------------------------------------------------*/

Public void updateAndCreateTextRsc
(
long rscId,  /* => resource id */
void *rsrcP  /* => Resource loaded from tempFileH */
)
{
   long baseRscSize, itemRscSize;
   void *rscP;
   DItem_TextRsc *nrsrcP = rsrcP;
   DItem_TextRsc *newTextRscP;
   void *tmpRscP, *auxInfoP;
   long sAuxInfo;
```

```
if (rsrcP)
{
     /* Before doing anything else, modify the resource - since
        we may be changing the size of the resource as well as
        modifying information within it, we must locate all variable
        information, copy out what we need and also calculate the
        size of the new resource. */

     /* First lets get past the label to the access string */
     tmpRscP = mdlDialog_fixResourceAddress((char *)rsrcP,
                              (char *)nrsrcP->label +
                         nrsrcP->labelLength, sizeof (long));

     /* Get the length of the access string and save the string */
     sAuxInfo = *((long *)tmpRscP) + sizeof(long);
     auxInfoP = malloc((int)sAuxInfo);
     memcpy(auxInfoP, tmpRscP, (int)sAuxInfo);

     /* Now determine the new size of the resource. First get the
        size of everything up to the access string loc, then the
        then the whole resource. */
     baseRscSize = (long)(mdlDialog_fixResourceAddress((char *)rsrcP,
               (char *)nrsrcP->label + strlen (dialogP->text) + 1,
               sizeof (long)) - rsrcP);
     itemRscSize = (long)mdlDialog_fixResourceAddress (NULL,
          (char *)(baseRscSize + sAuxInfo), sizeof (long));

     /* Now resize the resource and modify the info. */
     nrsrcP = mdlResource_resize(rsrcP, itemRscSize);

     /* If couldn't resize the resource, error */
     if (!nrsrcP)
     {
          sprintf(dialogP->text, "Error on resize (temp): %d",
                    mdlErrno);
          mdlDialog_openAlert(dialogP->text);
          mdlSystem_exit(-1, TRUE);
     }

     /* Assign the new information to the resource */
     nrsrcP->attributes = dialogP->itemAttributes & 0x0ffff;
     nrsrcP->mask = dialogP->mask;
     nrsrcP->maxSize = dialogP->textAttrs.maxSize & 0x0ff;
     newdiRP->itemArg = dialogP->itemArg;
     strcpy(nrsrcP->formatToDisplay, dialogP->textAttrs.ftd);
     strcpy(nrsrcP->formatToInternal, dialogP->textAttrs.fti);
     if (!dialogP->textAttrs.max[0])
     {
          if (dialogP->textAttrs.min[0])
               strcpy(dialogP->textAttrs.max,
```

```
                        dialogP->textAttrs.min);
        }
        else
        {
             if (!dialogP->textAttrs.min[0])
                  strcpy(dialogP->textAttrs.min,
                        dialogP->textAttrs.max);
        }
        strcpy(nrsrcP->minimum, dialogP->textAttrs.min);
        strcpy(nrsrcP->maximum, dialogP->textAttrs.max);

        /* Copy new label into the resource */
        nrsrcP->labelLength = strlen(dialogP->text) + 1;
        if (strlen (dialogP->text))
             strcpy(nrsrcP->label, dialogP->text);
        else
             nrsrcP->label[0] = '\0';
        /* Address the access string and copy that information back
        /* into the resource */

        rscP = (void *)mdlDialog_fixResourceAddress((char *)nrsrcP,
                        (char *)nrsrcP->label + nrsrcP->labelLength,
                             sizeof(long));
        memcpy(rscP, auxInfoP, sAuxInfo);
        free(auxInfoP);

        /* Now write the modified resource into memory */
        mdlResource_write(nrsrcP);
        mdlResource_free(nrsrcP);

        /* To address Resource Manager bug, close & reopen rsc file */
        mdlResource_closeFile(dialogP->tempFileH);
        mdlResource_openFile(&dialogP->tempFileH,
                             tempRscFileName, TRUE);
        /* Reload the resource */
        nrsrcP = mdlResource_load(dialogP->tempFileH,
                                  RTYPE_Text, rscId);
}

/* Now create a new text resource with no access string and the same
   same label as the one we just updated append with "-New"
   We first need to determine the size of our new rsc by going to the
   label location, adding in the len of the existing label and the
   length of the extension. Then we need to add in the length of the
   NULL access string to get the full resource size */
baseRscSize =
     (long)(mdlDialog_fixResourceAddress((char *)nrsrcP,
        (char *)nrsrcP->label + nrsrcP->labelLen + strlen("-New"),
        sizeof(long)) - nrsrcP);
itemRscSize = (long)mdlDialog_fixResourceAddress (NULL,
```

```
                        (char *)(baseRscSize + sizeof(long) + 1,
                        sizeof(long));

    /* Now malloc the resource, initialize it, and add it to the
       the resource file */
    if (newTextRscP = malloc(itemRscSize))
    {
         memset(newTextRscP, 0, itemRscSize);
         newTextRscP->attributes = TEXTATTR_NOCONCAT;
         newTextRscP->mask = NOMASK;
         newTextRscP->maxSize = 45;
         newdiRP->itemArg = NOARG;
         strcpy(newTextRscP->formatToDisplay, "%s");
         strcpy(newTextRscP->formatToInternal, "%s");

         /* Copy new label into the resource */
         newTextRscP->labelLength = nrsrcP->labelLen + strlen("-New");
         strcpy(newTextRscP->label, nrsrcP->label);
         strcat(newTextRscP->label, "-New");

         /* Address the access string and indicate that a NULL
            string is enclosed - no access string */
         rscP = (void *)mdlDialog_fixResourceAddress
                        ((char *)newTextRscP,
                        (char *)newTextRscP->label +
                        newTextRscP->labelLength, sizeof(long));
         (long *)rscP = 1L;

         /* Now add the resource to the file */
         mdlResource_add(dialogP->tempFileH, RTYPE_Text, rscId+100,
                        newTextRscP, itemRscSize, NULL);
    }
}
```

Binary Portability

Binary portability means a binary file created on one computer can be copied to and read on other computers of the same or different architecture. When a file is binary portable, it does not need to be converted to the native computer's format for its data to be read properly. The main benefit of having binary portable files is that users who run MicroStation on a variety of hardware platforms can share data freely without caring about the differences of their hardware configurations.

Although binary portability may seem like an obvious and simple goal, differences in computer architecture and compiler technologies can make it very challenging to achieve. The goal of the Binary Portability functions is to simplify this task.

The way data is written to a file for a given computer is a reflection of how that data was represented in the computer's memory. Therefore, where two computers differ in their representation of data types in memory, they will also differ in their representations on disk. At the same time, dumping data to disk, verbatim, is expedient from a programming perspective and fast in terms of execution time.

Let's look at examples of how the same data written to files on different computers can appear drastically different on disk. Using the following structure:

```
struct orderStruct
{
   long orderNumber;
   char orderType;
   short orderQuantity;
};
```

If we wrote it to a binary file with the following values:

```
orderNumber = 1000
ordertType = 'a'
orderQuantity = 50
```

Hexadecimal dumps of the data files would appear like this on the following computers:

Computer	Hex dump
MS-DOS based PC	E8 03 00 00 61 32 00
Intergraph Clipper Workstation	E8 03 00 00 61 00 32 00
SUN SPARCstation	00 00 03 E8 61 00 00 32

Even though it is the same information, it is represented differently on the different computers. The reason stems mainly from the various machine architectures. (It should be noted, however, that different operating systems, and even compilers, can differ in the way they represent data on the same computer).

Differing data representations

Here are the main factors that will lead to different data representations in computer memory, and therefore in a disk file:

Alignment restrictions

Some computers impose restrictions on the location (in memory) of certain data types. They require that the address of the data be a multiple of the size of the data type. For example, a long integer (4 bytes in size) would have to begin in memory at an address that was a multiple of 4. A double precision floating point datum would have to start at

an address that was a multiple of 8 bytes. On the other hand, some computers don't care. The PC (running MS-DOS, MSC C compiler) does not have any memory alignment restrictions. These differences in alignment restrictions (or the lack thereof) can be seen in the `orderStruct` example above. Since `orderQuantity` is a short integer, the Clipper Workstation requires a pad byte after the `orderType` structure member to ensure the short alignment for `orderQuantity`. The PC, however, requires no such padding. As a result, the PC wrote the two members out to disk as `61 32 00` whereas the Clipper workstation wrote them out as `61 00 32 00`.

Byte ordering

Computers differ in the way they order the bytes that compose multi-byte data types. For example, on some computers, the low-order byte of a long integer will occupy the lowest memory address of the integer. On other computers, the high-order byte will be first (in the lowest memory position). Computers of the first type are known as **little endian** computers. Computers that place the high-order byte of a multi-byte data type first are known as **big endian** computers. In the `orderStruct` example, the Sun SPARCstation wrote the `orderNumber` value `0x000003E8` to disk as `00 00 03 E8`, whereas the PC and the Clipper wrote it to disk as `E8 03 00 00`. From this we can infer that the SPARCstation is a big endian machine (like the HP, Macintosh and IBM RS-6000), and the PC and Clipper workstation are little endian machines.

Floating point standards

Although most microcomputers adhere to the IEEE standards for floating point data representation, a few (DEC VAX computers for instance) allow for other representations, the difference being in the number of bits devoted to the mantissa and exponent of the number.

Bitfield representation

Generally, the more frequently bitfields are used, the more hardware-dependent (and sometimes compiler-dependent) a data file becomes. This is due to the very relaxed definition that has been given for their implementation in the C programming language. (Although the use of bitfields in structures written to binary portable files is not recommended, there is limited support for them when using the Binary Portability functions).

MicroStation file types and their portability

Design Files

Prior to version 4.0 of MicroStation, the only binary portable files were design files and cell libraries. These file formats include little endian byte ordering, IEEE and VAX double precision floating point data, no alignment restrictions, and miscellaneous data types taken from the VAX/PDP architecture. As new element types have been developed, the format has been more strictly little endian, no alignment, and IEEE

double oriented. From here on in this document, this format will be referred to as "packed little endian" format.

MicroStation takes care of converting elements read from or written to design files. The only exceptions are application defined elements (Type 66, Level 20) and user data linkages on all elements. Application developers who update design files directly via these two exceptions are encouraged to generate their data in packed little endian format.

4.x Resource Files

In version 4, MicroStation resource files were introduced. These files, by default, were not binary portable. Applications could make them binary portable, however, by generating them in packed little-endian format and using the binary portability routines to convert them after `mdlResource_load` calls ("on the way in") and before `mdlResource_write` or `mdlResource_add` calls ("on the way out"). The routines used by the application to convert the resources were `mdlCnv_bufferFromFileFormat` and `mdlCnv_bufferToFileFormat`, respectively.

5.0 Resource Files

In version 5, the resource file format has been enhanced to include information about the file's format. This allows 5.0 resource files to be converted automatically by the MicroStation Resource Manager. Hence 5.0 resource files are binary portable. Unlike design files and portable 4.x resource files, 5.0 resource files are not generated in packed little endian format. Instead, 5.0 resource files are formatted according to the platform on which they are generated. This makes resource file accesses fastest on the native platform of the file since no conversion is required there.

Binary portable files

Prior to version 4.0 of MicroStation, the only binary portable files were design files and cell libraries. These file formats were largely inherited from the IGDS application on which MicroStation was originally based. The original format includes little endian byte ordering, IEEE and VAX double precision floating point data, no alignment restrictions, and miscellaneous data representations taken from the VAX/PDP architecture. As new element types have been developed, the format has been more strictly little endian, no alignment, IEEE double oriented, and will continue to stay that way. From here on in this document, this format will be referred to as "packed little endian" format.

Application developers who update design files directly using User Attribute Data Linkages on elements and/or User-defined Elements (Type 66, Level 20) have also been encouraged to generate their data in packed little endian format to maintain the file's portability.

Along with MicroStation version 4.0 came a new file type: the resource file. Although the MicroStation resource compiler did not generate resource files in binary portable

format, it was still possible to create binary portable resource files using Resource Manager functions to write the data. Again, packed little endian format was used internally and was encouraged for use among application developers.

As the number and complexity of MicroStation applications increased, and as MicroStation became available on more platforms, it became apparent that both the developers of MicroStation and its third party applications would need a set of automated tools for ensuring that files remained binary portable. Hence, the Binary Portability 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.

Four Recipes for Binary portable data

This section outlines four different scenarios for how an application might use the binary portability routines.

A. Type 66 Level 20 Elements (Application Elements)

When building your application

1. Create a **.mt** file that contains createdatadef statements for every struct that will be used to define the data portion of an application element. The **.mt** file must include the header files where the structures are defined. It must also include pragmas for treating the structure as packed little endian data. For example:

```
#pragma packedLittleEndianData
#include <myfile.h>  /* This defines "mytype66datastruct" */
createDataDef (mytype66datastruct,
MY_TYPE66_DATADEF_ID);
```

2. Update your application's make file to compile the **.mt** file using the **rsctype** compiler. The resulting **.r** file must be compiled with **rcomp**,

and the subsequent **.rsc** file must be lib'ed into your application's **.ma** file using **rlib**.

At execution time, when creating application elements

1. Initialize an element buffer to zeroes.
2. Call `mdlCnv_bufferToFileFormat` to convert your application data into the data portion of the element buffer starting at the 20th word of the element. Save the output (file format) buffer size from `mdlCnv_bufferToFileFormat`. The data definition ID is the one you assigned in step 1 (i.e., `MY_TYPE66_DATADEF_ID`).

```
typedef struct applelem
{
     Elm_hdr ehdr;       /* 14 words */
     Disp_hdr dhdr;      /*  4 words */
     short signature;    /* 1 word */
     short dataPortion[768 - 19];
} ApplElement;
```

3. Fill in the element header:
   - `type` (`MICROSTATION_ELM`)
   - `level` (`APPLICATION_LEVEL`)
   - `wordsToFollow = 17 + (size of converted application data+1)/2`
   - `attributeIndex = wordsToFollow - 14`
   - `xlow, ylow, zlow` (`mdlCnv_toScanFormat(MINI4)`)
   - `xhigh, yhigh, zhigh` (`mdlCnv_toScanFormat(MAXI4)`)
   - `signature = (assigned to your application by BSI)`. You must byte-swap the signature on big endian machines.
4. Add the element to the design file.

At execution time, when reading application elements

Use `mdlCnv_bufferFromFileFormat` to convert the data portion of the element to the native machine's format.

B. User Attribute Linkages

When building your application

1. Create a **.mt** file that contains createdatadef statements for every struct that will be used to define the data portion of your user attribute linkages. The **.mt** file must include the header files where the structures are defined. It must also include pragmas for treating the structure as packed little endian data.

For example:

```
#pragma packedLittleEndianData
#include <myfile.h>  /* This defines "mylinkagestruct" */
createDataDef(mylinkagestruct, MYLINKAGE_DATADEF_ID);
```

2. Update your application's make file to compile the **.mt** file using the **rsctype** compiler. The resulting **.r** file must be compiled with rcomp, and the subsequent **.rsc** file must be lib'ed into your application's **.ma** file using **rlib**.

At execution time

Use the `mdlLinkage_...` routines for appending, extracting and deleting user data linkages to/from elements. The `mdlLinkage_...` routines take data definition IDs and call the `mdlCnv_...` functions internally.

C. 4.x Resource Files

When building your application

1. Create a **.mt** file that contains createdatadef statements for every struct that will be used to define a binary portable resource. The **.mt** file must include the header files where the structures are defined. It must also include pragmas for treating the structure as packed little endian data. For example:

```
#pragma packedLittleEndianData
#include <myfile.h>  /* This defines "mystruct", used below. */
createDataDef (mystruct, MYSTRUCT_DATADEF_ID);
```

2. Update your application's make file to compile the **.mt** file using the **rsctype** compiler. The resulting **.r** file must be compiled with **rcomp**, and the subsequent **.rsc** file must be lib'ed into your application's **.ma** file using **rlib**.

At execution time

1. Call `mdlResource_openFile` to open the target file.
2. Call `mdlResource_queryFileHandle` to ensure that the file is 4.x format. If it is 5.0 format, see the next scenario.
3. Call `mdlResource_load` to load the target resource.
4. Call `mdlCnv_bufferFromFileFormat` to convert the data to the current machine format. The data definition ID will be the same as the one you assigned in the `createDataDef` statement in step 1.

If a data type will be converted often, you can avoid having the data definition loaded and interpreted by `mdlCnv_bufferFromFileFormat` each

time by saving the conversion rules generated by `..._bufferFromFileFormat` the first time your data definition is used.

5. Use the resource.
6. `mdlCnv_bufferToFileFormat` to convert the data back to file format. Use the same data definition ID (or conversion rules if you have them) used in step 6.
7. Call `mdlResource_write` to write the updated resource back to the file or `mdlResource_add` to add a new resource to the file.

D. 5.0 Resource Files

When building your application

If all your resources are generated by rcomp, you do not need to do anything special when building your application. Version 5.0 of **rcomp** will automatically generate data definitions for the resources it compiles. Steps 1-4 are only necessary if you have resources that only get generated at execution time using Resource Manager functions.

1. Update your header files to have `resourceclass` statements for every `typedef` that will be used to define resources created solely by resource manager function calls.
2. Create a **.r** file that simply includes the header files updated in the previous step.
3. Update your make file to **rcomp** the **.r** file from step 2 with the **-r** option. This causes **rcomp** to generate a data definition for each of the resourceclass statements encountered in your header files. The data definition ID will be the same as the resourceclass ID. (The **-r** option can be turned on by setting the **bmake** macro `rcompOpts = -r`).
4. Update your make file to **rlib** the **.rsc** file generated from the previous step into your application's **.ma** file.

At execution time

When dealing with 5.0 resource files, the version 5.0 Resource Manager makes all calls to the binary portability routines for the application. The application will only need to make sure that the appropriate data definitions exist in a place where the resource manager may find them. When loading 5.0 resources, the resource manager will search for data definitions of the corresponding type in all the resource files opened by the calling application as well as those opened by MicroStation.

1. If you know you are dealing with 5.0 resource files or 4.x resource files with the same format as the current platform, there is no special action to take at execution time. Just open the file (`mdlResource_openFile`) and load the resources (`mdlResource_load`). If there is a chance that you might be loading resources from a "foreign" 4.x file that has packed little endian resources, proceed to step 4 in the previous scenario, "4.x Resource Files". Likewise, when adding resources, there is no special action to be taken unless you are dealing with a 4.x resource file.

When an application adds resources via `mdlResource_add`, the resource manager will automatically add the appropriate data definition in the destination resource file, too. If the resource manager cannot find the appropriate data definition, the add succeeds, but that particular resource may not be loadable on another platform.

Linkage Functions

Although this section has talked about portable resource information, the Binary Portability Library provides for portable data structures in design files as well via the `mdlLinkage_...` functions which indirectly call the `mdlCnv_buffer...` functions documented above. This provides for portable data in both user attribute data assigned to existing MicroStation element types as well as in Application elements (Type 66/ Level 20). See "Element Linkage" in the MDL Function Reference Manual for descriptions of the `mdlLinkage_...` functions.

Resource Source Generator

The Resource Source Generator is used to generate source resource (**.r**) and include (**.h**) files from compiled (binary) resource (**.rsc**) or MDL application (**.ma**) files. The source files are given the name of the compiled file with the appropriate **.r** and **.h** suffixes and are saved in the same directory as the compiled file.

The following source files are generated from the binary resource file (**xxxxxxxx** is the base name of the compiled resource file):

File	Description
xxxxxxxx.r	Base resource file containing all recognized resource data contained in the binary file.
xxxxxxxx.h	Base header file containing all necessary resource-based defines (for example resource IDs).
xxxxmsgs.r	Message and string list resources, if any, are defined in the binary resource file.

File	Description
xxxxxcmd.r	Command table resource.
xxxxxtxt.h	Static text defines referenced by the xxxxxxxx.r file. The defines generated in this file are generally of the format TEXT_rrrr#_<string> where rrrr is the resource class (for example, dBox for a dialog box resource), # is the resource ID, and string is the text with all punctuation and white space (tabs and spaces) removed.

Generate Resource Source dialog box

When the Resource Source Generator is loaded, the Generate Resource Source dialog box opens. Use this dialog box to select the binary resource files and MDL application files for processing. To add a file to list of files to process, select the file in the Files list box and click the Add button. To remove a file from the list of files to process, select the file in the Resource Files to Process list box and click the Remove button.

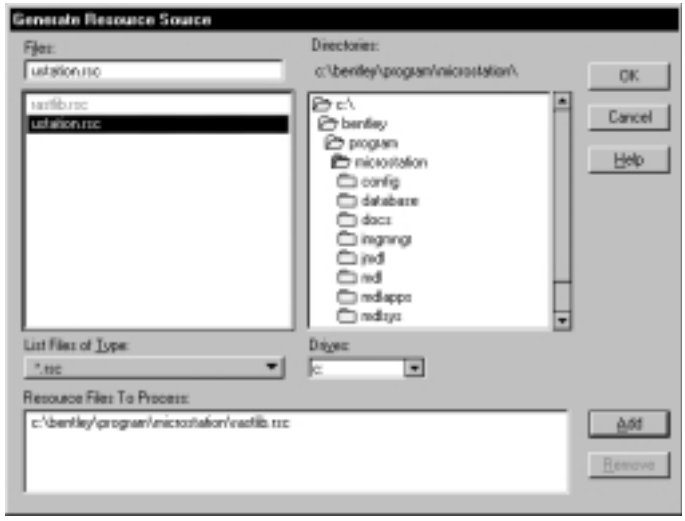


Figure 6.2 Generate Resource Source dialog box.

Click the OK button to generate the source files for the binary files listed in the Resource Files to Process list box.

The following resource classes are recognized by the Resource Source Generator:

Class	Resource Definition	Class	Resource Definition
dBox	Dialog Box	list	List (obsolete-use lBox items instead)
CmdT	Generic Command Table	mBar	Menu Bar
Cmds	MicroStation Command Table	pdMn	Text Pulldown Menu
str#	Message List	pdOM	Pulldown Option Menu
strL	String List	pdCM	Pulldown Color Picker Menu
cPck	Color Picker	oBtn	Option Button
gnrc	Generic	pBtn	Push Button
iFrm	Icon Command Frame	sBar	Scroll Bar
iPal	Icon Command Palette	dTxt	Text
iCmd	Icon Command	mTxt	Multiline Text
iCSm	Icon Command Small Icon	tBtn	Toggle Button
iCLg	Icon Command Large Icon	syns	Synonyms
Icon	Icon	prcB	Precision Button
levm	Level Map	cmnd	Command
lBox	List Box		

Resource Utility Programs

Resource utility programs include the conversion utility and file dump utility. Resource utility programs let you create and manipulate resource files.

cmdcnvert conversion utility for command table source files

The **cmdcnvrt** utility converts old-style (pre-MicroStation PC 4.0) **.txt** command table source files into the new-style **.r** source files read by the MicroStation Resource Compiler. After the conversion is complete, the output **.r** file must be edited to have a `#define` statement for each `CT_ commandtablename` referenced in the file. The syntax of the **cmdcnvrt** utility is as follows:

```
cmdcnvrt [-flags] file
```

`-flags` can be one of the following:

value	description
`-oFile`	Set name of output file.
`-v`	Verbose: show progress of conversion.

rdump resource librarian dump utility

The **rdump** utility dumps the contents of a resource file. It is useful if you are having trouble loading resources and want to examine the contents of a **.r** file. The syntax of the **rdump** utility is as follows:

```
rdump [-flags] file
```

`-flags` can be one of the following:

value	description
`-t<class>`	Restrict the dump to a particular type of resource. class is the resource class as specified in the resourceclass statement. It can be specified in hex, decimal or character constant form. For example, `-t'Msgs'` and `-t0x4d736773` are the same.
`-r<resourceID>`	Restrict the dump to a particular resource ID.
`-v`	Verbose: produce a detailed dump of each resource.

7 Building Applications

This chapter demonstrates the normal process required to build an MDL application.

Building Application Process

The process consists of the following steps:

- Creating a makefile and using the **bmake** utility.
- Creating and compiling an application command table.
- Generating resource files from C type definitions.
- Compiling resources.
- Compiling MDL source files.
- Using the MDL librarian (**mlib**) (optional).
- Linking the application source files.
- Using the resource librarian (**rlib**) to "put it all together."

Each step corresponds to a section of this chapter. "Creating a Makefile and Using the bmake Utility" on page 7-2 describes the developer's primary task when creating a new application; developing the makefile. All the other steps should be automated by placing their specific commands within the makefile.

This chapter will not tell you which MDL functions to use to accomplish specific tasks, or how to arrange your basic source code. It assumes that you have already created either an outline or flowchart to organize the operation of your application, and that you are able to research the required functions needed from within the MDL Function Reference Manual.

To demonstrate how the MDL tools are used, this chapter will follow the construction of the MDL sample application **basic.ma**. This is the same MDL application that is discussed in "A Complete Example" on page 18-1. Therefore, you may want to review these two sections of the manual in conjunction with one another.

Creating a Makefile and Using the bmake Utility

The MDL development environment includes a utility called bmake that produces executable images from **makefiles**. These makefiles contain the commands and directives needed to build your application. bmake was developed by and for Bentley Systems specifically for MicroStation development.

In programming environments such as MDL, changes to one file often causes other files to become obsolete. bmake recognizes when files have changed and detects and recreates any obsolete files based on the relationships between the file creation times.

For example, suppose you make a change to one of your MDL source files. For your change to be reflected in the application's executable, you must first run the MDL compiler on your source (.mc) file to generate or make its object (.mo) file. Next, you must run the MDL linker on your object file(s) to create the application (.ma) file.

In your makefile, you would first list your object file as a **target file** and tell bmake that it **depends** on the source file. When the target file is older than any of its dependents, the **build commands** that recreate the target (in this case, the MDL compiler) are executed. Next you list your application file as a target file that depends on the object file. bmake always processes makefiles linearly from beginning to end, so that both the object and the application files are rebuilt (assuming there are no errors).

Using file extensions to differentiate types of files is good programming practice. bmake exploits this by allowing you to define **rules** that describe how to create a file with one extension from a file of another extension. After you define these rules, you do not have to explicitly tell bmake how to recreate every target file. It infers the build commands from the file extension of the target and its dependencies and your rules.

In many instances, you will collect your rules and definitions into a single make include (.mki) file like the MicroStation mdl.mki file which contains the default macros and build rules for MicroStation MDL applications. This file is then included by your makefile to provide the needed macro and rule definitions.

You should use bmake instead of similar programs you may have used in other development environments for the following reasons:

- bmake allows include files for defining items common to many makefiles.
- bmake supports **conditional compilation** in makefiles (as in `ifdef`, `indef`, `elif`, `else`, `endif`).
- bmake allows the inclusion of other makefiles and header files containing default rules and macros.
- bmake allows different rules that apply to files in different directories.

- **bmake** allows multiple targets for the same dependency list.
- **bmake** does not have the typical memory limitations of other make programs.
- **bmake** can create link files from the dependency list in a makefile, making it easier to add new source files.
- **bmake** lets you define constants on the command line.
- **bmake** is available on every platform that MDL is available on.
- The same makefiles can be used on every platform on which MDL is available.

It would be possible to use bmake for development with other products if a new set of rules were created. Be aware, however, that one of **bmake**'s greatest strengths is its platform independence, so any advantage of using it would be lost if the rules were not designed with this in mind.

Makefile format

The makefile format is similar to the format for input files to other make programs. Each makefile must contain a series of records, each of which must be one of the following types:

- Macro
- Rule
- Dependency
- Conditional

Rules, conditionals and dependencies must be separated by blank lines.

A '\' (backslash) as the last character on a line signals that the line continues. (To end a line with the backslash character, use two consecutive backslashes).

The '#' character denotes a comment. **bmake** ignores the remainder of the line following this character. Comment lines cannot be continued over multiple lines.

Each makefile record type is described below.

Example

A listing of the makefile for **basic.ma** can be found in "A Complete Example" on page 18-1.

Macros

There are three locations where a bmake macro may be defined:

1. In the body of a makefile (or in `BMAKE_OPT`).
2. As an environment variable, outside of bmake.
3. On the command line using the `-d` command line option.

bmake looks for definitions in the above order, so macros defined in makefiles take precedence over those defined elsewhere.

Macros defined in a makefile are not case sensitive, but those defined otherwise will be if you are working under UNIX (MS-DOS and Macintosh are never case sensitive).

To define a macro in the body of a makefile, use one of the following formats:

format	description
`macro = definition`	Standard assignment. This is the most commonly used method.
`macro =% definition`	Standard but value being assigned is expanded before assignment.
`macro + definition`	Definition is appended to existing macro value.
`macro +% definition`	Definition is appended to existing macro value, and is expanded before being appended.

bmake performs macro substitution when it reads `$_macro_` in the input makefile, where the '_' characters signify the type of expansion:

macro	description
`$(name)`	Expand by iterative substitution.
`${name}`	Expand by iterative substitution removing the last character if it is a path separator character.
`$[name]`	Expand by value; expand the macro to its literal string value without doing any further substitution.

These macros are reserved and expand as follows:

macro	expansion
`$@`	the current target file
`$?`	all dependency files that are newer than the target file
`$=`	the newest dependency file

macro	expansion
$<	the current dependency file
$*	the base filename of the target file
$%	the directory of the first dependency

bmake also predefines the following macros:

platform	macro(s)
MS-DOS	msdos
Macintosh	mc68000, macintosh
Vax	vax
Clipper	unix, ip32, clipper
Sun OS	unix, sparc
Solaris	unix, sparc, SUNSVR4
HP 700	unix, hp700
Windows NT	winNT
DEC Alpha	winNT, _ALPHA_
Silicon Graphics	unix, SGIMIPS
IBM RS-6000	unix, rs6000

Example

This is a simple macro from basic.mke:

```
21 BaseDir = $(MS)/mdl/examples/basic/
```

Advanced Example

Suppose you have some modules that you want to compile with the default .c.o rule, except you want -I$(appDir) included on the compiler command line of the PC. Here is one way to accomplish this:

```
%if pm386
     savecopt =% $[copt]
     copt + -I${appDir}
%endif
     myfile1$(oext) : myfile1.c
     myfile2$(oext) : myfile2.c
%if pm386
     copt =% $[savecopt]
%endif
```

Rules

A **rule** in a makefile defines how to build targets with a given extension from a dependency with a given extension. The general format of a rule is as follows:

```
.[dir1;dir2;...;dirN]depExt.targetExt:
    buildCommands
```

If the directories section `[dir1;dir2;...]` is missing, the rule applies to files in any directory. If a target file depends on more than one file, the extension of the first one in the list is used to find the rule. If more than one rule applies to a target, the last rule in the makefile has precedence. For this reason, you should put rules that are directory-specific after any non-directory-specific rules in the makefile.

Example (from mdl.mki)

```
.mt.r:
    $(msg)
    > $(o)temp.cmd
    -o$@
%if privateInc
    -i$(privateInc)
%endif
    $(rscCompIncs)
    $(altIncs)
    -i$(publishInc)
    -i$(publishIdsInc)
    -i$(stdlibInc)
    $%$*.mt
    <
    $(RTypeCmd) @$(o)temp.cmd
    ~time
```

Dependencies

A **dependency** in a makefile specifies that one or more target files depends on one or more dependent files. The general format of a dependency is as follows:

```
target1 target2 ... targetN: dep1 dep2 ... depN
    buildCommands
```

If any dependent files are newer than any target files, the build commands are executed.

Dependencies are always processed in their makefile order, so you should organize your makefiles accordingly. (That is, if one target depends on another target, its dependency should appear later in the makefile.)

Build commands are a sequence of lines, executed one after the other, to build the target file. If a build command returns an error, bmake terminates. If there are no build commands, bmake attempts to use a predefined rule.

Example

```
53 $(o)basic.mo : $(baseDir)basic.mc $(baseDir)basic.h
```

(where `(o)` has been defined as the object directory.)

The first character in a build command can have special significance:

character	meaning
`-`	Ignore the status returned from this command. Do not terminate if the command returns an error.
`@`	Do not echo, even when bmake is not operating in silent mode.
`|`	Echo only this line, even in silent mode. This is useful for informational messages to the user.
`~`	Use a built-in bmake command (see below).
`>filename`	Write all lines in the makefile until a < to the specified file. This is typically used to write the dependencies to a file to generate a link file.

The following are built-in bmake commands:

command	description
`~CURRENT`	Print the current time.
`~TIME`	Set the target file's modification date to the date of the newest dependency file. This should generally be the last build command for a target file.

Conditional

A **conditional** includes or excludes sections of the makefile, depending on whether macros are present. The following types of conditionals are supported:

directive	description
`%ifdef`	If the macro exists, include subsequent lines.
`%if`	If the expression yields a non-zero result, include the subsequent lines. The conditional `%if` defined is functionally equivalent to `%ifdef`.
`%ifndef`	If the macro does not exist, include subsequent lines.

directive	description
%else	Use with %ifdef and %ifndef.
%elif	Use with %ifdef and %ifndef.
%endif	Close an ifdef or if block.
%undef	Undefines a macro.

Example

```
14 %if defined (msdos)
```

Include Files

As mentioned earlier, bmake allows you to include other files (.mki files) into the current makefile using the %include preprocessor directive.

The files included must be written using the same syntax as bmake makefiles. That is they must be other makefiles or files containing default macro and rule definitions. Examples of this can be found in all distributed MDL example applications.

Example

```
27 %include mdl.mki
28 %include mdlexmpl.mki
...
78 %include $(BaseDir)basicrsc.mki
```

As shown above, basic.mke includes three additional files to assist in, and complete, the application build process:

file	description
mdl.mki	This is the main MicroStation MDL make include file. It contains the default rules and macro definitions that should always be used when building MDL applications, Dynamic Link Modules (DLM), and external programs.

file	description
mdlexmpl.mki	This include file is an example of a project- and/or application-specific make include file. It provides overrides to certain macros defined in **mdl.mki** so that files are placed in a different location as they are built and also includes rules which build the output directories needed if they do not already exist. You should use this file as a template for building your own project and application make include file. As a rule, your makefile should never assume output directories exist.
basicrsc.mki	This include file is actually another makefile which contains the steps necessary to complete the building of the **basic** application. The steps defined in this file cause the language-specific portions of the application to be compiled and combined with the generic portions to create the final application (**.ma**) file. It is recommended that the language-specific portions of an application be separated from the rest of the application if the application needs to be delivered in different languages. This makefile is listed in the "Complete Example" chapter with **basic.mke**.

Starting the bmake utility

After installation, the **bmake** utility is in the (MS)**\mdl\bin** directory.

Many of the MDL tools, including **bmake**, require that the MS environment variable be set so that they can find the MicroStation home directory. If you performed a standard installation, the definition of MS would be:

```
set MS=c:\ustation            (DOS)
export MS=/usr/mstation       (UNIX)
```

The command line syntax for starting the **bmake** utility is as follows:

BMAKE [OPTIONS] MAKEFILE

Normally, [options] are not specified on the command line, but in the BMAKE_OPT environment variable so that the user does not have to specify them every time **bmake** is used. To use BMAKE_OPT, add it to your operating system's environment variable list.

bmake supports multiple include paths. They may be specified via either the command line or the environment variable BMAKE_OPT.

The standard BMAKE_OPT definitions are as follows:

```
export BMAKE_OPT=-I$MS/mdl/include/     (UNIX)
set BMAKE_OPT=-I$(MS)\mdl\include       (DOS)
```

Therefore, the command to make **basic.ma** without BMAKE_OPT set would be:

```
BMAKE -I$(MS)\MDL\INCLUDE BASIC
```

But with BMAKE_OPT set, it would be shortened to:

```
BMAKE BASIC
```

Existing make files that contain paths to include files need not be changed if they are only to be used under DOS or UNIX; merely exchange '\' and '/' whenever appropriate. However, if make files will ever be used on any other platform, they should be modified to conform to the above standard.

Command line options are as follows:

option	description
-a	Always build. For every dependency, execute the build command regardless of whether dependent files are newer than the target files. This option causes bmake to remake all files.
-d	Define a macro. For example, including -ddebug=1 on the command line is the same as specifying debug=1 in the makefile.
-i	Ignore errors. Ordinarily, bmake terminates when a build command returns a non-zero exit status. With this option, bmake ignores errors and continues.
-l	List targets. bmake prints the target file before executing build commands (helpful for debugging).
-m	Missing files are acceptable. Ordinarily, bmake terminates if any dependency files are missing. With this option, bmake continues.
-n	No execution. List, but do not execute the build commands.
-p	Print macros as defined (helpful for debugging makefiles).
-s	Enter silent mode. Do not print build commands as they are executed.
-t	Touch target files. Set the target file date to the date of the latest dependency file rather than executing the build commands.

option	description
-I	Specify the include (*.mki) file path.
-S (PC only)	Specify a swap file path. When specified, bmake will swap as much of itself out of memory as possible, thus allowing more space for the user application. Syntax is: `-S<temp file path>` For example: `bmake -Sc:\temp dlogdemo`. If the `BMAKE_OPT` environment variable is set to `-S<temp file path>`, this option will be set every time bmake is run.

Compiling an Application Command Table

MicroStation lets an application define its own command language in a command table. The keywords in the command language and the command syntax (the order of words in commands) are determined by the contents of the command table resource. Commands are parsed word by word.

The command table resource is created with the MicroStation resource compiler, rcomp. Input to rcomp is a resource source (.r) file. The syntax of this file is modeled after the C programming language. For more information on the MicroStation resource compiler, see "Compiling Resources" on page 7-19.

The first step in creating an application command table is to describe the command syntax in an ASCII file. The command table source file can be built with any convenient text editor. Within this file, a system-defined C language structure called Table defines a hierarchy of **command word tables**. This hierarchy defines the words in the application's command language and command syntax.

The first word of any application command must appear in the **root command word table**, the first command word table in the hierarchy. The command word table used for subsequent words in the application's command will be obtained from the root table's **sub command word table**. The appropriate subtable is determined by the options specified for the matching line in the current command word table. Each successive word in an application's command will be obtained further down the hierarchy in this manner.

The command word source file also contains the following:

- Include (`#include`) statements to include C header files.
- Comments, which the rcomp utility ignores.
- An optional version specified in a C `#pragma` statement that specifies the file version.

- An optional identity or `Ident` string specified in a C `#pragma` statement for the compiler to place at the beginning of the resource file.
- Table variable declarations that delineate the beginning of command word tables.
- Initialization statements for each command word contained in the command word table.

Example

A complete listing of **basiccmd.r**, the command table for **basic.ma**, can be found in "A Complete Example" on page 18-1.

The Table resource declaration defines a command word table. The syntax is as follows:

```
Table <tableidentifier>=
{
     (initialization statements go here, one per command word)
};
```

`<tableidentifier>` represents an 8-bit unsigned integer.

Typically, `#define` statements set up constants for the tableids as follows:

```
#define CT_MAINTABLE 1   /* root command word table */
Table CT_MAINTABLE=
{
     (initialization statements)
};
```

The main (or root) command word table must be assigned a `<tableidentifier>` of 1. The main command word table contains an entry for each keyword that can be used as the first word in a command for the application.

Example

The command table for the **basic** example, **basiccmd.r**, is very simplistic:

```
42 #define CT_NONE 0
43 #define CT_BASIC 1
   ...
49 Table CT_BASIC=
50 {
51     {1, CT_NONE, INPUT, NONE, "OPENMODAL"},
52 };
```

Each valid command word in the table has one initialization statement. The format for each initialization statement is as follows:

```
{<number>, <subtableidentifier> (or zero if no subtable),
 <commandclass>, <options>, "<commandword>"},
```

The number field constructs the command number for the entire command. The range is 1 to 255 for the first three words in a command and 1 to 15 for the last two words if the command has five words. A command can be no longer than five words.

The `commandword` field contains the word that the parser attempts to match. Case is not significant, and the quotation mark delimiters are required. Space, '/' and '=' characters are not allowed.

In the main table, each number must be in the 1-255 range. In a subtable of the main table, each number must also be in the 1-255 range.

When a word in a command is parsed against a command word table, a match is found if one of the following conditions exists:

- Every letter in the test word matches the corresponding letter in a command word in the table and a match exists for no other words.
- The test word is a command word.

Thus, according to the command word table below, the **TE** key-in matches 3 and the **LINE** key-in matches 1. However, the **LI** key-in does not match (is ambiguous).

```
Table CT_ELEMENTS=
     { /* Subtable    CommandClass    Options    CommandWord */
     { 1, CT_NONE,      INHERIT,       NONE,      "LINE" },
     { 2, CT_NONE,      INHERIT,       NONE,      "CURVE" },
     { 3, CT_NONE,      INHERIT,       NONE,      "TEXT" },
     { 4, CT_NONE,      INHERIT,       NONE,      "LINESTRING" }
};
```

The intelligence is added to a command word table with the `<subtableidentifier>`, `<commandclass>` and `<options>` fields in the initialization statement.

The `<subtableidentifier>` field specifies the command word subtable for the next word in the command. (Parsing always starts with the root command word table).

The `<commandclass>` field stores information about the command class in the command word table. MicroStation or an application program can

retrieve and use this information. The classes and their corresponding values are as follows:

Class	Value	Class	Value
PLACEMENT	1	COMPRESS	15
VIEWING	2	REFERENCE	16
FENCE	3	DATABASE	17
PARAMETERS	4	DIMENSION	18
LOCKS	5	LOCATE	19
USERCOMMAND	6	TUTORIAL	20
MANIPULATION	7	WORKINGSET	21
SHOW	8	ELEMENTLIST	22
PLOT	9	UNDO	23
NEWFILE	10	SUBPROCESS	24
MEASURE	11	VIEWPARAM	25
INPUT	12	VIEWIMMEDIATE	26
CELLLIB	13	WINDOWMAN	27
FILEDESIGN	14	DIALOGMAN	28

The possible values are 1 to 63. The first 48 classes are reserved for MicroStation, and the last 15 are available for application use. A command's class is determined by the class specified in the last matched word for which a class is defined. In the above example, the **DELETE ELEMENT** command is in class `MANIPULATION` (7), but the **DELETE CELL** command is in class `CELLLIB` (13).

The <options> field on the initialization statement is a bit field, where several binary flags are joined with the logical **OR** operator. The following flags can be joined with **OR**:

```
DEF | REQ | TRY | CMDSTR (n)
```

The `DEF` (default) flag specifies the default value of the word for a particular table. For example, consider the following command word table hierarchy:

```
Table CT_MAIN=
{
  {1, CT_DELETE, MANIPULATION, NONE, "DELETE"},
  {2, CT_ELEMENTS, PLACEMENT, REQ, "PLACE"},
  {3, CT_ACTIVE, PARAMETERS, REQ, "ACTIVE"}
};

Table CT_ELEMENTS=
{
```

```
    {1, CT_NONE, INHERIT, NONE, "LINE"},
    {2, CT_NONE, INHERIT, NONE, "CURVE"},
    {3, CT_NONE, INHERIT, NONE, "TEXT"},
    {4, CT_NONE, INHERIT, NONE, "LINESTRING"},
    {5, CT_NONE, INHERIT, NONE, "POINT"}
};

Table CT_DELETE=
{
    {1, CT_NONE, INHERIT, DEF, "ELEMENT"},
    {2, CT_NONE, INHERIT, NONE, "VERTEX"},
    {3, CT_NONE, INHERIT, NONE, "PARTIAL"},
    {4, CT_NONE, CELLLIB, NONE, "CELL"}
};

Table CT_ACTIVE=
{
    {1, CT_COLORS, INHERIT, TRY, "COLOR"},
    {2, CT_NONE, INHERIT, NONE, "WEIGHT"},
    {3, CT_NONE, INHERIT, NONE, "STYLE"}
};

Table CT_COLORS=
{
    {1, CT_NONE, INHERIT, NONE, "WHITE"},
    {2, CT_NONE, INHERIT, NONE, "BLUE"},
    {3, CT_NONE, INHERIT, NONE, "GREEN"},
    {4, CT_NONE, INHERIT, NONE, "RED"},
    {5, CT_NONE, INHERIT, NONE, "YELLOW"},
    {6, CT_NONE, INHERIT, NONE, "VIOLET"},
    {7, CT_NONE, INHERIT, NONE, "ORANGE"}
};
```

When the user keys in **DELETE**, the parser extracts the default from the next table. In this case, the **DELETE ELEMENT** command is generated. If the `CT_DELETE` subtable has no entries marked as `DEF`, the **DELETE** command is generated.

The `REQ` (required) flag tells the parser that a selection from the specified subtable is required. Thus, if the user keys in **PLACE** in this example, an error (insufficient arguments) is returned. The `REQ` flag is appropriate when a subtable is also specified in the `<subtableidentifier>` field.

The `TRY` (try parse) flag tells the parser to accept a value from a command word table if there is a match, but do not generate an error if no match exists. If there is no match, the parser saves the non-matching portion for application parsing. The parser needs the `TRY` flag to accept **ACTIVE COLOR RED** or **ACTIVE COLOR 4** in the above example. Without this option, an entry that does not match a command in the subtable results in an error. The `TRY` flag is needed only when a subtable is also specified in the `<subtableidentifier>` field.

The CMDSTR (command string) macro associates a command string in the application's registered command number, MessageList, with the command in the command word table. The message in the command number, MessageList, indicated by CMDSTR (n) displays every time the command is executed. For more information on registering an application's command number, MessageList, see the mdlState_registerStringIds function. For a description of MessageLists, see "Compiling Resources" on page 7-19.

If a string is successfully parsed with two different command tables, MicroStation uses the result that parsed the most keywords. For example, if the user keys in PLACE LINE MYWAY and an MDL application has a parse table that matches all three keywords, MicroStation generates the command for the application.

Syntax

To execute rcomp directly:

Key-in: RCOMP [-H] INPUT-FILE

The output command table resource file has the same name as the input file, with the .rsc extension. If the -h command line option is specified, an output file with the generated command numbers is produced in the format of a C include file. The file also has the same name as the input file, with the .h extension. This .h file is a programming aid that defines a mnemonic for each command and its corresponding command number.

Example

The command table entries in basic.mke read:

```
42 $(genSrc)basiccmd.h : $(BaseDir)basiccmd.r
43 $(o)basiccmd.rsc : $(BaseDir)basiccmd.r
```

These call the .r.h: and .r.rsc: rules in mdl.mki, which subsequently call rcomp.

If rcomp detects an error while processing the input file, the line containing the error is reported. One possible error is:

- Unrecognized constant: a constant provided on a resourceclass statement or within the body of a command word table declaration was not defined (#define) earlier in the program.

  The command numbers generated by rcomp (using the -h option) are 32-bit integers constructed as follows:

- The number in the initialization statement for the word matching the first word goes in the command's high byte.

- If there is a second word and a `<subtableidentifier>`, the number for the matching word goes in the second highest byte. Similarly, the match for the third word goes in the third-highest byte. The matching numbers go in the high four bits and low four bits of the low byte, for the fourth and fifth words.

When MicroStation successfully parses a user key-in, it generates a command number in the same manner. MicroStation creates a queue element that contains the command number and task ID of the application associated with the command table. When MDL receives a `CMDNUM` queue element for an MDL application, it uses the task ID to determine the application to execute the command. It then scans the list of command numbers associated with the application to determine the function to call.

MDL functions are linked with function numbers at compile time. The `cmdNumber` keyword associates a function with a command number. The `cmdNumber` keyword designates a list of function numbers handled by the function. Typically, the command numbers come from a **.h** file generated by **rcomp**. For example, assume that the generated **.h** file contains the following lines:

```
#define CMD_CHANGE 0x01000000
#define CMD_CHANGE_TEXT 0x01010000
#define CMD_CHANGE_TEXT_SINGLE 0x01010100
```

The source file would include the **.h** file and a declaration with the following format:

```
changeTextSingle(char *unParsedP)
cmdNumber CMD_CHANGE, CMD_CHANGE_TEXT, CMD_CHANGE_TEXT_SINGLE
{
     ...
}
```

Generating Resource Files from C Type Definitions

The **rsctype** utility compiles structure and union declarations (usually stored in a **.mt** file) and generates a resource source (**.r**) file containing structure and union definitions. **rcomp** compiles the source file and generates the resource file. The built-in functions described in this section can use the resource file.

MDL provides the C expression built-in functions to evaluate expressions at run-time. If the expressions contain references to fields contained in structures or unions, the corresponding structures and unions must be defined in a resource file.

A source program to be processed by **rsctype** uses C syntax for defining structures and unions. However, it cannot contain declarations that allocate space and it cannot contain executable statements.

The format of an **rsctype** program is typically as follows:

- A list of header files that contain the desired structure and union definitions.
- Structure and union definitions included in-line.
- `publishStructures` statements specifying structures to be defined in the resource source file.

The syntax for a `publishStructures` statement, shown in the following example, is identical to the syntax for a function call.

```
publishStructures(structureName[, structureName, ...])
```

Any number of structure and union names can be included in the list of names.

Example

This is the **.mt** file for **basic.ma**:

```
30#include "basic.h"
31publishStructures(basicglobals);
```

The larger and more complex the MDL application, the more `#includes` and additional structures that the **.mt** file will contain.

Predefined macros

rsctype provides built-in macros to be used in conditional compilations. It provides all definitions provided with **mcomp**. It also defines the `type_resource_generator` variable.

rsctype, like **mcomp**, uses the `MDL_COMP` environment variable to obtain arguments. The string specified by `MDL_COMP` is prepended to the command line arguments before the arguments are processed. Any combination of command line arguments can be specified with the `MDL_COMP` environment variable. The options must be separated by blanks.

`MDL_COMP` generally specifies include directories. The following `MDL_COMP` definitions specify two include directories:

```
export MDL_COMP=-i$MS/mdl/include/ -I/usr2/myhdrs (UNIX)
set MDL_COMP=-i$(MS)\mdl\include -Ic:\myhdrs      (DOS)
```

Syntax

The command line syntax for **rsctype** is as follows:

```
rsctype [-flag, -flag,...] input-file
```

`-flag` can be one of the following:

flag	description
-d<name>=<value>	Define name with value value as it is defined by a define statement in the source file. This option does not require `=value`.
-p	Display preprocessor output.
-v	Verbose: show the compiler's progress.
-o<filename>	Specify the output filename. If this option is not specified, the output filename is created when **.r** is appended to the input filename prefix.
-i<dir>	Add the specified directory to the list of directories searched for include files. The MDL compiler supports up to 40 include directories.

If only the file prefix is specified for the input filename, **rsctype** appends **.mt** to the name.

Example

The type resource entries in **basic.mke** read:

```
48 $(o)basictyp.r : $(BaseDir)basictyp.mt $(BaseDir)basic.h
49 $(o)basictyp.rsc : $(o)basictyp.r $(BaseDir)basic.h
```

These call the `.mt.r:` and `.r.rsc` rules in **mdl.mki**, which subsequently call **rsctype** and **rcomp**, respectively.

Compiling Resources

As described earlier in "Compiling an Application Command Table," the resource compiler **rcomp** generates resources from ASCII source files, storing them in resource files to be used by MDL applications. Resource compiler source files are usually denoted by the file extension, **.r**. The default extension for generated output files is **.rsc**.

Because of the wide variety of data that can be stored in a resource file, this section will not describe the construction of resource source files. For information on doing so, refer to the "Resources" chapter.

Example

A listing of **basic.r**, the resource file for **basic.ma**, can be found in "A Complete Example" on page 18-1.

Command line syntax

```
rcomp [options] source_filename
```

The following options are supported:

flag	description
-d<name>=<value>	Define name with the value value as if it is defined by a `#define` statement. This option does not require =value.
-f<format>	Specify the output format. Options include **clipper**, **hp700**, **mac**, **pc**, **sparc** and **4x**, which specifies 4.x compatibility.
-h	Generate a C header file of `CMD_` defines when compiling command tables.
-ho<filename>	Generate a C header file of `CMD_` defines when compiling command tables and allow the filename to be specified.
-i<dir>	Search this directory for include files.
-o<filename>	Specify explicit output filename. If this option is not specified, the output filename is determined from the input filename. The default extension for output files is **.rsc**.
-p	Display preprocessed output.
-pi	Trace include file history. If this flag is defined, then every time the compiler encounters an `#include` preprocessor directive, it displays both the name of the file that contains the `#include` statement, and the name of the file to be included.
-r	Generate a definition resource for each `resourceclass`.
-t<directory>	Directory to search for `#pragma translate` resources. If not specified, current directory is searched.
-v	Verbose: show progress of the compiler.

The command line switches and input filename can be provided in any order. Response files can be used to specify command line switches as well as the input filename to the compiler as the following example shows:

```
RCOMP -ddebug @args.txt
```

The contents of **args.txt** are as follows:

```
"-v -ic:\mydir\myhdr.-h input.r"        (MS-DOS)
"-v -i/usr/mydir/myhdr.-h input.r"      (UNIX)
```

Example

The resource entry in **basicrsc.mki** reads:

```
22 $(rscObjects)basic.rsc : $(BaseDir)basic.r $(langSpec)basictxt.h \
23                                $(privateInc)basic.h
```

This calls the `.r.rsc:` rule in **mdl.mki**, which subsequently calls **rcomp**.

Compiling MDL Applications

The **mcomp** utility compiles MDL source code (usually stored in a **.mc** file) and generates an MDL output (**.mo**) file that is ready to be linked using **mlink**.

The command line syntax for the MDL compiler is as follows:

```
mcomp [-flag, -flag,...] input-file
```

`-flags` can be one of the following:

flag	description
`-b`	Warn if the input file contains a reference to a built-in function that hasn't been declared.
`-c`	Compile only: on UNIX machines, the compiler automatically invokes the MDL linker if this flag is not specified and the source file compiles successfully. If this flag is specified, the linker is never invoked automatically. On the PC, the linker is not invoked automatically.
`-d<name>=<value>`	Define name with the value value as if it was defined by a `#define` statement. This option does not require value.
`-g`	Dump all debugging information to the output file.

flag	description
-i<dir>	Add the specified directory to the list of directories searched for include files. The MDL compiler supports up to 40 include directories.
-o<filename>	Specify the output filename. If this option is not specified, the output filename is created when **.mo** is appended to the input filename prefix.
-p	Display preprocessor output.
-pi	Trace include file history. If this flag is defined, then every time the compiler encounters a `#include` preprocessor directive, it displays both the name of the file that contains the `#include` statement, and the name of the file to be included.
-t<directory>	Directory to search for `#pragma translate` resources. If not specified, current directory is searched.
-v	Verbose: show the compiler's progress.

If only the file prefix is specified for the input filename, the MDL compiler appends **.mc** to the name.

mcomp, like **rsctype**, uses the `MDL_COMP` environment variable to obtain arguments. The string specified by `MDL_COMP` is prepended to the command line arguments before the arguments are processed. Any combination of command line arguments can be specified with the `MDL_COMP` environment variable. The options must be separated by blanks.

`MDL_COMP` generally specifies include directories. The following `MDL_COMP` definitions specify two include directories:

```
export MDL_COMP=-i$MS/mdl/include/ -i/usr2/myhdrs (UNIX)
set MDL_COMP=-i$(MS)\mdl\include -ic:\myhdrs      (DOS)
```

Under MS-DOS, the command line is restricted to 128 bytes. To use a longer command line, use command line indirection as follows: place the command line in a file and run **mcomp** with the command: `mcomp @<filename>`.

Example

The MDL source entry in **basic.mke** reads:

```
53 $(o)basic.mo : $(BaseDir)basic.mc $(BaseDir)basic.h
```

This calls the `.mc.mo:` rule in **mdl.mki** which subsequently call **mcomp**.

mcomp provides built-in macros to be used in conditional compilations. The MDL compiler's built-in definitions are the same as the result of the following preprocessor directive:

```
#define <variable-name> 1
```

The built-in macros for **mcomp** (and **rcomp** and **rsctype**) are:

platform	macro(s)
MS-DOS	`mdl`, `dos`, `pm386`
Macintosh	`mdl`, `mc68000`, `macintosh`
Vax	`mdl`, `vax`
Clipper	`mdl`, `unix`, `ip32`, `clipper`
Sun OS	`mdl`, `unix`, `sparc`
Solaris	`mdl`, `unix`, `sparc`, `SUNSVR4`
HP 700	`mdl`, `unix`, `hp700`
Windows NT	`mdl`, `winNT`
DEC Alpha	`mdl`, `winNT`, `_ALPHA_`
Silicon Graphics	`mdl`, `unix`, `SGIMIPS`
IBM RS-6000	`mdl`, `rs6000`

`mdl` is always defined and includes or excludes code based on whether **mcomp** or another compiler is being used.

The macro `__LINE__` expands to the line number of the file that contains the statement. `__FILE__` expands to a string that contains the name of the file that contains the macro. `__DATE__` expands to a string that contains the date the file is compiled. `__TIME__` expands to a string that contains the time the file was compiled. The statement:

```
printf ("line number %d in %s, compiled %s at %s\n",
    __LINE__, __FILE__, __DATE__, __TIME__);
```

prints:

```
line number 4 in sample.mc, compiled Oct 6 1993 at 13:10:42
```

Using the MDL Librarian

The MDL librarian, mlib, is used to combine multiple MDL object (.mo) files into a larger, linkable, MDL library (.ml) file. Use of this utility is optional, and many MDL applications, including basic.ma, are built without using mlib.

The command line syntax for the MDL librarian is as follows:

MLIB COMMAND[MODIFIERS] LIBRARY-FILE [INPUT-FILES]

The commands are as follows:

command	**description**
a	Add the files to the library file.
d	Delete the files from the library file.
t	Display a table of contents.
x	Extract the files from the library.

The modifiers are:

modifier	**description**
v	Verbose.
c	Do not display a message if the library file is created.

The following command requests a verbose table of contents of the mathlib.m library file:

MLIB -TV MATHLIB.ML

mlib stores any type of file in a library file. If the file being stored is an MDL object file, mlib extracts information from the file's symbol table and stores the information in the library file's header. This activity enables mlink to process the library faster when deciding which object images from the library file should be used.

Linking MDL Applications

The MDL linker (mlink) combines MDL object (.mo) or MDL library (.ml) files into a resource (.mp) file called a program file. The program file can be used as an application file, or it can be combined with other resource files. This combination creates an application file that contains all resources that the application requires.

If the program file will be merged into an application file with other program files, a task ID must be specified in the link step. Each program in the application file must have a unique task ID.

mlink accepts MDL object files and MDL library files as input. When processing a library file, **mlink** includes a file from the library only if the file satisfies any outstanding unresolved references. **mlink** performs multiple passes of a library file, stopping when a pass does not include any more files from the library.

The command line syntax for **mlink** is as follows:

MLINK [-FLAG, -FLAG,] INPUT-FILES

The `-flag` parameter can have the following values. (A space must separate the option and flag value, and a hyphen '-' must precede the flag value.)

`flag`	**description**
-a[name]	Designate the name of the application file created by **mlink**. If this name is not specified, **mlink** creates the name by appending **.ma** to the prefix of the first input filename.
-cs[options]	Control usage of common stack. Valid combinations are **-cs**, **-cson** and **-csoff**. By default, the common stack is used.
-g<options>	Control the amount of debugging information added to the output file. If **-g** or **-gd** is specified, **mlink** provides all information the MDL debugger can use. If **-gn** is used or if the **-g** flag is omitted, **mlink** provides only enough information for MDL to generate a useful message if the MDL application causes a fault at runtime. If **-go** is specified, all debugging information is omitted.
-m<options>	Request a load map and specify the options. With **l**, the labels will be sorted by location. With **n**, the labels will be sorted by name.
-s[stack-size]	Request a specific stack size in bytes. If no stack size is requested, **mlink** assigns a stack size. It displays the value after linking the application. If the default stack size is too small, a stack overflow occurs at runtime, causing MDL to abort the application and display a message indicating that the stack overflowed. If this happens, link the application specifying a larger stack size. If the default stack size is too large, memory is wasted.
-t[taskID]	Specify a task ID to be associated with the application. The task ID must have 15 bytes or less. **mlink** converts the task ID to upper-case.
-v	Verbose: show **mlink**'s progress.

Prior to version 5 of MicroStation, every MDL application had its own stack. This method wasted memory because different applications rarely used the stack at the same time. Therefore, the concept of a common stack was introduced in version 5, allowing nearly all applications to use the same stack. An application can use the common stack unless it calls the function `mdlInput_waitForMessage`. When an application using the common stack calls `mdlInput_waitForMessage`, MicroStation aborts the application and displays the message "MDL detected invalid use of suspend."

An application created with the 4.0 version of **mlink** gets its own stack. This allows 4.0 applications to be protected from any incompatibilities introduced by the use of the common stack.

An application created with the 5.0 version of **mlink** uses the common stack unless **-csoff** is specified in the link step.

If a program file does not have a specified task ID, the task ID is generated from the application filename when the program is loaded. See "Running MDL Applications" on page 8-1, for more information.

Under DOS, the command line is restricted to 128 bytes. To use a longer command line, use command line indirection. Place the command line in a file. Run **mlink** with the command: `mlink @<filename>`.

Example

The link portion of **basic.mke** reads:

```
57$(o)basic.mp : $(basicObjs)
58     $(msg)
59     > $(o)make.opt
60     $(linkOpts)
61     -a$@
62     $(basicObjs)
63     <
64     $(linkCmd) @$(o)make.opt
65     ~time
```

This rule calls the **linkcmd** macro which is mapped to **mlink**.

Resource Librarian

The resource librarian is used to merge resource files that have been created from a variety of sources. Its primary use is to build the final application (**.ma**) form of an MDL project. Because its output format is a standard compiled resource file, **rlib** can also be used to merge **.rsc** files. **rlib** can accept the following types of files as input:

- Files generated by the resource compiler (**.rsc** files).
- MDL program (**.mp** or **.ma**) files.
- Files created by MDL applications using the `mdlResource_createFile` function.

The following command line syntax is used for the resource librarian:

RLIB [OPTIONS] FILE 1, FILE 2 ... FILE N

If a file is provided without an extension, **.rsc** is assumed. **.rsc** files are broken down and merged together according to `resourceclass`. They are then written to the output file.

The following options are supported:

`flag`	**description**
-f<format>	Specify the output format. Options include **clipper**, **hp700**, **mac**, **pc**, **sparc** and **4x**, which specifies 4.x compatibility.
-o<filename>	Specify the output filename. If this option is not specified, the output filename is determined from the first input filename. The default extension for output files is **.rsc**. If you are building the final form of an MDL application, be sure to specify **.ma**.
-r<resourceID>	Restrict the librarian's operation to a particular resourceID.
-s"string"	Specify the output file's identification string.
-t<class>	Restrict the librarian's operation to a particular `resourceclass`. class is the `resourceclass` identifier as specified in the `resourceclass` statement. It can be specified in hex, decimal or character constant form. For example, `-t'Msgs'` and `-t0x4d736773` are equivalent.
-v	Verbose: show progress of the librarian.

Input files and command line switches can be provided in any order on the command line. Response files can also be used as they are with the resource compiler.

When adding customized resources into **ustation.rsc** with **rlib**, the **-f4x** option should be used to ensure that **ustation.rsc** remains in 4.x resource file format.

Example

The resource library portion of **basicrsc.mki** reads:

```
25$(mdlapps)basic.ma : $(basicRscs)
26     $(msg)
27     > $(rscObjects)make.opt
28     -o$@
29     $(basicRscs)
30     <
31     $(rscLibCmd) @$(rscObjects)make.opt
32     ~time
```

This rule calls the **rscLibCmd** macro which is mapped to **rlib**.

8 Running MDL Applications

This section describes how an MDL program is loaded, how its commands are started, and how it is unloaded.

MDL distinguishes between **programs**, **application files** and **tasks**. These terms are all defined in "MDL Applications" on page 1-5.

Relationships Between Terms

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 contains one or more programs.
- When a program is loaded, MDL creates a task.
- When a program is unloaded, MDL destroys the task.

Loading an MDL Program

All methods of loading an MDL program allow an application filename and**task ID** to be specified, but only the application filename is required.

When the application filename is specified, only the root of the filename is required, although the full filename can be provided. When trying to locate the file, MDL looks for a file with extension **.ma** or **.rsc**. It looks for the file in the paths specified by the environment variables `MS_EXE` and `MS_MDL`. If **example** is specified as the root of the application filename, MDL looks for the files in the following order:

1. **example.ma** in the directories specified by `MS_EXE`.
2. **example.ma** in the directories specified by `MS_MDL`.
3. **example.rsc** in the directories specified by `MS_EXE`.
4. **example.rsc** in the directories specified by `MS_MDL`.

If the task ID is not specified, MDL strips the file extension and path from the full file specification to derive the root filename. It converts the root filename to upper-case

and uses the result as the task ID. Once MicroStation has the task ID, it uses it as follows, regardless of whether the task ID is generated or specified. MicroStation tries to load the program that has the required task ID. If the application file does not contain a program with that task ID, MicroStation then tries to load a program that does not have a constant task ID. If it finds one, it loads the program and assigns the task ID.

An MDL program can be loaded by one of the following:

- An **MDL LOAD <name>** command.
- Selecting Load from the MDL Applications menu.
- Another MDL task calling the built-in function `mdlSystem_loadMdlProgram`.
- MicroStation initialization. MicroStation loads programs specified by the MDL environment variables `MS_INITAPPS` and `MS_DGNAPPS`.
- An application startup element (type 66) in the design file.

The syntax for the **MDL LOAD** command is as follows:

MDL LOAD [DEBUG | NODEBUG] <application filename>[,task ID] [parameters]

Any number of parameters can be specified. The parameters are separated by blanks and supplied to the MDL task started by the **MDL LOAD** command.

Typically, the simplest form of the **MDL LOAD** key-in is used. For example, to load the **example** application, key in **MDL LOAD <example>**.

The environment variables `MS_DGNAPPS` and `MS_INITAPPS` specify programs to load during MicroStation initialization. Each environment variable can specify a list of programs. If a list specifies more than one program, the specifications must be separated by semicolons. The format of a program specification is as follows:

```
application-file-name[,task ID][/d]
```

Typically, only the application filename is used. `/d` tells MDL to activate the debugger after loading the program. An example of a specification is as follows:

```
MS_DGNAPPS=calculat;qdim;myapps,macros/d
```

`MS_INITAPPS` specifies MDL programs to be loaded before MicroStation opens the design file or enters graphics mode. These programs are typically “front-end” programs used to select or manage files. See “Using MS_INITAPPS Applications” on page 8-5 for more information on these programs.

MS_DGNAPPS specifies MDL programs to be loaded immediately before MicroStation opens the first design file. Use MS_DGNAPPS to specify MDL programs that you want loaded automatically when you use MicroStation.

An application startup element in the design file specifies a program to be loaded when the design file is loaded. When MicroStation opens a design file, it searches for application startup elements when it looks for the TCB and other control elements. If it finds a startup element, it queues an MDL LOAD command that loads the program specified in the element with any command line arguments stored there. The MDL LOAD commands queued this way are processed immediately after the file displays. See `mdlSystem_createStartupElement` for more information.

After loading an MDL program, MicroStation tries to execute the task's `main` function. The `main` function is optional and is needed only if the task needs to perform initialization. If the task does not contain a `main` function or if a `main` function does not explicitly call an exit function, the task remains resident waiting to process commands. If the task exists when the load request occurs and the task has a reload function, MDL calls the reload function. (See `userSystem_reloadProgram` for more information on reload functions).

The arguments to `main` are similar to those for standard C. *argc* contains a count of the number of arguments. *argv* is an array of pointers to the arguments. *argv*[0] points to the name of the MDL application file. *argv*[1] points to a string specifying the source of the load request. If the program is loaded as the result of an environment variable (such as MS_INITAPPS), *argv*[1] points to the name of the environment variable. If the program is loaded as the result of an MDL LOAD command or from MDL Applications menu, *argv*[1] points to the string "USER". If the program is loaded as the result of an `mdlSystem_loadMdlProgram` call, *argv*[1] points to the string "MDLTASK". If the program is loaded as the result of an application startup element, *argv*[1] points to the string "STARTUP". The remaining arguments are provided as part of the load request.

Unloading an MDL Program

An MDL program can be unloaded by one of the following:

- An MDL UNLOAD command.
- A call to exit or `mdlSystem_exit` from the MDL task.
- MicroStation shutting down.
- A fatal error in the MDL program.
- A call to `mdlSystem_unloadMdlProgram` from an MDL task.
- The MDL debugger QUIT command.

When unloading an MDL program, MicroStation first calls the task's unload user hook if the task has specified one. (See `userSystem_unloadProgram` for more information.) The user hook can reject the unload request. If the unload request is not rejected, MicroStation cleans up most of the task's resources such as open files and allocated memory. There are specific items that are not cleaned up: 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 and the `mdlStringList_destroy` function to free memory allocated through string list functions.

Once all memory is cleaned, MicroStation removes the task from memory. Finally, if the unloaded program was started by another MDL task, the other task is notified.

The syntax for the unload command is as follows:

MDL UNLOAD <task ID>

Using Commands in MDL Tasks

This section describes how to declare a function as an MDL command and how to start the command. See "State Control Functions" for information on command organization.

When a function is declared, it can be designated as a command. To designate a function as a command, specify that the function's name is available as a command name, specify a list of command numbers to be associated with the function, or both.

To specify that a function's name is available as a command name, specify `cmdName` before the function name. To associate command numbers with a function, specify `cmdNumber` and a list of command numbers after the argument declarations for the function. The following are examples of the syntax for command declarations. For each example, the executable portion of the function would follow the function declaration, but is not shown here. Also, `CMD_TEST1` and `CMD_TEST2` are macros defined as integer constants.

```
cmdName void testCommand(char *unparsedP)

void testCommand () cmdNumber CMD_TEST1, CMD_TEST2

cmdName testCommand(char *unparsedP) cmdNumber CMD_TEST1
```

If a function's name is available as a command name, the command can be started with an MDL COMMAND key-in. The syntax for the MDL COMMAND key-in is:

MDL COMMAND [task ID,] <command name> <parameter>

If multiple tasks are loaded and these tasks have commands with the same name, task ID gets the command from the intended task. Otherwise, only command name is required.

If the task ID and command name are specified, a comma must separate them. Otherwise, MDL interprets the first string as the command name and the second string as a parameter. The key-in MDL COMMAND TASK,TESTCOMMAND attempts to start the TESTCOMMAND (that is, execute the testCommand function) in the TASK. The MDL COMMAND TESTCOMMAND PARAMS key-in starts the CMD command, supplying a pointer to PARAMS as the argument unparsedP.

If a task has a command table loaded and the task's commands have command numbers associated with them, the task's commands can be accessed with the commands specified in the command table. When this method is used, the key-in consists entirely of keywords specified in the application command table. MDL COMMAND should not be part of the key-in. See "Compiling an Application Command Table" on page 7-11, for more information.

Aborting an MDL Task

While a task is running, it can be aborted by pressing <Ctrl-C> or <Ctrl-Break>. Pressing these keys lets you regain control from a hung MDL task. A task can inhibit this capability.

See `mdlSystem_userAbortEnable`, `mdlSystem_extendedAbortEnable` and `mdlSystem_extendedAbortRequested` for more information.

Using MS_INITAPPS Applications

Many times an MDL application may need to present a controlled environment to the user upon startup. The application uses MicroStation's graphical user interface to let the user select a project or other preferences relating to the work session. Alternatively, an MDL application can operate MicroStation in a batch mode, where no graphics appear on the screen and no user interaction occurs. The `MS_INITAPPS` environment variable lets MicroStation accomplish both of these objectives.

As described above, programs specified with the `MS_INITAPPS` environment variable are started when MicroStation is activated. These programs are started after MicroStation has initialized internal variables for MDL and other necessary components and before MicroStation has entered graphics mode. The MDL_INITAPPS programs are started in the order specified in the environment variable. While a list of programs are allowed as initial applications, only one is needed.

The MS_INITAPPS program's `main` function is called with *argv*[1] pointing to `MS_INITAPPS`. The program uses this method of calling the function to determine that it is starting as an initial application, rather than being invoked by the user during normal MicroStation operation. *argv*[2] ...*argv*[*argc*-1] are the command line arguments that MicroStation was started with. Command line arguments that will be processed by the MS_INITAPPS applications (and ignored by MicroStation) start with `-i`.

When the program is designed to provide a graphical front end to the user, the application should follow these steps:

1. Put MicroStation in graphics mode by calling `mdlSystem_enterGraphics`.
2. If the program will remain loaded while the user is working on the design file, it should designate a function to be called when MicroStation reloads it by calling `mdlSystem_setFunction`.
3. If the front-end program will let the user specify a design file on the MicroStation command line, the program should examine the command line arguments that started MicroStation and determine whether a design file was specified. (The design file name will be the only command line argument that does not start with the '-' character). If a design file was specified, the program should skip the remaining steps and go directly to step 6, where MicroStation resumes control.
4. Perform any other initialization code, such as setting up other user functions to be called in response to MicroStation events.
5. Display the application's opening dialog box
6. Return control to MicroStation by returning from `main`.

The front-end program regains control when its dialog hook functions are called. Presumably, the program will let the user select a design file to work on in MicroStation. To accomplish the transition from the front-end program to MicroStation, the program follows these steps:

1. Close its dialog box using the `mdlWindow_close` function.
2. Enter the user's chosen design file using `mdlSystem_newDesignFile`.
3. Either exit entirely (using exit) or relinquish control to MicroStation and wait for its reload program user function to be called. This function tells the program to reopen its dialog box and once again supervise interaction with the user.

If the MS_INITAPPS program exits without first opening a design file, MicroStation terminates and returns to the operating system prompt.

If, on the other hand, an MS_INITAPPS program is designed for batch processing in MDL and will not use graphics or interact with the user, it should follow these steps:

1. Process the command line arguments that MicroStation is started with, particularly the command line switches starting with `-i`.
2. Process the design file(s) as needed, using `mdlSystem_newDesignFile` to load them.
3. Close the last design file using `mdlSystem_closeDesignFile`.
4. Call exit without a design file active to cause MicroStation to return to the operating system prompt.

9 Debugging MDL Applications

The MDL debugger provides full source-level debugging capabilities for MDL. In addition to providing standard debugging commands, it also provides a C-expression evaluator.

Debugging

The first time a particular MDL application is to be debugged, its name must be specified. To load an application and activate the MDL debugger from MicroStation, key in the following, where `<application>` is the application name:

MDL LOAD DEBUG <APPLICATION>

To debug a previously loaded application, key in the following:

MDL DEBUG [APPLICATION]

The application name is an optional part of the command, and need only be specified if the MDL application is unloaded, or another MDL application has been loaded since you last debugged the first application. If the application is loaded with the MDL LOAD command without the DEBUG modifier, MDL loads the symbol table before activating the debugger. Otherwise, it only activates the debugger. MDL DEBUG operates as an immediate command. It does not disrupt the state of the currently active MicroStation command.

It is possible to have debugging information loaded for more than 1 application at a time, but the debugger can only be in command mode for 1 application at a time. For example, it is possible to load 3 MDL applications with debugging, but when the debugger prompts for a command it is only possible to set breakpoints and display values for the same application.

The `mdlSystem_enterDebug` function activates the debugger from an MDL application. If the application does not have the debugging information loaded, MDL loads the symbol table.

Debugger Input & Output

Input to the debugger is always entered at a MicroStation keyboard.

On the PC, the debugger's output goes to standard out, and the debugger uses some ANSI sequences in the output. The following are standard PC configurations for debugging with MDL:

- Use two monitors but configure MicroStation for one. Start MicroStation from the monitor MicroStation will not use. All debugger output will go to the monitor where MicroStation was started.
- Run with standard out directed to an external terminal or terminal emulator by redirecting MicroStation standard out from the command line or by using the MicroStation environment variable `MS_DBGOUT` to specify the output destination. For example, `MS_DBGOUT=com1` writes all output to `com1`. The debugger uses the keyboard even if the output is redirected.
- Run with standard out displayed to MicroStation's DOS window. This is the default method. If you start MicroStation from a screen that MicroStation uses and you do not specify `MS_DBGOUT`, then the debugger's output is displayed to MicroStation's DOS window.

If the debugger uses a separate monitor or uses MicroStation's DOS window, then ANSI.SYS must be loaded for all of the debugger's output to display correctly.

A debugger command can be entered when the mdb> prompt displays in the debugger's output area. Multiple commands can be entered on one line. These commands must be separated by semicolons.

The debugger supports some command line editing on all platforms. Characters can be entered anyplace in a command line that is being edited. Only insert mode is available.

On the PC, use the up and down arrow keys to move back through the command history. To move through a command line, use the left and right arrow keys. Press <Ctrl> and left arrow simultaneously to move back a word. Press <Ctrl> and the right arrow simultaneously to move forward a word. Use the HOME key to move to the beginning of the command line. Use the END key to move to the end of the command line. Use the DELETE key to delete the character to the right of the cursor. Use the BACKSPACE key to delete the character to the left of the cursor. Use <Esc> to clear the command line.

On UNIX workstations, use <Ctrl-P> and <Ctrl-N> to move through the command history. <Ctrl-B> moves the cursor left one character in the command line. <Ctrl-F> moves the cursor right in the command line. <Ctrl-W> moves the cursor forward a word. <Ctrl-A> moves the cursor to the start of the command line. <Ctrl-E> moves the cursor to the end of the command line. Backspace deletes the character to the left of

the cursor. <Ctrl-D> deletes the character to the right of the cursor. Press the escape key to clear the command line.

Use <Ctrl-R> to search through the debugger's command history. Press <Ctrl-R> followed by a string and <Enter> or <Return> to cause the debugger to search for a specific string. <Ctrl-R> followed by <Enter> or <Return> causes the debugger to the repeat the search using the same search string.

You can suspend the debugger's output by pressing <Ctrl-S>; you can resume it by pressing any other key. You can abort the debugger's output using <Ctrl-C>. <Ctrl-Break> also works on the PC. It overrides any other queued keystroke, so it may work faster. When the output is aborted, MDL displays the prompt and waits for another command.

On the PC, ! may be entered to run a DOS command. ! followed by a command runs the command. ! without the command starts a DOS session. To return to the debugger from DOS, key in **EXIT**. The setting in the External Programs field of the User Preferences menu determines whether MicroStation frees memory before starting the command.

How the Debugger Finds Source Code

When the MDL debugger displays source code, it reads the code from the source code file. It looks for the source code file in the following places, in order:

1. When looking for the source file, the debugger first uses the name exactly as it is specified in the compile command or makefile.
2. Next it looks for it in the current directory and in the directories specified by `MS_DBGSOURCE`.
3. Finally, it looks in the directory specified that contains the **.ma** application file.

Therefore, when make files specify the full directory path, `MS_DBGSOURCE` is not needed. A typical setting for `MS_DBGSOURCE` is the following:

```
MS_DBGSOURCE = c:\ustation\mdl\examples\misc\;
c:\ustation\mdl\examples\plashape\
```

Preparing an Application for Use With the Debugger

To prepare an application for debugging, you must compile and link it with the **-g** modifier. When the **-g** modifier is specified to the compiler, the compiler dumps all debugging information to the object file. If the **-g** modifier is not specified, the compiler dumps only the function names to the debugger portion of the object file.

When the **-g** modifier is specified to the linker, the linker copies all available debugging information to the application file. If the **-g** modifier is not specified, the linker copies only the function names and source filenames to the application's debugging information.

Using C Expressions With the Debugger

C expressions can also be entered at the debugger prompt. C expressions are used with the MDL debugger to display and modify data in memory. The debugger accepts C expressions, which include the standard operators, variables and structure definitions. The debugger does not accept declarations or function calls for functions in the MDL program. The debugger does accept function calls for MDL's built-in functions.

The debugger knows the type and location of variables. It knows the format of structures referenced, so structure names can be used in casting. However, it does not know of `typedefs` included in the source code.

The debugger evaluates the expression and performs any assignments included in the statement. It then displays the result of the expression. If the result is an array, it displays all members of the array. If the result is a structure, it displays all members of the structure.

To display the elements of an array, enter only the array name. To display an array address, request the address of the first element. For example, if the source code contains the declaration `char sampleArray[10];`, the debugger command sampleArray would display the values of all 10 array elements. The `&sampleArray[0]` command would display the address of the array's first element.

To display data with a format different from the format declared in the program, use a cast or use the display command. For example, if the source code contains the declaration `int *pInteger` and you want to view the second byte as a character, key in `*((char *)pInteger+1)`. You can cast a constant as the array's address with an expression like `(long [10])0x12728`. This command displays 10 `longs` starting at the address `0x12728`. The expression `(long [10])&sampleArray[4]` displays 10 `longs` starting at the address of the fourth element in `sampleArray`.

When you enter a request, the debugger first tries to parse it as a command. If parsing for a keyword does not find a match, the entire line is evaluated as a C expression. In ambiguous cases, the first word is interpreted as a reserved keyword. To specify a C expression that may be mistaken for a command, enclose the expression in parentheses. For example, to display the value of a variable `b`, enter the command (`b`). Entering `b` without parentheses displays a list of break points.

Debugger Commands

This section defines the debugger commands: ALIAS, BREAKPOINT, CALLS, DISPLAY, DOWN, DUMP, EVALUATE, GO, HELP, MEMORY, QUIT, RECORD, SCOPE, SET, STEP, SYMBOLS, TYPE and WATCH.

Debugger command syntax

Like MicroStation commands, debugger commands may be entered with only the shortest string possible to uniquely identify it. For example, for the QUIT command, any of the following commands are acceptable: Q, QU, QUI and QUIT.

The format of the commands is as follows:

```
command/modifier arguments[,arguments..]
```

Commands and modifiers are keywords. For any keyword, you need to enter only enough characters to uniquely identify the keyword.

Modifiers are usually optional. A command may take more than one modifier.

The ALIAS command

The ALIAS command defines simple aliases for the debugger's command interface. When it starts processing a command line, the debugger first determines if the first token in the command line corresponds to an alias. If so, the debugger replaces the token with the alias string and repeats the process.

This substitution process is iterative but not recursive. The debugger continues this process until there are no matches, but a given alias can only be used once in the expansion of a given command line.

For example, consider how the debugger handles the command line **next** if **next** is aliased to **s**, and **s** is aliased to **s/into**. The debugger first substitutes **s** for **next**. Then it checks the aliases again and substitutes **s/into** for **s**. Then it checks the aliases again. This time, it does not use the definition of **s** because it has already been used on this command line.

The search for alias is case sensitive. Therefore, it is possible to alias **s** to one command and **S** to another command.

The syntax for the alias command is:

ALIAS <NAME> <SUBSTITUTION STRING>

The substitution string may appear with or without quotation marks. If quotation marks are used, they are not included as part of the alias definition. Quotation marks are required if the alias definition includes a semi-colon. Without the quotation marks, the semi-colon is interpreted as a statement terminator.

Use the **/delete** modifier to delete an alias. If the alias name is specified, then that alias is deleted. If the **/all** modifier is specified after **/del**, then all of the aliases are deleted. If **/all** is not specified and no name is specified, then the user is queried regarding each alias.

The following examples show ways to use variations of the **ALIAS** command:

Variation	Description
alias	lists the currently defined aliases.
a estr	prints the current definition of the alias **estr**.
alias estr e/str	defines the alias **estr** to mean **e/str**.
alias/del/all	deletes all aliases.
alias/del estr	deletes the alias for **estr**.

The BREAKPOINT command

The **BREAKPOINT** command lists or modifies break points. It accepts a series of code points separated by commas as arguments. A code point can be specified by a function name, a line number, or a period to refer to the current address. The line number may be preceded by a string to specify a filename. Following are the acceptable modifiers for the **BREAKPOINT** command:

```
/delete, /<count>
```

The following examples show ways to use variations of the **BREAKPOINT** command:

Variation	Description
b 125	sets a break point at line 125 of the current source file.
break testFunc	sets a break point at the function `testFunc`.

Variation	Description
b main, "test.mc"127	sets break points at main and line 127 of the test.mc file.
br/del	deletes all break points. The user is queried for each break point before deletion.
br/del main	deletes the break point at `main`.
b/5	sets a break point at the current location that causes a break every fifth time the instruction at the current address is executed.

The BREAKPOINT command may contain a statement block on the same line. The statement consists of a list of MDL debugger commands. The entire list is enclosed in curly braces. The statements are separated by semi-colons. The following example shows how the statement block can be used:

```
b displayElement {if (elP->ehdr.type != 3) {eval/str "type is"; elP->ehdr.type;go}}
```

When the breakpoint occurs, the MDL debugger evaluates the conditional (elP->ehdr.type != 3). If it is true, then the debugger executes the following three debugger commands:

```
eval/str "type is"
elP->ehdr.type
go
```

Otherwise, it stops at the breakpoint.

The following breakpoint never stops. It just displays the type if it is not 3.

```
b displayElement {if (elP->ehdr.type!=3) {eval/str "type is"; elP->ehdr.type;go} else
{go}}
```

The CALLS command

The CALLS command determines the sequence of calls that caused the current function to be reached. The output of calls also reports the parameter values to each function. The stack trace also prefixes each line with a frame number. This frame number can be used as input to the SCOPE command

This command uses the following modifier:
/<count>

Variation	Description
ca/5	displays a stack trace, displaying 5 calls.

The DISPLAY command

The DISPLAY command displays data in formats other than the format implied by the type of data. The MDL debugger also accepts EVALUATE as a synonym for DISPLAY.

The DISPLAY command supports modifiers for selecting data size, for selecting output radix, and for controlling how strings are displayed.

The modifiers /char, /short, /long and /double select data size. Without one of these modifiers, the debugger determines the data size based on the expression supplied as an argument. If one of these modifiers is used, it can be combined with a repeat count in the following format:

```
/<type>:<count>
```

The modifiers /hex, /octal and /decimal specify how integer values and pointers should be displayed.

The modifier /str specifies that the debugger should display strings for fields that are of type array of characters or array of `unsigned chars`, and all fields that are of type pointer to `char` or pointer to `unsigned char`.

The modifier /wstr specifies that the debugger should display strings for fields that are strings of type `MSWideChar`. When the debugger encounters an array of type `MSWideChar` or a pointer to `MSWideChar`, it converts the `MSWideChar` strings to a multi-byte string, and then displays the multi-byte string.

Variation	Description
dis/str *tcb	display the contents of the `TCB`. Interpret as strings all fields that are arrays of `chars` or `unsigned chars`, and all fields that are pointers to `char` or `unsigned char`.
dis/long:10 *tcb	display the first ten 4-byte words of the `TCB`.
dis/str/hex chTextInfo	display the contents of the structure `chTextInfo`. Interpret as strings all fields that are arrays of `chars` or `unsigned chars`, and all fields that are pointers to `char` or `unsigned char`. Display all numerical fields in hex.

Variation	Description
e strcpy (buff, "new text")	copy the string new text into the character array buff, and then display the address of `buff`.
dis/long:20 (long *)0x40051234	display 20 longs starting at the address `0x40051234`. `(long *)` is included in the command, because the display command expects the argument to be a pointer type.

The DOWN command

The DOWN command selects the scope the debugger uses for evaluating expressions. It moves the context one step farther from the scope that contains the last execution point.

See "The UP command" on page 9-14 and "The SCOPE command" on page 9-12 for more information.

The DUMP command

The DUMP command provides a formatted dump of a block of memory, giving both the ASCII and numeric representations of the data.

Use one of the following modifiers to specify the data size: /char, /short or /long. Also, a repeat count may be specified as an modifier. Both the count of bytes to dump and the data size are remembered from the previous dump command if they are not specified.

If an address is specified, it is used as the starting address for the dump. If no address is specified, the DUMP command starts with the first address following the previous dump.

The following examples show ways to use variations of the DUMP command:

Variation	Description
dump/200 textRscP	dump 200 bytes starting at the address pointed to by `textRscP`.
dump/long &textRscP	dump 64 bytes starting at the address of `textRscP`. The numbers are interpreted as `long`s.
dump	Dump more data starting at the first address beyond the previous block, using the same data size and repeat count as the last dump command.

The GO command

The GO command tells MicroStation to resume program execution. It can also specify some temporary break points. It accepts a code point specification as arguments. This specification can be a line number, function name, or period for the current address.

The following examples show various ways to use the GO command:

Variation	Description
g	tells MicroStation to run until the next break point is encountered.
g 125	tells MicroStation to run until the next break point is encountered or until an instruction corresponding to the code at line 125 is executed.
g .	tells MicroStation to run until the next break point is encountered or until the current instruction is encountered again.

The HELP command

The HELP command displays helpful information on the MDL debugger's commands.

The command uses the following modifiers: /alias, /breakpoint, /calls, /display, /down, /go, /if, /memory, /quit, /record, /scope, /set, /step, /symbols, /type, /up and /watch.

The following examples show various commands for which HELP information can be obtained:

Variation	Description
help/br	displays information on the BREAKPOINT command.
help/calls	displays information on the CALLS commands.

The IF command

The IF command evaluates an expression and conditionally executes the debugger commands that follow the conditional clause. The statements that follow the conditional clause must be enclosed in {curly braces}. The IF command may have an else clause.

The IF command is intended primarily for use with break points. See "The BREAKPOINT command" on page 9-6 and "The IF command" on page 9-10 for examples.

The MEMORY command

The MEMORY command displays the starting addresses of all blocks of memory currently allocated by the application. The list does not contain the addresses of freed blocks.

All blocks in the list were allocated with `malloc`, `calloc` or `realloc`. The block may have been allocated by the application calling these functions directly, or by the application calling other built-in functions that call the memory allocation functions.

The following modifier can be used:
/size

If the environment variable `MS_DEBUGMDLHEAP` is defined, then MicroStation remembers the size of every block of memory allocated with `malloc`, `calloc` or `realloc` by every MDL program. Therefore, when it displays the address of each block of memory it can also display the size. Use the **/size** modifier to tell MicroStation to display the sizes.

The QUIT command

The QUIT command exits MicroStation from the debugger and unloads the application.

Under DOS the modifier **/ustation** is supported. This causes MicroStation to exit back to the command line. This modifier is intended only as a last resort. Neither MicroStation nor MDL applications are given any chance to clean up.

The RECORD command

The RECORD command causes MicroStation to start recording the debugger's input and output.

The following modifiers can be used:
/input, **/output**, **/all**, **/append**, **/replace**, **/stop**, **/default**

/input specifies that only the input from the user should be recorded. **/output** specifies that only the debugger's output should be recorded. **/all** specifies that both the input and output should be recorded.

/append specifies that the information should be appended to the recording file. **/replace** specifies that the recording file should be replaced if it currently exists.

/stop means stop recording.

/default means that current command should not start or stop record. The modifiers and file name are saved as default for the subsequent record commands.

After the modifiers, specify a file name in a quoted string.

The defaults are to append to the file mdldebug.log, recording both input and output.

The following examples show ways to use variations of the RECORD command:

Variation	Description
record/default/repl "/usr2/tmp/mdb.log"	sets the defaults so that the file /usr2/tmp/mdb.log is used. It is replaced every time the debugger starts recording.
record	start recording using the current defaults.
record/st	stop recording.

The SCOPE command

The scope command lets the programmer set the scope to any scope in the stack trace, or to any source file in the application. The syntax for the scope command is:

```
scope <new frame number>
scope .
scope <source file name>
```

To get the frame number, use the calls command. Frame number 0 always refers to the currently executing frame. As with the other debugger commands, the dot refers to the current execution point. Therefore, scope 0 and scope . are synonymous.

scope <source file name> sets the scope to a specific source file. After this command is executed, there is no function scope available. Only the file scope is available.

The UP and DOWN commands can be used to change the scope to the next or previous scope.

The SET command

The SET command controls various aspects of the debugger's behavior.

Use the SET CASE command to control whether the MDL debugger's C expression evaluation is case sensitive. **set case on** makes it case sensitive. **set case off** makes it case insensitive. By default, the debugger is not case sensitive.

Use the SET TAB command to control the tab stop intervals used by the debugger when displaying source code. By default, the debugger uses a tab stop interval of 8. The command **set tab 1** sets the tab stop interval to 1, essentially turning off tab expansion.

The STEP command

The STEP command controls debugger activation. When used with one of the following modifiers, this command specifies when the debugger will be activated:

Variation	Description
step/into	indicates that the debugger will be activated the next time an instruction corresponds to a new line of code.
st/over	indicates that the debugger will be activated the next time an instruction corresponds to a new line of code in the same function.
st/from	indicates that the debugger will be activated the next time an instruction is executed in the function that called the current function.
st/<count>	specifies the number of times the criteria must be met before the debugger is activated.

The SYMBOLS command

The SYMBOLS command lists the MDL application's symbols that can be used in debugger commands.

This command accepts a string used to restrict the list of symbols displayed as an argument. The following modifiers are used: /functions, /variables and /structures.

The following examples explain various ways the SYMBOLS command is used:

Variation	Description
symbols/f pla	displays all function names that contain `pla`.
sym/st	displays the names of structures and unions that can be used in casts.
sy/f/v av	displays the names of all functions and variables that contain the string `av`.

The TYPE command

The TYPE command types the line following the last line typed. It accepts the argument <line numbers>. A line number can be provided as a number, a number preceded by a source filename, a dot to refer to the line corresponding to the current execution address, or a function name. If no line number is specified, the line following the last line printed is specified.

The following modifier can be used:
/<count>

The following examples show ways to use variations of the TYPE command:

Variation	Description
type/20 "main.mc"1	types lines 1 to 20 of master.mc.
t/-20.	types the line containing the current execution address and the 19 lines preceding it.
t/30 dochangesingle	types the line containing the declaration of the function `doChangeSingle`, and the subsequent 29 lines.

The UP command

The UP command selects the scope the debugger uses for evaluating expressions. It moves the context 1 step closer to the scope that contains the last execution point.

See"The DOWN command" on page 9-9 and "The SCOPE command" on page 9-12 for more information.

The WATCH command

The WATCH command invokes the MDL debugger when a location in memory is modified. This command is only supported in the DOS version.

On the PC, WATCH is implemented with the hardware debug registers. A maximum of 4 watch points exists. Because of restrictions for the hardware registers, WATCH can watch only one byte rather than the entire variable specified.

The argument for a watch instruction must be an expression that evaluates to a pointer type. If the program contains the declaration `int i1;`, `&i1` is a valid argument. However, `i1` is not since it evaluates to type integer. If the program contains the declaration `int *pInteger;`, `pInteger` and `&pInteger` are valid arguments. `pInteger` watches the address `pInteger` points to. `&pInteger` watches the address of the variable `pInteger`.

The following modifier is used with this command:
/<count>

Memory Debugging

To enable memory debugging, define the environment variable `MS_DEBUGMDLHEAP`. In most cases, you will want to set it to the name of your application. If memory debugging is not enabled, MicroStation should run exactly as before. If it is enabled, the performance of MicroStation will suffer in some cases and MicroStation will use more memory.

The following debugging tools are available:

1. When memory is freed, the entire block is set to 1. This should cause catastrophic errors in the MDL application if it continues to use the memory block after freeing it.
2. Every block of memory is bracketed by special markers. Both `free` and `realloc` check these markers to verify that they are intact. If either marker is corrupted, then MicroStation prints out an error message, the first 20 bytes of the block, the last 16 bytes of the block, and the first 4 bytes after the block. It also aborts the MDL application.
3. The MicroStation command MDL HEAP <task ID> causes MicroStation to display a list of blocks that the application has allocated. If `MS_DEBUGMDLHEAP` is defined, MicroStation also examines the markers on all of the blocks and reports if any are corrupted.
4. The MDL debugger command memory also causes MicroStation to display a list of blocks that the application has allocated. If `MS_DEBUGMDLHEAP` is defined, MicroStation examines the markers on all of the blocks and reports if any are corrupted.
5. If the application specified by `MS_DEBUGMDLHEAP` calls any of the heap management functions (`malloc`, `calloc`, `realloc` or `free`), then MicroStation examines all of the blocks currently allocated by that

application. If MicroStation finds a corrupted marker on any of the blocks, then it displays the relevant data and aborts the application.

If MS_DEBUGMDLHEAP is defined, then the first tools 1 through 4 are available. If MS_DEBUGMDLHEAP specifies the task ID of an application, then 5 works for that application. That is, MicroStation scans the application's `malloc` list on every call to `malloc`, `calloc`, `realloc` and `free`. If MS_DEBUGMDLHEAP is defined as ALL, then 5 works for all applications.

MicroStation converts the value of MS_DEBUGMDLHEAP to upper case, so the case is not important. If MS_DEBUGMDLHEAP is defined, then MicroStation displays the message `MS_DEBUGMDLHEAP is defined as <value>` at startup.

Automatic Fault Reporting

When MicroStation encounters a fault in an MDL program it examines the environment variables MS_DEBUGFAULT and MS_MDLTRACE to determine how to report the fault.

Regardless of whether these environment variables are defined, MicroStation writes messages to the command window providing information on the type of error and where the error occurred.

If MS_DEBUGFAULT is defined, MicroStation tries to invoke the MDL debugger for the application that caused the fault. MicroStation tries to do this even if the application was not being debugged. MDL developers should always have MS_DEBUGFAULT enabled so they do not need to try to reproduce unexpected faults. MicroStation ignores MS_DEBUGFAULT if the fault occurs in an application that does not have any debugging information. An application has some debugging information unless it was linked with the **-go** modifier.

If MS_MDLTRACE is defined, MicroStation prints debugging information to stdout. MicroStation prints the cause of the fault and the location where the fault occurred. It also displays the name of each MDL application it encounters while unwinding the stack. For example, if an error occurs in an MDL application's dialog hook function that was called by MicroStation's Dialog Manager (the DITEMS MDL task), then MicroStation would first print a description of the error, then information on the application containing the hook function, and then information on DITEMS.

Debugger Initialization File

The first time it is invoked in a MicroStation session, the MDL debugger checks for the environment variable MS_DBGINIT. If it is defined, then the debugger interprets the definition as a file name. The debugger opens the file and reads the file, interpreting the contents as debugger commands.

10 Documentation

No software product is complete without adequate documentation. At its most primitive, documentation tells users how to install the software, what basic features are included, and how they are used. But complete documentation goes much farther, including such features as lessons and tutorials, illustrations, glossaries and on-line, context sensitive help. Internationally distributed software also offers documentation in a variety of languages.

Useful Documentation

Documentation is an extension of the user interface and is an integral and required part of each feature.

This chapter will provide you with the basic information needed to create useful and consistent documentation for your application. It does not dictate the platform or the software to be used to create documentation (within certain limitations; see below), and it assumes that you are familiar with the post-production steps necessary to see your documentation through to final-copy. The primary objectives of the design of printed and on-line documents are to make them easy to read and to make it easy to find information. As always, form should follow function.

Development Methods

Most word processors understand only two data structures, the paragraph and the character string. Many let you tag paragraphs with what are commonly referred to as **styles** or **tags**; some also let you tag character strings. Each style defines the appearance of the text it tags. The collection of styles that determine a document's overall appearance is commonly called a **style sheet** or **format catalog**.

The Help creation tools are best used with a well-structured document. A document that is logically divided into many short sections by use of chapters, headings, and subheadings produces the most easily navigable help file. Each chapter or heading title becomes the title of a help article, and the subtopics within it become the "children" of the help article.

Tools

All printed and on-line documentation for MicroStation and MDE was produced using FrameMaker and Microsoft Word, both of which are available on a variety of platforms. The MicroStation 5.0 on-line help system reads files in the compiled Intergraph Help (I/Help) format and, for compatibility with earlier releases, help resources. I/Help is a platform-independent file format, so you only have to compile help on one platform for all your releases.

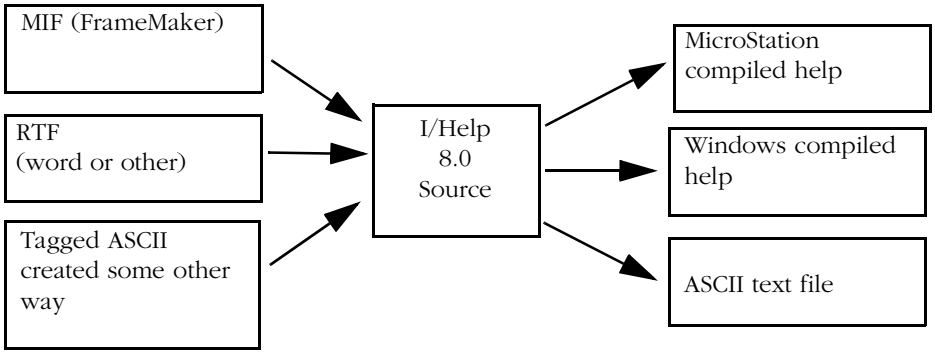


Figure 10.1 On-line help configuration.

You can construct your own I/Help source files by manually tagging an ASCII text version of your documentation, then compiling with the I/Help compiler. However, the preferred method of on-line help generation is to use the **mif2s** or **rtf2s** utilities, available upon request, to convert MIF (generated by FrameMaker) and RTF files (generated by MS Word) into I/Help source format. Templates for both FrameMaker and MS Word are available on request that will help you format your documentation with style sheets that the on-line help reader can recognize. Any product that can both import and export MIF or RTF formatted files without a loss of style-sheet definitions can be used to generate on-line help.

The available RTF template may not be compatible with WordPerfect.

You can also use your own style sheet to create your printed and on-line documentation. If you choose this option, you must edit the **mif2s.set** or **rtf2s.set** file so that the conversion utility will know how to read your style sheet.

The settings files **mif2s.set** and **rtf2s.set** define how FrameMaker and Microsoft Word styles are converted to I/Help. Because there are typically far more styles than I/Help tags, many styles can be, and typically are, mapped to one I/Help category. The settings files are text-based, and are divided into sections consisting of a category name enclosed in brackets followed by one or more style names, one per line. Style names must be contained in quotation marks ("") if the style name has any white space in it. Any line beginning with a semi-colon ';' is a comment and is ignored.

The contents of the settings files are listed below:

Setting File Group	Styles Listed Therein Are
[TopicBodyPara]	Passed as plain text. Paragraph numbers are ignored. Two carriage returns are placed at the end of each paragraph.
[ListEntryPara] (MIF only)	Considered list entries. Paragraph numbers or bullet characters are included if present. Paragraph text is indented. One carriage return is placed at the end of each paragraph.
[LastListEntryPara] (MIF only)	Treated the same as [ListEntryPara] styles, except that two carriage returns are placed at the end of each paragraph.
[IgnorePara]	Ignored.
[AttentionPara]	Treated as body text except that user-definable words are inserted before them. Note, Hint, Warning and Procedure are the supplied defaults. Syntax is: "<stylename>"<whitespace>"<prepend string>"
[AttentionChar]	Treated the same as [AttentionPara] styles, except that it allows the use of one style for all attention paragraphs, with each containing a specific character to signify the specific attention message. Syntax is: "<stylename>"<whitespace><attn character> <whitespace>"<prepend string>" Be aware that <attn character> must be specified as a single character, not as an ASCII value in decimal or hex notation.
[HelpTogglePara] (RTF only)	Used to signal the help converter to ignore any and all subsequent paragraphs until another paragraph in one of these styles is encountered. This is useful for omitting documentation from on-line help that must appear in the printed version.
[MonoPara]	Displayed in the Help window in a monospaced (non-proportional) font.
[MonoFont]	Displayed in the Help window in a monospaced (non-proportional) font.
[HyperTextFont] (RTF only)	Made into hypertext jumps if a destination can be found. A warning is issued in the log file if no destination could be found.
[ContextPara]	Used to mark an article for later retrieval by context sensitive help. Styles used for this purpose are usually conditional and/or hidden in the style sheet.
[KeyinPara]	Treated as body text except that the word Key-in: is inserted before the paragraph. Also, if header files have been specified "[HeaderFiles]"), the help conversion tools attempt to match this text to key-in commands listed in the files. Similar to [ContextPara] (above), [KeyinPara] is used to mark articles for later retrieval by help tracking.

Setting File Group	Styles Listed Therein Are
[TableTitlePara] (MIF only)	Displayed as table titles.
[TableEntryPara] (MIF only)	Flowed into table cells.
[TopicTitlePara]	Used by the conversion utilities to specify the logical depth of each type of heading paragraph. Paragraphs in these styles are converted into topic titles. The numbers indicate the logical depth of each heading type within the document, "1" being the lowest level (Chapter, for example). Syntax is: <name><whitespace><depth (1 or higher)>
[SubTopicTitlePara]	Paragraphs in these styles are NOT converted into topic titles. Their logical depths are lower than those of [TopicTitlePara]. Instead of becoming topic titles, they are displayed in bold font with a blank line before them.
[ConvertXRef] (MIF only)	Converted into hypertext jumps, provided the destination is one of the header paragraph types listed in [TopicTitlePara].
[IgnoreXRef] (MIF only)	Passed through as plain text with no warning message and no attempt to convert into hypertext.
[HeaderFiles]	This section actually lists a set of filenames, rather than paragraph styles. The files specified should be command list header files generated by the MDL resource compiler. The help conversion tools attempt to match text marked as key-ins in the MIF or RTF file with commands in the generated header files, and insert data needed for context sensitivity and tracking into the help source file.
[SpaceAfterCutoff] (RTF only)	Instead of listing one or more styles, this section contains only a single number of the range 1-72. Any paragraph in the source document that is styled with a "Space After" that is equal to or greater than this value (in points) will be followed by a blank line when converted into on-line help.

Any style encountered in a MIF or RTF file that is not listed in the corresponding settings file generates a warning message, and text marked with that style is passed as [TopicBodyPara].

The settings file groups may be in any order, organize them as you wish.

Converting to Help

Once the MIF or RTF files have been created, they need to be converted into the I/Help intermediate format. This gives all help sources a common ground from which they can be converted to any final format. MIF files are converted into I/Help **.s** files using the **mif2s** utility. RTF files are converted into **.s** files using the **rtf2s** utility.

The syntax for the **mif2s** conversion utility is:

MIF2S **[arguments]** <input file>

arguments can be any combination of the following:

flag	description
-t	Specify title of the output file. Enclose the title in quotes if it contains spaces. Default is `Help`.
-o	Specify output file. If you do not specify an output file when invoking **MIF2S**, you will be prompted for one when it runs.
-s	Specify the settings file. If not specified, **mif2s.set** is the default.
-t	Suppress output of **topics.lst**.
-n	Add line numbers to the **.s** file for debugging purposes. This argument is rarely used, but can be useful if the **.s** file is not turning out properly.
-d	Specify version, language, and logical name of the output file. Default is `01.00.00.00:english:help`. This argument is rarely used, as the information it specifies is only used by Intergraph help conversion programs.

input file is a single **.mif** file or a book file that contains a series of **.mif**s.

You can store the **mif2s** utility anywhere you like, but it should be invoked from the directory in which the **.mif** files are stored. See Example below for more information.

Example

```
C:\MIFS> \doctools\mif2s -t"Application Help" bookfile.mif -oapplhelp.s
```

Note the following points about the line above:

- The source **.mif** files are stored in **C:\MIFS**, and we were sure to invoke **mif2s** from that directory.
- The input to the utility was a bookfile, and that bookfile, as well as the FrameMaker documents to which it referred, were exported to **.mif** format prior to running **mif2s**.

- The filenames in the bookfile do not have to be changed to have **.mif** suffixes.
- The order of the arguments, such as the input and output files does not matter.

After **mif2s** has finished writing the **.s** file, it also writes out the file **topics.lst**. This file can be examined to see if all the intended topics converted correctly.

The syntax for the **rtf2s** conversion utility is:

RTF2S [arguments] [input file]

arguments can be any combination of the following:

flag	description
-t	Specify title of the output file. Enclose the title in quotes if it contains spaces. Default is `Help`.
-l	Specify name of a listfile containing the RTF filenames to process. This argument should not be specified if an input file is to be designated.
-o	Specify output file. If you do not specify an output file when invoking **RTF2S**, you will be prompted for one when it runs.
-d	Specify version, language, and logical name of the output file. Default is `01.00.00.00:english:help`. This argument is rarely used, as the information it specifies is only used by Intergraph help conversion programs.

input file is the first RTF file in the series. It should only be designated if a listfile is not specified.

Unlike the **mif2s** utility above, the **rtf2s** utility does not have to be invoked from the directory in which the RTF files are stored. This is because the listfile can contain complete paths to the source RTF files, allowing them to be stored in multiple subdirectories.

Example

```
C:> \doctools\rtf2s -t"Application Help" -llistfile.txt -
oapplhelp.s
```

Note the following points about the line above:

- The source RTF files can be stored anywhere as long as the paths to them in **listfile.txt** are correct.

- The input to the utility was an ASCII listfile that contains nothing but the fully-qualified filenames of the source RTF files. The files are listed in the order in which they are to be processed; that is, the order in which they are to appear in the final helpfile.
- The order of the arguments, such as the input and output files does not matter.

Documentation Principles

Documentation is part of the product. The only reason documentation exists is to help users use the software. Typically, if software is hard to comprehend, some users may say the documentation is unclear; others may say the software is too hard to use. In many cases, both the software and the documentation may need improvement.

Developers should be aware that creating easy-to-use software and quality documentation cannot be accomplished separately. Create a good user interface first. In general, the goal is a program that is "self-documenting."

- Procedures are as intuitive as possible.
- The program provides assistance when needed; either when the user requests it or when the software senses it is required. Assistance can come from Help or from the software itself; ideally there really is no obvious difference Ñ all components work together.
- The "assistance" never interferes with the user who already knows what to do, and shortcuts are provided for advanced users.

Bentley Systems recommends following these basic principles when creating documentation:

- On-line and printed versions should be created from a single source whenever possible.
- Organize documents to work well both in print and on-line. If there is a conflict between what is optimal for on-line versus print, the on-line version has precedence.
  Tools for converting source materials to on-line documents are designed to minimize human intervention.
- Effective illustrations should be used to clarify information that is difficult to explain in words.
- All document text should be tagged according to its purpose, with minimal hard formatting (bold, italics, and so on) directly applied.
- Keep terminology and graphic symbols consistent so that they will be clear to users of all supported systems. Where possible, BSI

terminology is consistent with the OSF/Motif and Microsoft Windows standards.

- Where possible, content that applies to several products or platforms should be re-used.
- Documents should be written and designed to make translating into other languages as easy as possible.

Conventions are explained in the MicroStation documentation in the following places:

Section	Explains
"Document Conventions" at front of each guide	Terminology and typographic conventions.
Lessons in User's Guide	Meaning of important terms and operations, such as `Reset`, `data point`, `tentative point` and `accept`.
Glossary	Defines terms.

Relevant sections of MicroStation's documentation

In many cases, applications need not re-create the sections in the MicroStation User's Guide that cover basic operating concepts. Most users should complete at least the following to learn basic MicroStation operations:

- In Introducing MicroStation, "Getting Started."
- In the User's Guide, "Fundamentals" and "Basic 2D Design and Drafting."

Once completed, the user should be able to operate most parts of your application's interface. Because you know the parts of the documentation that are most important to your application, you can:

- Supplement the material in MicroStation's documentation with more detail or detail specific to your application.
- Give alternate recommendations to those in the MicroStation documentation for users of your application or at your site.
- Tell users to ignore those parts of the MicroStation documentation that are not relevant to using your application.

Why your application should have on-line documentation

Developers should provide on-line help that can be viewed in MicroStation's Help window for the following reasons:

- It is usually neither cost-effective nor user-friendly to have your own Help system.
- Users have come to expect high-quality on-line manuals.

BSI will supply FrameMaker or Microsoft Word templates to interested third-parties and will answer questions about these templates. You can customize these templates if necessary.

The I/Help 8.0 tags used by MicroStation are covered in the on-request document "I/Help 8.0 Format."

Since each developer's document production needs are unique, contact Bentley Support at Bentley Systems if you have questions about on-line or printed document production, or require any of the aforementioned items.

Suggested Application Document Organization

Larger applications can have a separate user's and reference guide, and perhaps even developer's guide.

Applications should supply a glossary of terms that are specific to the application and not generally understood by potential users. If you follow the MicroStation conventions closely, most terms are already defined in MicroStation's glossary.

Including these appendices is useful for most applications:

- Tools listed alphabetically.
- Settings listed alphabetically: All settings used by the application and how they are set.
- Commands listed alphabetically by key-in name.

Information for context-sensitive help

For context-sensitive help, tag at least one paragraph with a `[ContextPara]` designated style. The information in this paragraph is used to generate the help topic's logical name.

All tools must have a key-in; many dialog items should. If a dialog item has a key-in, the key-in should be used to generate the help logical name since it appears in the document anyway.

MDL applications

When a feature that is an MDL application can be started from the GUI, it is documented in that place and the fact that it is an MDL application is de-emphasized.

Distributed MDL applications (.mas) should have MicroStation help .hc, Windows help .hlp and ASCII .txt files with the same filename.

Opening the Help window from MDL applications

Three key-in help commands are provided that should be useful to the MDL programmer.

HELP CONTENTS [filename]

opens the help window if necessary and displays the table of contents of the specified file, or of the main MicroStation help file if none is specified. The filename should have no extension (.hc is appended) and should have no path. The help system searches for the file in the directories specified in the environment variable MS_HELPPATH.

HELP SEARCH [filename]

opens the help window if necessary, opens the specified help file or the main help file, and opens the Search window for a search of the file.

HELP TOPIC [filename] [topic logical name]

opens the help window if necessary and attempts to display the given topic within the given file.

Dialog items that activate the help system can be constructed easily using these commands. For example, an MDL application called MYAPP might have a pull-down menu item like this:

```
{TXT_HelpOnMyAPP, NOACCEL, ON, NOMARK, NOSUBMENU, 0, 0,
            CMD_MDL_KEYIN, "USTNHELP HELP CONTENTS myapp"};
```

11 Element Descriptors

An **element descriptor** is a multiple linked list of elements that an application can easily traverse. All memory associated to the element descriptor is allocated by MicroStation.

Element Descriptors

An application should never `malloc` an element descriptor directly.

Element descriptors are an *in memory representation* of elements in the design file; they do not reside in the design file. When a complex element (for example, a cell or text node) is read into an element descriptor, the element descriptor read function sets up all of the nesting for that complex element.

MicroStation updates the header information in complex elements when the elements in the element descriptor are written back to disk. This removes the responsibility of setting words to follow (WTF), `levelMask`, etc., from the application.

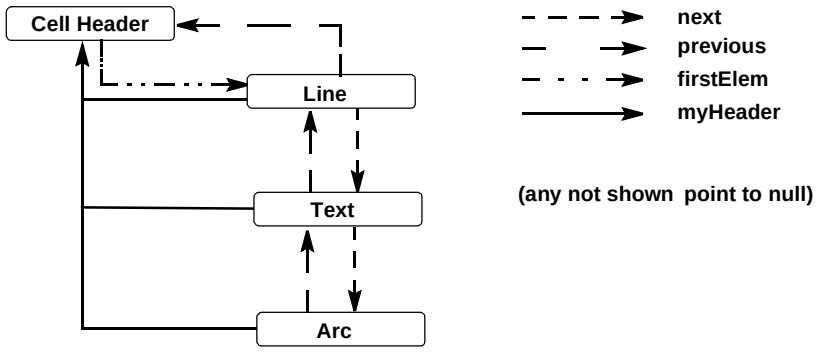
Element descriptors can represent cells and shared cells, complex chains and shapes, or a group of elements such as the selection set or fence contents.

Element descriptor header information includes:

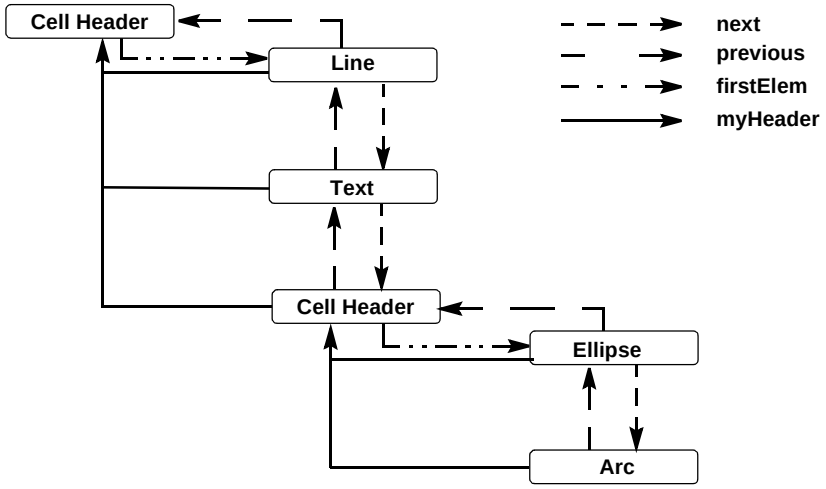
- Pointers to: next, previous, parent and child element descriptors.
- Flags to determine if the element in the element descriptor is a complex header element and, if so, if the data in that header accurately reflects the data in the rest of the element descriptor.
- Two long words for application information storage - these can the value of a pointer in your application, file position, or other application-specific information. *This is not user attribute linkage information.* User attribute linkage information is stored in the `el` portion of the element descriptor.

```
struct msElementDescr
{
   struct
   {
        MSElementDescr *next;          /* ptr to first entry in list */
        MSElementDescr *previous;      /* ptr to last entry in list */
        MSElementDescr *myHeader;      /* ptr to my hdr */
        MSElementDescr *firstElem;     /* ptr to first elem if header*/
        int            isHeader;       /* is this a complex header */
        int            isValid;        /* INTERNAL USE ONLY */
        long           userData1;      /* available for user */
        long           userData2;      /* available for user */
   } h;
   MSElement el;  /* elem data */
};
```

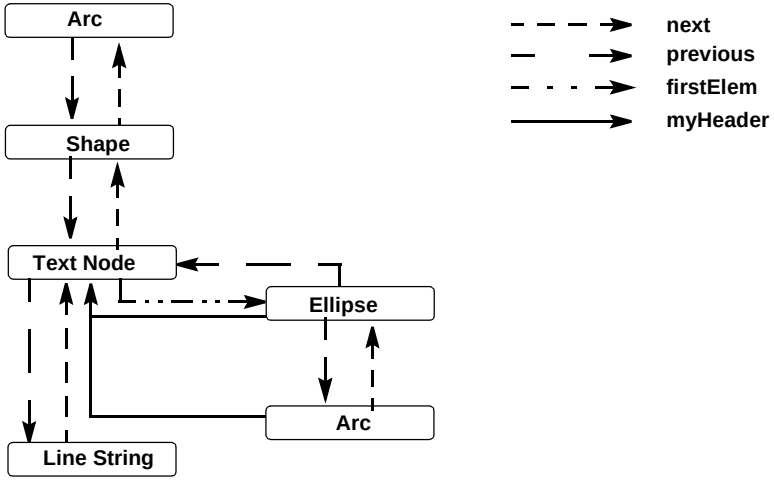
Sample element descriptors



This is an in-memory representation of a cell with three primitive elements. `myHeader` points back to the cell header from each component element.



This is an in-memory representation of a cell with two primitive elements and on nested cell.



This is an in-memory representation of a selection set.

Recursive programming

A recursive function is a function that calls itself. Element descriptors, because of their multiple links, lend themselves quite well to recursive programming.

Recursive programming and element descriptors are perfectly suited for nested complex elements because at each level of nesting the application simply calls the recursive function to do any processing.

Recursive programs can be difficult to debug because it is often difficult to determine how deeply nested the program is at any given time.

Stack usage must be carefully monitored when recursively processing element descriptors. Each invocation of the recursive function uses stack space for all of its automatic variables.

This is an example of a recursive function that modifies an element descriptor:

```
Private void modElementDescriptor
(
MSElementDescr *elmdP
)
{
   do
   {
        /* if this is a header element, call with first child element */
        if (elmdP->h.isHeader)
             modElementDescriptor(elmdP->h.firstElem);
        else
             changeElement (elmdP->el);
   } while (elmdP->h.next != NULL);
}
```

Functions that use element descriptors

Functions that use element descriptors include all `mdlElmdscr_…`, `mdlModify_…`, `mdlCell_…`, `mdlSharedCell_…`, `mdlBspline_…` and `mdlDynamic_setElmDescr`, with more to come.

Element descriptor validation

Element descriptors validate the data in complex headers, including range, class and level bit map, words in description, number of elements, number of poles and maximum length of a text element in a text node.

Validation is automatic when an element descriptor is added to the design file.

To force an element descriptor to be validated, set `elmdP->isValid` to `FALSE`.

Never set `elmdP->isValid` to `TRUE`. An application can check if the flag is `TRUE`, but should never set it to `TRUE`.

Cautions about memory use and element descriptor size

Element descriptors can use a lot of memory. Although MicroStation allocates the memory for an element descriptor, an application must tell MicroStation to free the memory allocated to an element descriptor when the application is finished with it. For debugging, periodically use the command SHOW HEAP ELMD to determine if all memory allocated to element descriptors has been freed.

Do not copy to/from element descriptors using `sizeof(MSElement)`. The el portion of the element descriptor is dynamically allocated to only hold the size of the actual element. To get the size of the `el` portion of the element descriptor, call `mdlElmdscr_igdsSize`.

12 Standard C Functions

This section lists the standard C functions that MDL currently supports. It also lists standard C functions that are replaced by similar functions.

ANSI Standard Functions

For ANSI standard functions, the function prototypes are provided in the standard header files. For example, the prototypes for `fopen` and `fread` are included in **stdio.h**, and the prototypes for `qsort` and `bsearch` are in **stdlib.h**. The following headers are provided:

"assert.h"	"ctype.h"	"errno.h"	"float.h"	"limits.h"
"math.h"	"setjmp.h"	"stdarg.h"	"stddef.h"	"stdio.h"
"stdlib.h"	"string.h"	"time.h"		

All the headers listed on p. 241 of "The C Programming Language," 2nd Edition, are included except **signal.h** and **locale.h**. The functions defined in **signal.h** and **locale.h** do not apply to MDL, and are not supported.

The functions in the header files above are not documented separately unless the functions that MDL provides differ from the functions ordinarily provided in C libraries. Exceptions in MDL are as follows:

- `stdin`, `stdout` and `stderr` are supported. Only DOS and UNIX platforms support `stdin`, and when MicroStation is in graphics mode it does not make sense to use `stdin`. In fact, once MicroStation is in graphics mode the behavior of `stdin` is undefined. `stdin` should only be used in MDL programs started as `MS_INITAPPS` applications.
- `atexit` is replaced by the user function `userSystem_unloadProgram`. This function is specified to MicroStation with `mdlSystem_setFunction`.
- `system` is replaced by `mdlExternal_...` functions.

As implemented in MicroStation PC, the `fopen` function opens files with deny-none file-sharing mode by default. To modify the sharing mode for a given MDL task, use the `mdlFile_setDefaultShare` function. If the DOS SHARE utility is installed, the sharing mode indicates the type of access that other network users can have to the file. See the `mdlFile_setDefaultShare` function documentation for more information.

MDL supports two functions for retrieving the value of an environment variable. `mdlSystem_getenv` retrieves the value of a MicroStation environment variable, and getenv retrieves the value of an operating system environment variable.

The `printf` class of functions are no longer restricted to generating strings of 600 bytes or less (as mentioned in the documentation for `mdlOutput_vprintf` in previous versions of the MDL documentation). Now there is no limit on the size of strings they can generate.

MDL supports ANSI standard format strings for the `scanf` and the `printf` classes of functions. In addition to supporting all standard format specifiers, these functions also support `%w` for working units.

A double-precision value must be supplied for the corresponding argument when the working units format is used with one of the `printf` functions. These functions generate a value in standard working units format (MU:SU:PU). Units are never included in the generated value. If the # modifier is used, the generated value is in the format specified in the Format field of the Coordinate Readout box. Based on this setting, the generated string can contain only master units; master and secondary units; or master, secondary and primary units. If the # modifier is not used, the generated value always contains master, secondary and primary units. If a minimum width is included in the format, the field is padded to that width. If padding is required, the generated value is right justified in the field unless the format specification contains an '-'. This value indicates left justification. For example, the format string `%-40w` tells the functions to generate a left justified working units specification with a minimum field width of 40 characters.

When a format specifier is supplied to a `scanf` function, the corresponding argument must point to a double-precision variable. The format of the data in the input stream or string must be `<value>:<value>:<value>`. Each value can be an integer or floating point number. Expressions are not permitted. The `scanf` functions can handle working unit specifications generated by `printf` functions using the `%w` format without a # modifier.

ANSI-Compliant Function Definition Files (.fdf)

The function prototypes for MDL-specific built-in functions (those listed in the MDL Function Reference Manual) are in **.fdf** files in the **mdl/include** directory. The names of these files correspond closely to the categories of MDL functions. For example, all of the `mdlElement_…` functions are defined in **mselemen.fdf**, and all of the `mdlOutput_…` functions are defined in **msoutput.fdf**.

While **.fdf** files can be included in both MDL source code and DLM source code, the ANSI standard header files included with MDE can be included only in MDL code. When compiling source for a DLM, use the ANSI header files provided with the native C compiler.

The MDL compiler has built-in knowledge of all of the built-in functions, so it is not necessary to use any of the **.fdf** files. However, the declarations provided in the **.fdf** files are more complete than those built into the compiler, so the compiler can do better type checking if the **.fdf** files are used.

13 MDL Built-In Variables

Several variables that are global to MicroStation are common to MDL applications. These variable names and their types are known to the MDL compiler and MDL programs may access them without declaring them. However, as a convenience, MDL programs may declare them as `extern` but may not change their type. A program can cast pointers to other data types to point to these variables.

Variables

Many MDL built-in variables are structures or unions. While the MDL compiler knows the variable's name or the variable's type name, it does not know the type definition. Therefore, it cannot determine the variable components such as member names for structures. For this reason, any built-in variable that is not a base type (such as `int`, `double` or `char`) must have its type definition included before it is referenced. If a built-in variable that is not a base type is referenced before it is defined, an incomplete definition error will result at compile time. In the following table, if the variable has a derived type, the name of the include file that contains the definition is specified.

Type	Variable	Description
`short`	`dgnBuf[]`	Current element during many MicroStation operations, that can also be accessed from User Commands.
`MSStateData`	`statedata`	Information that relates to the current state function. It is defined in **global.h**.
`Tcb`	`*tcb`	All transient information about the current design file and is defined in **tcb.h**.
`Mgds_modes`	`mgds_modes`	All mode information about the current MicroStation execution and is defined in **global.h**.
`char`	`mgdsPrompt[35]`	The prompt MicroStation writes in the prompt field, which is **uStn>** by default. Use `mdlState_setKeyinPrompt` to set the prompt for one command.
`short`	`element_drawn[8]`	A bit mask, one bit per element type. Bit `n` is set if element type `n` is a displayable element.

Type	Variable	Description
int	mdlErrno	Additional error status for various MDL functions. The possible values are defined in mdlerrs.h.
int	errno	Additional error status for various operating system functions. MicroStation does not generate the values. Instead, the native operating system generates them.
MSGraphConfig	graphConfig	A description of the graphic configuration. Defined in global.h.
long	mdlCommandNumber	The number of the most recently started MDL application command.

In addition to these built-in variables, MDL has built-in variables that should be treated as read-only. These variables are floating point constants that are frequently used.

These constants are actually in writable memory. If their values are accidentally changed, fatal errors will occur in MicroStation.

Floating point constant name	Value	Floating point constant name	Value
fc_onehalf	0.5	fc_30	30.0
fc_zero	0.0	fc_60	60.0
fc_1	1.0	fc_90	90.0
fc_m1	-1.0	fc_180	180.0
fc_2	2.0	fc_270	270.0
fc_3	3.0	fc_360	360.0
fc_4	4.0	fc_750	750.0
fc_5	5.0	fc_1000	1000.0
fc_10	10.0	fc_10000	10000.0
fc_100	100.0	fc_100000	100000.0
fc_p1	0.1	fc_2pi	2.0*π
fc_p01	0.01	fc_epsilon	0.00001
fc_p001	0.001	fc_mm_per_in	25.40
fc_p0001	0.0001	fc_360000	360000.0
fc_pi	p	fc_iang_to_rad	(π/180.0)/360000.0
fc_180overpi	180.0/π	fc_rad_to_iang	(180.0*360000.)/π
fc_piover180	π/180.0	fc_miang_to_rad	-1.0*(π/180.0)/360000.0
fc_piover2	π/2.0	fc_rmaxi4	RMAXI4

Floating point constant name	Value	Floating point constant name	Value
fc_piover3	π/3.0	fc_rmini4	RMINI4
fc_piover4	π/4.0	fc_rmaxui4	RMAXUI4
fc_piover6	π/6.0	fc_tan30	0.5773502692

14 Sample MDL Applications

MicroStation provides a number of sample applications with MDL. These examples are provided in subdirectories of the **mdl\examples** directory. This chapter briefly describes these sample applications.

Examples

The **mdl\examples\doc** directory supplements this manual. The examples in this directory illustrate how to use the built-in functions but do not add any functionality to MicroStation. You are not likely to use these examples with MicroStation. Typically, each source file in the module is self-contained. Each source file illustrates the functions for a particular class of built-in functions. For example, the functions beginning with the `mdlSystem_…` prefix are illustrated in the source file, **system.mc**.

The **mdl\examples\misc** directory contains small, complete examples. Typically, these examples perform some functionality, but are not really useful.

The other subdirectories of MDL examples contain more complex examples. Some of these examples implement extensions that MicroStation users have requested. These examples illustrate fundamental concepts instead of just illustrating the calling sequences of functions.

The example in **mdl\examples\plashape** illustrates many fundamental concepts needed for commands that create new elements.

The example in **mdl\examples\trumpet** illustrates the use of the current transformation and element descriptors.

The example in **mdl\examples\chngtxt** illustrates many fundamental concepts needed for commands that modify existing elements.

The example in **mdl\examples\rasticon** illustrates many fundamental concepts needed for applications that are started before MicroStation starts. It also illustrates how to use MicroStation windows other than the standard MicroStation views. This application is an icon editor used to create the icon menus provided with MicroStation.

The examples in mdl\examples\calculat illustrate how to implement a preprocessor for processing key-ins before MicroStation processes them. When this application is installed, full C syntax can be used for entering MicroStation commands.

The other subdirectories of the mdl\examples directory also contain examples. They also discuss fundamental concepts of MDL applications. These examples are referenced throughout this manual in the areas where the concepts are introduced.

Building sample MDL applications

The MDL development tools, including a make utility, are in the directory, mdl\bin. This directory must be in your path for the MDL utilities to execute.

The examples that have their own directories all have makefiles. To build one of these, change your current directory to the directory that contains the example. Then run BMAKE specifying the name of the application. The environment variable, MS, must specify the root directory where MicroStation is installed. MS is not a MicroStation environment variable. It is an operating system environment variable. If MicroStation is installed in the \ustation directory, enter the following sequence of commands to build the CHNGTXT application:

```
set MS=\ustation\
set BMAKE_OPT=-I\ustation\mdl\include
cd \ustation\mdl\examples\chngtxt
bmake chngtxt
```

Running MDL applications

➤ **To run MDL applications from MicroStation**

1. At the MicroStation prompt, key in MDL LOAD <application> to load the application.
2. Select an application command by keying in either MDL COMMAND <command name> or the application command.
3. Unload the MDL application by keying in MDL UNLOAD <application> (an optional step). If further application commands might be used, the application may remain loaded.

For example, to load the CHNGTXT example application, key in the following at the MicroStation prompt: MDL LOAD chngtxt

The commands in the CHNGTXT example are CHANGE TEXT SINGLE, CHANGE TEXT FENCE, CHANGE TEXT ALL and FENCE CHANGE TEXT. All of these commands prompt for the string to be replaced and the replacement string; they then edit text elements accordingly.

The PLASHAPE example has a single command, PLASHAPE. This command expects the user to enter data points to define the shape and then Reset when finished.

The RASTICON example has a dialog box interface. There are no commands.

The commands in the TRUMPET example are TRUMPET and MULTI. The application does not have a command table, so the commands must be accessed through MDL COMMAND. To run TRUMPET, key in MDL LOAD trumpet and then either MDL COMMAND MULTI or MDL COMMAND trumpet. A 3D design file is required.

The MDL Debugger can be used to step through the example applications and the user's own applications. The MS_DBGSOURCE environment variable must be defined to point to the directory containing the .mc source file. For example, to use the debugger with the CHNGTXT example, set MS_DBGSOURCE to \ustation\mdl\examples\chngtxt\.

dlogdemo MDL Example

The dlogdemo MDL example is designed to demonstrate how the various aspects of the MicroStation Dialog Box Manager work from a programming standpoint. dlogdemo demonstrates all of the standard dialog items at least once. We recommend that programmers who intend to utilize the Dialog Box Manager take the following steps to familiarize themselves with its operation:

1. Experiment with MicroStation's dialog boxes and tool palettes to get a feel for the way they work. You can (and should) refer to the documentation for details, but because a large part of the effort that "figure things out" without documentation, you may want to try learning without a manual on occasion.
2. Scan chapter 7 of MDL manual, which introduces several important concepts of the dialog box manager architecture.
3. Compile, load, and experiment with the basic, dlogdemo and rasticon examples. To compile each, change the default directory appropriately:

Basic: \ustation\mdl\examples\basic

Dlogdemo: \ustation\mdl\examples\dlogdemo

Chngtxt: \ustation\mdl\examples\chngtxt

Then issue the appropriate make command, such as bmake dlogdemo for dlogdemo. To load each application, enter MicroStation and key in MDL LOAD <application>. Note that the make programs will place the completed application (with .ma

extension) in the **\ustation\mdlapps** directory. The environment variable `MS_MDL` should point to this directory by default, so it should not be necessary to specify a path when loading the applications.

An alternate method of loading the applications is to choose MDL Applications from the User pull-down menu to open the MDL settings box. Then select the application and click the Load button.

4. Briefly examine the source files in each directory. Note that the basic application demonstrates a compact application and some important concepts involving modal and non-modal dialog boxes. The **dlogdemo** application illustrates all of the standard dialog box items. dlogdemo is not a good example of a user interface, but rather an attempt to demonstrate how all of the standard dialog manager items are implemented, and how they can interact. The files that are used to construct this example have many comments embedded in them to explain how they work. This example may in some cases act as a "programming template" for new application dialog boxes. The rasticon application illustrates the use of the dialog box manager in a more realistic fashion, including advanced concepts such as the "generic" item. It is also the application you will want to utilize when you create your own icons for your applications.
5. Re-read "Dialog Box Manager Overview" on page 15-1. Keep in mind that the best example of user interface design is MicroStation itself. Any and all of the techniques and tools used in MicroStation's user interface are available to application programs, as well as MicroStation's native dialog boxes. As you start creating hook functions, refer to the "Dialog Box Manager" chapter of the MicroStation MDL Function Reference Manual for descriptions of the built-in functions provided by the Dialog Box Manager.

chngtxt MDL Example

The **chngtxt** MDL example is designed to demonstrate how various aspects of the element location and modification programming capabilities are utilized. **chngtxt** demonstrates how to perform element selection based with user assistance (single element, selection set, and fence) and without user assistance (all elements) as well as modifying the elements which were selected. This example also uses simple dialog box operations to start commands as well as setting control variables as well as creating new commands the user can key into the Command Window.

We recommend that programmers who intend to utilize this functionality take the following steps to familiarize themselves with its operation:

1. Compile the application from the **\ustation\mdl\examples\chngtxt** directory and then run it run within MicroStation using the **MDL LOAD CHNGTXT** command. Commands can be either keyed into the Command Window or selected from the Chngtxt dialog box menu bar. The programmer has the ability to change the text associated with a text element or text node element for a single element, all elements in a selection set, all elements meeting the current fence selection criteria, or all elements within the design file.
2. Review "Dialog Box Manager Overview" on page 15-1 to become more familiar with the standard C functios provided by MDL and the Dialog Box Manager functionality available to the application program.
3. Review "Building sample MDL applications" on page 14-2, loading into MicroStation, and how each command is associated with an application function to implement that command.
4. Examine the source file associated with this application to see exactly how this application performs the operations on the design file. Refer to the MicroStation MDL Function Reference Manual for additional information about the functions used by this application.

15 Dialog Box Manager Overview

The MicroStation dialog box manager controls the operation of dialog boxes in the rectangular windows on the screen that accept and process user input. Dialog boxes contain dialog items, which are on-screen graphical entities the user manipulates to change an application's variables. The MicroStation dialog box manager implements the standard dialog box items that are common to graphical user interfaces.

Overview

MicroStation's entire graphical user interface, including over 120 dialog boxes and 2000 dialog box items, was implemented using the MicroStation dialog box manager.

Dialog boxes and dialog box items are usually defined in resource compiler ASCII source files. Changes in dialog box details can then be incorporated into an application with only a resource compile and link. In these cases, a full recompile and relink of an application's MDL source files is not necessary. In addition, items can be dynamically added, removed, set and hidden at run-time by calling dialog box manager library functions.

The MicroStation dialog box manager is "application aware." It can directly modify application state variables, using only information specified in a resource file. As a result, a large part of the interaction between the dialog box manager and client application is specified by defining data, not by writing program code. Other window systems force the programmer to write a support function for every type of user action.

Sometimes a dialog box item is too complex to be specified in a resource file. In such cases, the default behavior of an item can be changed or enhanced by attaching a hook function to that item. Using techniques described in this manual, the MicroStation dialog box programmer can write dialog and item hook functions that customize the behavior of the dialog box manager.

The next section in this chapter discusses those features of the MicroStation dialog box manager that set it apart from other graphical user interface programming systems. The remainder of the chapter explains concepts that need to be clearly understood when creating MDL applications that use the dialog box manager.

The "Standard Dialog Box Items" on page 16-1discusses all of the dialog box items supported by the dialog box manager.

The “Dialog Box Manager Hook Functions” on page 17-1 discusses the structure of dialog and item hook functions and the messages that can be sent to each.

The “A Complete Example” on page 18-1 steps you through a complete example of an MDL application that opens a dialog box.

The “Dialog Box Style Guidelines” on page 19-1 contains dialog box style guidelines, hints on debugging dialog boxes, and general techniques useful when creating dialog boxes.

“Dialog Box Manager Functions” in the MicroStation MDL Function Reference Manual describes all of the MDL library functions relating to the dialog box manager.

Features

The MicroStation dialog box manager incorporates many features that distinguish it from other graphical user interface programming systems. This section discusses these unique features and how they benefit MDL dialog box programmers.

Application aware dialog items

Regardless of the underlying programming system being used, details about the items in the dialog box must be specified when creating a dialog box. MicroStation’s dialog box manager is unique because the specification of a dialog box item can include detailed application knowledge. The dialog box manager item handlers, which implement the actual functionality of dialog box items, can use this information to modify an application variable without further effort by an MDL dialog box programmer.

For example, when a toggle button (also known as a check box or check button) is defined in a resource file, an application variable can be specified to determine whether the toggle button is on or off. When the user changes the state of the toggle button on the screen, the dialog box manager also changes the application variable. A MicroStation command, also defined as part of the item specification, can be placed in the input queue at this time. No program code needs to be written to support these actions.

Therefore, fully functional dialog boxes can be produced without any MDL programming at all, simply by defining dialog box item resources. These resources can be quickly created by copying existing item specifications and then modifying the copy to suit the new requirements.

Advanced functionality

MicroStation's dialog box items are highly comprehensive, and reduce the tasks of the programmer in a variety of ways. For example:

- Text items, which allow the user to enter text strings, can automatically ensure that the string is in range. Other systems require that a programmer write explicit range checking code for every text item that needs it.
- The color picker item makes it easy to let the user select a color by picking it from an array of choices.
- The tool palette item makes it easy to create a "floating tool palette" with tear off sub-palettes and "slam-down" items attached to individual icon tools.

Many situations require giving the end-user multiple means for accessing and modifying a particular setting. Sometimes, users are more comfortable with typing in the new value for a parameter. At other times, they want several choices when changing the setting. It also may be desirable to create a different dialog box that changes the setting-though the setting can already be changed with an existing dialog box. By using a synonym resource, the MicroStation dialog box manager can correctly change the appearance of the appropriate dialog box items no matter what dialog box they appear in. Synonym resources are discussed in detail in a later section.

Reusable dialog items

The MicroStation dialog box manager separates an item's "item list specification" (which mainly contains position information) from its "item resource specification" (which contains the "intelligence" of the item, including its application knowledge). This separation allows a single item resource specification to be reused in multiple dialog boxes and requires only the changing of the positioning information.

To use an existing dialog item in a different dialog box, only its resource type and resource ID are required. No knowledge of the contents of its item resource specification is needed. In particular, some of the items defined within MicroStation's resource files can be used by other applications. The "OK" and "Cancel" push buttons are good examples of dialog items defined in MicroStation's resource files that are reused extensively by other applications.

Flexible architecture

Usually, the more powerful dialog item handlers are, the less flexible they become. If the default behavior of an item is appropriate in a specific instance, it is not likely to be appropriate in many other situations. To counteract this problem, MicroStation's dialog box manager lets a programmer gain control at a low level and "defeat" (enhance) the

manager's default processing. Usually, the default behavior is correct because so much application-specific behavior can be specified within a dialog item. But when the need arises, the MDL dialog box programmer can create and attach dialog and item hook functions to a dialog box or dialog item to gain more control of special situations.

Typically, hook functions are required when:

- The appearance of an item cannot be determined from the state of a single application variable.
  OR
- When it is necessary for changes in the state of one item to cause complex changes in several other dialog items.

The generic item allows for the creation of a custom dialog item without requiring the construction of a dialog item handler. The MDL programmer has control over the appearance of a generic item and its response to mouse and keyboard events.

Since item handlers are separate loadable modules, changes can be shipped in small files without having to deliver an entirely new version of MicroStation. As new or updated item handlers are developed, software developers will be able to incorporate them easily for use with their applications.

No complicated input loop

An MDL dialog box application does not contain an input loop, since the MicroStation dialog box manager's internal input loop sends events directly to the appropriate item or dialog box. Hook functions, which can be attached to dialog boxes or dialog items to modify their default behavior, are therefore independent from each other and easier to write. These hook functions don't need to know about the existence or behavior of other dialog boxes or items to perform their work. They don't need to have switch statements that require information on all of the dialog boxes present in an application or all the items within a dialog box. This results in more modular functions and an easier to maintain application.

In addition, the MDL dialog box manager handles events for system controls internally. The MDL dialog box programmer need not be concerned with the management of extraneous events that can and should be handled automatically.

Fast prototyping

Since functional dialog boxes can be specified mainly in resource files, the prototype phase of program design can be completed very efficiently. For each successive design and test iteration, the resource compiler need only compile one small resource file in the entire application does not have to be recompiled or linked. Any program code can be kept to a minimum at this stage.

Because the resource syntax is virtually standard C data definitions, with the full power of the C-like pre-processor, programmers will find it easy to learn and use. The pre-processor `#define` directive is used extensively when specifying dialog item positions and sizes. This allows quick modifications to the position or size of multiple items by changing a single constant. The MDL pre-processor and the powerful basic interface components, combined with fast turnaround, results in a very effective prototyping tool.

In fact, dialog boxes can be designed by end-users unfamiliar with programming. The end-users can lay out the look and feel of the user interface for an application, while programmers simultaneously write the underlying processing code.

Portable across multiple hardware platforms

Because MDL portability is a primary design criterion for MicroStation, it is possible to convert most MDL applications to any platform that MicroStation supports with little effort. The programmer does not have to rewrite any program code or recreate any dialog box resources. The user interface portion of MicroStation and other in-house applications were developed with the same tools that are available to any application developer. These applications typically only require a recompilation and a link to run on different hardware platforms, and to retain their look and feel on different platforms.

User interface independent

The dialog box manager is interface independent. Your application can use any MDL dialog function, and the dialogs (and items in them) that are created will automatically conform to the platform-specific interface and the user preference settings. You, as a developer, need not worry if the GUI in use is Motif, Microsoft Windows, Macintosh, X-Windows, or other standard. The dialog box manager will handle the specifics for you.

Designed for internationalization

Applications can be designed with all of their text strings placed in a separate resource source file. Only this file needs to be translated to convert dialog box item labels and application messages to a different language. Program source code does not have to be supplied to translators. This protects proprietary information, while also making it easy to locate what needs to be translated.

Dialog Box Manager Basic Concepts

MicroStation windows

A **window** is a rectangular area of the screen that has a distinctive border or window frame. Windows are the visible areas of the screen where an application interfaces with the user.

Most windows have a **title bar** across the top. On the left side of the title bar is the optional window **menu button**. When a user presses this button, the pull-down **window menu** displays. The window menu button and window menu are also known as the system menu button and the system menu. The contents of the window menu are fixed and cannot be changed by an MDL programmer. On the right side of the title bar are the optional window **control buttons**. These provide a short-cut for invoking actions contained in the window menu without having to pull it down.

A user can change the size of a window with window **resize borders**.

The part of a window that is within the border or frame is known as the **content area** or **client area**.

There are two types of windows that can be used with MicroStation: **view** windows and **dialog box** windows. Several different operations can be performed on MicroStation windows, whether views or dialog boxes. See "Graphic Functions" in the MicroStation MDL Function Reference Manual for more information on window related functions.

MicroStation view windows

MicroStation view windows allow the user to display and change design files. A pointer to a view window can be obtained by calling `mdlWindow_viewWindowGet`.

MicroStation dialog box windows and dialog items

MicroStation dialog box windows are windows that can contain dialog items. A **dialog item** is a user interface control that allows the user to change application settings or invoke application commands by directly modifying on-screen graphical entities. Typically, dialog items look and act like their real world counterparts, so users can intuit how to operate them. A toggle button, for example, looks and acts like the push-in toggle switches on stereo equipment. Remember that since a dialog box is a type of MicroStation window, any operation that can be performed on a window using the `mdlWindow_...` functions can also be performed on a dialog box.

Prior to Version 5.5, view windows could not contain dialog items. However, with the release of Version 5.5, view windows now have dialog box items on the right and bottom borders.

Resources

A resource is application data that has a programmer defined format (a resource type), is stored in a file (a **resource file**), and is accessed by specifying an identifying long integer (a **resource ID**).

The resource manager is essentially a database manager used for manipulating application data. Programmers can define their own data structures and store them in a resource file. These resources can then be accessed by specifying the resource type and resource ID of the data they wish to use. The dialog box manager makes extensive use of resources and the resource manager to partition off the "user interface look" part of an application from its underlying implementation. In fact, the resource compiler, resource librarian, and resource manager were all created to handle issues brought up during the design stage of the dialog box manager.

By specifying all aspects of the look of an application in resources within a resource file, specific details of the user interface can be changed without rewriting application code. For example, customization for foreign languages involves only the shipment of a different resource file with new text strings; the base application does not have to be modified in any way.

The use of resources also speeds the design of application user interfaces. Changing dialog boxes typically involves only the recompiling of the involved resources, not a recompile and link of the entire application. This allows prototyping to proceed at a rapid pace. Much of the look and feel of an application can be developed before any of the actual implementation begins.

Also, since resources normally reside on disk, they reduce the run-time memory requirements of an application.

To gain complete portability, MicroStation uses its own resource file format that is essentially the C syntax for initializing data structures. Once the resource file that specifies a dialog box is written for one platform, it can be recompiled without change for use on a different platform.

Each occurrence of a particular resource type used by an MDL application should have its own unique resource ID. For example, in a given application, there can only be one resource whose resource type is `RTYPE_DialogBox` (defined to be `dBox`) and whose resource ID is 1.

The dialog box manager searches MicroStation's resource files whenever a related resource cannot be found in an application's open resource files. An MDL application can therefore use any of MicroStation's dialog box manager related resources (such as the dialog items present in MicroStation's dialog boxes). To allow this capability without creating resource ID conflicts with other developers, all of MicroStation's built-in resources have negative resource ID numbers. These resource IDs are defined in the header file **dlogids.h**. Other developers should use only positive resource ID numbers.

Resource ID numbers should be defined in their own .h file. This file will then be included in both an application's resource files, and in its MDL program files that need to refer to specific item IDs.

See "Resources" and "Resource Management Functions" in the MicroStation MDL Function Reference Manual for more information about the MicroStation resource manager.

Dialog item resource specifications and item list specifications

The dialog box manager design allows dialog items to be reusable. To accomplish that goal, a dialog box item is composed of two separate pieces: a dialog item resource specification and a dialog item list specification.

A **dialog item resource** specification contains the complete behavioral description of an item and is uniquely identified by the combination of its resource type and resource ID. Other item information includes:

- The text string label associated with the item.
- Numbers used to find a description of the item in MicroStation's help system.
- A string that indicates what application variable to control.

The item's location within a particular dialog box is not specified. Therefore, the same item can be used and placed in different dialog boxes by referencing its resource type and resource ID. It will look the same and affect application variables identically in each instance.

There is one item list per dialog, and each item within a dialog box has an entry in that list - a **dialog item list specification**. A dialog item list specification not only specifies the resource type and resource ID of the item to include in the dialog box, but also its position and size. Most other user interface toolkits pack positioning information with other item information in their item lists, thus preventing the item's reuse in a different dialog box. See "DialogItemRsc Structure" on page 16-6 for more information on dialog item lists.

Sometimes most of an item's resource specification is usable in another situation, except for a small portion, such as the item's label or the string that specifies which application variable to control. To handle those situations, the dialog box manager allows for overriding of certain item resource specification fields with information specified in the item list. Currently it is possible to override an item's label or access string with item list specifications. For example, the standard MicroStation toggle button item that controls the TCB grid lock variable could be included in a dialog box and labeled "Lock cursor to grid" instead of "Grid Lock."

The file **dlogids.h** contains the resource IDs of all of the dialog items that are used in MicroStation. Most of these items can be used in a dialog box simply by including the proper resource type and the listed resource ID in the dialog's item list. The programmer does not have to specify the details of the item's resource specification. MicroStation's predefined resource for that item will automatically be used.

Dialog item state: internal value versus external state

The state of a dialog box item is composed of two parts: the item's internal value and the item's external state.

An item's **internal value** is the simple C variable, hidden within the item's private data structures, which is referenced whenever the item needs to be displayed. Currently all items, except the text item, keep their values in signed long integers. The text item keeps its internal value as a `NULL` terminated string. For example, a toggle button is implemented with a hidden variable that can be either 0 or 1, indicating whether the toggle button is currently depressed or not. MDL dialog box manager functions exist to get or set an item's internal value.

An item's **external state** is the value of the application data that the item controls. For example, suppose a toggle button controls the TCB variable for grid lock, `tcb->control.grid_lock`. In this case, the item's external state matches the value of `tcb->control.grid_lock`. The application data that determines an item's external state is not necessarily a simple C variable. With hook functions, which are discussed later, it is possible to consider multiple pieces of data as the item's state. There are MDL dialog box functions to get or set an item's external state.

Since items are used to graphically depict the state of application data in an intuitive way, an item's internal (and therefore displayed) value usually matches its external state. When an item is initially created, the item's external state is retrieved and the internal state is set accordingly. The item's initial appearance thus matches the state of the application data being controlled. Whenever the user interacts with an item, changing its appearance and therefore its internal value, the item's external state is also updated to reflect the change.

An item's internal value and external state thus represent two different parts of an item's state. The internal value represents the state of the item's appearance on the screen. The external state represents what the application thinks the item's state should be. In "Modeless and modal dialog boxes" on page 15-10 and "Item synchronization" on page 15-12, it is important to understand this distinction.

Referencing Application Variables from Resource Files

Among the information that can be included in an item resource specification is a string that determines which application variable the item controls.

The application variable underlying a particular item is specified in the `accessStr` field of the item resource specification. `accessStr` is a `NULL` terminated string containing a C expression that, when evaluated at run-time, results in a variable reference. See "Generating Resource Files from C Type Definitions" on page 7-17 for a general discussion of MDL's special facility for evaluating C expression strings at run-time. This facility allows the specification in a resource file of how that item affects application data.

For a C expression to be evaluated at run-time, any variables that are referenced must be published with one of the `mdlDialog_publish…` functions. (The lower level `mdlCExpression_symbolPublish` function can also be called, but the `mdlDialog_publish…` functions are easier to use.) In addition, if any variable is a structure or a pointer to a structure, the `mdlDialog_publish…` functions require that **type declaration resources** exist for that structure in an open resource file. Type declaration resources contain detailed information about a structure that allow it to be analyzed at run-time. The **rsctype** utility program converts standard C structure declarations that are `#included` in a type definition file, commonly known as a **.mt** file, into type declaration resources. See "Generating Resource Files from C Type Definitions" on page 7-17 for more information on the **rsctype** utility and generating type declaration resources.

An **underlying application variable** is the application defined variable that a dialog item directly controls and that determines an item's external state. By using C expression strings in an item resource specification, the underlying application variable can be automatically modified by the dialog box manager whenever the internal value of an item changes (whether from user interaction with the item or from calls to MDL functions). Usually, no program code is necessary to have the item's appearance match the state of an item's underlying application variable.

Modeless and modal dialog boxes

Modeless dialog boxes allow the user to interact with other dialog boxes. Operations such as drawing, modification of items in other dialogs, and keying in MicroStation commands can occur while a modeless dialog is displayed. MicroStation's Command window and the Element Attributes dialog box are examples of modeless dialog boxes.

While a **modal dialog box** is displayed, all user interactions must be focused within that single dialog. The dialog box manager will ignore mouse presses that occur while the cursor is outside a modal dialog box. Modal dialog boxes do not have window menu buttons on the left side of the title bar or window control buttons on the right side. MicroStation's Preferences dialog box is an example of a modal dialog box. Modal

dialog boxes are used when an application must have some information before it can continue, or a complete set of information must be entered before other actions can be taken.

Push button items are used to dismiss modal dialog boxes or possibly to launch another modal dialog box for more information. When dismissing a modal dialog box, users can accept the changes they have made to items within a modal dialog box by pressing a push button that is usually labeled "OK." Alternatively, users can discard any changes they have made to the items in a modal dialog box by pressing a push button that is usually labeled "Cancel." All modal dialog boxes should include one or both of the push buttons whose resource IDs are `PUSHBUTTONID_OK` or `PUSHBUTTONID_Cancel`. These predefined buttons automatically take care of all the details needed to dismiss a modal dialog box.

To implement the behavior associated with the Cancel and OK push buttons, the items in a modal dialog have their internal values decoupled from their external states. Changing the on-screen appearance of an item in a modal dialog box does not usually change the application variable that the item controls. No action is necessary to discard the changes to items within a modal dialog box because the application variables associated with the items have not been modified. However, when the user accepts the changes in the appearance of the items in a modal dialog box by clicking the OK push button, the dialog box manager must force the external state of each item to match the item's internal value.

Modal dialog boxes can be invoked in two different ways, by using the standard `mdlDialog_open` function or by using `mdlDialog_openModal`. The `mdlDialog_openModal` function does not return control to its caller until the user has dismissed the dialog box. The first argument of the `mdlDialog_openModal` function indicates which button the user pressed when dismissing the dialog box. An application should call `mdlDialog_openModal` when it needs more information from the user before it can continue processing.

A modal dialog box opened with `mdlDialog_open` must be handled differently. The `mdlDialog_open` function returns immediately to its callerÑit does not wait until the user has dismissed the dialog box. To determine which type of button was pressed to dismiss the modal dialog box, a dialog hook function must be attached to the dialog box. The `DIALOG_MESSAGE_DESTROY` message is sent when the dialog is dismissed, and the `actionType` field of that message indicates which button was pressed. An application should call `mdlDialog_open` when a complete set of information must be entered before other actions can be invoked, but the information is not needed immediately.

A modal dialog box should have the `DIALOGATTR_MODAL` bit set in the attributes field of its dialog box resource specification. See "DialogBoxRsc Structure" on page 16-2 for more information on the attributes field.

Item synchronization

An item's state has two parts as discussed in "Dialog item state: internal value versus external state." An item is **synchronized** when the item's internal value (and appearance) is forced to match the item's external state. In other words, after synchronization, the item's appearance on the screen is guaranteed to match the application data the item represents.

Synonym resources

In certain situations, a change in the external state of one item requires the simultaneous change in the appearance (and consequently the internal state) of other items. Synonym resources can perform this simultaneous change.

Synonym resources are simply a list of items, where each item is specified by its resource type and resource ID. Some items (such as toggle buttons, text and option buttons) can have a synonym resource ID contained in their resource item specifications. Whenever one of those item's external state changes, a synchronize message is sent to all the items contained in its associated synonym resource. This forces the synonym resource item's appearances to match the state of the application variable they control.

For example, MicroStation's Element Attributes dialog box has two dialog items in it, both of which give a different view of the active element color (both are "synonyms" for the active element color). These are the text and color picker items next to the "Color:" label. Whenever either one changes the active element color, the other item must reflect that change. Both items are associated with the following synonym resource:

```
DItem_SynonymsRsc SYNONYMID_ElementColor=
{
   {
        {Text, TEXTID_ElementColor},
        {ColorPicker, COLORPICKERID_ElementColor},
   }
};
```

Therefore, when the text item is used to change the active element color, a synchronize message is sent to the color picker item. The color picker item's appearance will be forced to match the application variable the item controlsÑin this case, the active element color. The result is that when the text item changes the active element color, the color picker's appearance will also automatically be changed. A similar sequence of events will occur if the color picker is used to change the active element color.

Remember that when a dialog box is modal, its items' states remain unchanged until the user selects the "OK" push button. If synonym resource lists need to be used within a modal dialog box, the `DIALOGATTR_ALWAYSSETSTATE` bit must be set in the attributes

field of the dialog box resource. Here, special handling must be performed if the normal "Cancel" push button behavior (to not modify any the dialog box's underlying application variables) is needed. The values of the application variables affected by the dialog box must be stored somewhere when the dialog box is first opened, and then restored if the user activates the "Cancel" push button. These actions can be performed by a dialog hook function that handles the `DIALOG_MESSAGE_CREATE` and `DIALOG_MESSAGE_DESTROY` messages.

Keyboard focus

The dialog item that is currently processing user keystrokes is called the **keyboard focus** item or **input focus** item. The current keyboard focus item is visibly distinguishable in some way. For example, a text item with the keyboard focus contains the text cursor and a scroll bar of a list box item with the keyboard focus is darkened. When a dialog item becomes the item that receives user keystrokes, it is said to gain the focus. When the focus moves to another item, the first focus item is said to lose the focus. Not all dialog items can process user keystrokes. These items are said to be unable to accept the keyboard focus. An item that can accept the keyboard focus is said to be **focusable**.

A focusable dialog item can validate its contents before it gives up the keyboard focus. If the user has typed in an invalid string, the item can refuse to lose the focus. If the contents are invalid, a focusable item will typically beep and reset its contents to the previous valid value.

The dialog box that contains the keyboard focus item is called the **keyboard focus dialog box**.

MicroStation will usually move the keyboard focus to the MicroStation Command Window when the user clicks in a window that doesn't contain any focusable items (such as view windows). This action is called **keyboard focus auto-switching**. Keyboard focus auto-switching prevents the inadvertent modification of dialog items when the user is switching between modifying dialog items, performing drawing actions, and keying-in application commands. Dialog boxes can optionally turn off auto-switching if automatically moving the keyboard focus to the command window is not desired by setting the `DIALOGATTR_NOAUTOSWITCH`.

Sinking dialog boxes

Most modeless dialog boxes should have the `DIALOGATTR_SINKABLE` bit set in the attributes field of their dialog box resource specifications. See "DialogBoxRsc Structure" on page 16-2 for more information on the attributes field. A sinkable dialog box can be sent behind view windows by the user, and has a sink icon in the right side of its title bar. When a dialog box is "sunk," any overlapping view windows will obscure the dialog box.

Normally, modeless dialog boxes "float" in front of all view windows. It is usually impossible to bring a view window in front of a dialog box, and view windows never obscure dialog boxes. MicroStation is set up this way so the user can always see MicroStation's current settings. Dialog boxes, however, can be large and obscure too much of the view windows into the design file. In these cases, it is convenient to allow the user to sink a dialog box. The dialog box is not closed, and usually the layout of the windows can be arranged so the important parts of the dialog box are still visible. If the user needs to see the full dialog box again, clicking on any visible part of the dialog box border will "unsink" it and bring it to the front of all the windows.

The only modeless dialog boxes that should not be sinkable are those that contain tool palettes. Tool palettes should always float above the view windows.

Specifying coordinates

Positions of items within dialog boxes are specified in local coordinates. The local coordinate system has its origin in the upper left corner of the content area of a dialog box. Y values increase as one moves toward the bottom of a dialog box.

Two different units specify the locations and sizes of dialog boxes and dialog items: **pixel units** and **dialog coordinate units**. Pixel units correspond to actual screen pixels. A dialog coordinate unit is 1/12 of a dialog box's current font height.

Locations and sizes in resource files are always specified in dialog coordinate units, not pixels. By using dialog coordinates, items will be correctly placed and sized when the user changes the size of the dialog font, or moves the dialog to a screen that has a different font size associated with it. Each screen can have a different font size to handle different screen resolutions. Otherwise, a dialog box that looks fine on a 640 x 480 pixels screen will look too small on a 1280 x 1024 screen. In this case, the user will probably desire a larger font on the second screen.

To make specifying coordinates in terms of characters easier, three constants are defined in **dlogbox.h**:

```
#define DCOORD_RESOLUTION 12
#define XC   (DCOORD_RESOLUTION/2)
#define YC   DCOORD_RESOLUTION
```

`XC` can be thought of as the approximate width of one character, and `YC` as one character height. The formulas for computing dialog dimensions are as follows:

```
Dialog width = charWidth * XC * FontPixelHeight/DCOORD_RESOLUTION
Dialog height = charHeight * YC * FontPixelHeight/
DCOORD_RESOLUTION
```

For example, suppose that screen 0 has a dialog font that is 10 pixels high, and screen 1 has a dialog font that is 18 pixels high. In a resource file, a dialog is defined to have

a width of `20*XC` and a height of `10*YC`. In other words, the dialog will be about 20 "characters" wide and about 10 "characters" tall. On screen 0 the dialog will be 100 pixels wide (20 * 12/2 * 10/12 = 100) and 100 pixels tall (10 * 12 * 10/12 = 100). On screen 1, with the larger dialog font, the dialog will be 120 pixels wide (20 * 12/2 * 18/12 = 120) and 180 pixels tall (10 * 12 * 18/12 = 180). Notice that the pixel width and height of the dialog is the same though different numbers of characters were specified. This is because the definition of `XC` assumes that a character is half as wide as it is tall.

The `GENY` macro, defined in **dlogbox.h**, can be used to convert row numbers to y dialog coordinates. This is useful if there are arranged dialog items. For example, specifying `GENY(1)` for the y coordinate of one item, and then `GENY(2)` for a second item will properly vertically position the two items, with an appropriate gap between them. The `GENY` macro is an example of using a C macro to aid in dialog box construction. Other macros can be created based on `GENY` if a different spacing scheme is needed.

If the dialog font is 12 pixels high, there is a one-to-one correspondence between dialog coordinates and pixels.

When using dialog box manager calls to perform drawing operations, positions and sizes are usually specified in pixels (not the dialog coordinates used in resource files).

All references to a point use the `Point2d` structure. All references to a rectangle use the `BSIRect` structure. Note that these structures use long ints to specify positions. The `sextent` structure by contrast uses shorts. It uses shorts to save space in resource files where it is used to specify the location of dialog items. Be careful not to do the following:

```
rect.origin = sextent.origin;
```

This must be done instead:

```
rect.origin.x = sextent.origin.x;
rect.origin.y = sextent.origin.y;
```

These data structures are defined in **basetypes.h**.

Global coordinates are used to indicate the actual position of a pixel on the screen. The origin of the global coordinate system is the upper-left corner of the screen. Y values increase as one moves toward the bottom of the screen. A screen number must always be specified when drawing directly to the screen.

The `BSIRect` structure was previously called `Rectangle`, however, the rectangle name conflicts with a structure name in Windows NT, so for MDL we renamed `Rectangle` to `BSIRect`.

Text font

All dialog boxes have a current font associated with them. Whenever the width of a character or a string is determined or text is drawn, the dialog box manager uses the current font.

Fonts are specified by a font index. For each screen, MicroStation stores several different font specifications for predefined purposes. For dialog boxes, only two fonts are of interest: those specified by the indexes `FONT_INDEX_DIALOG` and `FONT_INDEX_BOLD`. `FONT_INDEX_DIALOG` is the font index of the standard dialog font and `FONT_INDEX_BOLD` is the font index of the standard bold dialog font.

Color

Several dialog box manager built-in functions allow the specification of a `colorIndex`. This is not a direct reference to the MicroStation color table. Instead, it is an index into an array that locates the position within the MicroStation color table of 12 standard colors. See **msdefs.h** for a list of all the predefined colors.

Most dialog box programmers will not directly draw into a dialog box. If they do, they will mainly be using `WHITE_INDEX`, `LGREY_INDEX`, `DGREY_INDEX` and `BLACK_INDEX`.

The `mdlWindow_lineStyleSet` function allows the specification of a direct index into the current design file's color table. It should only be used in very special circumstances (like color picker dialogs). See "Graphic Functions" in the MicroStation MDL Function Reference Manual for more information on drawing with color.

The dialog box manager's internal architecture

Three major sub-systems within MicroStation handle dialog boxes: the dialog box manager functions, the item handler functions, and MDL programmer-supplied hook functions.

The sub-systems communicate by sending various messages. A **message** is a data structure that has a header that indicates the type of message and a union of additional information that depends on the message type. A function that is used to receive messages gets one and only one argument passed to it: a pointer to a message structure. It replies by setting various fields in the message itself.

All user interaction and most programmer interaction with a dialog box first goes through a dialog box manager routine. The dialog box manager routines control the general flow of user interaction with the dialogs. These routines know the basic information about a dialog box such as its position and size, and the position, size, and type of items within the dialog box. In particular, these routines have access to an array of `DialogItem` that is contained in memory for each dialog box. See "DialogItemRsc Structure" on page 16-6 for more information. The dialog box manager routines have

no knowledge about the details of specific dialog item types. That knowledge is contained in the dialog item handlers.

A **dialog item handler** is a loadable MDL module that contains all the default functionality associated with a type of dialog item. Every dialog item has a resource type associated with it. This type is used to specify a structure that is initialized in a resource file-the item resource specification. It also determines which item handler will be sent messages when the dialog box manager routines need to manipulate the item.

The item handler for a dialog item type knows how to create, destroy, draw, move, resize, get the value of, and set the value of that type of item. If applicable, it also knows how to respond to mouse button presses or releases inside the item, or how to accept keystrokes. If the item type can automatically control an application variable, the item handler knows how to get and set the item's external state.

A **hook function** is an MDL programmer-written function that can be used to modify the default behavior of the dialog box manager functions and the item handler functions. A hook function can be attached to a dialog box, in which case it is a **dialog hook function**, or to an individual dialog item, in which case it is an **item hook function**. Item hook functions are generally used to implement inter-item communications, where changing the state of one item requires the simultaneous complex change of other items. Keep in mind that this situation can sometimes also be handled by the judicious use of synonym resources. Dialog hook functions are usually used to allocate and initialize data structures that will be used throughout the life of a dialog box by certain dialog items.

Generally, some user action will cause the dialog box manager to send a message to the appropriate item handler with an indication of which specific item is to receive the message. The item handler can then optionally pass the message on to an MDL programmer-supplied item hook function to see if the hook function wants to handle that type of message. If it does, the item handler will typically just return without further action. If the item hook function does not want to handle that type of message or if there isn't an item hook function attached to the item, then the item handler will perform its default processing for that type of message. The item handler or item hook function can set various fields in a message to return information to the function that sent the message.

Each item handler can decide whether and when to pass messages on to an attached item hook function. Therefore, it is important to carefully read the item type description of an item to determine which messages will be sent to a particular item hook function. See "Item Hook Functions" on page 17-28 for more information.

In addition to an item hook function, which is attached to a specific item and is only sent messages relevant to that item, a dialog hook function can be attached to the entire dialog box. Dialog hook functions are sent messages whenever a user interaction occurs anywhere in the dialog box to which it is attached. The dialog box manager routines directly send messages to dialog hook functions. They are not sent

first to item handler functions. Whether a message is sent to a dialog hook function does not depend on the item for which the message is ultimately destined. Instead, the messages the MDL programmer has said the dialog hook function is interested in determine whether a message is sent. After that the hook function will only be sent those types of messages.

For example, suppose that the user moves a dialog box on the screen. The dialog box has one toggle button item inside it. There is a dialog hook function, which has indicated an interest in update messages, attached to the dialog box and an item hook function attached to the toggle button. The following sequence of events occurs:

1. The window manager redraws the dialog box border at the new position. It sends a message to the dialog box manager that it should draw the contents of the window.
2. The dialog box manager sends a "draw" message to the item handler associated with each item in the dialog box. Here a "draw" message is sent to the toggle button item handler. This message contains a pointer to the specific item to draw.
3. The toggle button item handler draws a toggle button using information contained in the pointer to the specific toggle button. It does not bother to pass along the "draw" message to the item hook function that is attached to the toggle button. Since a toggle button's item hook function could not perform any useful action upon receipt of a "draw" message, it is best that it not be sent in the first place. The drawing of a toggle button is performed by the toggle button item handler; it is not necessary or desirable that a hook function be allowed to modify this behavior.
4. The dialog box manager next sends an "update" message to the dialog hook function. This could be used to draw anything into the dialog box.

Since the dialog box manager routines have no built-in knowledge about specific items (all it knows is the type of items), it is very simple to add new item types by publishing new item handlers. No changes need to be made to any of the dialog box manager routines themselves to support new item types.

Hook function IDs

When referencing hook functions within a resource file, a long integer ID is specified. To avoid conflicts with other developers, MicroStation uses negative numbers for its hook function IDs. Other developers should only use positive numbers for their hook function IDs. Note that dialog hook function IDs and item hook function IDs use the same **number space**. Number space is akin to name space or the concept of *scope*, in that it is an area reserved for references, and no members within that area can be

identical. In other words, the number 1 cannot be used for both a dialog hook function ID and an item hook function ID.

Hook function ID numbers should be defined in the same file in which resource IDs (such as item resource ID numbers and dialog box resource ID numbers) are defined. This file will then be included in both an application's resource files and in the MDL program file that contains the `main` function.

To make an application's hook functions visible to the dialog box manager, the hook function ID numbers and hook function addresses need to be published with a call to the `mdlDialog_hookPublish` function. Once this function is called, the dialog box manager can determine the address of the appropriate hook function when given a hook function ID number.

Dialog Box Manager Header Files

The header file **dlogbox.h** contains the declaration of the dialog box manager resource structures. It must be included in all resource files that contain dialog box manager resource definitions.

The header file **dlogitem.h** contains structures that hook functions need to refer to. In particular, it contains the declarations for dialog messages and dialog item messages. **dlogitem.h** must be included in all MDL program files that contain hook functions. **dlogitem.h** automatically includes **dlogbox.h**.

The header file **dlogman.fdf** contains function prototypes for all the dialog box manager functions. When this file is included, the MDL compiler can find dialog box manager function argument errors. This file is also useful as a quick reference to the arguments to all of the dialog box manager functions. For each dialog box manager function, the function arguments are listed with a description of the argument, and an indication of whether the argument is an input to or an output from the function.

The header file **dlogids.h** contains the IDs of all MicroStation resources. If an application wishes to use one of these resources it should include **dlogids.h**.

The Dialog Box Manager Library

The file **ditemlib.ml** contains all of the item-type-specific dialog box manager functions. All MDL programs that use the dialog box manager routines need to link with **ditemlib.ml**.

Creating A Dialog Box

➤ **To create a dialog box**

1. Create a dialog box resource instance. This contains the title of the dialog box, its attributes, size, help reference ID and a list of items. See "DialogBoxRsc Structure" on page 16-2 for more information.
2. Create item resource instances for the items that are used in the dialog box. An item resource instance specifies the item's label, the application variable it controls, its item hook function, and other information. See "Standard Dialog Box Items" on page 16-1 for more information.
3. If the item resource instances refer to application variables that are structures or pointers to structures, type definitions must be created for those variables. Place a `publishStructures` statement in the application's **.mt** type definition file for each referenced structure. Generate the type resource file by running the **rsctype** utility on the application's type definition file. See "Referencing application variables from resource files" and "C expression handling" in the MicroStation MDL Function Reference Manual and "Generating Resource Files from C Type Definitions" on page 7-17 for more information.
4. Compile the application's resource file. See"Compiling an Application Command Table" on page 7-11 for more information.
5. If item or dialog hook functions have been used in the dialog box, these functions must be added to the MDL application. The dialog box manager also needs to be informed of the IDs and addresses of these hook functions when the application runs. To do this, call `mdlDialog_hookPublish` from the application's main function. See "Hook Functions" on page 17-1 and "Hook function IDs" on page 15-18 for more information.
6. The resource file that contains the dialog box must be opened when the application runs. To do this, place a call to `mdlResource_openFile` inside the application's `main` function. See "Resource Management Functions" in the MicroStation MDL Function Reference Manual for more information.
7. If application variables have been referenced from the dialog box's item resource instances, then those variables need to be published in the main function of the MDL application. To do this, call the appropriate `mdlDialog_publish...` functions. See "Referencing Application Variables from Resource Files" on page 15-10 for more information.
8. Compile and link the MDL application if there are item or dialog hook functions, or resource file referenced variables. Be sure to link with

the dialog box manager library **ditemlib.ml**. See "Linking MDL Applications" on page 7-24 for more information.

9. (Optional) Merge the application's resource files with the MDL program file to create an application file. See "Resource Librarian" on page 7-27 for more information on combining resource files.
10. Load the MDL application by keying in: **MDL LOAD <applicationName>**.
11. Display the dialog by keying in: **DIALOG <dialogId applicationName>**. Application variables are automatically modified and hook functions called as the user interacts with the dialog. Alternatively, `mdlDialog_open` can be called from the application's `main` function or from a function that is attached to an application's command table.

Remember that the use of the MDL **bmake** utility can automate steps 3, 4, 8 and 9.

Dialog Box Manager Sample Programs

See the **basic** application, in the **mdl\examples\basic** directory, for an example of an MDL application that opens a simple dialog box. The source for this example can also be found in "A Complete Example" on page 18-1.

See the **dlogdemo** application, in the **mdl\examples\dlogdemo** directory, for a more involved MDL application that shows the use of all of the standard dialog box items. The **dlogdemo** application with the **basic** application are meant to be used as templates when creating new applications that make use of the dialog box manager.

See the **iconedit** application, in the **mdl\examples\rasticon** directory, for examples of using dialog box manager routines and the generic item.

Recommended Reading

The following reading suggestions will help you design a successful user interface for your MDL application:

OSF/Motif Style Guide

OSF/Motif Style Guide, Open Software Foundation, Englewood Cliffs, NJ: Prentice Hall, 1990.

OSF/Motif UserÕs Guide, Open Software Foundation, Englewood Cliffs, NJ:Prentice Hall, 1990.

IBM Systems Application Architecture, Common User Access

Systems Application Architecture, Common User Access: Advanced Interface Design Guide, IBM Corporation, 1989.

Microsoft Guidelines

Microsoft Windows Software Development Kit, Guide to Programming, Version 3.0, Microsoft Corporation, 1990.

The GUI Guide, International Terminology For The Windows® Interface, Microsoft Corporation, 1993.

Apple Computer User Interface Guidelines

Macintosh Human Interface Guidelines, Apple Computer Inc., Reading, MA: Addison-Wesley, 1993.

Making it Macintosh, Macintosh Human Interface Guidelines Companion, CD-ROM, Apple Computer Inc., Reading, MA: Addison-Wesley, 1993.

Inside Macintosh Series, Apple Computer Inc., Reading, MA: Addison-Wesley, 1992-1993. Specific volumes concerning user interface design include:

- Inside Macintosh: Overview
- Inside Macintosh: Macintosh Toolbox Essentials

General Interface Guidelines

The Art of Human-Computer Interface Design, Edited by Brenda Laurel, Addison-Wesley, 1990. Tog on Interface, Bruce Tognazzini, Addison-Wesley, 1991.

16 Standard Dialog Box Items

This chapter discusses the standard dialog box items that are supported by the MicroStation dialog box manager.

Structured Items

In addition, the structures used to define dialog boxes and dialog box item lists within resource files are explained in detail. This is followed by a description of those structure members that are common to multiple dialog item resource specifications.

Each dialog box item description includes: instructions for creating an item list specification, instructions for creating an item resource specification, a list of the messages that will be sent to item hook functions, and a list of relevant dialog box manager functions.

The following standard dialog items are discussed in this chapter:

- Label Item (page 16-9)
- Group Box Item (page 16-11)
- Separator Item (page 16-12)
- Toggle Button Item (page 16-13)
- Push Button Item (page 16-16)
- Option Button Item (page 16-21)
- Scroll Bar Item (page 16-26)
- Text Item (page 16-28)
- Multi-line Text Item (page 16-33)
- Color Picker Item (page 16-38)
- Level Map Item (page 16-41)
- Menu Bar Item (page 16-43)
- Text Pull-down Menu (page 16-47)
- Option Pull-down Menu (page 16-52)

- **Color Picker Item** (page 16-38)
- **Tool Palettes** (page 16-57)
- **Icon command frame** (page 16-58)
- **Icon command palette** (page 16-60)
- **Icon command** (page 16-62)
- **List Box Item** (page 16-77)
- **Generic Item** (page 16-90)
- **Button Group Items** (page 16-93)
- **Sash Item** (page 16-98)
- **Scale Item** (page 16-100)
- **Popup Menu Item** (page 16-104)
- **Radio Button Item** (page 16-106)
- Combo Box Item

DialogBoxRsc Structure

The `DialogBoxRsc` data structure (defined in **dlogbox.h**) defines a dialog box in a resource file. Its header section contains information for the entire dialog (like its width and height), and then a list of items.

```
typedef struct dialogboxrsc
{
   ULong      attributes;          /* dialog attributes */
   int        width;               /* dialog coords */
   int        height;              /* dialog coords */
   ULong      helpInfo;            /* help for entire dialog */
   char       *helpSource;         /* help task ID */
   long       dialogHookId;        /* dialog hook ID */
   long       parentDialogId;      /* to inform when destroyed */

#if defined (resource)
   char       label[];             /* dialog label (title) */
   DialogItemRsc itemList[];       /* array of dialog items */
#else
   long       labelLength;         /* length of label (title) */
   char       label[1];            /* dialog label (title) */
#endif

} DialogBoxRsc;
```

Field	Description
attributes	Specifies the attributes of the dialog box. It is constructed by combining the constants from the "attributes Value" table below with the logical OR operator.
width	Specifies the width of the dialog box's content area in dialog coordinate units.
height	Specifies the height of the dialog box's content area in dialog coordinate units.
helpinfo helpSource	Currently unused. Had been used in the past to specify the dialog information for version 4 help resource files. These two fields may be used with version 5 help files in the future, so do not reuse them for any other information.
dialogHookId	Specifies the ID of a dialog hook function. See "Dialog Box Manager Hook Functions" for more information. If the dialog box does not have a dialog hook function, specify NOHOOK for this field.
parentDialogId	Specifies the ID of the parent dialog box when one dialog box needs to launch another dialog box. Then this field is used to determine which dialog to inform when the child dialog box is destroyed. Use NOPARENTID if the dialog box is not launched from another dialog box, or the parent dialog does not need to be informed of the child's destruction.
label	Contains the title of the dialog box. This string will display in the title bar of the dialog box.
itemList	Is an array of DialogItemRsc and lists dialog items that the dialog box contains. The DialogItemRsc structure is discussed after the following table.

The following table lists possible values for the attributes field (described above) and the meaning of those values:

attributes Value	Meaning
DIALOGATTR_DEFAULT	Is the default attribute. None of the other attributes in this table are set.
DIALOGATTR_MODAL	Indicates the dialog box is modal. It will not have a window control icon on the left side of the title bar or icons on the right side of the title bar. See "Modeless and modal dialog boxes" on page 15-10 for more information on modal dialogs.

attributes Value	Meaning
DIALOGATTR_GROWABLE	Indicates the dialog box can be resized. It will have resize corners drawn in the window border and the window manager will allow the user to change the size of the dialog box.
DIALOGATTR_SINKABLE	Indicates the dialog box can be sunk. See "Sinking dialog boxes" on page 15-13 for more information on sinking.
DIALOGATTR_UNCLOSEABLE	Indicates the dialog box cannot be closed. MicroStation's command window is an example of an uncloseable dialog box.
DIALOGATTR_NOAUTOSWITCH	Indicates the keyboard focus will not automatically switch to the default focus window when a click occurs in a window that cannot accept the focus. See "Keyboard focus" on page 15-13 for more information on keyboard focus auto-switching.
DIALOGATTR_CLOSEONNEW	Indicates that the dialog box should be closed when a design file is opened. This attribute is used when the contents of a dialog box depend on the characteristics of the file opened.
DIALOGATTR_ALWAYSSETSTATE	Modal dialogs usually do not set the underlying application's state until the user clicks on the OK button. If this attribute is set, the dialog box manager will set the application state when the user interacts with an item. See "Modeless and modal dialog boxes" on page 15-10 for more information on modal dialogs.
DIALOGATTR_BOTHVIRTUAL	If the platform supports two virtual screens, setting this attribute indicates that the dialog is to appear on both virtual screens.
DIALOGATTR_NOAUTOACTIVATE	Indicates that the dialog manager should not automatically activate the default items (typically the OK and Cancel buttons) in a dialog box when the user presses the <Return> or <Esc> keys.
DIALOGATTR_NOSYNCHONNEW	Indicates that the dialog manager should not synchronize the dialog items in this dialog box when a new design file is opened.
DIALOGATTR_NORIGHTICONS	Indicates that the dialog manager should not place any window control icons in the right side of the window title bar. That is, no Sink or Minimize icons are displayed in the title bar.

attributes Value	Meaning
DIALOGATTR_NOAUTOFOCUS	Indicates that the dialog manager should not automatically set the input focus to the dialog box when it is opened. By default, the dialog manager assigns input focus to a newly opened dialog box if it has any items which can receive input focus.
DIALOGATTR_DONTCAUSEAUTOSWITCH	Indicates that the dialog manager should not switch the input focus to the default dialog box if the user clicks in this NON-FOCUSABLE dialog box. This attribute should only be used on dialog boxes which contain no focusable items.
DIALOGATTR_AUTOUNLOADAPP	Indicates that the dialog manager should automatically unload the mdl application which owns this dialog box when the dialog box is closed.
DIALOGATTR_DONTDRAWBEVEL	Indicates that the dialog manager should not draw a beveled border around the dialog box when the GUI mode is Motif. By default, all windows/dialog boxes are bordered by a beveled rectangle in Motif GUI mode (palettes are always beveled in Motif). This attribute is ignored in GUI modes other than Motif.
DIALOGATTR_REQUESTBACKINGSTORE	Indicates that the dialog manager should use backing store for the display area of the dialog box if the user has allowed the use of backing store within MicroStation in the user preferences. If, either backing store is not being used due to the user preferences setting or due to the absence of this attribute on the dialog box, the application must be able to draw the contents of the dialog box itself as needed.

In the `DialogBoxRsc`, the `#if defined (resource)` construction specifies two variable-length arrays when used with the resource compiler. The first is an array of `char`; the second is an array of `DialogItemRsc`. The resource compiler precedes all variable length arrays with a `long`, which contains the number of array members.

This syntax is meaningless in C, where only the first variable-length array can be directly referenced. The C compiler cannot determine the start of the second array because its position depends on the length of the first. Therefore, the hidden long generated by the resource compiler becomes visible and only the first variable-length array is declared This information can be used at runtime to find the start of the second array. The construct `#if defined (resource)` is used in a similar fashion in other data structures in this chapter.

DialogItemRsc Structure

Each item in a dialog box needs to have an DialogItemRsc entry in the DialogBoxRsc field, itemList. The order that items are listed is important in two respects: the items are drawn in the order specified, and the input focus moves from item to item based on this order. The DialogItemRsc data structure is declared as follows:

```
typedef struct dialogitemrsc
{
   Sextent extent;      /* item area, origin (in dialog coords), if
                           width/height is zero, use dimensions
                           specified in item */
   long type;           /* item type */
   long id;             /* item ID */
   byte attributes;     /* item attributes */
   long itemArg;        /* item argument */

#if defined (resource)
   char label[];        /* item label */
   char auxInfo[];      /* item auxiliary information */
#else
   int  labelLength;    /* length of item label */
   char label[1];       /* item label */
#endif

} DialogItemRsc;
```

The meaning of some of the DialogItemRsc fields depends on the type of item specified. See the descriptions of the various item types for the exact interpretation.

Field	Description
extent	Specifies the area (x position, y position, width and height) that the item will cover in the dialog box. This field is in dialog coordinate units, and it's origin is the upper left corner of the dialog box. For many item types, 0 can be specified for the width or height. The handler for that item type will then calculate a reasonable default if possible.
type	Specifies the type of item to create. Valid values for this field are ColorPicker, Generic, GroupBox, IconCmdFrame, Label, LevelMap, ListBox, MenuBar, MLText, OptionButton, PushButton, ScrollBar, Text, ToggleButton, Sash, Scale, RadioButton, MenuBarX, LevelPicker, ToolBox, IconCmdX, ButtonGroup and IconCmdFrameX.
id	Specifies the resource ID of the item instance to load. The dialog box manager calls the resource manager with type and ID to load the correct item resource. The item types, Label and GroupBox, do not need to load an associated item resource. All information needed to specify these types is contained in the DialogItemRsc field so the ID field can be set to any value (usually 0).

Field	Description
attributes	Specifies the initial general state of the item. This field can be ENABLED (ON) to allow an item to be manipulated or DISABLED (OFF). A disabled item is dimmed, cannot accept the input focus, and ignores user actions. Most items should be ENABLED (ON). In order to make the dialog item initially invisible, use the logical **OR** operator to combine the HIDDEN constant with the appropriate enabled constant. For example, with ENABLED | HIDDEN, the dialog item is initially invisible; when it becomes visible with a call to **mdlDialog_itemShow**, it will be automatically enabled.
itemArg	Contains different values for different item types. For example, label items use this field to specify the kind of justification to use. Most items do not currently use this field.
label	Represents a string that lets you override the label of the item for a particular dialog box. A default label is defined in each item resource. The label specified in a DialogItemRsc overrides any label specified in the item resource. This field lets you use the defined behavior of an item, but label the item differently. The label's use depends on the item type.
auxInfo	Contains additional item-specific information. Currently, this field is used only to override the access string defined in an item resource. This ability enables the item to act as it is defined in the item resource but affect a different application variable.

The following is an example of a complete dialog box resource. This is the Element Attributes dialog box, and it is defined in MicroStation's resource file.

```
#define D1 YC/2
#define D2 YC
#define XS 7*XC + XC/2
#define X2 14*XC

DialogBoxRsc DIALOGID_ElementAttributes=
{
   DIALOGATTR_DEFAULT | DIALOGATTR_SINKABLE | DIALOGATTR_CLOSEONNEW,
   25*XC, 10*YC + YC/2, NOHELP, MHELP, NOHOOK, NOPARENTID,
   "Element Attributes",
{
{{XS, GENY(1), 5*XC, 0}, Text, TEXTID_ElementLevel, ON, 0, "", ""},
{{XS, GENY(2)+D1, 5*XC, 0}, Text, TEXTID_ElementColor, ON, 0, "", ""},
{{X2, GENY(2)+D1, 0, 0}, ColorPicker, COLORPICKERID_ElementColor,
                                                    ON, 0 ,"", ""},
{{XS, GENY(3)+D1, 5*XC, 0}, Text, TEXTID_ElementStyle, ON, 0, "", ""},
{{X2, GENY(3)+D1, 0, 0}, OptionButton, OPTIONBUTTONID_ElementStyle,
                                                    ON, 0, "", ""},
```

```
{{XS, GENY(4)+D1, 5*XC, 0}, Text, TEXTID_ElementWeight, ON, 0,"", ""},
{{X2, GENY(4)+D1, -58, 0}, OptionButton, OPTIONBUTTONID_ElementWeight,
                                                    ON, 0, "", ""},
{{XS, GENY(5)+D2, 13*XC, 0}, OptionButton, OPTIONBUTTONID_ElementArea,
                                                    ON,0,"", ""},
{{XS, GENY(6)+D2, 13*XC, 0}, OptionButton, OPTIONBUTTONID_ElementClass,
                                                    ON,0,"", ""},
{{XS, GENY(7)+D2+YC/4, 0, 0}, ToggleButton, TOGGLEID_ElementFill,
                                                    ON, 0,"", ""},
}
};

#undef D1
#undef D2
#undef XS
#undef X2
```

Common Item Resource Fields

Most item resource data structures discussed in "Standard dialog box items" have many fields in common. These common fields will be discussed here rather than with each individual item resource description.

Field	Description
`commandNumber`	Contains a command number that will be put at the end of MicroStation's input queue when the item is activated. (The user releases the data button when the cursor is within the item, or, if the item accepts keystrokes, the user presses spacebar or enter when focus is on item).
`commandSource`	Specifies the task that will execute the command associated with `commandNumber`. 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. (If `MCMD` is specified, `commandNumber` must be a MicroStation command number defined in **cmdlist.h**.)
`unparsed`	Specifies a string that will be placed with `commandNumber` on MicroStation's input queue when an item is activated. This field is useful when a command that cannot be fully parsed into a command number needs to be specified. Specifying unparsed simulates the user keying in the command indicated by `commandNumber`, a space character, and then the unparsed string.
`helpInfo`	Specifies the help reference ID for an item. If no help exists for an item, set this field to `NOHELP`.

Field	Description
helpSource	Specifies both the type of help (command or topic) and the task that owns the help file to use for the item. If no help exists for the item, set this field to MHELP.
itemHookId	Specifies the ID of an item hook function. See "Hook function IDs" on page 15-18 for more information. If an item does not have an item hook function, specify NOHOOK for this field.
itemHookArg	Specifies a value that an item hook function uses to determine its operation. itemHookArg is stored in an item's RawItemHdr structure. See "RawItemHdr structure" on page 17-32 for more information.
label	Contains a string that lets you specify a default label for the item. This default label can be overridden by the DialogItemRsc label field, which is part of a dialog box's item list.
accessStr	Specifies the underlying application variable that the item will control. This field should contain a C expression that references a variable published with a call to an mdldialog_publish… function. An example of this type of expression is tcb->control.grid_lock. (tcb points to one of MicroStation's global data areas and is automatically available for any resource file to reference).
synonymsId	Indicates the DItem_SynonymsRsc instance to use when synching a list of synonyms after an item's state changes. See "Synonym resources" on page 15-12 for more information about synonyms.

Label Item

The label item is used to draw a text string somewhere in a dialog. The text can be forced to wrap at word breaks and can be left, right or center-justified. The text of a label can be changed with a call to mdlDialog_itemSetLabel.

The label item does not need an item resource to complete its appearance, and is completely specified by information in the item list.

Item list specification

The DialogItemRsc field extent specifies the location of the label. Most label items are only a single line high. In this case, set the height to 0. The label handler will automatically make the label height equal to the current dialog font height.

The width can also be 0. In this case, the width of the item will be the width of the original label contents. If you later set the label to a longer string, it will be truncated

since it will no longer fit in the same area on the screen. If the label can be changed dynamically, a width the length of the longest anticipated string should be specified.

The `type` field should be `Label`.

The dialog box manager does not use the ID since a label item does not have an item resource. It can be used, however, to differentiate between labels when a dialog has many label items. In this case, the ID would be used with a call to `mdlDialog_itemGetByTypeAndId`.

The `attributes` field can be `ON` or `OFF`, and optionally combined with `HIDDEN`. It will usually be `ON`.

The `itemArg` field specifies justification, word-wrapping, and font style. Justification options are `ALIGN_LEFT`, `ALIGN_RIGHT` and `ALIGN_CENTER`. If no alignment is specified, `ALIGN_LEFT` is assumed. To specify word wrap, use the logical OR operator to combine with `LABEL_WORDWRAP`. To specify the bold dialog font (instead of the default normal dialog font), use the logical OR operator to combine with `LABEL_FONT_BOLD`.

The `label` field contains the text of the label item.

The `auxInfo` field is not used with label items.

The following is an example of a label item list specification:

```
{{XC, YC, 30*XC, 0}, Label, 0, ON, ALIGN_LEFT|LABEL_FONT_BOLD,
  "Hello, World", ""}
```

Item resource specification

A label item does not have an item resource.

Item hook function messages

Since a label item does not have an item resource, it cannot have an item hook function and will not receive item messages.

Label item functions

The `mdlDialog_labelSetAttributes` function allows the application program to changes the justification and weight of the label text.

Group Box Item

The group box item is used to draw a rectangle with a 3D appearance around a group of dialog items. A label can be drawn on the top edge of the rectangle, near the left side of the group box. The text of a label can be changed by calling `mdlDialog_itemSetLabel`.

The group box item doesn't need an item resource to complete it's appearance and is completely specified by information in the item list.

Item List Specification

The `DialogItemRsc` field `extent` specifies the location of the group box. Both a positive width and height should be specified. The group box handler cannot determine reasonable defaults.

The `type` field should be `GroupBox`.

The `ID` is unused by the dialog box manager, since a group box item does not have an item resource. It can be used, however, to differentiate between group boxes when a dialog has a number of group box items. In that case you would use the ID in a call to `mdlDialog_itemGetByTypeAndId`.

The `attributes` field can be `ON` or `OFF` and optionally combined with `HIDDEN`. It will usually be `ON`.

The `itemArg` field is unused and should usually be set to 0.

The `label` field contains the label of the group box item. This is placed over the top edge of the group box, near the left edge.

The `auxInfo` field is not used with group box items.

The following is an example of a group box item list specification:

```
{{XC, YC, 20*XC, 10*YC}, GroupBox, 0, ON, 0, "Snap", ""}
```

Item resource specification

A group box item does not have an item resource.

Item hook function messages

Since a group box item does not have an item resource, it cannot have an item hook function and will not receive item messages.

Group box item functions

There are no group box item functions.

Separator Item

The seprator item is used to draw a vertical or horizontal seprator line the height or width of a dialog box.

The separator item doesn't need an item resource to complete its appearance and is completely specified by information in the item list.

Item List Specification

The `DialogItemRsc` field `extent` specifies the location of the separator. The X or Y origin point of the line is defined and the width/height should be set to 0. The seprartor handler determines whether the separator line is vertical or horizontal by whether the X origin is 0 (if non-zero, the separator is assumed to be vertical).

The `type` field should be `Separator`.

The `ID` is unused by the dialog box manager, since a seprator item does not have an item resource. It can be used, however, to differentiate between seprator when a dialog has a number of separator items. In that case you would use the ID in a call to `mdlDialog_itemGetByTypeAndId`.

The `attributes` field can be `ON` or `OFF` and optionally combined with `HIDDEN`. It will usually be `ON`.

The `itemArg` field is unused and should usually be set to 0.

The `label` field is unused and should usually be set to `""`.

The `auxInfo` field is not used with separartor items.

The following is an example of a separator item list specification:

```
{{0, 8*YC, 0, 0}, Separator, 0, ON, 0, "", ""}
```

Item resource specification

A separator item does not have an item resource.

Item hook function messages

Since a separator item does not have an item resource, it cannot have an item hook function and will not receive item messages.

Separator item functions

There are no separator item functions.

Toggle Button Item

The toggle button item (also known as a check button or a check box) shows the state of an application variable that is either on or off.

Item list specification

The `DialogItemRsc` field `extent` specifies the location of the toggle button. The x position specifies the location of the left side of the toggle button's 3D square, not the start of the toggle button's text label. The y position specifies both the top of the 3D square and the top of the text label. If the width is 0, the width of the label is used. If the height is 0, the height of the current dialog font is used.

The `type` field should contain `ToggleButton`.

The `ID` determines the `DItem_ToggleButtonRsc` instance to load.

The `attributes` field can be `ON` or `OFF`, and optionally combined with `HIDDEN`. It will usually be `ON`.

The `itemArg` field is unused and should usually be set to 0.

The `label` field overrides the label contained in the `DItem_ToggleButtonRsc` instance.

The `auxInfo` field currently overrides only the access string contained in the `DItem_ToggleButtonRsc` instance.

The following is an example of a toggle button item list specification:

```
{{XC, YC, 0, 0}, ToggleButton, TOGGLEID_LockGrid, ON, 0, "", ""}
```

Item resource specification

The toggle button item is defined in a resource file with the following structure:

```
typedef struct ditem_togglebuttonrsc
{
   ULong commandNumber;
   ULong commandSource;
   long synonymsId;
   ULong helpInfo;
   ULong helpSource;
   long itemHookId;
   long itemHookArg;
   ULong mask;
   char invertFlag;

#if defined (resource)
   char label[];
   char accessStr[];
#else
   long labelLength;
   char label[1];
#endif

} DItem_ToggleButtonRsc;
```

The DItem_ToggleButtonRsc structure has the following unique fields. (See "Common item resource fields" for a description of any field not in this table).

Field	Description
mask	If invertFlag is FALSE, when the toggle button is ON (depressed and checked) the bitwise OR operator combines mask with the variable specified by accessStr. When the toggle button is OFF (raised and unchecked) the bitwise AND operator combines the 1's complement of mask with the variable specified by accessStr. If invertFlag is TRUE, when the toggle button is OFF (raised and un-checked) the bitwise OR operator combines mask with the variable specified by accessStr. When the toggle button is ON (depressed and checked) the bitwise AND operator combines the 1's complement of mask with the variable specified by accessStr.
invertFlag	See the description for mask.

The following is an example of a toggle button item resource. This item is used in the Locks and Toggles dialog boxes, and is defined in MicroStation's resource file.

```
DItem_ToggleButtonRsc TOGGLEID_LockGrid=
{
   CMD_LOCK, MCMD, NOSYNONYM, NOHELP, MHELP, NOHOOK, NOARG, 0x1,
   NOINVERT, "Grid Lock", "tcb->control.grid_lock"
};
```

Item hook function messages

The following messages are sent to item hook functions that are attached to toggle button items:

DITEM_MESSAGE_CREATE

DITEM_MESSAGE_INIT

DITEM_MESSAGE_DESTROY

DITEM_MESSAGE_BUTTON

DITEM_MESSAGE_SYNCHRONIZE

DITEM_MESSAGE_GETSTATE

DITEM_MESSAGE_SETSTATE

DITEM_MESSAGE_STATECHANGED

DITEM_MESSAGE_QUEUECOMMAND

Toggle button item functions

`mdlDialog_toggleButtonGetInfo` retrieves detailed information about a toggle button.

`mdlDialog_toggleButtonSetInfo` sets various aspects of a toggle button.

Push Button Item

The push button item activates commands when the data button is pressed and then released while the cursor is on the push button. A push button can also be activated by pressing the enter key if a user has given focus to that button by moving via the tab key.

Item list specification

The `DialogItemRsc` field `extent` specifies the location of the push button. If width is 0 the width of the label, plus an extra character on each side, is used. Since all buttons in a given dialog should have the same width, the width usually needs to be specified. `BUTTON_STDWIDTH` can be used for buttons with labels that have only five or six characters. Height should be 0. This height lets the push button item handler set a height that is the same for all push buttons.

The `type` field should contain `PushButton`.

The `ID` determines the `DItem_PushButtonRsc` instance to load.

The `attributes` field can be `ON` or `OFF` and optionally combined with `HIDDEN`. It will usually be `ON`.

The `itemArg` field is unused and should usually be set to 0.

The `label` field can override the label contained in the `DItem_PushButtonRsc` instance.

The `auxInfo` field is not used with push button items.

The following is an example of a push button item list specification:

```
{{2*XC, 5*YC, BUTTON_STDWIDTH, 0}, PushButton,
PUSHBUTTONID_Precision, ON, 0, "", ""}
```

Item resource specification

The push button item is defined in a resource file with the following structure:

```
typedef struct ditem_pushbuttonrsc
{
   char isDefault;/* TRUE if this is default button */
   ULong helpInfo;
   ULong helpSource;
   long itemHookId;
   long itemHookArg;
   ULong commandNumber;
   ULong commandSource;

#if defined (resource)
   char unparsed[];
   char label[];
#else
   long unparsedLength;
   char unparsed[1];
#endif

} DItem_PushButtonRsc;
```

The `DItem_PushButtonRsc` structure has the following unique fields. See "Common Item Resource Fields" on page 16-8 for a description of any field not in this table.

Field	Description
`isDefault`	Indicates whether the push button is a dialog box's default button. Possible values are listed in the `isDefault` table below.
`itemHookArg`	When the standard push button item hook function is invoked by declaring `HOOKITEMID_Button_StandardAction` in the `itemHookId` field, `itemHookArg` specifies the `actionType` for the push button. `actionType` determines the behavior of push buttons in modal dialogs. It is sent to a dialog hook function as part of the `DIALOG_MESSAGE_DESTROY` message. The values of `itemHookArg` listed in the `itemHookArg` table below have special meanings when the standard push button item hook function is attached to a push button in a modal dialog.

The following table lists possible values for the `isDefault` field (described above) and the meaning of those values:

isDefault Value	Meaning
`DEFAULT_BUTTON`	The push button is a dialog box's default button. The push button is outlined with a recessed rectangle. The button will automatically be activated if the user presses the **<Enter>** key. In some situations, a double-click in a dialog box should also activate the default button. This action must be implemented manually by calling `mdlDialog_pushButtonActivate`. In any dialog box there should be only one default push button.
`NOT_DEFAULT_BUTTON`	The button is not a dialog box's default button.
`CANCEL_BUTTON`	The push button is a dialog box's cancel button. This type of push button is automatically activated if the user presses the **<Esc>** key and typically means that the operations to be performed by the dialog box should be aborted (cancelled). A dialog box should never contain more than one cancel button.
`PBUTATTR_ALIGNCENTER`	This attribute automatically centers the push button label in the push button display rectangle. This is the default attribute.
`PBUTATTR_ALIGNLEFT`	This attribute causes the push button label text to be left aligned in the push button display rectangle.
`PBUTATTR_ALIGNRIGHT`	This attribute causes the push button label text to be right aligned in the push button display rectangle.

The following table lists possible values for the `itemHookArg` field (described above) and the meaning of those values:

itemHookArg Value	Meaning
`ACTIONBUTTON_APPLY`	When the push button is activated, the `mdlDialog_itemSetState` function will be called for each item in the dialog box. The dialog box will not be closed.
`ACTIONBUTTON_RESET`	When the push button is activated, the `mdlDialog_itemSynch` function is called for each item in the dialog box. The dialog box will not be closed.

itemHookArg Value	Meaning
ACTIONBUTTON_OK	When the pushbutton is activated, the `mdlDialog_itemSetState` function will be called for each item in the dialog box. The dialog box will then be closed.
ACTIONBUTTON_CANCEL	When the push button is activated, the dialog box will then be closed.

Multi-lined push button items can be displayed by placing newline characters '\n' in the push button item label.

The following are examples of push button item resources:

```
DItem_PushButtonRsc PUSHBUTTONID_Precision=
{
   NOT_DEFAULT_BUTTON, NOHELP, MHELP,
   NOHOOK, NOARG, CMD_POINT_DEFAULT, MCMD, "", "Apply"
}

DItem_PushButtonRsc PUSHBUTTONID_MultiLineLeft=
{
   NOT_DEFAULT_BUTTON | PBUTATTR_ALIGNLEFT, NOHELP, MHELP, NOHOOK,
   NOARG, CMD_POINT_DEFAULT, MCMD, "", "Left\nAligned\nMultilined"
}
```

Item hook function messages

The following messages are sent to item hook functions that are attached to push button items:

DITEM_MESSAGE_CREATE

DITEM_MESSAGE_DESTROY

DITEM_MESSAGE_BUTTON

DITEM_MESSAGE_SYNCHRONIZE

DITEM_MESSAGE_QUEUECOMMAND

Push button item functions

Using the `mdlDialog_pushButtonActivate` function produces the same result as pressing the data button and then releasing it while the cursor is in the indicated push button. Thus, button and queue command messages are sent to any attached item hook function. If the hook function does not handle the DITEM_MESSAGE_QUEUECOMMAND message or if no hook function exists, the push button's non-zero `commandNumber` is placed at the end of MicroStation's input queue.

The `mdlDialog_pushButtonSetDefault` function dynamically changes the default button in a dialog box.

The `mdlDialog_pushButtonSetCancel` function dynamically changes the cancel button in a dialog box.

`mdlDialog_pushButtonGetInfo` retrieves detailed information about a push button.

`mdlDialog_pushButtonSetInfo` sets various aspects of a push button.

Several default push buttons, are provided with MicroStation, and included in MicroStation's resource file. These may be included in dialogs by including in the `DialogItemList` Resource specification and optionally, overriding their labels in the label entry of the dialog item list entry. These are, (see Dlogids.h):

	ID	Default Label
PUSHBUTTONID_RESET	-1	Apply
PUSHBUTTONID_OK	-2	Reset
PUSHBUTTONID_CANCEL	-3	Ok
PUSHBUTTONID_DEFAULT	-4	Cancel
PUSHBUTTONID_OKNotDefault	-5	Default
PUSHBUTTONID_	-8	Ok
PUSHBUTTONID_	-7	Yes
PUSHBUTTONID_	-8	No
PUSHBUTTONID_	-9	Retry
PUSHBUTTONID_	-10	Stop
PUSHBUTTONID_	-11	Help
PUSHBUTTONID_	-12	Yes to All

Option Button Item

The option button item lets the user select one choice from a mutually exclusive set of choices. It has an optional label right-justified next to a beveled rectangle that shows the currently selected choice. When the user presses the mouse button while the cursor is in the currently selected choice, the full list of choices displays. The user drags over the various option menu choices and releases the mouse button on the new selection. If the mouse button is released outside of the popped-up menu, the current selection is unchanged. The option choices can be text strings or icons

Option buttons should be used only to choose application settings. Do not use option buttons to invoke action commands (such as invoking another dialog box).

Item list specification

The `DialogItemRsc` field `extent` specifies the location of the beveled rectangle that displays the option button's current choice. Any label associated with the option button will be right-justified next to the left edge of this beveled rectangle. If the option button is using text subitems and the width is 0, the width of the longest text subitem label will be used. If several option buttons are vertically aligned, a non-zero width should be specified to ensure that these option buttons have the same width. If height is 0, as it normally should be, the height of the current dialog font will be used.

If the option button uses iconic subitems, the default width and height are the width and height of the first subitem. With iconic subitems, a width sometimes needs to be specified in pixels, not dialog coordinate units. To specify pixels, use a negative number.

The `type` field should be `OptionButton`.

The `ID` determines the `DItem_OptionButtonRsc` instance to load.

The `attributes` field can be `ON` or `OFF` and optionally combined with `HIDDEN`. It will usually be `ON`.

The `itemArg` field is unused and should usually be set to 0.

The `label` field can override the label contained in the `DItem_OptionButtonRsc` instance.

The `auxInfo` field can currently override only the access string contained in the `DItem_OptionButtonRsc` instance.

The following is an example of an option button item list specification:

```
{{7*XC, 7*YC, 13*XC, 0}, OptionButton, OPTIONBUTTONID_ElementClass,
  ON, 0, "", ""}
```

Item resource specification

The option button item is defined in a resource file with the following structure:

```
typedef struct ditem_optionbuttonitemrsc
{
   ULong iconType;
   long iconId;
   ULong commandNumber;
   ULong commandSource;
   ULong value;
   ULong mask;
   byte enabled;

#if defined (resource)
   char label[];
#else
   long labelLength;
   char label[1];
#endif
} DItem_OptionButtonItemRsc;

typedef struct ditem_optionbuttonrsc
{
   long synonymsId;
   ULong helpInfo;
   ULong helpSource;
   long itemHookId;
   long itemHookArg;

#if defined (resource)
   char label[];
   char accessStr[];
   DItem_OptionButtonItemRsc optionButtonItems[];
#else
   long labelLength;
   char label[1];
#endif

} DItem_OptionButtonRsc;
```

The `itemHookArg` field of the `DItem_OptionButtonRsc` structure controls how the option button is displayed and initialized as described below:

Value	Meaning
`OPTNBTNATTR_NEWSTYLE`	Display the Motif option button bump on the right end of the option button display rectangle. All option buttons should specify this attribute in order to display this bump which allows the user to distinguish more readily between an option button and a push button.
`OPTNBTNATTR_DONTADDBUMPWIDTH`	This attribute indicates to the item handler that the bump width should not be added to the width defined in the option button extent data when creating the option button item. By default, the item handler assumes that the extent information only includes enough space to display the option button selection text and that it must add additional space to display the bump without overwriting the text. If this attribute is used, the item handler assumes the width specifies enough space for both the text and the bump.

The only unique field in the `DItem_OptionButtonRsc` structure is the array of `DItem_OptionButtonItemRscs` that is the option button's subitem list. For consistency with MicroStation's dialog boxes, an option button's label should always end with a colon.

The `DItem_OptionButtonItemRsc` structure has the following unique fields. (See "Common item resource fields" for a description of any field not in this table.)

Field	Description
`iconType`	Indicates the type of icon the subitem is using. This field should usually be Icon if you are using `IconRscs`. If not using icons, specify `NOTYPE`.
`iconId`	Specifies the ID of the icon instance to load and display as the subitem. If you are not using icons, specify `NOICON`.

<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td>value</td><td>Is used with mask to determine the currently selected subitem. Only one subitem can be selected at once. mask indicates the relevant bits of the variable specified by accessStr. These bits, shifted so the rightmost bit of the mask becomes bit 0, are compared to value. If the mask is 0x0E (1110 binary), the masked value will be shifted right 1 bit before being compared to value. The subitem whose shifted masked bits match value becomes the currently selected subitem.
For example, suppose bits 2 and 3 of the variableflags will be tested. mask will then be 0xC (1100 binary). If bit 2 is on, the first subitem should be selected. If bit 3 is on, the second subitem should be selected. If both bits are on, the third subitem is selected. value and mask should be set to the following for each subitem:
<table>
<tr><th>subitem</th><th>value</th><th>mask</th><th>flags value that will select subitem</th></tr>
<tr><td>0</td><td>1</td><td>0xC</td><td>4 (= 1 << 2)</td></tr>
<tr><td>1</td><td>2</td><td>0xC</td><td>8 (= 2 << 2)</td></tr>
<tr><td>2</td><td>3</td><td>0xC</td><td>12 (= 3 << 2)</td></tr>
</table></td></tr>
<tr><td>mask</td><td>Is used with value to determine the currently selected subitem. If the entire variable specified by accessStr is being compared to value, set this field to NOMASK (which is defined to be 0xFFFFFFFF).</td></tr>
<tr><td>enabled</td><td>Determines the state (enabled or disabled) of the subitem when the option button is first created. If this field is set to ON, the subitem can be selected; when it is set to OFF, the subitem will be dim and cannot be selected.</td></tr>
<tr><td>label</td><td>Specifies the subitem's text string if NOTYPE and NOICON are specified for iconType and iconId.</td></tr>
</table>

The following is an example of an option button item resource. This item is used in the Element Attributes dialog box, and is defined in MicroStation's resource file.

```
DItem_OptionButtonRsc OPTIONBUTTONID_ElementClass=
{
   NOSYNONYM, NOHELP, MHELP, NOHOOK, OPTNBTNATTR_NEWSTYLE,
   "Class:", "tcb->props",
   {
        {NOTYPE, NOICON, CMD_ACTIVE_LEVEL, MCMD, 0, 0x000F, ON,
         "Primary"},
        {NOTYPE, NOICON, CMD_ACTIVE_LEVEL, MCMD, 2, 0x000F, ON,
         "Construction"}
   }
};
```

Item hook function messages

The following messages are sent to item hook functions that are attached to option button items:

```
DITEM_MESSAGE_CREATE
DITEM_MESSAGE_INIT
DITEM_MESSAGE_DESTROY
DITEM_MESSAGE_BUTTON
DITEM_MESSAGE_SYNCHRONIZE
DITEM_MESSAGE_GETSTATE
DITEM_MESSAGE_SETSTATE
DITEM_MESSAGE_STATECHANGED
DITEM_MESSAGE_QUEUECOMMAND
```

Option button item functions

`mdlDialog_optionButtonGetNItems` returns the number of subitems in an option button.

`mdlDialog_optionButtonGetItemInfo` retrieves detailed information about a subitem of an option button.

`mdlDialog_optionButtonSetItemInfo` sets various aspects of a subitem of an option button.

`mdlDialog_optionButtonDeleteItem` deletes a subitem from an option button.

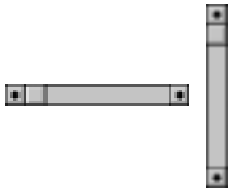
`mdlDialog_optionButtonDeleteAll` deletes all subitems from an option button.

`mdlDialog_optionButtonInsertItem` inserts a subitem into an option button.

`mdlDialog_optionButtonSetEnabled` sets the state (enabled or disabled) of an option button's subitem.

Scroll Bar Item

The scroll bar item can be used to show the state of a variable that ranges between a set of values.



Item list specification

The `DialogItemRsc` field `extent` specifies the location of the scroll bar. The x position specifies the left edge and the y position specifies the top edge of the scroll bar. The orientation of the scroll bar depends on the width and height fields. If width is 0, the scroll bar handler will make a vertical scroll bar. If height is 0, a horizontal scroll bar will be created. Both the width and height cannot be 0.

The `type` field should be `ScrollBar`.

The `ID` determines the `DItem_ScrollBarRsc` instance to load.

The `attributes` field can be `ON` or `OFF` and optionally combined with `HIDDEN`. It will usually be `ON`.

The `itemArg` field is unused and should usually be set to 0.

The `label` field is ignored for scroll bar items.

The `auxInfo` field can currently override only the access string contained in the `DItem_ScrollBarRsc` instance.

The following is an example of a scroll bar item list specification:

```
{{XC, YC, 10*XC, 0}, ScrollBar, SCROLLBARID_ColorPal_Hue, ON, 0, "", ""}
```

Item resource specification

The scroll bar item is defined in a resource file with the following structure:

```
typedef struct ditem_scrollbarrsc
{
   long itemHookId;
   long itemHookArg;
   int minValue;
   int maxValue;
   int incAmount;
   int pageIncAmount;
   double sliderSize;

#if defined (resource)
   char accessStr[];
#else
   long accessStrLength;
   char accessStr[1];
#endif

} DItem_ScrollBarRsc;
```

The `DItem_ScrollBarRsc` structure has the following unique fields. (See "Common item resource fields" for a description of any field not in this table.)

Field	Description
`minValue`	Specifies the value associated with the minimum position of the scroll bar slider.
`maxValue`	Specifies the value associated with the maximum position of the scroll bar slider.
`incAmount`	Specifies the amount the current value of the scroll bar is changed when the user clicks on the scroll bar arrows.
`pageIncAmount`	Specifies the amount the current value of the scroll bar is changed when the user clicks in the page area of the scroll bar. The page areas are the areas between the slider and the scroll bar arrows.
`sliderSize`	Specifies the size of the slider relative to the entire size of the scroll bar. `sliderSize` must be between 0.0 and 1.0. The slider's size will never be less than the width of vertically oriented scroll bars or the height of horizontally oriented scroll bars.

The following is an example of a scroll bar item resource:

```
DItem_ScrollBarRsc SCROLLBARID_ColorPal_Hue=
{
   NOHOOK, NOARG, 0, 359, 1, 10, 0.10, ""
};
```

Item hook function messages

The following messages are sent to item hook functions attached to scroll bar items:

```
DITEM_MESSAGE_CREATE
DITEM_MESSAGE_INIT
DITEM_MESSAGE_DESTROY
DITEM_MESSAGE_BUTTON
DITEM_MESSAGE_GETSTATE
DITEM_MESSAGE_SETSTATE
DITEM_MESSAGE_STATECHANGED
DITEM_MESSAGE_QUEUECOMMAND
```

Scroll bar item functions

The `mdlDialog_scrollBarSetRange` function sets the minimum and maximum limits of a scroll bar item. The `mdlDialog_scrollArrowDraw` function draws a scroll-bar-item-like arrow in a dialog box.

`mdlDialog_scrollBarGetInfo` retrieves detailed information about a scroll bar.

`mdlDialog_scrollBarSetInfo` sets various aspects of a scroll bar.

View window icons and scroll bars

MicroStation has added optimized viewing commands and scroll bars to the view window borders. The user will have the option of removing these icons and scroll bars if screen *real estate* is the primary concern. This new feature means that the content rectangle of the view window is not necessarily the same as the usable view area. A new MDL routine `mdlView_getViewRectangle` should be used instead of `mdlWindow_contentRectGetLocal` or `mdlWindow_contentRectGetGlobal` if the dimensions of the usable view rectangle are desired.

Text Item

The text item allows editable text strings to be entered. The format of the string can be specified when external application variables are being modified by the text item.

The strings can be range checked against specified minimum and maximum string values. Custom validation can also be performed for an item by attaching a item hook function that processes the DITEM_MESSAGE_FOCUSOUT message. Horizontal scrolling allows the editing of more characters than would otherwise fit inside the text item. The standard undo, cut, copy, and paste operations are supported. Selecting of multiple characters can be done by dragging the mouse cursor through ranges of text.

Item list specification

The DialogItemRsc field extent specifies the location of the beveled rectangle that contains the text to edit. Any label associated with the text item will be right-justified next to the left edge of this beveled rectangle. If width is 0, the width of the beveled rectangle is calculated with the maximum number of characters the text item can contain. An explicit width should be specified. If height is 0, as it normally should be, the height of the current dialog font will be used.

The type field should be set to Text.

The ID determines the DItem_TextRsc instance to load.

The attributes field can be ON or OFF and optionally combined with HIDDEN. It will usually be ON.

The itemArg field may be set to TEXT_MASK_LABELABOVE to specify that the label should appear above the beveled rectangle. Otherwise, itemArg should usually be set to 0.

The label field can override the label contained in the DItem_TextRsc instance.

The auxInfo field can currently override only the access string contained in the DItem_TextRsc instance.

The following is an example of a text item list specification:

```
{{XC, YC, 5*XC, 0}, Text, TEXTID_ElementLevel, ON, 0, "", ""}
```

Item resource specification

The text item is defined in a resource file with the following structure:

```
typedef struct ditem_textrsc
{
   ULong commandNumber;
   ULong commandSource;
   long synonymsId;
   ULong helpInfo;
   ULong helpSource;
```

```
    long itemHookId;
    long itemHookArg;
    byte maxSize;                   /* max # of chars in field */
    char formatToDisplay[16];       /* conv to display from internal */
    char formatToInternal[16];      /* conv to internal from display */
    char minimum[16];               /* minimum value */
    char maximum[16];               /* maximum value */
    ULong mask;                     /* only used with integer types */
    UShort attributes;              /* other attributes */

#if defined (resource)
    char label[];
    char accessStr[];
#else
    long labelLength;
    char label[1];
#endif

} DItem_TextRsc;
```

For consistency with MicroStation's dialog boxes, a text item's label should end with a colon. The `DItem_TextRsc` structure has the following unique fields. (See "Common item resource fields" for a description of any field not in this table)..

Field	Description
`maxSize`	The maximum number of characters that this text item can edit.
`formatToDisplay`	A `sprintf` format string to convert the value of the variable specified by `accessStr` to a string. Usually, the '`-`' flag character should indicate left justification. For example, `%-d` left-justifies an integer. Use `%w` to display the variable as a working units string.
`formatToInternal`	A `sscanf` format string to convert from the text item's string value to the format expected by the variable specified by `accessStr`.

Interpretation of item contents	formatToInternal value
integer	`"%ld"`
floating point number	`"%lf"`
working units number	`"%w"`
single character	`"%c"`
string	`""`

Field	Description
`minimum`	A string that contains the minimum value of the text item. Specify an empty string (‘ ‘) for no minimum value. The minimum value should be specified in a format that the `formatToInternal` string can process. For example, if `formatToInternal` is `%ld`, ‘0’ means that the text item must be greater than or equal to 0. If the item’s contents are interpreted in working units, the minimum string can be specified as a floating point number.
`maximum`	A string that contains the maximum value of the text item. Specify an empty string (‘ ‘) for no maximum. The maximum value should be specified in a format that the `formatToInternal` string can process. For example, if `formatToInternal` is `%ld`, ‘100’ means that the text item must be less than or equal to 100. If the item’s contents are interpreted in working units, the minimum string can be specified as a floating point number.
`mask`	This field is used only when the text item’s contents are interpreted as an integer. In this case, `mask` indicates which bits of the variable referenced by `accessStr` will be affected. For example, if `mask` is `0xF0`, `formatToInternal` is `%ld`, and the text item contains 3, the high nibble of the variable referenced by `accessStr` is set to 3. Usually, `mask` should be set to `NOMASK` (defined to be `0xFFFFFFFF`), which indicates that the entire variable is affected.
`attributes`	A short integer that specifies text item characteristics. It is constructed by combining the constants from the following table with the logical OR operator.

attributes Value	Meaning
TEXT_CONCAT	If commandNumber is added to the end of MicroStation's input queue (usually because the user has moved the input focus out of the item), the text item contents will be included as the unparsed part of the command. For example, when the user tabs out of the "Level: text" item in the Element Attributes dialog box, the command number CMD_ACTIVE_LEVEL is placed on the input queue. In addition, since CONCAT is specified as the text item's attribute, the item's contents are appended to the command number. For example, suppose the text item contains the string "5". Tabbing out of the "Level: text" item simulates the user keying in ACTIVE LEVEL 5.
TEXT_NOCONCAT	If commandNumber is added to the end of MicroStation's input queue (usually because the user has moved the input focus out of the item), the contents of the text item will not be included as the unparsed part of the command.
TEXT_READONLY	The beveled rectangle around the text item will not be drawn. The user cannot change the text item contents. However, the text item will respond to DITEM_MESSAGE_SYNCHRONIZE messages. This attribute implements a read-back label that automatically changes its text contents depending on the state of some application variable.
TEXT_NORESETONERROR	This attribute instructs the item handler to not reset the displayed value of the text item when an error has been detected for the text item.
TEXT_ABBREVFILENAME	This attribute instructs the item handler to abbreviate the file name displayed in the text item so that the most significant portion of the file name (the right most display portion) is visible in the text item and the full name is preceeded with a "..." to indicate abbreviation. This attribute implies, and should only be used in, read-only text items.

The following is an example of a text item resource. This item is used in the Element Attributes dialog box, and is defined in MicroStation's resource file.

```
DItem_TextRsc TEXTID_ElementLevel=
{
```

```
    CMD_ACTIVE_LEVEL, MCMD, NOSYNONYM, NOHELP, MHELP, NOHOOK, NOARG,
    2, "%-ld", "%ld", "1", "63", 0xff, CONCAT, "Level:",
    "tcb->ad1.params.actlev"
};
```

Item hook function messages

The following messages are sent to item hook functions that are attached to text items:

```
DITEM_MESSAGE_CREATE
DITEM_MESSAGE_INIT
DITEM_MESSAGE_DESTROY
DITEM_MESSAGE_SYNCHRONIZE
DITEM_MESSAGE_FOCUSIN
DITEM_MESSAGE_FOCUSOUT
DITEM_MESSAGE_KEYSTROKE
DITEM_MESSAGE_POSTKEYSTROKE
DITEM_MESSAGE_GETSTATE
DITEM_MESSAGE_SETSTATE
DITEM_MESSAGE_STATECHANGED
DITEM_MESSAGE_QUEUECOMMAND
```

Text item functions

`mdlDialog_textGetRange` gets a text item's minimum and maximum range strings.

`mdlDialog_textSetRange` sets the minimum and maximum range strings of a text item.

`mdlDialog_textGetInfo` retrieves detailed information about a text item.

`mdlDialog_textSetInfo` sets various aspects of a text item.

Multi-line Text Item

The multi-line text item allows editable multi-line text strings to be entered. Vertical scrolling is supported. The user controls vertical scrolling by using a scroll bar that is part of the multi-line text item. When the user presses the Enter key, a line break is entered. The standard cut, copy and paste operations are supported. Selecting of

multiple characters can be done by dragging the mouse cursor through ranges of text, and by performing keyboard navigation with the shift key held down.

Both word-wrap and horizontal scrolling are supported, although they are mutually exclusive. One or the other must be selected as part of the item definition.

If no-wrap is specified in the item definition, then horizontal scrolling is used. The user controls horizontal scrolling by moving the cursor. If the user tries to move the cursor out of the display area, then MicroStation scrolls the text so that the cursor remains in the display area. Since the user must use the cursor to control horizontal scrolling, this option is not useful for read-only multi-line text items.

If no-wrap is not specified, then word wrap is supported and horizontal scrolling is not supported. If a line exceeds the item's display area, the display is word wrapped at a break.

Multi-line text items used to display a lot of text should use horizontal scrolling since this is more efficient than word wrap. This should not be an issue for most applications.

See the "Text Item Functions" section of the "Dialog Box Manager" chapter of the MDL Function Reference Manual for more information.

Item list specification

The field `origin` in the `DialogItemRsc` field `extent` specifies the origin of the beveled rectangle that contains the text to edit. The field width in the `DialogItemRsc` field extent specifies the distance from left-most boundary of the left-most bevel, to the right-most pixel of the right-most bevel. The field height in the `DialogItemRsc` field extent is ignored. The actual height of the item is computed based on the `DItem_MultilineTextRsc` field `displayRows`. If a label is associated with the multi-line text item, it may be displayed above the item or to the left of the item. In either case, the label does not affect the positioning of the bevel. If the label is displayed to the left, then the origin of the label is computed as follows: the x coordinate is set to the origin of the bevel minus the width of the string minus the width of an average character; the y coordinate is set to the y coordinate of the bevel origin plus 2. The top of the label lines up with the bottom edge of the top bevel and there is a gap approximately 1 character long between the end of the label and the bevel. If the label is displayed above, then the origin of the label is computed as follows: the x coordinate is the same as the x coordinate of the origin of the bevel; the y coordinate is set to the y coordinate of the origin minus the character height minus one quarter of the character height.

The `type` field should be set to `MLText`.

The ID determines the DItem_MultilineTextRsc instance to load.

The attributes field can be ON or OFF and optionally combined with HIDDEN. It will usually be ON.

The itemArg field may be set to TEXT_MASK_LABELABOVE to specify that the label should appear above the beveled rectangle. Otherwise, itemArg should usually be set to 0.

The label field can override the label contained in the DItem_MultilineTextRsc instance.

The following is an example of a text item list specification:

```
{{XC, YC, 40*XC, 0}, MLText, MLTEXTID_TextEdit, ON, 0, "", ""}
```

Item resource specification

The multi-line text item is defined in a resource file with the following structure:

```
typedef struct ditem_multilinetextrsc
{
   long synonymsId;
   ULong helpInfo;
   ULong helpSource;
   long itemHookId;
   long itemHookArg;
   UShort attributes;  /* other attributes */
   long displayRows;

#if defined (resource)
   char label[];
#else
   long labelLength;
   char label[1];
#endif

} DItem_MultilineTextRsc;
```

The DItem_MultilineTextRsc structure has the following unique fields. (See "Common item resource fields" for a description of any field not in this table).

Field	Description
displayRows	The number of rows to be displayed inside the beveled rectangle.
attributes	A short integer that specifies text item characteristics. It is constructed by combining the constants from the following table with the logical OR operator.

attributes Value	Meaning
MLTEXTATTR_READONLY	This forces the field to be read-only. With this in effect, the field never accepts the focus. It is impossible to enter characters into the field, or to position the text cursor to the field. The scroll bar associated with the dialog box still accepts the focus, so a user may use it to scroll through text that has too many lines to be displayed at once.
MLTEXTATTR_NODISPLAYNLCHAR	By default, MicroStation always displays an end-of-line marker at the end of each line. This lets the user distinguish between a line wrap and the actual end of a line. If the MLTEXTATTR_NODISPLAYNLCHAR attribute is specified, MicroStation never displays the end-of-line marker.
MLTEXTATTR_AUTOHIDEDECOR	By default, MicroStation always displays the beveled border and scroll bar associated with a multi-line text item. If the MLTEXTATTR_AUTOHIDEDECOR attribute is specified along with the MLTEXTATTR_READONLY attribute, MicroStation will not display the bounding box and scroll bar unless the text enclosed in the item is large enough to warrant their existence.
MLTEXTATTR_ALWAYSBEVELED	This attribute is used to override part of the effects of the MLTEXTATTR_AUTOHIDEDECOR attribute. If the MLTEXTATTR_ALWAYSBEVELED attribute is specified along with the MLTEXTATTR_READONLY and MLTEXTATTR_AUTOHIDEDECOR attributes, MicroStation will always display the beveled border surrounding the multi-line text.
MLTEXTATTR_NOWRAP	This attribute turns off the default auto line wrap processing of the multi-line item handler. Using this attribute causes horizontal scrolling to be used instead of wrapping text to the next display line.
MLTEXTATTR_FIXEDFONT	This attribute is used to cause MicroStation to display the text in the multi-line text item in the fixed width display font. A proportional font, which is usually preferred, will be used if this is not defined.

The following is an example of a multi-line text item resource.

```
DItem_MultilineTextRsc MLTEXTID_MlTxtEdit1=
{
   NOSYNONYM, NOHELP, MHELP, HOOKITEMID_EditMlTxt, 1,
   0, MLTXT_EDIT_LINES, ""
};
```

Item hook function messages

The following messages are sent to item hook functions that are attached to text items:

DITEM_MESSAGE_CREATE

DITEM_MESSAGE_INIT

DITEM_MESSAGE_DESTROY

DITEM_MESSAGE_SYNCHRONIZE

DITEM_MESSAGE_FOCUSIN

DITEM_MESSAGE_FOCUSOUT

DITEM_MESSAGE_KEYSTROKE

DITEM_MESSAGE_POSTKEYSTROKE

DITEM_MESSAGE_GETSTATE

DITEM_MESSAGE_SETSTATE

An MDL program can provide the multi-line text string to MicroStation either by having an item hook handler respond to a DITEM_MESSAGE_GETSTATE message, or by calling **mdlDialog_itemSetValue**. For the DITEM_MESSAGE_GETSTATE message, the hook handler must set the field stringValueP of the value structure to point to the string. When using **mdlDialog_itemSetValue**, the MDL program uses the stringValueP parameter to provide a pointer to the string. In either case, MicroStation makes a copy of the string and operates on the internal copy.

An MDL program can retrieve the multi-line text string from MicroStation either by having an item hook handler respond to a DITEM_MESSAGE_SETSTATE message, or by calling **mdlDialog_itemGetValue**. For the DITEM_MESSAGE_SETSTATE message, the field stringValueP of the value structure points to MicroStation's copy of the string. The **mdlDialog_itemGetValue** function sets valueUnionP->charPFormat to point to MicroStation's copy of the string. The MDL program must not modify MicroStation's copy of the string.

The multi-line text string passed between MicroStation and the MDL program is one string with the line breaks marked by a special character. The value of this character is defined in the built-in variable textNLCharacter.

With MicroStation version 4.2 and later, multi-line text uses the standard new line character '`\n`' that is represented in C. Compatibility with older MDL applications that used `textNLCharacter` for line breaks will be maintained, as `textNLCharacter` will be defined as '`\n`.'

Multi-line Text item functions

`mdlDialog_mlTextGetInfo` retrieves detailed information about a multi-text item.

`mdlDialog_mlTextSetInfo` sets various aspects of a multi-text item.

`mdlDialog_mlTextGetCursor` returns the line number and position of the cursor.

`mdlDialog_mlTextSetCursor` sets the position of the cursor or highlight range.

`mdlDialog_mlTextGetLineCoords` translates a cursor position into a line number and offset.

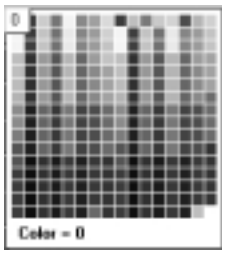
`mdlDialog_mlTextGetLineRange` gets the range in bytes of a line in the text buffer.

`mdlDialog_mlTextInsertString` inserts a string into the buffer.

`mdlDialog_mlTextTopRowNumber` gets the number of the line being displayed in position zero.

Color Picker Item

The color picker item lets the user select one color from a palette of 255 choices. It is a small colored square surrounded by a beveled rectangle. It has an optional label that is right-justified next to the left edge of the beveled rectangle. This label is usually omitted since color picker items usually have an associated text item to their left.



When the user presses the data button while the cursor is in the beveled rectangle, a palette of 255 colored squares displays. The user can then select a new color by releasing the data button while the cursor is in one of the 255 squares. If the user

releases the data button while the cursor is outside of the popped-up color palette, the current color associated with the color picker item is not changed.

Item list specification

The `DialogItemRsc` field extent specifies the location of the color picker. The x position specifies the location of the left side of the color picker item's 3D beveled rectangle. The y position specifies both the top of the 3D beveled rectangle. Width and height should be set to 0. Setting them to 0 ensures that the color picker item will look the same in different dialog boxes. The default width or height is the height of the current dialog font.

The `type` field should be `ColorPicker`.

The `ID` determines the `DItem_ColorPickerRsc` instance to load.

The `attributes` field can be `ON` or `OFF` and optionally combined with `HIDDEN`. It will usually be `ON`.

The `itemArg` field is unused and should usually be set to 0.

The `label` field can override the label contained in the `DItem_ColorPickerRsc` instance.

The `auxInfo` field can currently override only the access string contained in the `DItem_ColorPickerRsc` instance.

The following is an example of a color picker item list specification:

```
{{XC, YC, 0, 0}, ColorPicker, COLORPICKERID_ElementColor, ON, 0, "", ""}
```

Item resource specification

The color picker item is defined in a resource file with the following structure:

```
typedef struct ditem_colorpickerrsc
{
   ULong commandNumber;
   ULong commandSource;
   long synonymsId;
   ULong helpInfo;
   ULong helpSource;
   long itemHookId;
   long itemHookArg;
   long associatedTextId;
   ULong mask;
```

```
#if defined (resource)
   char label[];
   char accessStr[];
#else
   long labelLength;
   char label[1];
#endif

} DItem_ColorPickerRsc;
```

The DItem_ColorPickerRsc structure has the following unique fields. (See "Common item resource fields" for a description of any field not in this table).

Field	Description
associatedTextId	This field specifies the resource ID of a text item that will be associated with the color picker item. As the user drags the mouse over the popped-up color palette, the associated text item will be dynamically set to the index of the color square the cursor is on.
mask	mask indicates the bits of the variable specified by accessStr that are used to determine the current color index. Usually, mask should be set to NOMASK (defined as 0xFFFFFFFF), which indicates that the entire variable will be used to derive the color index.
accessStr	For a color picker, accessStr specifies a variable used to determine the current color index.

The following is an example of a color picker item resource. Defined in MicroStation's resource file, this item is used in the Element Attributes dialog box.

```
DItem_ColorPickerRsc COLORPICKERID_ElementColor=
{
   CMD_ACTIVE_COLOR, MCMD, SYNONYMID_ElementColor, NOHELP, MHELP,
   NOHOOK, NOARG, TEXTID_ElementColor, NOMASK, "","tcb->symbology.color"
};
```

Item hook function messages

The following messages are sent to item hook functions that are attached to color picker items:

DITEM_MESSAGE_CREATE

DITEM_MESSAGE_INIT

DITEM_MESSAGE_DESTROY

DITEM_MESSAGE_BUTTON

DITEM_MESSAGE_SYNCHRONIZE

```
DITEM_MESSAGE_GETSTATE
DITEM_MESSAGE_SETSTATE
DITEM_MESSAGE_STATECHANGED
DITEM_MESSAGE_QUEUECOMMAND
```

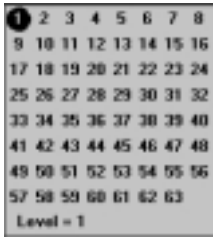
Color picker item functions

`mdlDialog_colorPickerGetInfo` retrieves detailed information about a color picker item.

`mdlDialog_colorPickerSetInfo` sets various aspects of a color picker item.

Level Map Item

The level map item shows the on/off display state of MicroStation's design file levels within a level map variable. A level map variable is an array of four short integers with each bit designating the on/off display state of a single level. The least significant bit of the first array element corresponds to the display state of level 1.



The level map item displays as an eight by eight array of numbers ranging from 1 to 63. The user turns on a level by single-clicking the data button while the cursor is over the number corresponding to the level. A range of levels is selected when the mouse is dragged over a range of numbers. Levels that are turned on are surrounded by a black rectangle. Dragging the data button with the shift button down will select a rectangular range of levels.

The active level is indicated by setting the `activeLevelAccessStr` field. This level will display with a surrounding black circle instead of a rectangle. The user selects the active level by double-clicking the data button. The display of the active level cannot be turned off.

Item list specification

The DialogItemRsc field extent specifies the location of the level map item. Since the width and height of a level map item is fixed, extent.width and extent.height should be set to 0.

The type field should be LevelMap.

The ID determines the DItem_LevelMapRsc instance to load.

The attributes field can be ON or OFF and optionally combined with HIDDEN. It will usually be ON.

The itemArg field is unused and should usually be set to 0.

The label field can override the label contained in the DItem_LevelMapRsc instance.

The auxInfo field can currently override only the access string contained in the DItem_LevelMapRsc instance.

The following is an example of a level map item list specification:

```
{{XC, YC, 0, 0}, LevelMap, LEVELMAPID_ViewLevels, ON, 0, "", ""}
```

Item resource specification

The level map item is defined in a resource file with the following structure:

```
typedef struct ditem_levelmaprsc
{
   ULong helpInfo;
   ULong helpSource;

#if defined (resource)
   char label[];
   char accessStr[];
   char activeLevelAccessStr[];
#else
   long labelLength;
   char label[1];
#endif

} DItem_LevelMapRsc;
```

The DItem_LevelMapRsc structure has the following unique fields. (See "Common item resource fields" for a description of any field not in this table).

Field	Description
accessStr	Represents a C expression that references a level map variable (an array of four short integers).
activeLevel-AccessStr	Represents a C expression that references a variable that contains the current active level. The active level will be surrounded with a circle and is selected with double-click of the data button. If an active level is not applicable, specify an empty string for this field.

The following is an example of a level map item resource. This item is used in the View Levels dialog box, and is defined in MicroStation's resource file.

```
DItem_LevelMapRsc LEVELMAPID_ViewLevels=
{
   NOHELP, MHELP, "View Levels",
   "msDialogState.viewLevelsP->levels",
   "msDialogState.viewLevelsP->activeLevel"
};
```

Item hook function messages

The level map item cannot have an associated item hook function.

Level map item functions

mdlDialog_levelMapGetInfo retrieves detailed information about a level map item.

mdlDialog_levelMapSetInfo sets various aspects of a level map item.

Menu Bar Item

The menu bar item is used to create a narrow rectangular region, across the top of a dialog box, that contains multiple pull-down menus. It is really a composite item. The menu bar itself is simply a placeholder for a number of pull-down menus. There are currently three types of pull-down menus: text, option and color picker menus. These pull-down menus are discussed in their own sections.

Item list specification

The DialogItemRsc field extent is unused for menu bar items. Menu bars always go at the top of a dialog box and extend across its entire width. It should be set to {0, 0, 0, 0}.

The type field should be MenuBar.

The ID determines the DItem_MenuBarRsc instance to load.

The attributes field is ignored for menu bar items because the menu bar is always enabled and displayed. The individual pull-down menus can be disabled however.

The itemArg field is unused and should usually be set to 0.

The label field is unused and should be set to "".

The auxInfo field is unused and should be set to "".

The following is an example of a menu bar item list specification:

```
{{0, 0, 0, 0}, MenuBar, MENUBARID_Main, ON, 0, "", ""}
```

Item resource specification

The menu bar item is defined in a resource file with the following structure:

```
typedef struct ditem_menubaritemrsc
{
   long type;
   long ID;
} DItem_MenuBarItemRsc;

typedef struct ditem_menubarrsc
   {
   long itemHookId;
   long itemHookArg;

#if defined (resource)
   DItem_MenuBarItemRsc menuBarItems[];
#else
   long nMenus;
   DItem_MenuBarItemRsc pulldownMenus[1];
#endif

} DItem_MenuBarRsc;
```

The only unique field in the DItem_MenuBarRsc structure is the array of DItem_MenuBarItemRsc that is the menu bar item's list of pull-down menus.

The DItem_MenuBarItemRsc structure has the following unique fields. (See "Common item resource fields" for a description of any field not in this table.)

Field	Description
type	This field is the type of pull-down menu to include in the menu bar item.
id	This field is the resource ID of the pull-down menu to include in the menu bar.

The following is an example of a menu bar item resource. This item is the MicroStation Command window menu bar, and is defined in MicroStation's resource file.

```
DItem_MenuBarRsc MENUBARID_Main=
{
   HOOKITEMID_MenuBar_CommandWindow, NOARG,
   {
        {PulldownMenu, PULLDOWNMENUID_File},
        {PulldownMenu, PULLDOWNMENUID_Edit},
        {PulldownMenu, PULLDOWNMENUID_Element},
        {PulldownMenu, PULLDOWNMENUID_Settings},
        {PulldownMenu, PULLDOWNMENUID_View},
        {PulldownMenu, PULLDOWNMENUID_Palettes},
        {PulldownMenu, PULLDOWNMENUID_User},
        {PulldownMenu, PULLDOWNMENUID_Help},
   }
};
```

Item hook function messages

The following messages are sent to item hook functions that are attached to menu bar items:

DITEM_MESSAGE_CREATE

DITEM_MESSAGE_DESTROY

DITEM_MESSAGE_BUTTON

Menu bar item functions

mdlDialog_menuBarAddAppMenu adds the Applications pulldown menu to the command window menu bar.

mdlDialog_menuBarAddCmdWinMenu adds a pulldown menu to the command window menu bar.

`mdlDialog_menuBarAttachMenu` adds a pulldown menu to a menu bar.

`mdlDialog_menuBarDeleteCmdWinMenu` deletes a pulldown menu from the command window menu bar.

`mdlDialog_menuBarDeleteAllItems` deletes all the subitems from a specified pull-down menu.

`mdlDialog_menuBarDeleteItem` deletes a subitem from a specified pull-down menu.

`mdlDialog_menuBarDeleteAllMenus` deletes all the menus from a menu bar.

`mdlDialog_menuBarDeleteMenu` deletes an entire pull-down menu from a menu bar item.

`mdlDialog_menuBarDetachMenu` removes a pulldown menu from a menu bar.

`mdlDialog_menuBarFind` finds a menu bar in a dialog box.

`mdlDialog_menuBarFindAppMenu` finds the Applications pulldown menu in the command window menu bar.

`mdlDialog_menuBarFindMenu` finds a pulldown menu in a menu bar.

`mdlDialog_menuBarFindItem` finds a particular kind of pull-down menu item based on a search ID.

`mdlDialog_menuBarGetCmdWinP` gets a pointer to a command window menu bar.

`mdlDialog_menuBarGetItem` gets a pointer to a pull-down menu item.

`mdlDialog_menuBarGetMenu` gets a pointer to a pull-down menu.

`mdlDialog_menuBarGetNItems` returns the number of items in a particular pull-down menu.

`mdlDialog_menuBarGetNMenus` returns the number of pulldown menus in a particular menu bar.

`mdlDialog_menuBarGetSelection` retrieves information about the menu item that was last selected from a particular menu bar.

`mdlDialog_menuBarInsertMenu` inserts a pull-down menu into a menu bar.

`mdlDialog_menuBarInsMenu` inserts a pull-down menu into a menu bar.

`mdlDialog_menuBarMenuGetEnabled` gets the state (enabled or disabled) of a menu.

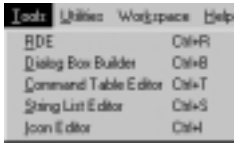
`mdlDialog_menuBarMenuSetEnabled` sets the state (enabled or disabled) of a pull-down menu.

`mdlDialog_menuBarMenuGetTitle` gets the title of a pull-down menu.

`mdlDialog_menuBarMenuSetTitle` sets the title of a pull-down menu.

Text Pull-down Menu

Text pull-down menus can contain items that display as text strings.



These items can have attached submenus. These submenus are indicated with a right-pointing triangle on the left side of the menu item. Text pull-down menus are usually used to activate a command action. Most pull-down menus in MicroStation's Command window are text pull-downs.

Item resource specification

The text pull-down menu item is defined in a resource file with the following structure:

```
typedef struct ditem_pulldownmenuitemrsc
{
#if defined (resource)
   char label[];
   ULong accelerator;
   byte enabled;
   byte mark;
   long subMenuType;
   long submenuId;
   ULong helpInfo;
   char helpTaskId[];
   long pulldownItemHookId;
   long searchId;      /* used to search for item */
   ULong commandNumber;
   char commandTaskId[];
   char unparsed[];
#else
   long labelLength;
```

```
   char label[1];
#endif

} DItem_PulldownMenuItemRsc;

typedef struct ditem_pulldownmenursc
{
   ULong helpInfo;
#if defined (resource)
   char helpSource;
   long pulldownHookId;
   ULong attributes;
   char menuTitle[];
   DItem_PulldownMenuItemRsc pulldownItems[];
#else
   long helpTaskLength;
   char helpTaskId[1];
#endif

} DItem_PulldownMenuRsc;
```

The `DItem_PulldownMenuRsc` structure has the following unique fields. (See "Common item resource fields" for a description of any field not in this table.)

Field	Description
`attributes`	Specifies the attributes of the text pull-down menu. This field will normally be `ON | ALIGN_LEFT` and is constructed by combining the constants from the following table with the logical OR operator:
`menuTitle`	Contains the title of the menu. This field should be set to `""` if the text pull-down menu is a submenu. (In this case, its title cannot display.) Put a tilde '~' before the character in the title that will be the mnemonic. The mnemonic is the key the user presses to activate the menu item when navigating the menu bar with the keyboard.
`pulldownItems`	Represents an array of `DItem_PulldownMenuItemRsc`. See the next table for a description of the `DItem_PulldownMenuItemRsc` structure.

The following table lists possible values for the attributes field (described above) and the meaning of those values:

attributes Value	Meaning
`ON`	The text pull-down menu will be initially enabled.
`OFF`	All subitems in the text pull-down menu will be disabled (dimmed). The title of the menu will also be dimmed (if the pull-down menu is not a submenu).

attributes Value	Meaning
ALIGN_LEFT	Aligns the pull-down menu on the left side of the menu bar. This attribute is ignored for submenus.
ALIGN_RIGHT	Aligns the pull-down menu on the right side of the menu bar. By convention, only a Help menu should be right-aligned. This attribute is ignored for submenus.

The DItem_MenuBarItemRsc structure has the following unique fields. (See "Common item resource fields" for a description of any field not in this table).

Field	Description
label	Specifies the text string that will display as the menu's item. Put a tilde '~' before the character in the title that will be the mnemonic. The mnemonic is the key the user presses to activate the menu item when navigating the menu bar with the keyboard.
accelerator	Specifies the virtual key code to use as the menu item's accelerator. If the dialog box in which this menu resides has the input focus, pressing the accelerator key will activate the menu item.
enabled	Specifies whether the menu item is initially enabled. Usually this field should be ON (enabled).
mark	Specifies the type of graphic mark to place on the left side of the menu item before the text string. This field can have one of the values listed in the mark table below.
subMenuType	Specifies the type of the menu item's submenu. The choices are NOSUBMENU, PulldownMenu, PulldownOptionMenu and PulldownCPickerMenu.
subMenuId	Specifies the resource ID of the menu item's submenu. If no submenu is needed, use 0 or NOSUBMENU.
pulldownItemHookId	Specifies the ID of an item hook function.

Field	Description
searchId	Specifies a user-definable number that should be unique in each menu. Many functions use this number to find a menu item. This number is more useful than specifying a menu item index when the contents of the menu have been dynamically changed.
commandTaskId	Specifies the task that will execute the command associated with commandNumber. Setting comandTaskId to OTASKID, which is #defined to be an empty string, indicates that the task that owns (originally creates) the dialog should execute the command. Setting comandTaskId to MTASKID, which is #defined to be '+', indicates that MicroStation should be used to execute the command. (commandNumber must be a MicroStation command number defined in **cmdlist.h**.) If you need to specify a task different than the owner or MicroStation, put the task name here.

The following table lists possible values for the mark field (described above) and the meaning of those values:

mark Value	Meaning
NOMARK	No mark will display.
MARK_TOGGLE_OUT	A raised toggle button will display.
MARK_TOGGLE_IN	A recessed and filled toggle button will display.
MARK_RADIO_OUT	A raised radio button will display.
MARK_RADIO_IN	A recessed and filled radio button will display.
MARK_RIGHT_ARROW	A right-pointing triangle will display.

The following is an example of a text pull-down menu resource. This item is the MicroStation Command window Element menu, and is defined in MicroStation's resource file.

```
DItem_PulldownMenuRsc PULLDOWNMENUID_Element=
{
   NOHELP, MHELP, NOHOOK, ON | ALIGN_LEFT, "Element",
{
{"~Color",   NOACCEL, ON, NOMARK, PulldownCPickerMenu,
             PULLDOWNCPICKERMENUID_ElementColor, NOHELP,MHELP, NOHOOK,
             MENUSEARCHID_Element_Color, NOCMD, MTASKID,""},
{"~Style",   NOACCEL, ON, NOMARK, PulldownOptionMenu,
             PULLDOWNOPTIONMENUID_ElementStyle, NOHELP,MHELP, NOHOOK,
             MENUSEARCHID_Element_Style, NOCMD, MTASKID,""},
```

```
{"~Weight",  NOACCEL, ON, NOMARK, PulldownOptionMenu,
             PULLDOWNOPTIONMENUID_ElementWeight, NOHELP,MHELP, NOHOOK,
             MENUSEARCHID_Element_Weight, NOCMD, MTASKID,""},

{"~Info",    NOACCEL, ON, NOMARK, 0, NOSUBMENU, NOHELP,MHELP, NOHOOK,
             MENUSEARCHID_Element_Info, CMD_ANALYZE, MTASKID,""},
{"-",        NOACCEL, OFF,NOMARK,0,NOSUBMENU,NOHELP,MHELP,
             NOHOOK, NOARG, NOCMD, MTASKID,""},

{"~Attributes",NOACCEL, ON, NOMARK, 0, NOSUBMENU,NOHELP,MHELP, NOHOOK,
             MENUSEARCHID_Element_Attributes, CMD_DIALOG_ATTRIBUTES,
             MTASKID,""},

{"-",        NOACCEL, OFF,NOMARK,0,NOSUBMENU,NOHELP,MHELP, NOHOOK,
             NOARG,NOCMD, MTASKID,""},

{"~B-splines",NOACCEL, ON, NOMARK, 0, NOSUBMENU,NOHELP,MHELP, NOHOOK,
             MENUSEARCHID_Element_BSplines, CMD_MDL_LOAD, MTASKID,
             "Splines Element"},

{"~Dimensions",NOACCEL, ON, NOMARK, PulldownMenu,
             PULLDOWNMENUID_SubDimension, NOHELP, MHELP, NOHOOK,
             MENUSEARCHID_Element_Dimensions, CMD_DIALOG_DIMPLACE,
             MTASKID,""},

{"~Multi-lines",NOACCEL, ON, NOMARK, 0, NOSUBMENU, NOHELP, MHELP,
             NOHOOK, MENUSEARCHID_Element_MultiLines,
             CMD_DIALOG_MULTILINE, MTASKID, ""},

{"~Text",    NOACCEL, ON, NOMARK, 0, NOSUBMENU,NOHELP,MHELP, NOHOOK,
             MENUSEARCHID_Element_Text, CMD_DIALOG_TEXT, MTASKID,""},
}
};
```

Item hook function messages

The following messages are sent to item hook functions that are attached to text pull-down menus:

DITEM_MESSAGE_DRAW

DITEM_MESSAGE_BUTTON

DITEM_MESSAGE_QUEUECOMMAND

Text pull-down menu specific functions

`mdlDialog_textPDMItemGetInfo` retrieves detailed information about a particular menu item in a text pull-down menu.

`mdlDialog_textPDMItemSetInfo` sets various aspects of a menu item in a text pull-down menu.

`mdlDialog_textPDMItemInsert` inserts a menu item into a text pull-down menu.

`mdlDialog_textPDMItemIns` inserts a menu item into a text pull-down menu.

`mdlDialog_textPDMItemSetEnabled` changes a text pull-down menu item's state (enabled or disabled).

`mdlDialog_textPDMItemSetLabel` sets a text pull-down menu item's text label (the item's displayed text string).

`mdlDialog_textPDMItemSetMark` sets a text pull-down menu item's mark.

Option Pull-down Menu

Option pull-down menus are similar to text pull-down menus, except that they add the capability to include non-text menu items. The currently selected choice is indicated by a right-pointing triangle on the left side of an option menu item. Option pull-down menus can display either text strings or icons, but not both. Option pull-down menus cannot have submenus.

Item resource specification

The option pull-down menu item is defined in a resource file with the following structure:

```
typedef DItem_OptionButtonItemRsc DItem_PulldownOptionItemRsc;
typedef struct ditem_pulldownoptionmenursc
{
   long synonymsId;
   ULong helpInfo;
   ULong helpSource;
   long pulldownHookId;
   ULong attributes;

#if defined (resource)
   char menuTitle[];
   char accessStr[];
   DItem_PulldownOptionItemRsc pulldownOptionItems[];
#else
   long menuTitleLength;
```

```
    char menuTitle[1];
#endif

} DItem_PulldownOptionMenuRsc;
```

The `DItem_PulldownOptionMenuRsc` structure has the following unique fields. (See "Common item resource fields" for a description of any field not in this table.)

Field	Description
`attributes`	Specifies the attributes of the option pull-down menu. This field will normally be `ON | ALIGN_LEFT` and is constructed by combining the constants from the attributes table (below) with the logical OR operator.
`menuTitle`	Contains the title of the menu. It should be set to `""` if the option pull-down menu is a submenu. (In this case, its title cannot display). Put a '~' before the character in the title that will be the mnemonic. The mnemonic is the key the user presses to display the menu when navigating the menu bar with the keyboard.
`pulldownOptionItems`	Represents an array of `DItem_PulldownOptionItemRsc`. This structure is the same as the `DItem_OptionButtonItemRsc` structure. See the description of the option button item for more information on the fields of this structure.

The following table lists possible values for the attributes field (described above) and the meaning of those values:

attributes Value	Meaning
`ON`	The option pull-down menu will be initially enabled.
`OFF`	All subitems in the option pull-down menu will be disabled (dimmed). The title of the menu will also be dimmed (if the pull-down menu is not a submenu).
`ALIGN_LEFT`	This attribute is ignored for submenus. The pull-down menu will be aligned on the left side of the menu bar.
`ALIGN_RIGHT`	This attribute is ignored for submenus. The pull-down menu will be aligned on the right side of the menu bar. By convention, only a Help menu should be right aligned.

The following is an example of an option pull-down menu resource. This item is the MicroStation Command window Element Style submenu, and is defined in MicroStation's resource file.

```
DItem_PulldownOptionMenuRsc PULLDOWNOPTIONMENUID_ElementStyle=
{
   NOSYNONYM, HELPID_TEXT_ElementStyle, MHELPTOPIC,
   HOOKITEMID_StylePulldown_CmdWindow, ON | ALIGN_LEFT,
   "~Line Style", "",
{
{Icon, ICONID_LineStyle0, CMD_ACTIVE_STYLE_CSELECT, MCMD, 0, NOMASK,
ON, "~0"},
{Icon, ICONID_LineStyle1, CMD_ACTIVE_STYLE_CSELECT, MCMD, 0, NOMASK,
ON, "~1"},
{Icon, ICONID_LineStyle2, CMD_ACTIVE_STYLE_CSELECT, MCMD, 0, NOMASK,
ON, "~2"},
{Icon, ICONID_LineStyle3, CMD_ACTIVE_STYLE_CSELECT, MCMD, 0, NOMASK,
ON, "~3"},
{Icon, ICONID_LineStyle4, CMD_ACTIVE_STYLE_CSELECT, MCMD, 0, NOMASK,
ON, "~4"},
{Icon, ICONID_LineStyle5, CMD_ACTIVE_STYLE_CSELECT, MCMD, 0, NOMASK,
ON, "~5"},
{Icon, ICONID_LineStyle6, CMD_ACTIVE_STYLE_CSELECT, MCMD, 0, NOMASK,
ON, "~6"},
{Icon, ICONID_LineStyle7, CMD_ACTIVE_STYLE_CSELECT, MCMD, 0, NOMASK,
ON, "~7"},
{Icon, ICONID_LineStyleCustom, CMD_DIALOG_LSTYLESETUP, MCMD, 0, NOMASK,
ON, "~c"},
{Icon, ICONID_LineStyleEdit, CMD_DIALOG_LSTYLEEDIT, MCMD, 0, NOMASK,
ON, "~e"},
}
};
```

Item hook function messages

The following messages are sent to item hook functions that are attached to option pull-down menus:

DITEM_MESSAGE_CREATE

DITEM_MESSAGE_DESTROY

DITEM_MESSAGE_BUTTON

DITEM_MESSAGE_GETSTATE

DITEM_MESSAGE_SETSTATE

DITEM_MESSAGE_STATECHANGED

DITEM_MESSAGE_QUEUECOMMAND

Option pull-down menu functions

`mdlDialog_optionPDMItemGetInfo` retrieves detailed information about a particular menu item in an option pull-down menu.

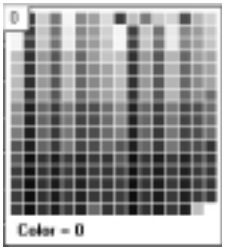
`mdlDialog_optionPDMItemSetInfo` sets various aspects of a menu item in an option pull-down menu.

`mdlDialog_optionPDMItemInsert` inserts a menu item in an option pull-down menu.

`mdlDialog_optionPDMItemSetEnabled` changes an option pull-down menu item's state (enabled or disabled).

Color Picker Pull-down Menu

The color picker pull-down menu acts in a similar manner to the standard color picker item that is placed in the content area of dialog boxes. It selects a current color index. The color picker pull-down menu cannot have submenus.



Item resource specification

The color picker pull-down menu item is defined in a resource file with the following structure:

```
typedef struct ditem_pulldowncpickermenursc
  {
  long synonymsId;
  ULong helpInfo;
  ULong helpSource;
  long pulldownHookId;
  ULong attributes;
  ULong commandNumber;
  ULong commandSource;
  ULong mask;

#if defined (resource)
  char menuTitle[];
```

```
    char accessStr[];
#else
    long menuTitleLength;
    char menuTitle[1];
#endif

} DItem_PulldownCPickerMenuRsc;
```

The `DItem_PulldownCPickerMenuRsc` structure has the following unique fields. (See "Common item resource fields" for a description of any field not in this table.)

Field	Description
`attributes`	Specifies the attributes of the color picker pull-down menu. It will normally be `ON | ALIGN_LEFT` and is constructed by combining the constants from the attributes table (below) with the logical OR operator.
`mask`	Indicates which bits of the variable specified by `accessStr` are used to determine the current color index. Usually, `mask` should be set to `NOMASK` (defined to be `0xFFFFFFFF`), which indicates that the entire variable will be used to determine the index.
`menuTitle`	Contains the title of the menu. This field should be set to `""` if the color picker pull-down menu is a submenu. (In this case, its title cannot display). Put a '~' before the character in the title that will be the mnemonic. The mnemonic is the key the user presses to display the menu when navigating the menu bar with the keyboard.

The following table lists possible values for the attributes field (described above) and the meaning of those values:

attributes Value	Meaning
`ON`	The color picker pull-down menu will be initially enabled.
`OFF`	All subitems in the color picker pull-down menu will be disabled (unselectable). The title of the menu will also be dimmed (if the pull-down menu is not a submenu).
`ALIGN_LEFT`	This is the default setting and should always be set. The pull-down menu will be aligned on the left side of the menu bar.

The following is an example of a color picker pull-down menu resource. This item is the MicroStation Command window Element Color submenu, and is defined in MicroStation's resource file.

```
DItem_PulldownCPickerMenuRsc PULLDOWNCPICKERMENUID_ElementColor=
{
   NOSYNONYM, NOHELP, MHELP, HOOKITEMID_ColorPicker_CmdWindow, ON,
   CMD_ACTIVE_COLOR_CSELECT, MCMD, NOMASK, "", "tcb->symbology.color"
};
```

Item hook function messages

The following messages are sent to item hook functions that are attached to color picker pull-down menus:

DITEM_MESSAGE_CREATE

DITEM_MESSAGE_DESTROY

DITEM_MESSAGE_BUTTON

DITEM_MESSAGE_GETSTATE

DITEM_MESSAGE_SETSTATE

DITEM_MESSAGE_STATECHANGED

DITEM_MESSAGE_QUEUECOMMAND

Color picker pull-down menu functions

There are no color picker pull-down menu functions.

Tool Palettes

Tool palettes contain icons that, when activated, add a command to MicroStation's input queue. There are two types of MicroStation's tool palettes: icon command frames and icon command palettes.

Icon command palettes can contain only icon commands. These icon commands can have associated items. These items will display when the icon is selected and are the icon's popdown items. (See the MicroStation User's Guide for an explanation of these items).

Icon command frames can contain icon commands or icon command palettes. The icon commands that are directly in a command frame cannot have associated item lists. Each icon command palette in a command frame has a current icon, the icon command from the palette that is displayed in the parent icon command frame. When

the data button is pressed while the cursor is over a icon command palette in a command frame, the full palette is popped up. The user can then select an icon command from the palette, or tear off the palette. (To tear off a palette, drag the cursor off from the palette until an outline of the palette displays. The palette can then be placed anywhere on the screen).

Icon command frame

An icon command frame should be the only item in a dialog box. Icon command frames should usually be vertically oriented (which means that they have more rows than columns).

Item list specification

The `DialogItemRsc` field `extent` is unused for icon command frame items. The dialog box that contains an icon command frames is resized so the command frame fills the dialog box. The `extent` should be set to `{0, 0, 0, 0}`.

The `type` field should be `IconCmdFrame`.

The `ID` determines the `DItem_IconCmdFrameRsc` instance to load.

The `attributes` are ignored for icon command frame items. The icon command frame is always enabled and displayed.

The `itemArg` field is unused and should usually be set to 0.

The `label` field is unused and should be set to "".

The `auxInfo` field is unused and should be set to "".

The following is an example of a icon command frame item list specification:

```
{{0, 0, 0, 0}, IconCmdFrame, ICONCMDFRAMEID_Main, ON, 0, "", ""}
```

Item resource specification

The icon command frame item is defined in a resource file with the following structure:

```
typedef struct ditem_iconcmdframersc
{
   int nColumns;
   int nRows;
   ULong helpInfo;
   ULong helpSource;

#if defined (resource)
```

```
    char label[];
    DialogItemSpec iconPieces[];
#else
    long labelLength;
    char label[1];
#endif

} DItem_IconCmdFrameRsc;
```

The DItem_IconCmdFrameRsc structure has the following unique fields. (See "Common item resource fields" for a description of any field not in this table).

Field	Description
nColumns	Specifies the number of columns in the icon command frame.
nRows	Specifies the number of rows in the icon command frame.
label	Specifies the title of the icon command frame. The dialog box title will be set to this string's value.
iconPieces	Specifies the contents of the icon command frame. iconPieces is an array of DialogItemSpec members. The type field of an iconPieces member can be either IconCmd or IconCmdPalette. The id field of an iconPieces member indicates the DItem_IconCmdRsc or DItem_IconCmdPaletteRsc instance to load.

The following is an example of a icon command frame item resource. This item is the MicroStation Main tool box, and is defined in MicroStation's resource file.

```
DItem_IconCmdFrameRsc ICONCMDFRAMEID_Main=
{
   2, 10, NOHELP, MHELP, "Main",
   {
        {IconCmd, ICONCMDID_ChooseElement},
        {IconCmdPalette, ICONCMDPALETTEID_Line},
        {IconCmdPalette, ICONCMDPALETTEID_Arc},
        {IconCmdPalette, ICONCMDPALETTEID_LineString},
        {IconCmd, ICONCMDID_PlaceOpenBSpline},
        {IconCmdPalette, ICONCMDPALETTEID_Polygon},
        {IconCmdPalette, ICONCMDPALETTEID_Fillet},
        {IconCmdPalette, ICONCMDPALETTEID_Circle},
        {IconCmdPalette, ICONCMDPALETTEID_Point},
        {IconCmdPalette, ICONCMDPALETTEID_Cell},
        {IconCmdPalette, ICONCMDPALETTEID_EnterData},
        {IconCmdPalette, ICONCMDPALETTEID_Text},
        {IconCmdPalette, ICONCMDPALETTEID_MirrorElement},
        {IconCmdPalette, ICONCMDPALETTEID_CopyElement},
        {IconCmdPalette, ICONCMDPALETTEID_Change},
        {IconCmdPalette, ICONCMDPALETTEID_Modify},
```

```
        {IconCmdPalette, ICONCMDPALETTEID_Chain},
        {IconCmdPalette, ICONCMDPALETTEID_Drop},
        {IconCmdPalette, ICONCMDPALETTEID_View},
        {IconCmd, ICONCMDID_DeleteElement},
    }
};
```

Item hook function messages

The icon command frame item cannot have an associated item hook function.

Icon cmd frame item functions

`mdlDialog_icFrameGetItemInfo` retrieves detailed information about an item in an icon command frame.

`mdlDialog_icFrameSetItemInfo` sets various aspects of an item in an icon command frame.

`mdlDialog_icFrameGetNItems` returns the number of items in an icon command frame.

`mdlDialog_icFrameGetNSubItems` returns the number of subitems in an icon command frame palette item.

`mdlDialog_icFrameSelectIcon` selects (highlights) an icon in an icon command frame/palette.

Icon command palette

Icon command palettes are different from other items in that no dialog box is associated with them when their resources are defined. In fact, they cannot be placed directly inside a dialog box by including them in dialog item list specification. Instead only an icon command palette resource is created in a resource file. The special function `mdlDialog_openPalette` is then used to display the palette. At that point, the dialog box is automatically created on-the-fly to contain the specified icon command palette.

Icon command palettes should usually be horizontally oriented. (They should have more columns than rows.) Horizontally oriented palettes simplify laying out popdown items in the palette's icon commands.

Item resource specification

The icon command palette item is defined in a resource file with the following structure:

```
typedef struct ditem_iconcmdpalettersc
{
   int nColumns;
   int nRows;
   int defaultItem;
   ULong helpInfo;
   ULong helpSource;
   long itemHookId;
   long itemHookArg;
#if defined (resource)
   char label[];
   int iconCmds[];
#else
   long labelLength;
   char label[1];
#endif

} DItem_IconCmdPaletteRsc;
```

The `DItem_IconCmdPaletteRsc` structure has the following unique fields. (See "Common item resource fields" for a description of any field not in this table).

Field	Description
`nColumns`	Specifies the number of columns in the icon command palette.
`nRows`	Specifies the number of rows in the icon command palette.
`defaultItem`	Specifies the index of the icon command to display when the icon command palette is contained in an icon command frame.
`label`	Specifies the title of the icon command palette. When the icon palette displays in its dialog box, this string sets the dialog box's title.
`iconCmds`	Specifies the contents of the icon command palette. `iconCmds` is an array of `DItem_IconCmdRsc` resource ids. This array determines the `DItem_IconCmdRsc` instances to load. The number of entries in the `iconCmds` array can be less than `nRows * nColumns`, in which case the empty slots in the palette are left blank when the palette is displayed.

The following is an example of an icon command palette item resource. This item is the MicroStation Lines tool box, and is defined in MicroStation's resource file.

```
DItem_IconCmdPaletteRscICONCMDPALETTEID_Line=
{
   8, 2, 0, NOHELP, MHELP, NOHOOK, NOARG, "Lines",
   {
        ICONCMDID_PlaceLine,
        ICONCMDID_PlaceLineAA,
        ICONCMDID_ConstBisectorLine,
        ICONCMDID_ConstBisectorAngle,
```

```
            ICONCMDID_PlaceTerminator,
            ICONCMDID_ConstLineMin,
            ICONCMDID_ConstTanPerp,
            ICONCMDID_ConstTanBetween,
            ICONCMDID_ConstLineAAToPt,
            ICONCMDID_ConstLineAAFromPt,
            ICONCMDID_ConstLineAAToPtK,
            ICONCMDID_ConstLineAAFromPtK,
            ICONCMDID_ConstPerpToElem,
            ICONCMDID_ConstPerpFromElem,
            ICONCMDID_ConstTanToElem,
            ICONCMDID_ConstTanFromElem,
    }
};
```

Item hook function messages

No messages are sent to the item hook function.

Icon cmd palette item functions

`mdlDialog_icPaletteGetItemInfo` retrieves detailed information about a item in an icon command palette.

`mdlDialog_icPaletteSetItemInfo` sets various aspects of an item in an icon command palette.

`mdlDialog_icPaletteGetNItems` returns the number of items in an icon command palette.

`mdlDialog_icPaletteSelectIcon` selects (highlights) an icon in an icon command palette.

Icon command

Icon command items cannot be directly placed in a dialog box item list. They must be referenced by an icon command palette item or an icon command frame item.

Each `DItem_IconCmdRsc` structure has associated `IconCmdSmallRsc` and `IconCmdLargeRsc` instances. These resources contain the icon's actual raster bitmaps. These three resources are connected by making their resource IDs identical. For example, if an instance of `DItem_IconCmdRsc` has an ID of 5, the IDs of the corresponding `IconCmdSmallRsc` and `IconCmdLargeRsc` should also be 5.

For consistency with MicroStation's icon commands, small icons should be 23 pixels wide by 23 pixels high, and large icons should be 31 pixels wide by 31 pixels high.

Item resource specification

The icon command item is defined in a resource file with the following structure:

```
typedef struct ditem_iconcmdrsc
{
   ULong helpInfo;

#if defined (resource)
   char helpTaskId[];
   ULong attributes;
   ULong commandNumber;
   char commandTaskId[];
   char unparsed[]
   char enabledAccessStr[];
   DialogItemRsc iconItems[];
#else
   long helpTaskLen;
   char helpTaskId[1];
#endif

} DItem_IconCmdRsc;
```

The `DItem_IconCmdRsc` structure has the following unique fields. (See "Common item resource fields" for a description of any field not in this table).

Field	Description
`attributes`	Specifies the attributes of the icon command. This field will normally be 0 and is constructed by combining the constants from the attributes table (below) with the logical OR operator.
`commandTaskId`	Specifies the task that will execute the command associated with `commandNumber`. Setting `comandTaskId` to `OTASKID`, which is `#defined` to be an empty string, indicates that the task that owns (originally creates) the dialog should execute the command. Setting `comandTaskId` to `MTASKID`, which is `#defined` to be "+", indicates that MicroStation should be used to execute the command. (`commandNumber` must be a MicroStation command number defined in cmdlist.h.) If you need to specify a task different than the owner or MicroStation, put the task name here.

Field	Description
enabledAccessStr	Is currently unused and should be set to "".
iconItems	An array of DialogItemRsc and the list of popdown dialog items associated with the icon command. This list is created the same way that a normal dialog item list is created. See the "DialogBoxRsc Structure" and "DialogItemRsc Structure" sections in this chapter for more information on specifying dialog item lists. Only icon commands included in icon command palettes can have an item list. Icon command frames will not display popdown item lists. When used in an icon command item list specification, the extent member of DialogItemRsc specifies positions relative to the bottom left of the parent icon command palette, not the upper left corner of the dialog box containing the icon command.

attributes Value	Meaning
ICONCMDATTR_DONTSINGLESHOT	Indicates that this icon command can not be double clicked upon to enter single-shot operational mode. By default, double-clicking on an icon causes MicroStation to enter single-shot mode for the selected icon.
ICONCMDATTR_DONTAUTOSELECT	Indicates that this icon resource should not be highlighted (selected) when the command associated with the icon is queued to MicroStation.

The following is an example of an icon command resource. This is the MicroStation icon command for the keyin PLACE LINE ANGLE and can be found in the Lines tool box. It is defined in MicroStation's resource file.

```
DItem_IconCmdRsc ICONCMDID_PlaceLineAA=
{
   NOHELP, MHELP, 0, CMD_PLACE_LINE_ANGLE, MTASKID, "", "",
   {
      {{16*XC, GENY(1), 10*XC, 0}, Text, TEXTID_ActiveAngle,ON,0,"","},
   }
};

IconCmdSmallRsc ICONCMDID_PlaceLineAA=
{
   23, 23, FORMAT_MONOBITMAP, BLACK_INDEX, "Pl Line At AA",
   {
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x00, 0x08, 0x00, 0x00,
        0x20, 0x00, 0x00, 0x80, 0x00, 0x02, 0x00, 0x00,
        0x08, 0x00, 0x00, 0x20, 0x00, 0x00, 0x80, 0x00,
```

```
        0x02, 0x00, 0x00, 0x08, 0x82, 0x00, 0x21, 0x04,
        0x00, 0x85, 0x1c, 0x02, 0x0a, 0x28, 0x08, 0x3e,
        0xf8, 0x20, 0x45, 0x10, 0x80, 0x00, 0x01, 0x6d,
        0xb6, 0xc0, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00, 0x00,
    }
};

IconCmdLargeRsc ICONCMDID_PlaceLineAA=
{
    31, 31, FORMAT_MONOBITMAP, BLACK_INDEX, "Pl Line At AA",
    {
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x01, 0x00, 0x00, 0x00, 0x04, 0x00, 0x00,
        0x00, 0x10, 0x00, 0x00, 0x00, 0x40, 0x00, 0x00,
        0x10, 0x00, 0x00, 0x00, 0x40, 0x00, 0x00, 0x01,
        0x00, 0x00, 0x00, 0x04, 0x60, 0x60, 0x00, 0x10,
        0xc0, 0xc0, 0x00, 0x41, 0x81, 0x80, 0x01, 0x04,
        0x84, 0x80, 0x04, 0x09, 0x09, 0x00, 0x10, 0x12,
        0x12, 0x00, 0x40, 0x7e, 0x7e, 0x01, 0x00, 0x84,
        0x84, 0x04, 0x01, 0x09, 0x08, 0x10, 0x00, 0x00,
        0x00, 0x40, 0x00, 0x00, 0x00, 0x8c, 0xcc, 0xcc,
        0xc0, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x00,
    }
};
```

Item hook function messages

The icon command item cannot have an associated item hook function.

Icon cmd item functions

`mdlDialog_selectIconsById` changes the selection state of an icon by its resource ID.

`mdlDialog_selectIconsByIdNoMsg` changes the selection state of an icon by its resource ID but does not display the command message inthe command window.

`mdlDialog_selectIconsByCmd` changes the selection state of an icon by its command number.

`mdlDialog_selectIconsByCmdNoMsg` changes the selection state of an icon by its command number but does not display the command message inthe command window.

Palettes vs. tool boxes

The default MicroStation user interface uses tool boxes rather than icon command palettes. Still, palettes remain completely supported in MicroStation, but do not have the advanced features of tool boxes.

As in the Microsoft Office products, tool boxes are resizable and dockable. Additionally, tool boxes support tool tip and tool description help and allow items to be grouped. To support docking and to preserve screen space, tool boxes do not have pop-down fields. The Tool Settings dialog is used for this purpose.

Application developers are encouraged, but not required, to switch to tool boxes.

Tool Boxes

Notice that a tool box must be contained in a dialog box definition and that icon command resources can be shared between MDL applications using the “owner=” specification.

Example of a resized tool box.

```
DialogBoxRsc TOOLBOXID_SmartTools=
{
  DIALOGATTR_TOOLBOXCOMMON, 0, 0, NOHELP, MHELP, NOHOOK,
  NOPARENTID, "",
  {
    {{0, 0, 0, 0}, ToolBox, TOOLBOXID_SmartTools, ON, 0, "", ""},
  }
};

DItem_ToolBoxRsc TOOLBOXID_SmartTools=
{
  NOHELP, MHELPTOPIC, NOHOOK, NOARG, 0, TXT_SmartToolsTitle,
  {
    {{0, 0, 0, 0}, IconCmd,ICONCMDID_SmartLine, ON, 0, "",
     "owner=\"SMRTLINE\""},
    {{0, 0, 0, 0}, IconCmd, ICONCMDID_DropElement, ON, 0, "",
     "owner=\"DROP\""},
    {{0, 0, 0, 0}, IconCmd, ICONCMDID_MatchElement, ON, 0, "",
     "owner=\"MATCH\""},
    {{0,0,0,0},IconCmd,ICONCMDID_DeleteElement,ON,1,"",""},
  }
};
```

Tool boxes utilize the exact same icon command resources (`DItem_IconCmdRsc`) used by palettes. No icon rework is necessary. To add tool tip and tool description information, an extended attributes resource must be added.

```
/* IconCmd resource w/ extended attributes to implement tool tip help */
DItem_IconCmdRsc ICONCMDID_PlaceMyLine=
{
  NOHELP, OHELPTASKIDCMD, 0, CMD_PLACE_LINE, MTASKID, "", "",
  {
  }
}

extendedAttributes
{{
{EXTATTR_FLYTEXT,TXT_Flyover_PlaceMyLine}, /* Appears in statusarea */
  {EXTATTR_BALLOON, TXT_Balloon_PlaceMyLine}, /* Appears in pop-up */
}};
```

For a frame to include a tool box, the extended frame resource (`DItem_IconCmdFrameXRsc`) must be used. Note that a tool box containing items other than icon command resources cannot be included in a frame.

```
/* Sample Tool Frame Dialog Box - IconCmdFrameX is the only item */
DialogBoxRsc DIALOGID_SampleToolFrame=
{
  DIALOGATTR_DEFAULT | DIALOGATTR_NORIGHTICONS, 0, 0, NOHELP, MHELP,
  NOHOOK, NOPARENTID, TXT_SampleFrame,
  {
{{0,0,0,0}, IconCmdFrameX, ICONCMDFRAMEID_MainSample, ON, 0, "", ""},
  }
};

DItem_IconCmdFrameXRsc ICONCMDFRAMEID_MainSample=
{
  2, 2, NOHELP, MHELP, 0, TXT_SampleFrame,
  {
    {ToolBox, TOOLBOXID_Sample, "V550NEW"},
    {ToolBox, TOOLBOXID_Polygons, MTASKID},
    {ToolBox, TOOLBOXID_Ellipses, MTASKID},
    {ToolBox, TOOLBOXID_Tags, MTASKID},
  }
};
```

Converting Icon Palettes to Tool Boxes

To illustrate a simple case of conversion from an icon command palette to a tool box, use the Lines-Sample icon command palette from the PALETTE MDL application. This command palette is delivered with MicroStation V5 MDL examples.

```
/* Resource definition of the icon command palette shown below */
DItem_IconCmdPaletteRsc ICONCMDPALETTEID_LineSample=
{
  7, 1, 0, NOHELP, MHELP, NOHOOK, NOARG, TXT_LinesSample,
  {
    ICONCMDID_SamplePlaceLine,
    ICONCMDID_SamplePlaceLineAA,
    ICONCMDID_SampleConstBisectLine,
    ICONCMDID_SampleConstBisectAngle,
    ICONCMDID_SampleConstLineMin,
    ICONCMDID_SampleConstTanPerp,
    ICONCMDID_SampleConstTanBetween,
  }
};
```

Resource Definition of the icon command palette from PALETTE.R.

The first step in the conversion is creating a dialog box resource definition for the tool box. The dialog box resource contains one item, the tool box. Note that the dialog box and tool box resources have the same resource identifier. Since tool boxes are really just dialog boxes, you might want to differentiate dialog boxes from tool boxes in your application. This can be accomplished by numbering your dialog boxes and tool boxes with a different series of numbers. For example, start numbering dialog boxes with resource identifier 100 and tool boxes with resource identifier 200.

```
DialogBoxRsc TOOLBOXID_LineSample=
{
  /* tool box default attributes, (see dlogbox.h) */
  DIALOGATTR_TOOLBOXCOMMON,0,0,NOHELP,MHELP,NO,NOPARENTID,TXT_Sample,
  {
    {{0, 0, 0, 0}, ToolBox, TOOLBOXID_LineSample, ON, 0, "", ""},
  }
};
```

Next, define the tool box resource by populating the `DItem_ToolBoxRsc` structure. This structure is defined in dlogbox.h. The items contained in the tool box are declared in a manner similar to declaring items within a dialog box, that is, by populating the `DialogItemRsc` structure.

The auxiliary information field, (the last parameter) is `NULL` when reusing MicroStation PowerDraft's icon commands, but contains the string: `"owner=\"PALETTE\""` for the icon commands that belong to other MDL applications. The string, in quotes, that follows the "`owner=`" string is the task identifier of the application that owns the icon command. Since our PALETTE application owns the icon commands in this tool box, we could have left the auxiliary information field blank. We chose to fill in the field

with "owner="PALETTE"" to show how we might reuse an icon command that exists in some other MDL application. The only instance necessary to fill in this field is when an application, other than MicroStation or your MDL application, owns the icon command.

The item argument field of the DialogItemRsc structure can be thought as a group index. If all items in the dialog should be part of the same group, then assign the same value to all instances of the item argument field. If the items should be in different groups, then assign the same item argument value to all items within the group. The group index should be defined in the dialog item list structure in ascending order.

```
DItem_ToolBoxRsc TOOLBOXID_LineSample=
{
  /* DItem_ToolBoxRsc structure (see dlogbox.h) */
  NOHELP, MHELPTOPIC, NOHOOK, NOARG, 0, TXT_Sample,
  {
    /* DialogItemRsc structure (see dlogbox.h) */
    {{0, 0, 0, 0}, IconCmd, ICONCMDID_SamplePlaceLine, ON, 0, "",
      "owner=\"PALETTE\""},
    {{0, 0, 0, 0}, IconCmd, ICONCMDID_SamplePlaceLineAA, ON, 0, "",
      "owner=\"PALETTE\""},
    {{0, 0, 0, 0}, IconCmd, ICONCMDID_SampleConstBisectLine, ON, 1, "",
      "owner=\"PALETTE\""},
    {{0,0,0,0}, IconCmd, ICONCMDID_SampleConstBisectAngle, ON, 1, "",
      "owner=\"PALETTE\""},
    {{0, 0, 0, 0}, IconCmd, ICONCMDID_SampleConstLineMin, ON, 1, "",
      "owner=\"PALETTE\""},
    {{0, 0, 0, 0}, IconCmd, ICONCMDID_SampleConstTanPerp, ON, 2, "",
      "owner=\"PALETTE\""},
    {{0, 0, 0, 0}, IconCmd, ICONCMDID_SampleConstTanBetween, ON, 2, "",
      "owner=\"PALETTE\""},
  }
};
```

With the above tool box resource definition, we have the tool boxes shown below:

Tool box sized horizontally.

Tool box sized vertically.

There are two major differences between an icon command palette and a tool box:

- Tool boxes can be resized, (shown above) and docked to the edges of the application window.

- All pop-down items that you may have associated with your icon command will always be displayed in the tool settings window in MicroStation PowerDraft.

Adding a tool box

Now that you have a handle on the tools necessary to build a tool box, let's look at the more complex case of incorporating a tool box into an icon command frame in place of an icon command palette. The declaration of the tool box to be added to the icon command frame is the same as the aforementioned declaration, but the tool box can only contain icon commands.

Convert your icon command palettes to tool boxes as demonstrated above. Then change your icon command frame resource to an extended icon command frame resource. Before modifying the PALETTE application, you should have the icon command frame with the resource definition below.

```
/* Resource definition for sample icon command frame from PALETTE */
DialogBoxRsc DIALOGID_MainSampleFrame=
{
  DIALOGATTR_DEFAULT | DIALOGATTR_NORIGHTICONS,
  XC, YC, NOHELP, MHELP, HOOKDIALOGID_PaletteHook, NOPARENTID,
TXT_Sample,
  {
    {{0,0,0,0}, IconCmdFrame, ICONCMDFRAMEID_MainSample,ON,0,"","},
  }
};

DItem_IconCmdFrameRsc ICONCMDFRAMEID_MainSample=
{
  1, 4, NOHELP, MHELP, TXT_Sample,
  {
    {IconCmd, ICONCMDID_SampleChooseElement},
    /* icon command palette to convert to a tool box */
    {IconCmdPalette, ICONCMDPALETTEID_LineSample},
    {IconCmdPalette, ICONCMDPALETTEID_ArcSample},
    {IconCmd, ICONCMDID_SampleDeleteElement},
  }
};
```

Figure 16.5 Sample icon command frame from PALETTE application.

To add a tool box when modifying the icon command frame definition, the dialog box definition for the icon command frame will be basically the same—except that instead of including an `IconCmdFrame` in the dialog box definition, include an extended version of the `IconCmdFrame` resource, `IconCmdFrameX`.

```
DialogBoxRsc DIALOGID_MainSampleFrame=
{
  DIALOGATTR_DEFAULT | DIALOGATTR_NORIGHTICONS, XC, YC,
  NOHELP, MHELP, HOOKDIALOGID_PaletteHook, NOPARENTID, TXT_Sample,
  {
    {{0,0,0,0},IconCmdFrameX,ICONCMDFRAMEID_MainSample,ON,0,“”,“”},
  }
};
```

The changes made to the icon command frame (in bold) allow us to add a tool box to this frame. Note, there are two additional parameters in the extended resource

- The attribute field, which is reserved for future use and should be zero.
- The owner task identifier, which is the MDL task identifier of the application that owns the icon command palette or icon command. When specifying ownership of the items on a frame, it is best to specify the task explicitly, rather than using `OTASKID`.

```
DItem_IconCmdFrameXRscICONCMDFRAMEID_MainSample=
{
  /* DItem_IconCmdFrameXRsc (see dlogbox.h) column rows help help
     attributes label identifier source */
  1, 4, NOHELP, MHELP, 0, TXT_Sample,
  {
    /* DialogItemSpecX (see dlogbox.h) item type item identifier owner
       owner task identifier */
    {IconCmd, ICONCMDID_SampleChooseElement, “PALETTE”},
    {ToolBox, TOOLBOXID_LineSample, “PALETTE”},
    {IconCmdPalette, ICONCMDPALETTEID_ArcSample, “PALETTE”},
    {IconCmd, ICONCMDID_DeleteElement, “”},
  }
};
```

For this example above, we have only converted the first icon command palette that exists in this icon command frame. We have also changed the last icon command in the frame from `ICONCMDID_SampleDeleteElement` to `ICONCMDID_DeleteElement` to show that you can reuse one of MicroStation PowerDraft's icon commands (notice that the owner task identifier is a `NULL` string).

Incorporating tool tips and descriptions

While no changes to icon commands are required to make them part of the tool box, you can incorporate Tool Tips (balloon help) and Tool Descriptions (flyover help) into your icon commands. Balloon help is displayed in a yellow text box near a toolbox item when you place the pointer on top of the item, and the pointer is stationary for about 0.5 seconds. Flyover help is displayed in MicroStation PowerDraft's status area when the balloon help is displayed over an item or when the pointer passes over (flies over) the icon command.

Notice that we have extended the resource definition for the icon command to include the `extendedAttributes` information, rather than create a new resource type to include this information. The resource compiler and dialog manager have been enhanced to read and process this extended information for dialog items. By extending resource definitions, we can maintain backward compatibility with existing dialog items that you might include in a tool box, rather than creating an `IconCmdX` resource, for instance. If you add dialog items to your toolbox without adding the extended attribute information, then the items will display and operate properly, but you will not have the balloon and flyover help features enabled.

When creating the text strings for balloon and flyover help, follow these guidelines:

- For balloon help (or tool tip), use the command name if the string is not too long for the yellow box. We recommend that the string length should not exceed 30 to 40 characters. Capitalize the first letter of each word. The maximum string length for balloon help is 80 characters.
- For flyover help (or tool description), use a *one-liner* that describes the command. Capitalize the first word in the sentence and do not end the sentence with a period. The maximum string length for flyover help is 80 characters.
- Balloon and flyover help are only supported for:

  Icon commands that exist in icon command frames or tool boxes

  Dialog items that exist in tool boxes

- Use balloon help for a dialog item in a tool box to replace the item's label. You should not use a label for a dialog item that is placed in a tool box.

```
/* Icon command resource definition, which includes extended
   definition for balloon and flyover help */
DItem_IconCmdRsc ICONCMDID_SamplePlaceLine=
{
  NOHELP, OHELPTASKIDCMD, 0, CMD_PLACE_LINE, MTASKID, "", "",
  {
  }
```

```
}

extendedAttributes
{{
  /* extended attribute type Text for balloon and flyover help */
  {EXTATTR_FLYTEXT, TXT_Flyover_SamplePlaceLine},
  {EXTATTR_BALLOON, TXT_Balloon_SamplePlaceLine},
}};
```

In the above example, the following definitions are declared in the header file, palettxt.h.

```
#define TXT_Balloon_SamplePlaceLine "Place Sample Line"
#define TXT_Flyover_SamplePlaceLine "Place sample line flyover help"
```

Figure 16.6 Place Sample Line flyover help.

It really is easy to convert existing icon command palettes to tool boxes. And with this knowledge, you can quickly and easily blend your application into MicroStation PowerDraft or the next MicroStation upgrade.

Tool box summary

With the introduction of MicroStation PowerDraft, tool boxes effectively replace palettes. Tool boxes have many advantages over palettes. Tool boxes are:

- Resizable
- Dockable
- Consistent with MicroSoft Office products

Additionally, tool boxes:

- May contain items other than icon command items.
- Support fly-over help.
- Allow commands to be grouped.

Some special rules apply to tool boxes:

- A tool box must be contained in a dialog box resource definition.

- A tool box does not have slam-down fields, (all slam-downs display in the Tool Settings dialog).
- For a frame to include a tool box, the extended frame resource (`DItem_IconCmdFrameXRsc`) must be used.
- Tool boxes which contain items other than icon command resources cannot be included in a frame.

The format of the tool box resource is as follows, (from dlogbox.h):

```
typedef struct ditem_toolboxrsc
{
    ULong    helpInfo;
    ULong    helpSource;
    long     itemHookId;
    long     itemHookArg;
    ULong    attributes;    /* incl. groupbox, resource, folder */
#if defined (resource)
    char     label[];            /* Default Label, used for groupBox and
                                    Folder types */
    DialogItemRsc itemList[];/* List of items in this container */
#else
    long labelLength;
    char label[1];
#endif
} DItem_ToolBoxRsc;
```

Converting palettes to tool boxes

Conversion of palettes to tool boxes is straightforward. The tool box's resource definition contains a dialog item resource for each item in the tool box.

➤ **To convert each existing palette to a tool box by following these steps**

1. Create a tool box definition, using the format in dlogbox.h.
2. For each icon command resource in the existing palette, create a dialog item resource in the tool box item list.
3. Create a dialog box definition and assign it the same resource id as the new tool box.
4. List only the tool box item in the new dialog box's item list.

The icon command resources do not require modification.

Although palettes will continue to be supported, their appearance will only confuse our customers. Users would be frustrated to not be able to dock and resize palettes.

Icon Command Resources

In MicroStation Version 4 and Version 5, it is possible for tool settings not to display properly. This is especially true when using key-ins, a tablet/paper menu or commands tagged to function keys.

The problem stemmed from a design inconsistency/flaw in `IconCmdRscs`. Since the tool settings item list is associated to the icon, (instead of the command that the icon executes), the tool settings could only be found if the icon was in memory (displayed). Users of tablet/paper menus found this problem because they typically keep their screens free of palettes/tool boxes since all of the commands are right on the menu.

This problem is resolved in MicroStation with the introduction of a new resource type, `CmdItemListRsc`. Since it is command based, (command numbers are used as the IDs) `CmdItemListRsc`'s let MicroStation find the tool settings for the current command no matter how the command was called. MicroStation now looks for a `CmdItemListRsc` associated with the current command before checking for an item list in an `IconCmdRsc` resource.

This enhancement does not change the resource definition of `IconCmdRsc`'s. However, is recommended to remove the item list from the `IconCmdRsc` when you have added a `CmdItemListRsc`. `CmdItemListRsc`'s have been added and tool settings removed from `IconCmdRsc`'s for all MicroStation and MicroStation MDL application commands.

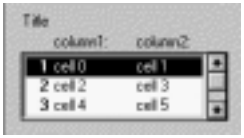
Adding `CmdItemListRsc`'s is an easy task, since it's item list is of the same form as the item list in the `IconCmdRsc`. See the example below which was taken from V551NEW.R from the delivered MDL example called V551NEW.

The only restriction is that the command and it's associated `CmdItemListRsc` must be in the same application.

```
#if defined (MSVERSION) && (MSVERSION < 0x551)
/* ----- MicroStation V5 & PowerDraft Method ----- */

DItem_IconCmdRsc ICONCMDID_PlaceSomething=
{
  NOHELP, MHELP, 0, CMD_PLACE_SOMETHING, OTASKID, "", "",
  {
   {{2*XC,GENY(1),0,0},
    ToggleButton,TOGGLEID_LockGrid,ON,0,"",""},
   {{2*XC, GENY(2), 0, 0},
    ToggleButton, TOGGLEID_LockGraphicGroup,
     ON,0,"",""},
  }
};
#else /* ----- MicroStation 95 Method ----- */
/* The above IconCmdRsc gets updated and a CmdItemListRsc resource
   is added.*/
```

```
DItem_IconCmdRsc ICONCMDID_PlaceSomething=
{
  NOHELP, MHELP, 0, CMD_PLACE_SOMETHING, OTASKID, "", "",
  {
    /* Tool settings in CmdItemListRsc */
  }
}

extendedAttributes
{
  {
    /* Tool Description/Flyover Help appears in status area */
    {EXTATTR_FLYTEXT, TXT_Flyover_PlaceSomething},
    /* Tool Tip/Balloon Help appears in yellow text
       box near icon command */
    {EXTATTR_BALLOON, TXT_Balloon_PlaceSomething},
  }
};

CmdItemListRsc CMD_PLACE_SOMETHING=
{
  {
   {{2*XC,GENY(1),0,0},ToggleButton,
   TOGGLEID_LockGrid,ON,0,"",""},
   {{2*XC,GENY(2),0,0},ToggleButton,
   TOGGLEID_LockGraphicGroup,ON,0,"",""},
  }
};
#endif /* if defined (MSVERSION) && (MSVERSION < 0x551) */
```

List Box Item



The list box item allows the display and selection of multiple text strings. A list box has a vertical scroll bar that is used to scroll through the list of text strings when there are more strings than can be displayed within the list box item at one time.

List boxes can have multiple columns. Each column can have its own type of justification. Within a given row, there is one list box cell for every column. A list box with three columns would have three cells per row. An option exists when defining the list box to either allow selection of only whole rows or to allow individual cells to be selected. The handling of the selection of rows within a list box is defined by the attributes of the list box and the operational mode the list box is using.MicroStation uses four Motif Selection Modes: Single, Browse, Multiple and Extended which are described later in this section.

The list box item is designed to manipulate a string list. See “String List Editor” on page 4-24 for more information on creating, manipulating, and destroying string lists. The list box item assumes that there is one string list member per list box cell. The string list members should be stored in row order. The first string list member corresponds to the list cell at row 0, column 0, the second string list member corresponds to the list cell at row 0, column 1, and so forth.

There must be at least one auxiliary information field associated with each string member in the string list. This is specified by setting the `nInfoFields` argument greater than or equal to one when calling `mdlStringList_create`. The list box item uses the first information field of each string list member to store selection and control information for each cell in the list box. This field must not be modified by the MDL programmer. If application data needs to be associated with each list box cell, that data can be placed in additional information fields.

Every list box should have an item hook function attached to it. When the item hook function receives the `DITEM_MESSAGE_CREATE` message it should indicate the list box’s string list by calling `mdlDialog_listBoxSetStrListP`. The string list must be created prior to calling this function. The string list must have at least one auxiliary information field per member and be initialized with one string member for each cell in the list box. If no item hook function is attached to the list box or the hook function did not create and attach a string list to the list box, the list box item handler will allocate an empty string list with two auxiliary information fields and attach it to the list box.

When the item hook function receives the `DITEM_MESSAGE_DESTROY` it should generally destroy the string list that the list box is manipulating. The usual reason for not destroying the string list is when it will be used in further processing, for example,

plotting all the files that the user just selected. If no item hook function is associated with the list box or the hook function did not create the string list, the list box item handler will destroy the string list associated with the list box.

Additional rows can be added to the contents of a list box by inserting members into the string list that the list box is set to manipulate. This is done with the `mdlStringList_insertMember` function. Rows can be deleted from the contents of a list box by deleting members from the list box's string list. This is done with the `mdlStringList_deleteMember` function. The string that is displayed within an individual list box cell can be changed by calling the `mdlStringList_setMember` function. Be certain, however, not to ever change a string list member's first auxiliary information field. To be on the safe side, the `infoFieldsP` argument to `mdlStringList_setMember` can always be set to `NULL`. Whenever the number of rows in a list box is changed, the `mdlDialog_listBoxNRowsChanged` function must be called or `mdlDialog_listBoxSetStrListP` can be called again.

The item hook function will receive a `DITEM_MESSAGE_STATECHANGED` message whenever the current selection changes.

Item list specification

The `DialogItemRsc` field extent specifies the location of the list box. The x position specifies the location of the left side of the list box's beveled rectangle, not the start of the list box's text label. The y position specifies the top of the list box's beveled rectangle. The list box resource specification's column widths and `nRows` fields determine a list box's width and height. Therefore, set the item list's extent width and height to 0.

The `type` field should be `ListBox`.

The `ID` determines the `DItem_ListBoxRsc` instance to load.

The attributes field can be `ON` or `OFF` and optionally combined with `HIDDEN`. It will usually be `ON`.

The `itemArg` field is unused and should usually be set to 0.

The `label` field can override the label contained in the `DItem_ListBoxRsc` instance.

The `auxInfo` field is not used with list box items.

The following is an example of a list box item list specification:

```
{{XC, YC, 0, 0}, ListBox, LISTID_FileListFiles, ON, 0, "", ""}
```

Item Resource Specification

The list box item is defined in a resource file with the following structure:

```
typedef struct ditem_listcolumnrsc
{
   int width;
   int maxSize;        /* max # of chars in column */
   long attributes;

#if defined (resource)
   char heading[];
#else
   long headingLength;
   char heading[1];
#endif
} DItem_ListColumnRsc;

typedef struct ditem_listrsc
{
   ULong helpInfo;
   ULong helpSource;
   long itemHookId;
   long itemHookArg;
   long attributes;
   int nRows;
   int sizeNumColumn;

#if defined (resource)
   char label[];
   DItem_ListColumnRsc listColumns[];
#else
   long labelLength;
   char label[1];
#endif
} DItem_ListRsc;

typedef DItem_ListRsc DItem_ListBoxRsc;
```

The `DItem_ListBoxRsc` structure has the following unique fields.

Field	Description
`attributes`	Specifies the list box's attributes. It is constructed by combining the constants from the attributes table (below) with the logical **OR** operator.
`nRows`	Specifies the number of rows that will display.

Field	Description
sizeNumColumn	Enables the list box handler to automatically number the rows. The maximum number of digits that can be used to number the rows must be specified. For example if a list can have up to 99 rows, specify 2 here. The rows will not be numbered if 0 is specified.
listColumns	Represents an array of DItem_ListColumnRsc. This array will have one entry for each column in the list box. See the next table for a description of the DItem_ListColumnRsc structure.

attributes Value	Meaning
LISTATTR_LABELONSIDE	The list box's label will be placed to the left instead of at the default.
LISTATTR_NEVERSELECTION	The user will be unable to select any part of the list. The list in this case is used only to view strings.
LISTATTR_DRAGSELECTION	The user can select multiple rows by dragging the cursor or using the SHIFT and/or CTRL modifier keys when clicking on list box rows. Replaces the LISTATTR_RANGESELECTION attribute.
LISTATTR_DISJOINTSELECTION	When combined with LISTATTR_DRAGSELECTION, this value indicates that multiple disjoint selections can be made.
LISTATTR_NOSELECTION	The list does not require a selection to always be present.
LISTATTR_INDEPENDENTCOLS	Individual cells, instead of entire rows, can be selected.
LISTATTR_NOKEYSEARCH	Indicates that a pressed key does not cause a search thorugh the string list for a match on the first character in column 1 and position the list box pointer in that row.
LISTATTR_FIXEDFONT	Indicates that a fixed width font should be used when displaying text in the list box.
LISTATTR_DYNAMICSCROLL	Indicates that the list box contents should be scrolled as the user drags the scroll bar handle up and down in the list box scroll bar.

attributes Value	Meaning
LISTATTR_COLOREDROWS	Indicates that colored rows are to be used in this list box. If colored rows are to be used, the application must ensure that at least two auxiliary information fields are attached to the string list members for use by the list box item handler and tha t the color descriptor for the color to be used in the display of the row is stored in the second auxiliary information field. Specifying NULL in the second fields indicates that the default color should be used for display.
LISTATTR_DRAWOVERLAPPING	Indicates that any items overlapping the list box are to be redrawn when the list box is drawn/updated.

The following attributes cause the MicroStation list box item handler to emulate various Motif style selection models for list boxes and should not be used in conjunction with any of the LISTATTR_xxxxSELECTION attributes documented above:

attributes Value	Meaning
LISTATTR_SELSINGLE	Emulate the Motif Single Selection model which allows the selection of a single row in the list box. Clicking on the row selects the row and deselects the previously selected row.
LISTATTR_SELBROWSE	Emulate the Motif Browse Selection model which allows the selection of a single row in the list box. This model is the same as Single Selection, but additionally allows the user to browse the list box by dragging the cursor through the rows in the list box; highlighting each row as the cursor passes over it. Releasing the mouse button on a row selects the row and deselects the previously selected row.
LISTATTR_SELMULTI	Emulate the Motif Multiple Selection model which allows the selection of multiple rows in the list box. Rows are added to or removed from the selection list by clicking on them with the mouse.
LISTATTR_SELEXTENDED	Emulate the Motif Extended Selection model which allows the selection of multiple rows in the list box. This model is the same as Multiple Selection, but additionally allows the user to select a range of rows in the list box by dragging the cursor through the rows in the list box. Use if the SHIFT and/or CTRL modifier keys extends the selection capabilities of rows in the list box.

The DItem_ListColumnRsc structure has the following unique fields.

Field	Description
width	Specifies the column width in dialog coordinate units.
maxSize	Specifies the maximum number of characters that display in the column.
attributes	Specifies the column's attributes. The values listed in the attributes table below are valid.
heading	Specifies a heading for the column. The heading will be justified above the column according to the justification bits in attributes.

attributes Value	Meaning
ALIGN_LEFT	Up to maxSize characters will be left-justified in a column that is width wide.
ALIGN_RIGHT	Up to maxSize characters will be right-justified in a column that is width wide.
ALIGN_CENTER	Up to maxSize characters will be centered in a column that is width wide.

The following is an example of a list box item resource. This item is used in the Open file dialog box, and is defined in MicroStation's resource file.

```
DItem_ListBoxRsc LISTID_FileListFiles=
{
  NOHELP, MHELP, HOOKITEMID_List_FileListFile, NOARG, 0, 6, 0, "Files",
  {
   {14*XC, 14, 0, ""},
  }
};
```

Item hook function messages

The following messages are sent to item hook functions that are attached to list box items:

```
DITEM_MESSAGE_CREATE
DITEM_MESSAGE_DESTROY
DITEM_MESSAGE_SYNCHRONIZE
DITEM_MESSAGE_BUTTON
DITEM_MESSAGE_KEYSTROKE
DITEM_MESSAGE_STATECHANGED
```

List box item functions

`mdlDialog_listBoxDeleteAll` deletes all columns from a list box.

`mdlDialog_listBoxDeleteColumn` deletes one column from a list box.

`mdlDialog_listBoxDrawContents` redraws all or part of the list's visible contents.

`mdlDialog_listBoxEnableCells` sets the state (enabled or disabled) of a range of cells.

`mdlDialog_listBoxSetColInfo` sets the attributes of a list box column

`mdlDialog_listBoxGetColInfo` gets the attributes of a list box column.

`mdlDialog_listBoxGetDisplayRange` returns the range of cells currently displayed in the list box.

`mdlDialog_listBoxGetInfo` gets the attributes of a list box.

`mdlDialog_listBoxSetInfo` sets the attributes of a list box.

`mdlDialog_listBoxGetLocationCursor` gets the location of the selection cursor in a list box.

`mdlDialog_listBoxSetLocationCursor` sets the location of the selection cursor in a list box.

`mdlDialog_listBoxGetNColumns` returns the number of columns in a list box.

`mdlDialog_listBoxGetNextSelection` obtains the next selected cell after the specified cell.

`mdlDialog_listBoxGetSelectRange` returns the range of selected cells. If non-contiguous selection is enabled, not every cell in the range will be selected.

`mdlDialog_listBoxGetSelections` returns the location of all currently selected cells.

`mdlDialog_listBoxSetSelections` extends the list of currently selected cells.

`mdlDialog_listBoxGetStrListP` returns a pointer to the `stringList` that is currently connected to the list box.

`mdlDialog_listBoxSetStrListP` sets the `stringList` that the list box will manipulate. This function must be called when the function receives the `DITEM_MESSAGE_CREATE` message. It can also be called any time a list box's underlying `stringList` is changed completely.

`mdlDialog_listBoxInsertColumn` inserts a column into an existing list box.

`mdlDialog_listBoxIsCellSelected` returns an indication of whether a given cell is selected.

`mdlDialog_listBoxIsCellEnabled` returns an indication of whether a given cell is enabled.

`mdlDialog_listBoxLastCellClicked` returns the cell that the last data button click occurred in.

`mdlDialog_listBoxNRowsChanged` should be called any time the number of rows in the `stringList` attached to the list box changes. Calling this function correctly resizes the list box's scroll bar.

`mdlDialog_listBoxSelectCells` selects (highlights) a range of cells.

`mdlDialog_listBoxSetTopRow` sets the top row that displays in the list box.

`mdlDialog_listBoxSetTopRowRedraw` sets which row is the first displayed in a list box and optionally redraws the list box.

List boxes

If you have seen MicroStation PowerDraft or MicroStation, you may have noticed the File Open/Create dialog boxes contain folder icons for each directory name, and that the tool box selection dialog contains a list box containing check box icons. They are just two examples of an enhancement in the list box item handler that displays icons in a list box.

Modifying list boxes to include icons

- Modify the resource definition for your list box item. Include a new attribute, `LISTATTR_DRAWPREFIXICON`, which instructs the list box item handler to draw an icon to the left of any text that is included in the string list for a cell. You may also need to add extra space in the column for the icon to be drawn.

```
/* List box resource definition from LISTICON.R */
DItem_ListBoxRsc LISTBOXID_Choices=
{
  NOHELP, MHELP, HOOKITEMID_CheckListBox, NOARG,
  LISTATTR_DYNAMICSCROLL|LISTATTR_DRAWPREFIXICON|LISTATTR_NOSELECTION,
  5, 0, "",
  {
    {(WIDTH_FILEOPENDIRS_LISTBOX + 10)*XC, 30, 0, ""},
  }
};
```

- Create the icons that will be displayed in the list box. The icons used in the MDL example LISTBOX are MicroStation icons whose identifiers are defined in `$(MS)`/mdl/include/dlogids.h. You will have to create icons to support different dialog font sizes.

Figure 16.7 List boxes containing icons from LISTBOX application.

The dialog font sizes vary from 8 to 24 points. Your application should have an icon to support each available font size. In the LISTBOX application, we have taken a few different approaches. The list box containing directory names uses small (16 x 12 pixels), medium (24 x 18 pixels) and large (32 x 24 pixels) icons, depending on the current dialog font size. The Choices list box uses a different icon for each dialog font size. For example, for 10 point dialog font, we created a 10 x 10 pixel icon (resource definition shown below).

```
/* From USTATION.R */
```

```
IconRsc ICONID_ToggleOn10Pt=
{
  10, 10, FORMAT_MONOBITMAP, BLACK_INDEX, TXT_IconM67_ON,
  {
    0x00, 0x1f, 0xe6, 0x19, 0x4a, 0x4c, 0x93, 0x25, 0x29,
    0x86, 0x7f, 0x80, 0x00,
  }
};
```

- Modify the MDL code that manages the list box's string list. The following code is from the list box hook function `listicon_checkListBoxHook` for the Choices list box in the LISTBOX example application. The code for the `DITEM_MESSAGE_CREATE` message creates a string list with four information fields and calls the function `listicon_fileListPopulate` to populate the string list with choices and information to display the check box icons.

```
/* From listicon_checkListBoxHook() in LISTICON.MC */
case DITEM_MESSAGE_CREATE:
{
    int fontHeight = mdlDialog_fontGetCurHeight(dimP->db);
    int status;
    DialogItem *diP = dimP->dialogItemP;
    RawItemHdr *rihP = diP->rawItemP;
    StringList *strListP;
    /* ************************************************************ *
     * Create the string list with at least four info fields. These
     * info fields will be used as follows:
     * 0 - Reserved for the dialog manager
     * 1 - Reserved for colored row information
     *     (see documentation of LISTATTR_COLOREDROWS)
     *     (see MDL example NEWITEMS)
     * 2 - Resource ID of icon to display preceding that row of text.
     * 3 - Number of pixels to indent before drawing the icon.
    * ************************************************************ */
    strListP = mdlStringList_create(0, 4);

    /* ************************************************************ *
     * if string list create failed, tell the dialog manager
     * create failed
     * ************************************************************ */
    if (NULL == strListP)
    {
        dimP->u.create.createFailed = TRUE;
        return;
    }

    /* ************************************************************ *
     * populate the string list with text and icon information based on
```

```
         * the current font height.
         * ********************************************************** */
        if (SUCCESS !=
                (status=listicon_checkListPopulate(strListP,fontHeight)))
        {
            listicon_errorPrint(MSGID_CheckBoxPopulateErr);
            dimP->u.create.createFailed = TRUE;
            return;
        }

        /* associate the string list to the list box */
        mdlDialog_listBoxSetStrListP(rihP, strListP, 1);
        break;
    }
    ...
```

The function `listicon_checkListPopulate` is called from the list box hook function above when the list box is created. The string list for the choices list box is populated with ten choices with check box icons. The function `listicon_toggleCheckBox` assigns the check box icon information to the string list's information fields.

```
Private int listicon_checkListPopulate
(
StringList *strListP,   /* <= string list to populate */
int fontHeight          /* => current dialog font height */
)
{
    int status = SUCCESS, i, newIndex;
    if (NULL == strListP)
        return ERROR;
    for (i=0; I<10; i++)
    {
        char buff[50];
        sprintf(buff, "%d Choice", i);
        status=mdlStringList_insertMember(&newIndex,strListP,-1,1);
        if (SUCCESS != status)
            break;
        mdlStringList_setMember(strListP, newIndex, buff, NULL);
        /* Add the check box icon information to the string list */
        listicon_toggleCheckBox(strListP,newIndex,fontHeight,NULL);
    }
    return status;
}
```

The string list associated with the list box must have additional information fields that store the icon information for the members of the string list. The function `listicon_toggleCheckBox` (shown below) initially populates the string list's

information fields and toggles the check box when the user clicks on a row in the list box. The string list's information fields are populated according to the following table:

Field number	Use of the information field
0	Reserved for the dialog manager to store selection and control information for each cell in the list box.
1	Reserved for colored row information.
2	Resource identifier of the icon to display preceding a row of text.
3	Number of pixels to indent before drawing the icon.

In this function, the current font height determines which icon size to use for the string list. Field 2 in the information field is set to the appropriate icon's resource identifier; field 3 is set to the number of pixels that the icon is indented. For this example, all check box icons are indented by one pixel.

```
Private void listicon_toggleCheckBox
(
StringList *strListP, /* <> string list to store icon info */
int listIndex,    /* => index into string list */
int fontHeight,   /* => current dialog font height */
int *infoFieldsP  /* => current information field values */
)
{
    int iconRscID, status;
    int pixelOffset; /* number of pixels that icon is indented */
    /* ************************************************************ *
     * if the icon resource id was previously set, then use it's value,
     * otherwise, set the id to zero
     * ************************************************************ */
    iconRscID = (infoFieldsP) ? infoFieldsP[2] : 0;
    /* indent the icon by 1 pixel */
    pixelOffset = 1;

    /* ************************************************************ *
     * if the toggle is OFF then turn it ON. The toggle is initially
     * set to OFF.
     * ************************************************************ */
    if (fontHeight < 9)
    {
        iconRscID = (iconRscID==ICONID_ToggleOff8Pt) ?
                    ICONID_ToggleOn8Pt : ICONID_ToggleOff8Pt;
    }
    else if (fontHeight < 11)
    {
        iconRscID = (iconRscID==ICONID_ToggleOff10Pt) ?
```

```
                    ICONID_ToggleOn10Pt : ICONID_ToggleOff10Pt;
    }
    else if (fontHeight < 13)
    {
        iconRscID = (iconRscID==ICONID_ToggleOff12Pt) ?
                    ICONID_ToggleOn12Pt : ICONID_ToggleOff12Pt;
    }
    else if (fontHeight < 17)
    {
        iconRscID = (iconRscID==ICONID_ToggleOff14Pt) ?
                    ICONID_ToggleOn14Pt : ICONID_ToggleOff14Pt;
    }
    else if (fontHeight < 23)
    {
        iconRscID = (iconRscID==ICONID_ToggleOff18Pt) ?
                    ICONID_ToggleOn18Pt : ICONID_ToggleOff18Pt;
    }
    else
    {
        iconRscID = (iconRscID==ICONID_ToggleOff24Pt) ?
                    ICONID_ToggleOn24Pt : ICONID_ToggleOff24Pt;
    }

    /* ********************************************************** *
     * Set the third info field (index 2) to the resource ID of icon
     * to display preceding this row of text.
     * ********************************************************** */
    mdlStringList_setInfoField (strListP, listIndex, 2, iconRscID);
    /* ********************************************************** *
     * Set the fourth info field (index 3) to the number of pixels to
     * indent before drawing the icon.
     * ********************************************************** */
    mdlStringList_setInfoField(strListP, listIndex, 3, pixelOffset);
}
```

- Handle the FONT CHANGED event to modify icon information in the string list and represent the new font height of the dialog. List box text sizes change automatically when the current dialog font changes—but because icon resources are a fixed size, the programmer must update the icon resource identifier in the `StringList` according to the new font height. The application must request font change event messages from the dialog manager by setting the *fontChanges* member of `DialogHookInterests` to `TRUE` during the dialog box creation.

```
/* From listicon_mainBoxHook() in LISTICON.MC */
...
case DIALOG_MESSAGE_CREATE:
{
    /* ********************************************************** *
```

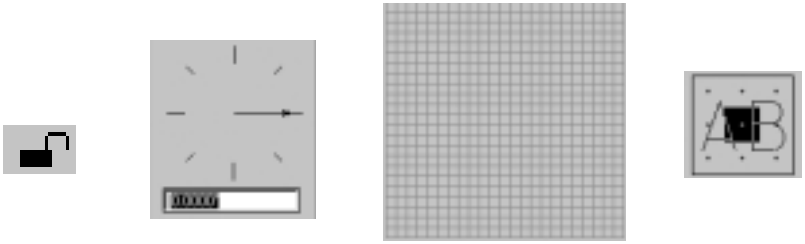
```
        * Establish an interest in Dialog font changes so we can change our
        * icon sizes when the dialog manager gives us the
        * DIALOG_MESSAGE_FONTCHANGED message.
        * ********************************************************** */
        dmP->u.create.interests.fontChanges = TRUE;
        break;
    };
    ...
```

When the user moves the dialog box to another screen that has a different font size, the dialog manager sends the `DIALOG_MESSAGE_FONTCHANGED` message to the dialog box hook function. When the hook function receives the message, the application can either recreate the string list for the list box or modify each icon resource identifier in each string list member's information field.

The MDL applications NEWITEMS and LISTBOX mentioned are delivered with MicroStation under the MDL\examples subdirectory. NEWITEMS was also delivered with MicroStation V5.0.

Generic Item

The generic item creates items whose appearance and response to mouse and keyboard messages can be customized. The item hook function manages all behavior of a generic item. Unlike all other items, the dialog box manager provides no default behavior for a generic item. The MDL program **rasticon** shows one use of the generic item. The source for rasticon is delivered with MicroStation in the **mdl\examples** directory.



Item list specification

The item hook function uses the `DialogItemRsc` field `extent` to specify the location of the generic item. If the `DITEM_MESSAGE_SETEXTENT` message is not handled, the upper left corner of the item will be at the position specified by `extent.origin` and the item's width and height will be `extent.width` and `extent.height`, respectively.

The `type` field should be `Generic`.

The `ID` determines the `DItem_GenericRsc` instance to load.

The `attributes` field can be `ON` or `OFF` and optionally combined with `HIDDEN`. It will usually be `ON`.

The `itemArg` field is unused and should usually be set to 0.

The `label` field can pass a string to the item hook function at `DITEM_MESSAGE_CREATE` time. `diMsg.u.create.labelP` will point at this field.

The `auxInfo` field is unused for generic items and should be set to "".

The following is an example of a generic item list specification:

```
{{XC, YC, 10*YC, 10*YC}, Generic, GENERICID_Cell, ON, 0, "", ""}
```

Item resource specification

The generic item is defined in a resource file with the following structure:

```
typedef struct ditem_genericrsc
{
   ULong helpInfo;
   ULong helpSource;
   long itemHookId;
   long itemHookArg;
} DItem_GenericRsc;
```

The following is an example of a generic item resource. This item is used in the Cells dialog box, and is defined in MicroStation's resource file.

```
DItem_GenericRsc GENERICID_Cell=
{
   NOHELP, MHELP, HOOKITEMID_Generic_Cell, NOARG
};
```

Item hook function messages

The following messages are sent to item hook functions that are attached to generic items:

```
DITEM_MESSAGE_CREATE
DITEM_MESSAGE_INIT
```

DITEM_MESSAGE_DESTROY

DITEM_MESSAGE_DRAW

DITEM_MESSAGE_FONTCHANGED

DITEM_MESSAGE_MOVE

DITEM_MESSAGE_SETEXTENT

DITEM_MESSAGE_BUTTON

DITEM_MESSAGE_SYNCHRONIZE

DITEM_MESSAGE_HIGHLIGHT

DITEM_MESSAGE_KEYSTROKE

DITEM_MESSAGE_FOCUSIN

DITEM_MESSAGE_FOCUSOUT

DITEM_MESSAGE_SETLABEL

DITEM_MESSAGE_GETVALUE

DITEM_MESSAGE_SETVALUE

DITEM_MESSAGE_GETSTATE

DITEM_MESSAGE_SETSTATE

DITEM_MESSAGE_STATECHANGED

DITEM_MESSAGE_QUEUECOMMAND

`dimP->dialogItemP->attributes.acceptsKeystrokes` should be set to `FALSE` at `DITEM_MESSAGE_CREATE` time if the item should not be able to accept the input focus. `KEYSTROKE`, `FOCUSIN` and `FOCUSOUT` messages will not be generated.

`dimP->dialogItemP->attributes.mouseSensitive` should be set to `FALSE` if the item should not be able to respond to mouse events. `BUTTON` messages will not be generated.

Generic item functions

There are no generic item functions.

Button Group Items

The button group item creates items whose appearance is similar to that of icon command palettes and icon commmand frames but which can be contained in a standard dialog box and operate in the same manner as option or radio buttons.

Button group items define a palette of icons which are displayed in a dialog box. The operation and display of the icons within the palette is defined by the item hook function and the attributes assigned to the item in the resource definition. To the user, a button group actually behaves like an iconic option button which is expanded out and allows radio button selection.

Item list specification

The `DialogItemRsc` field `extent` specifies the location of the button group item. The width and height of the item is defined by the `iconWidth` and `iconHeight` fields in the button group item resource definition The `extent.width` and `extent.height` fields are ignored and should be set 0.

The `type` field should be `ButtonGroup`.

The `ID` determines the `DItem_ButtonGroupRsc` instance to load.

The `attributes` field can be `ON` or `OFF` and optionally combined with `HIDDEN`. It will usually be `ON`.

The `itemArg` field is unused and should usually be set to 0.

The `label` field can override the label contained in the `DItem_ButtonGroupRsc` instance.

The `auxInfo` field can override the access string contained in the `DItem_ButtonGroupRsc` instance.

The following is an example of a button group item list specification:

```
{{XC, YC, 0, 0}, ButtonGroup, BGROUPID_SnapButtonWide, ON, 0, "", ""}
```

Item resource specification

The button group item is defined in a resource file with the following structure:

```
typedef struct ditem_buttongroupitemrsc
{
   ULong iconType;
   long iconId;
   ULong commandNumber;
   ULong commandSource;
   ULong value;
   ULong mask;
   ULong arrayIndex;
   ULong attributes;

#if defined (resource)
   char label[];
#else
   long labelLength;
   char label[1];
#endif

} DItem_ButtonGroupItemRsc;

typedef struct ditem_buttongrouprsc
{
   long synonymsId;
   ULong helpInfo;
   ULong helpSource;
   long itemHookId;
   long itemHookArg;
   ULong attributes;
   long nColumns;
   long nRows;
   long itemWidth;
   long itemHeight;
   long selectedFgColorIndex;/* only if lgrey highlight,
                                -1 = use icon's */

#if defined (resource)
   char label[];
   char accessStr[];
   DItem_ButtonGroupItemRscbuttonGroupItems[];
#else
   long labelLength;
   char label[1];
#endif

} DItem_ButtonGroupRsc;
```

The `DItem_ButtonGroupRsc` structure has the following unique fields. See "Common Item Resource Fields" on page 16-8 for a description of any field not in this table.

<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td>attributes</td><td>Specifies attributes which affect the button group item as a whole. It is constructed by combining the constants from the following table with the logical OR operator:

<table><tr><th>Value</th><th>Description</th></tr><tr><td>BGROUPATTR_HIGHLIGHT
-INVERT</td><td>Indicates that when an icon is selected, it should be highlighted by inverting the icon foreground and background colors.</td></tr></table></td></tr>
<tr><td>nColumns</td><td>Specifies the number of columns in the button group.</td></tr>
<tr><td>nRows</td><td>Specifies the number of rows in the button group.</td></tr>
<tr><td>itemWidth</td><td>Specifies the width of the icons contained in the button group in dialog coordinates. A negative value indicates the width is in pixels and a value of 0 indicates that the maximum icon width for the icons in the button group should define the item width. This field is usually 0.</td></tr>
<tr><td>itemHeight</td><td>Specifies the height of the icons contained in the button group in dialog coordinate units. A negative value indicates the height is in pixels and a value of 0 indicates that the maximum icon height for the icons in the button group should define the item height. This field is usually 0.</td></tr>
<tr><td>selectedFgColorIndex</td><td>Specifies a color index to be used as the foreground color in displaying the icon when it has been selected by the user. If this value is -1, then the icon's foreground color is used. This field is not used when the BGROUPATTR_HIGHLIGHTINVERT attribute is set.</td></tr>
<tr><td>buttonGroupItems</td><td>Specifies the contents of the button group item. buttonGroupItems is an array of DItem_ButtonGroupItemRscs that is the button group's subitem list. This structure has many of the same fields as the DItem_OptionButtonItemRsc structure.</td></tr>
</table>

The DItem_OptionButtonItemRsc structure has the following unique fields. See "Common Item Resource Fields" on page 16-8 for a description of any field not in this table.

Field	Description
iconType	Indicates the type of icon the subitem is using. NOTYPE is not supported for this field.
iconId	Specifies the ID of the icon instance to load and display as the subitem. NOICON is not supported for this field.
value	Is used with mask to determine the currently selected subitem. Only one subitem can be selected at once. mask indicates the relevant bits of the variable specified by accessStr. These bits, shifted so the rightmost bit of the mask becomes bit 0, are compared to value. If the mask is 0x0E (1110 binary) then the masked value will be shifted right 1 bit before being compared to value. The subitem whose shifted masked bits matches value becomes the currently selected subitem. For example, suppose bits 2 and 3 of the variableflags will be tested. mask will then be 0xC (1100 binary). If bit 2 is on, the first subitem should be selected. If bit 3 is on, the second subitem should be selected. If both bits are on, the third subitem is selected. value and mask should be set to the following for each subitem:
mask	Is used with value to determine the currently selected subitem. If the entire variable specified by accessStr is being compared to value, set this field to NOMASK (which is defined to be 0xFFFFFFFF).
enabled	Determines the state (enabled or disabled) of the subitem when the button group is first created. If this field is set to ON, the subitem can be selected; when it is set to OFF, the subitem will be dim and cannot be selected.
arrayIndex	Not currently supported.
attributes	Specifies subitem specific attribute characteristics. It is constructed by combining the constants from the following table with the logical OR operator:

subitem	value	mask	flags value that will select subitem
0	1	0xC	4 (= 1 << 2)
1	2	0xC	8 (= 2 << 2)
2	3	0xC	12 (= 3 << 2)

attributes Value	Meaning
BGROUPITEMATTR_DISABLED	Specifies that the icon is disabled and is not selectable by the user.
BGROUPITEMATTR_GREYED	Specifies that when the icon is drawn, it should be drawn using the icon style ICON_STYLE_DGREY (See the **mdlWindow_iconDraw** function in the MicroStation MDL Function Reference Manual for more information on icon drawing styles).

The following is an example of a button group item resource. This item is used in the Snap Mode button bar, and is defined in MicroStation's resource file.

```
DItem_ButtonGroupRsc BGROUPID_SnapButtonTwoRow=
{
   SYNONYMID_SnapButtonGroup, NOHELP, MHELP,
   HOOKITEMID_ButtonGroup_SnapMode, NOARG,
   0, 8, 2, 0, 0, WHITE_INDEX, "", "tcb->snapOverride",
   {
     {Icon, ICONID_NearSnapPoint, NOCMD, MCMD, 0, NOMASK, 0, 0, ""},
     {Icon, ICONID_KeyPointSnap, NOCMD, MCMD, 1, NOMASK, 0, 0, ""},
     {Icon, ICONID_MidPointSnap, NOCMD, MCMD, 2, NOMASK, 0, 0, ""},
     {Icon, ICONID_CenterSnapCircle, NOCMD, MCMD, 3, NOMASK, 0, 0, ""},
     {Icon, ICONID_OriginSnap, NOCMD, MCMD, 4, NOMASK, 0, 0, ""},
     {Icon, ICONID_BisectorSnap, NOCMD, MCMD, 5, NOMASK, 0, 0, ""},
     {Icon, ICONID_IntersectSnap, NOCMD, MCMD, 6, NOMASK, 0, 0, ""},
     {Icon, ICONID_TangentSnap, NOCMD, MCMD, 7, NOMASK, 0, 0, ""},
     {Icon, ICONID_TangentPointSnap, NOCMD, MCMD, 8, NOMASK, 0, 0, ""},
     {Icon, ICONID_PerpendicularSnap, NOCMD, MCMD, 9, NOMASK, 0, 0, ""},
     {Icon, ICONID_PerpendicularSnapPnt,NOCMD, MCMD, 10,NOMASK,0,0, ""},
     {Icon, ICONID_ParallelSnap, NOCMD, MCMD, 11,NOMASK, 0, 0, ""},
     {Icon, ICONID_ThruPointSnap, NOCMD, MCMD, 12,NOMASK, 0, 0, ""},
     {Icon, ICONID_PointOnSnap, NOCMD, MCMD, 13,NOMASK, 0, 0, ""},
     {Icon, ICONID_ResizeBoxToTall, NOCMD, MCMD, 14,NOMASK, 0, 0, ""},
   }
};
```

Item hook function messages

The following messages are sent to item hook functions that are attached to button group items:

DITEM_MESSAGE_CREATE

DITEM_MESSAGE_INIT

```
DITEM_MESSAGE_DESTROY
DITEM_MESSAGE_GETSTATE
DITEM_MESSAGE_SETSTATE
DITEM_MESSAGE_SYNCHRONIZE
DITEM_MESSAGE_STATECHANGED
DITEM_MESSAGE_QUEUECOMMAND
```

Sash Item

The sash item creates items whose appearance is a separator bar with a handle on it which can be used to divide a dialog box into separate horizontal panes. The user can grab the sash handle and drag the sash up and down the dialog box changing the amount of display space in each pane. The item hook function controls how far the sash can move and the resizing/redisplaying of items in each pane.

The hook function is responsible for performing the resizing of items on each side of the sash; the sash item handler does not do any item resizing itself. The item hook can, and should, perform the resizing using both a motion function hook and button up event processing logic.

Item list specification

The `DialogItemRsc` field `extent` specifies the horizontal location of the sash. The y position should be defined and the x, width and height fields are ignored and should be 0.

The `type` field should be `Sash`.

The `ID` determines the `DItem_SashRsc` instance to load.

The `attributes` field can be `ON` or `OFF` and optionally combined with `HIDDEN`. It will usually be `ON`.

The `itemArg` field is unused and should usually be set to 0.

The `label` field is unused for sash items and should be set to `""`.

The `auxInfo` field is unused for sash items and should be set to `""`.

The following is an example of a sash item list specification:

```
{{0, 0, 10*YC, 0}, Sash, SASHID_Example, ON, 0, "", ""}
```

Item resource specification

The sash item is defined in a resource file with the following structure:

```
typedef struct ditem_sashrsc
{
   ULong helpInfo;
   ULong helpSource;
   long itemHookId;
   long itemHookArg;
   long topBorder;     /* min dist. from top (in dcoords) */
   long bottomBorder;  /* min dist. from bottom (in dcoords) */
   ULong attributes;
} DItem_SashRsc;
```

The `DItem_SashRsc` structure has the following unique fields. See "Common Item Resource Fields" on page 16-8 for a description of any field not in this table.

Field	Description
`topBorder`	Specifies the minimum distance from the top of the dialog box that the sash can be moved.
`bottomBorder`	Specifies the minimum distance from the bottom of the dialog box that the sash can be moved.

The following is an example of a sash item resource.

```
DItem_SashRsc SASHID_Example=
{
   NOHELP, MHELP, HOOKID_Sash, NOARG, 5*YC, 5*YC, 0
};
```

Item hook function messages

The following messages are sent to item hook functions that are attached to sash items:

```
DITEM_MESSAGE_BUTTON
```

When handling the `DITEM_MESSAGE_BUTTON` message for a `BUTTONTRANS_DOWN` event, the item hook function can use a motion function (the `u.button.motionFunc` member) to allow the application to track cursor movement while the button is down. Using a motion function would allow the MDL application to dynamically resize the panes as they are "resized" by the sash item.

Sash item functions

There are no sash item functions.

Scale Item

The scale item can be used to show the state of a variable that ranges between a set of values. These items are similar to scroll bars except that they also allow the attachment of moving labels and implement the Motif scale item within MicroStation.

Item list specification

The `DialogItemRsc` field `extent` specifies the location of the scale. The x position specifies the left edge and the y position specifies the top edge of the scale. The orientation of the scale depends on the width and height fields. If width is 0, the scale handler will make a vertical scale. If height is 0, a horizontal scale will be created. Both the width and height cannot be 0.

The `type` field should be `Scale`.

The `ID` determines the `DItem_ScaleRsc` instance to load.

The `attributes` field can be `ON` or `OFF` and optionally combined with `HIDDEN`. It will usually be `ON`.

The `itemArg` field is unused and should usually be set to 0.

The `label` field can override the label contained in the `DItem_ScaleRsc` instance.

The `auxInfo` field can currently override only the access string contained in the `DItem_ScaleRsc` instance.

The following is an example of a scale item list specification:

```
{{XC, YC, 10*XC, 0}, Scale, SCALEID_ColorPal_AltColorSpace3, ON,
0, "", ""}
```

Item resource specification

The scroll bar item is defined in a resource file with the following structure:

```
typedef struct ditem_scalersc
{
   ULong commandNumber;
   ULong commandSource;
   long synonymsId;
   ULong helpInfo;
   ULong helpSource;
   long itemHookId;
   long itemHookArg;
   double minValue;
   double maxValue;
   double incAmount;
   double pageIncAmount;
   long associatedTextId;
   ULong attributes;
   char formatToDisplay[16];

#if defined (resource)
   char label[];
   char accessStr[];
   char minLabel[];
   char maxLabel[];
#else
   long labelLength;
   char label[1];
#endif

} DItem_ScaleRsc;
```

The DItem_ScaleRsc structure has the following unique fields. See "Common Item Resource Fields" on page 16-8 for a description of any field not in this table.

Field	Description
minValue	Specifies the value associated with the minimum position of the scale slider.
maxValue	Specifies the value associated with the maximum position of the scale slider.
incAmount	Specifies the amount the current value of the scale is changed when the user clicks on the scale arrows.
pageIncAmount	Specifies the amount the current value of the scale is changed when the user clicks in the page area of the scale. The page areas are the areas between the slider and the scale arrows.
associatedTextId	This field specifies the resource ID of a text item that will be associated with the scale item. As the user drags the mouse on the slider, the associated text item will be dynamically set to the value of the scale based upon it's current slider position.
attributes	Specifies the attributes of the scale item. It is constructed by combining the constants from the attributes table (below) with the logical OR operator.
formatToDisplay	An sprintf format string to convert the value of the scale item into a display value when the SCALE_SHOWVALUE attribute is used.
minLabel	A string which should be used as the label for the minimum value of the scale item.
maxLabel	A string which should be used as the label for the maximum value of the scale item.

The following table lists possible values for the attributes field (described above) and the meaning of those values:

attributes Value	Meaning
SCALE_HASARROWS	Indicates that the scale item should be drawn with single step adjustment arrows at each end of the scale.
SCALE_SHOWVALUE	Indicates that the current scale value should be displayed alongside the slider.
SCALE_LIMITSONSIDE	Indicates that the range limits for the scale are to be displayed at the side of the scale item and not at the ends. By default, scale ranges are displayed at the ends of the scale item. Limits are most commonly placed on the side when the scale is vertically oriented.

The following is an example of a scale item resource. This item is used in the Color Palette dialog box, and is defined in MicroStation's resource file.

```
DItem_ScaleRsc SCALEID_ColorPal_AltColorSpace3=
{
   NOCMD, LCMD, SYNONYMID_ColorPar_AltColorSpace3, NOHELP, MHELP,
   HOOKITEMID_Scroll_ColorSquare, NOARG,
   0.0, 100.0, 1.0, 5.0, TEXTID_ColorPal_AltColorSpace3,
   SCALE_HASARROWS, "%.0lf", "", "", "", ""
};
```

Item hook function messages

The following messages are sent to item hook functions attached to scale items:

DITEM_MESSAGE_CREATE

DITEM_MESSAGE_INIT

DITEM_MESSAGE_DESTROY

DITEM_MESSAGE_BUTTON

DITEM_MESSAGE_GETSTATE

DITEM_MESSAGE_SETSTATE

DITEM_MESSAGE_SETVALUE

DITEM_MESSAGE_STATECHANGED

DITEM_MESSAGE_SYNCHRONIZE

DITEM_MESSAGE_FOCUSIN

DITEM_MESSAGE_FOCUSOUT

DITEM_MESSAGE_KEYSTROKE

DITEM_MESSAGE_POSTKEYSTROKE

DITEM_MESSAGE_SYNCHRONIZE

DITEM_MESSAGE_QUEUECOMMAND

`RTYPE_Scale`, Auxilliary message type `SCALE_AUXMESSAGE_FORMATVALUE`

The `RTYPE_Scale` message is used to allow the item hook function to display textual information as the current item value instead of the current scale item value. This message is only sent when the `SCALE_SHOWVALUE` attribute is set on the scale item. The message is sent to the hook function each time the item handler needs to display the current value of the scale item alongside the scale slider. The `dimP->auxMessageType` field will be set to `SCALE_AUXMESSAGE_FORMATVALUE` and the `dimP->auxInfoP` field will point to a buffer into which the hook function can place an ASCII string to be

displayed as the current scale item value. The **newitems** MDL example application shows various usages of the scale item.

When handling the DITEM_MESSAGE_BUTTON message for a BUTTONTRANS_DOWN event, the item hook function can use a motion function (the u.button.motionfunc member) to allow the application to track cursor movement while the button is down.

Scale item functions

There are no scale item functions.

Popup Menu Item

The popup menu item creates a floating pulldown menu under the mouse cursor when opened. The item can be opened the item hook function during button event processing. This item is simply a reference to an existing pulldown menu resource which can be used as a popup menu by the application program.

An application program causes a popup menu to be displayed by using a dialog box hook function that handles the DIALOG_MESSAGE_BUTTON event for a dialog box containing a popup menu item in its item list. The hook function can cause a popup menu to be opened by setting the dmP->u.button.buttonRiP field to point to the raw item header of the popup menu item. This causes the popup menu to be displayed under the mouse cursor and the user to select a entry in that popup menu. Typically, the popup menu is opened on a BUTTONTRANS_DOWN event with some form of modifier key (SHIFT, CTRL or ALT key combined with the mouse down event).

Item list specification

The DialogItemRsc field extent to is ignored by the item handler and should be set to 0 in all subfields.

The type field should be PopupMenu.

The ID determines the DItem_PopupMenuRsc instance to load.

The `attributes` field should always be `HIDDEN`.

The `itemArg` field is unused and should usually be set to 0.

The `label` field is unused for popup menu items and should be set to `""`.

The `auxInfo` field is unused for popup menu items and should be set to `""`.

The following is an example of a popup menu item list specification:

```
{{0, 0, 0, 0}, PopupMenu, POPUPMENUID_ViewSnaps, HIDDEN, 0, "", ""}
```

Item resource specification

The popup menut item is defined in a resource file with the following structure:

```
typedef struct ditem_popupmenursc
{
   ULong helpInfo;
   ULong helpSource;
   long itemHookId;
   long itemHookArg;
   ULong attributes;
   ULong menuType;
   long menuId;
} DItem_PopupMenuRsc;
```

The `DItem_PopupMenuRsc` structure has the following unique fields. See "Common Item Resource Fields" on page 16-8.

Field	Description
`menuType`	Specifies the type of pulldown menu resource to be used as a popup menu. This value must be one of `PulldownMenu`, `RPulldownOptionMenu` or `PulldownCPickerMenu`.
`menuId`	Specifies the resource id of the pulldown menu resource to be used as a popup menu.

The following is an example of a popup menu item resource. This item is used in MicroStation's window environment, and is defined in MicroStation's resource file.

```
DItem_PopupMenuRsc POPUPMENUID_ViewSnaps=
{
   NOHELP, LHELP, NOHOOK, NOARG, 0, PulldownMenu, PULLDOWNMENUID_Snaps
};
```

Item hook function messages

The following messages are sent to item hook functions that are attached to popup menu items:

```
DITEM_MESSAGE_CREATE
DITEM_MESSAGE_INIT
DITEM_MESSAGE_DESTROY
```

Popup menu item functions

There are no popup menu item functions.

Radio Button Item

The radio button item shows the state of an application variable that can be any one of a selection of values. The currently selected value is identified by the 3D diamond displayed next to each value label. This item is actually comprised of two different resources: The radio button resource itself which describes one of the possible values an option can assume and the radio button list resource which defines all the radio button values which comprise the group. The collection of these two resources is called a radio button group and is operationally similar to the option button item and visually similar to the toggle button item.

When the user selects one of the options available for a radio button group, that option is highlighted and the previously highlighted option is un-highlighted. That is, radio buttons operate the same as an option button in that only one option can be selected at any one time, but are visually similar to toggle buttons in that they are all displayed to the user at all times are are displayed as toggle fields.

Item list specification

The `DialogItemRsc` field `extent` specifies the location of the radio button item. The x position specifies the location of the left side of the radio button's 3D diamond, not the start of the radio button's text label. The y position specifies both the top of the 3D diamond and the top of the text label. If the width is 0, the width of the label is used. If the height is 0, the height of the current dialog font is used.

The `type` field should contain `RadioButton`.

The `ID` determines the `DItem_RadioButtonRsc` instance to load.

The `attributes` field can be `ON` or `OFF`, and optionally combined with `HIDDEN`. It will usually be `ON`.

The `itemArg` field is unused and should usually be set to 0.

The `label` field overrides the label contained in the `DItem_RadioButtonRsc` instance.

The `auxInfo` field currently overrides only the access string contained in the `DItem_RadioButtonRsc` instance.

The following is an example of a radio button item list specification:

```
{{XC, YC, 0, 0}, RadioButton, RBUTTONID_Option4, ON, 0, "", ""}
```

Item resource specification

The list of radio button item instances is defined in a resource file along with the actual radio button item resources themselves. The list of radio button resources which comprise the radio button group is defined by the following structure:

```
typedef struct idlistrsc
{
#if defined (resource)
   long ids[];
#else
   long nIds;
   long idList[1];
#endif
} IdListRsc;
typedef IdListRsc DItem_RadioButtonListRsc;
```

This structure defines the list of `DItem_RadioButtonRsc` resource IDs which comprise the content of the radio button group. The `DItem_RadioButtonRsc` structure describes a component of a radio button group and is defined by the following structure:

```
typedef struct ditem_radiobuttonrsc
{
   ULong commandNumber;
   ULong commandSource;
   long synonymsId;
   ULong helpInfo;
   ULong helpSource;
```

```
    long itemHookId;
    long itemHookArg;
    ULong value;
    ULong mask;
    long radioButtonListId;       /* must specify, id of list
                                     of other rb's in group */

#if defined (resource)
    char label[];
    char accessStr[];
#else
    long labelLength;
    char label[1];
#endif

} DItem_RadioButtonRsc;
```

The DItem_RadioButtonRsc structure has the following unique fields. See ""Common Item Resource Fields" on page 16-8.

Field	Description
value	Is used with mask to determine the currently selected subitem. Only one subitem can be selected at once. mask indicates the relevant bits of the variable specified by accessStr. These bits, shifted so the rightmost bit of the mask becomes bit 0, are compared to value. If the mask is 0x0E (1110 binary) then the masked value will be shifted right 1 bit before being compared to value. The subitem whose shifted masked bits matches value becomes the currently selected subitem.
mask	Is used with value to determine the currently selected subitem. If the entire variable specified by accessStr is being compared to value, set this field to NOMASK (which is defined to be 0xFFFFFFFF).
radioButtonListId	Defines the resource ID of the DItem_RadioButtonListRsc which to which this radio button resource definition belongs. This defines the grouping to be used for this radio button value definition.

subitem	value	mask	flags value that will select subitem
0	1	0xC	4 (= 1 << 2)
1	2	0xC	8 (= 2 << 2)
2	3	0xC	12 (= 3 << 2)

For example, suppose bits 2 and 3 of the variableflags will be tested. mask will then be 0xC (1100 binary). If bit 2 is on, the first subitem should be selected. If bit 3 is on, the second subitem should be selected. If both bits are on, the third subitem is selected. value and mask should be set to the following for each subitem:

The following is an example of a radio button item resource. This item is used in the Color Palette dialog box, and is defined in MicroStation's resource file.

```
DItem_RadioButtonListRsc RADIOBUTTONLISTID_ColorInterp_Terminal=
{
   {
        RADIOBUTTONID_ColorInterp_Start,
        RADIOBUTTONID_ColorInterp_End,
   }
};

DItem_RadioButtonRsc RADIOBUTTONID_ColorInterp_Start=
{
   NOCMD, LCMD, NOSYNONYM, NOHELP, MHELP,
   HOOKITEMID_ColorInterp_RadioButtons, NOARG,
   0, 0xff, RADIOBUTTONLISTID_ColorInterp_Terminal,
   TXT1_ColorInterpRadioButtonStart, "msDialogState.interpColor"
};

DItem_RadioButtonRsc RADIOBUTTONID_ColorInterp_End=
{
   NOCMD, LCMD, NOSYNONYM, NOHELP, MHELP,
   HOOKITEMID_ColorInterp_RadioButtons, NOARG, 1, 0xff,
   RADIOBUTTONLISTID_ColorInterp_Terminal,
   TXT1_ColorInterpRadioButtonEnd, "msDialogState.interpColor"
};
```

Item hook function messages

The following messages are sent to item hook functions that are attached to radio button items:

DITEM_MESSAGE_CREATE

DITEM_MESSAGE_INIT

DITEM_MESSAGE_DESTROY

DITEM_MESSAGE_BUTTON

DITEM_MESSAGE_SYNCHRONIZE

DITEM_MESSAGE_GETSTATE

DITEM_MESSAGE_SETSTATE

DITEM_MESSAGE_STATECHANGED

Radio button item functions

There are no radio button item functions.

Tab Page Item

This document describes the use and construction of Tab Page dialog items. Included is a complete description of the MDL resources provided, the use of tab pages, and several programming examples. With this information, the knowledgeable user should be able to create customized interfaces to expand the capabilities of MicroStation products. The tab page item and tab page list item have been implemented in MicroStation SE.

Tab Page dialog items displaying Tab Options.

The tab page item is used to create dividers much like those in a notebook or the labels in a file cabinet. By using a tab page item, several pages for an application can be defined and displayed multiple in the same area of a dialog box. Each page consists of a set of information or a group of items that the application displays when the user selects the corresponding tab. This item is comprised of two different resources:

- The tab page list resource which lists the individual tab page items to be displayed.
- The tab page resource itself which lists the items to be displayed on the page and identifies the label on the tab.

Tab Page List

The tab page list item `DialogItemRsc` entry is included in the dialog box item list.

Item List Specification

The `DialogItemRsc` field `extent` specifies the location and size of the tab page list item. Depending on the placement of the tabs (top, bottom, left, right), the x position specifies the location of the left side of the tab page area or the left side of the tabs. Similarly, depending on the tab placement, the y position specifies either the top of the tabs to be included in the list or the top of the tab page display area. The size of the tab page area depends on the `width` and `height` fields. If `width` is 0, the width of the tab page area is based on the positioning and sizes of the different items on each of the

tab pages. If `height` is 0, the height of the tab page area is based on the sizes of the items on the pages.

- The `type` field should contain `TabPageList`.
- The `ID` determines the `DItem_TabPageListRsc` instance to load.
- The `attributes` field can be `ON` or `OFF`, and optionally combined with `hidden`. It will usually be `ON`.
- The `itemArg` field is unused and should usually be set to 0.
- The `label` field is unused and should be set to "".
- The `auxInfo` field is unused and should be set to "".

The following is an example of an item list specification for the tab page list item.

```
{{B1X,GENY(1),TPLW,0}, TabPageList, TPLISTID_ONE, ON, 0,"","},
```

Item Resource Specification

The tab page list resource is defined in a resource file along with the actual tab page resources themselves. Tab page list resources are defined by the following structure:

```
typedef struct ditem_tabpagelistrsc
   {
   long           width;         /* dialog coords */
   long           height;        /* dialog coords */
   long           synonymsId;
   ULong          helpInfo;
   ULong          helpSource;
   long           itemHookId;
   long           itemHookArg;
   ULong          attributes;
#if defined (resource)
   char           label[];
   DialogItemRsc  itemList[];
#else
   long           labelLength;
   char           label[1];
#endif
   } DItem_TabPageListRsc;

#if defind (resource)
   resourceclass DItem_TabPageListRsc RTYPE_TabPageList;
#define TabPageList RTYPE_TabPageList
#endif
```

The DItem_TabPageListRsc structure defines the list of DItem_TabPageRsc resource Ids which comprise the tab page list. This structure has the following unique fields. (See "Common item resource fields" for a description of any field not in this table.)

Field	Description
width	Specifies the overall width of the tab page display area, including the area for the tabs.
height	Specifies the overall height of the tab page display area, including the area for the tabs.
label	Unused and should be set to "".
itemList	An array of DialogItemRsc and lists the tab pages contained in this tab page list.
attributes	Specifies the attributes of the tab page list item. It is constructed by combining the constants from the attributes table (below) with the logical OR operator.

Value	Description
TABATTR_DEFAULT	Tab Attributes default includes: Tabs across the top, Single row of tabs (if tabs are too wide for display area, up-down buttons are generated for tab navigation), Tabs variable sized, centered labels & centered icons Tabs are medium height
TABATTR_TABSBOTTOM	Tabs across the bottom
TABATTR_TABSLEFT	Tabs along the left
TABATTR_TABSRIGHT	Tabs along the right
TABATTR_MULTIROW	Show multiple rows of tabs if too wide for a single row
TABATTR_TABSFITPAGEWIDTH	Tabs sized to fit the page area
TABATTR_TABSFIXEDWIDTH	Width of tabs based on largest label
TABATTR_LABELLEFTJUSTIFY	Left justify the tab label
TABATTR_LABELRIGHTJUSTIFY	Right justify the tab label
TABATTR_ICONLEFTJUSTIFY	Left justify the tab icon
TABATTR_ICONRIGHTJUSTIFY	Right justify the tab icon
TABATTR_TABSSHORT	Short tabs
TABATTR_TABSTALL	Tall tabs

The following is an example of a tab page list item resource.

```
DItem_TabPageListRsc TPLISTID_ONE =
{
   0, 0, NOSYNONYM, NOHELP, MHELP, NOHOOK, NOARG, TABATTR_DEFAULT,
   TXT_DialogTabs,
   {
   {{0,0,0,0}, TabPage, TABPAGEID_OPTIONS, ON, 0,"","},
   {{0,0,0,0}, TabPage, TABPAGEID_RADIOS,  ON, 0,"","},
   {{0,0,0,0}, TabPage, TABPAGEID_SCALES,  ON, 0,"","},
   }
};
```

Item Hook Function Messages

The following messages are sent to item hook functions that are attached to tab page list items: `DITEM_MESSAGE_CREATE`, `DITEM_MESSAGE_DESTROY`, `DITEM_MESSAGE_SYNCHRONIZE`, `DITEM_MESSAGE_BUTTON`, `DITEM_MESSAGE_KEYSTROKE`, `DITEM_MESSAGE_STATECHANGED`, `DITEM_MESSAGE_GETSTATE` and `DITEM_MESSAGE_SETSTATE`.

Tab Page

The individual tab page item `DialogItemRsc` entries are listed in the item list of the tab page list resource. The items to be displayed on a page are listed in the appropriate tab page resource definition.

Item List Specification

The `DialogItemRsc` field `extent` is unused for tab page items. The tab page list containing this tab page will place and size the tab according to the tab page list attributes.

The `type` field should contain `TabPage`.

The `ID` determines the `DItem_TabPageRsc` instance to load.

The `attributes` field can be `ON` or `OFF`, and optionally combined with `hidden`. It will usually be `ON`.

The `itemArg` field is unused and should usually be set to 0.

The `label` field is unused and should be set to “”.

The `auxInfo` field is unused and should be set to "".

The following is an example of a tab page item list specification.

```
{{0,0,0,0}, TabPage, TABPAGEID_OPTIONS, ON, 0,"","},
```

Item Resource Specification

The tab page resources are defined by the following structure:

```
typedef struct ditem_tabpagersc
   {
   ULong     commandNumber;
   ULong     commandSource;
   long      synonymsId;
   ULong     helpInfo;
   ULong     helpSource;
   long      itemHookId;
   long      itemHookArg;
   ULong     attributes;
   ULong     iconType;
   long      iconId;
#if defined (resource)
   char      label[];            /* Default Label */
   DialogItemRsc itemList[];     /* List of items in this container */
#else
   long      labelLength;
   char      label[1];
#endif
   } DItem_TabPageRsc;

#if defined (resource)
   resourceclass DItem_TabPageRsc RTYPE_TabPage;
#define TabPage RTYPE_TabPage
#endif
```

The `DItem_TabPageRsc` structure defines the list of items which comprise the display area of the tab page. This structure has the following unique fields. (See "Common item resource fields" for a description of any field not in this table.)

Field	Description
`iconType`	Indicates the type of icon the tab is using. This field should usually be `Icon` if you are using `IconRscs`. If not using icons, specify `NOTYPE`.

Field	Description
iconId	Specifies the ID of the icon instance to load and display on the tab. May be used in conjunction with the tab's label. If you are not using icons, specify NOICON. The tab page list attributes for icons determine the placement of the icon on the tab.
label	Specifies the tab's label. May be used in conjunction with the tab's icon. The tab page list attributes for labels determine the placement of the label on the tab.
itemList	An array of DialogItemRsc that lists the items contained in the display area of this tab page.

The following is an example of a tab page item resource.

```
DItem_TabPageRsc  TABPAGEID_OPTIONS=
{
   NOCMD, LCMD, NOSYNONYM, NOHELP, MHELP, NOHOOK, NOARG,
   TABATTR_DEFAULT, Icon, ICONID_TabOptions,
   TXT_TabPageOptions,
   {
   {{XRBL,Y11,0,0}, RadioButtonListX, RBLISTXID_TABPLACEMENT,
   ON, 0, "", ""},
   {{XGRP,Y11-YC,GBW,7*YC}, GroupBox, 0, ON, 0, TXT_TabPlacement, ""},
   {{X1,Y16,0,0}, ToggleButton, TOGGLEBTNID_AddTabPage, ON, 0, "", ""},
   }
};
```

The first item in a tab page item list must have a type and Id combination that is unique within the dialog box. Because most group box's are defined with an id of 0, we recommend you not start a tab page item list with a group box. In the example above, we use the RadioButtonListX as the unique item, as this item occurs only once in the dialog box.

Item Hook Function Messages

The following messages are sent to item hook functions that are attached to tab page items: DITEM_MESSAGE_CREATE, DITEM_MESSAGE_DESTROY, DITEM_MESSAGE_SYNCHRONIZE, DITEM_MESSAGE_BUTTON, DITEM_MESSAGE_KEYSTROKE, DITEM_MESSAGE_STATECHANGED, DITEM_MESSAGE_GETSTATE and DITEM_MESSAGE_SETSTATE.

Combo Box Item

The combo box item consists of a popup list and a selection field. The popup list presents the options a user can select and the selection field displays the current selection. The selection field may optionally be an edit field used to enter text not in the list. Any new values are then added to the list of available options.

Item List Specification

The `DialogItemRsc` field extent specifies the location and size of the combo box item selection field. The x position specifies the location of the left side of the selection field, and the y position specifies either the top of the selection field or, if the 'label on top' attribute is set, the top of the label. If width is 0, the width of the ComboBox selection field is calculated using the maximum number of characters the field cancontain. Specifying an explicit width is recommended. If height is 0, the height is calculated using the current dialog font.

The type field should contain `ComboBox`.

The ID determines the `DItem_ComboBoxRsc` instance to load.

The attributes field can be `ON` or `OFF`, and optionally combined with `hidden`. It will usually be `ON`.

The `itemArg` field is unused and should usually be set to 0.

The `label` field can override the label contained in the `DItem_ComboBoxRsc` instance.

The `auxInfo` field can currently override only the access string contained in the `DItem_ComboBoxRsc` instance.

The following is an example of an item list specification for the ComboBox item:

```
{{X_POS, Y_POS, CMBX_W, 0}, ComboBox, COMBOBOXID_STATES, ON,
0,"","",},
```

Item Resource Specification

The ComboBox resource is defined in a resource file with the following structure:

```
typedef struct ditem_comboboxrsc
  {
  ULong     commandNumber;
  ULong     commandSource;
  long      synonymsId;
  ULong     helpInfo;
```

```
    ULong     helpSource;
    long      itemHookId;
    long      itemHookArg;
    byte      maxSize;            /* max # of chars in text field */
    char formatToDisplay[16];/* format str to convert from internal */
    char formatToInternal[16];/* convert to internal from display str */
    char      minimum[16];        /* minimum value */
    char      maximum[16];        /* maximum value */
    ULong     mask;               /* only used with integer types */
    long      stringListId;       /* Id of initial string list used */
    long      nRows;              /* Number of rows to show in list */
    UShort    gapWidth;           /* Gap between textedit & button */
    UShort    listWidth;          /* Width of dropdown listbox */
    UShort    valueColumnIndex;   /* Index of Column to get value from */
    ULong     attributes;         /* other attributes */
#if defined (resource)
    char                label[];
    char                accessStr[];
    DItem_ListColumnRsc listColumns[];
#else
    long                labelLength;
    char                label[1];
#endif
} DItem_ComboBoxRsc;

#if defined (resource)
    resourceclass DItem_ComboBoxRscRTYPE_ComboBox extendedAttributes;
#define ComboBox RTYPE_ComboBox
#endif
```

For consistency with MicroStation's dialog boxes, a combo box item's label should end with a colon.

The following fields are described in the Text Item resource specification and are functionally identical in the combo box:

`maxSize`, `formatToDisplay`, `formatToInternal`, `minimum`, `maximum` and `mask`.

The following fields are described in the List Box Item resource specification and are functionally identical in the combo box:

`nRows` and `listColumns`.

The DItem_ComboBoxRsc structure has the following unique fields. (See "Common item resource fields" for a description of any field not in this table.)

Field	Description
stringListId	Resource Id of a String List to be used to fill the popup list.
gapWidth	The gap between the selection field and the popup list activation button (specified in dialog coordinates (dcoords)).
listWidth	The width of popup List Box (specified in dialog coordinates (dcoords)).
valueColumnIndex	Specifies the index of the list column to obtain the selection field value from.
attributes	Specifies the attributes of the combo box item. It is constructed by combining the constants from the attributes table (below) with the logical OR operator.

Valid constants for use with the attributes field:

Constant	Description
COMBOATTR_READONLY	The selection field is read only; no editting is allowed
COMBOATTR_SORT	Sort members in string list initially and as new values are added
COMBOATTR_DRAWPREFIXICON	Draw a specified icon preceding the text in the popup List Box
COMBOATTR_INDEXISVALUE	Use the list box member index as the value; not the actual value of the list box member
COMBOATTR_AUTOADDNEWSTRINGS	Automatically add new values to the string list after they are entered in the selection field
COMBOATTR_LABELABOVE	Place the label above the selection field

The following are two examples of a ComboBox item resource.

```
DItem_ComboBoxRsc COMBOBOXID_STATES=
  {
  NOCMD, LCMD, NOSYNONYM, NOHELP, MHELP, HOOKID_ComboStates,
  NOARG, 2, "", "", "", "", NOMASK, STRLISTID_ComboBoxStates, 15, 0,
  TW*4, 0, COMBOATTR_READONLY | COMBOATTR_INDEXISVALUE, "States:", "",
  { {TW, 2, 0, "Code"}, {0, 15, 0, "State Name"}, }
  };

DItem_ComboBoxRsc COMBOBOXID_EDITTABLE=
  {
  NOCMD, LCMD, NOSYNONYM, NOHELP, MHELP, HOOKID_ComboBoxes, NOARG,
```

```
    12, "%.1lf", "%lf", "0.0", "100.0", NOMASK, 0, 6, 4, 0, 0,
    COMBOATTR_AUTOADDNEWSTRINGS, "ComboBox 2:", "", { {0, 12, 0, ""}, }
};
```

Item Hook Function Messages

The following messages are sent to item hook functions that are attached to ComboBox items: `DITEM_MESSAGE_CREATE`, `DITEM_MESSAGE_DESTROY`, `DITEM_MESSAGE_SYNCHRONIZE`, `DITEM_MESSAGE_BUTTON`, `DITEM_MESSAGE_KEYSTROKE`, `DITEM_MESSAGE_STATECHANGED`, `DITEM_MESSAGE_GETSTATE` and `DITEM_MESSAGE_SETSTATE`.

ComboBox Item Functions

`mdlDialog_comboBoxGetInfo` - gets the attributes of a combo box.

`mdlDialog_comboBoxSetInfo` - sets the attributes of a combo box.

`mdlDialog_comboBoxGetTextP` - returns the `RawItemHdr` pointer of the attached text item.

`mdlDialog_comboBoxGetListBoxP` - returns the `RawItemHdr` pointer of the attached popup list box.

`mdlDialog_comboBoxGetStrListP` - returns a pointer to the string list that is currently connected to the combo box.

`mdlDialog_comboBoxSetStrListP` - sets the string list that the combo box will manipulate.

`mdlDialog_comboBoxSetPopupState` - may be used to programmatically open or close the popup list box.

SpinBox Item

The SpinBox item consists of a text field and up-down arrows. The arrows may be used to increment or decrement the numeric value in the text field.

Item List Specification

The `DialogItemRsc` field extent specifies the location and size of the SpinBox item. The x position specifies the location of the left side of the text field, and the y position specifies either the top of the text field or, if the 'label on top' attribute is set, the top of

the label. If width is 0, the width of the SpinBox is calculated using the maximum number of characters the text field can contain. Specifying an explicit width is recommended. If height is 0, the height is calculated using the current dialog font.

The `type` field should contain `SpinBox`.

The ID determines the `DItem_SpinBoxRsc` instance to load.

The `attributes` field can be `ON` or `OFF`, and optionally combined with `hidden`. It will usually be `ON`.

The `itemArg` field is unused and should usually be set to 0.

The `label` field can override the label contained in the `DItem_SpinBoxRsc` instance.

The `auxInfo` field can currently override only the access string contained in the `DItem_SpinBoxRsc` instance.

The following is an example of an item list specification for the SpinBox item:

```
{{X_POS, Y_POS, SPNBX_W, 0}, SpinBox, SPINBOXID_EXAMPLE, ON,
0,"","",},
```

Item Resource Specification

The SpinBox resource is defined in a resource file with the following structure:

```
typedef struct ditem_spinboxrsc
    {
    ULong    commandNumber;
    ULong    commandSource;
    long     synonymsId;
    ULong    helpInfo;
    ULong    helpSource;
    long     itemHookId;
    long     itemHookArg;
    byte     maxSize;            /* max # of chars in field */
    charformatToDisplay[16];/* format str to convert from internal */
    charformatToInternal[16];/* convert to internal from display str */
    double   minValue;           /* minimum value */
    double   maxValue;           /* maximum value */
    double   incAmount;          /* UpDown increment amount */
    ULong    mask;               /* only used with integer types */
    ULong    attributes;         /* other attributes */
#if defined (resource)
    char     label[];
```

```
    char    accessStr[];
#else
    long    labelLength;
    char    label[1];
#endif
    } DItem_SpinBoxRsc;

#if defined (resource)
   resourceclass DItem_SpinBoxRsc RTYPE_SpinBox extendedAttributes;
#define SpinBox RTYPE_SpinBox
#endif
```

For consistency with MicroStation's dialog boxes, a SpinBox item's label should end with a colon.

The following fields are described in the Text Item resource specification and are functionally identical in the SpinBox: `maxSize`, `formatToDisplay`, `formatToInternal`, `minValue`, `maxValue` and `mask`.

The `DItem_SpinBoxRsc` structure has the following unique fields. (See "Common item resource fields" for a description of any field not in this table.)

Field	Description
`attributes`	Specifies the attributes of the SpinBox item.

Valid constants for use with the attributes field:

Constant	Description
`SPINATTR_LABELABOVE`	Place the label above the text field.

The following is an example of a SpinBox item resource.

```
DItem_SpinBoxRsc SPINBOXID_Example=
    {
    NOCMD, LCMD, NOSYNONYM, NOHELP, MHELP, HOOKID_spinBox, NOARG,
    8, "%.1lf in", "%lf", 1.0, 100.0, 0.5, NOMASK, SPINATTR_LABELABOVE,
    "~Spin Box Label:", "appG.spinBoxValue"
    };
```

Item Hook Function Messages

The following messages are sent to item hook functions that are attached to SpinBox items: `DITEM_MESSAGE_CREATE`, `DITEM_MESSAGE_DESTROY`, `DITEM_MESSAGE_SYNCHRONIZE`, `DITEM_MESSAGE_BUTTON`, `DITEM_MESSAGE_KEYSTROKE`, `DITEM_MESSAGE_STATECHANGED`, `DITEM_MESSAGE_GETSTATE` and `DITEM_MESSAGE_SETSTATE`.

17 Dialog Box Manager Hook Functions

Hook functions give an MDL dialog box programmer the ability to modify or amplify the default behavior of the standard dialog items. For generic items, the item hook function actually implements the entire behavior of the item, thereby allowing complete creation of custom items.

Hook Functions

There are two types of hook functions: **dialog hook** functions and **item hook** functions. A dialog hook function is attached to a dialog box and receives messages that apply to the dialog box as a whole. An item hook function is attached to an item within a dialog box and receives messages that apply only to that item.

Dialog hook functions are passed a single argument, a pointer to a `DialogMessage`. Item hook functions are also passed a single argument, but this argument points to a `DialogItemMessage`.

Both hook function types are of type `void`. The hook functions respond to messages by setting the `msgUnderstood`, and other message fields that depend on the message type. Fields used to reply to the dialog box manager will be zero (`FALSE`) when sent to a hook function. The hook function sets a response field only if it needs to answer a non-zero (`TRUE`) result.

See “Dialog Box Manager Basic Concepts” on page 15-6 for more information on hook functions and messages.

See “Debugging Hook Functions” on page 19-3 for information on how to have the dialog box manager print detailed information about the messages it sends to dialog and item hook functions. This is the best technique for learning about the type, contents and order of the messages that can be sent to hook functions.

Dialog Hook Functions

A basic dialog hook function should look like the following:

```
Public void dialogName_dialogHook
(
DialogMessage *dmP
)
{
   dmP->msgUnderstood = TRUE;
   switch(dmP->messageType)
   {
        case DIALOG_MESSAGE_CREATE:
        {
             dmP->u.create.interests.updates = TRUE;
             dmP->u.create.interests.mouses = TRUE;
             ...
             break;
        }
        case DIALOG_MESSAGE_DESTROY:
        {
             ...
             break;
        }
        default:
        {
             dmP->msgUnderstood = FALSE;
             break;
        }
   }
}
```

Dialog hook functions are set up as one switch statement that tests for various messages and then handles them.

DialogMessage structure

The `dialogmessage` structure that is the only argument to a dialog hook function is declared as follows (from **dlogitem.h**):

```
typedef struct dialogmessage
{
   boolean msgUnderstood;    /* <= message understood? */
   int messageType;          /* => message type */
   DialogBox *db;            /* => dialog box handle */
   long dialogId;            /* => resource id of dialog box */
   void *userDataP;          /* => set by user during CREATE */
   union
   {
        /* structures that contain information
           that depend on messageType */
   } u;
} DialogMessage;
```

The structures in the union *u* will be discussed in the section on the individual dialog messages.

Field	Description
`msgUnderstood`	Indicates whether the dialog hook function understood the message. The dialog hook function sets this field to `TRUE` if it understands the message. If the message is not understood, the dialog box manager takes specific action depending on the value of `messageType`. Some messages also have a `hookHandled` member. This member indicates that the hook function not only understood the message, but handled it completely and the dialog box manager should not perform any further default processing.
`messageType`	Specifies the type of dialog message. The individual messages are discussed in the next section.
`db`	Contains an opaque pointer to the dialog box intended for the message. This pointer is usually just passed on to other dialog box manager functions.
`dialogId`	Contains the ID of the dialog box the message is intended for.
`userDataP`	Contains a pointer that can be set up in response to the `DIALOG_MESSAGE_CREATE` message. This pointer will then be passed to the dialog hook function for all other messages.

An “opaque” pointer is a pointer that an MDL program can receive from and pass to functions, but should never attempt to reference.

Dialog Hook Function Messages

The messageType field of a dialog message determines the message's type. Based on the field's value, additional information can be present in the u field of the DialogMessage structure. This section discusses each individual dialog message, including any additional information.

The following messages are listed in this section:

Focus Messages (page 17-10)	Sent
DIALOG_MESSAGE_CREATE	Before any DITEM_MESSAGE_CREATE messages are sent to the item hook functions.
DIALOG_MESSAGE_INIT	After DITEM_MESSAGE_CREATE messages are sent to all the item hook functions.
DIALOG_MESSAGE_BEFOREDESTROY	Before the dialog items have been sent the DITEM_MESSAGE_DESTROY message and the dialog box has been sent the DIALOG_MESSAGE_HIDE message.
DIALOG_MESSAGE_HIDE	When the dialog box is about to be hidden.
DIALOG_MESSAGE_DESTROY	When the dialog box is about to be destroyed.

Focus Messages (page 17-10)	Sent
DIALOG_MESSAGE_FOCUSIN	When the dialog box gains the input focus.
DIALOG_MESSAGE_FOCUSOUT	When the dialog box loses the input focus.
DIALOG_MESSAGE_ITEMFOCUSIN	When an item in the dialog box gains the input focus.
DIALOG_MESSAGE_ITEMFOCUSOUT	When an item in the dialog box loses the input focus.
DIALOG_MESSAGE_KEYSTROKE	When a key is pressed while the dialog box has the input focus.
DIALOG_MESSAGE_ACTIVATE	When a specific menu bar menu item or dialog item is activated.

Size Messages (page 17-16)	Sent
DIALOG_MESSAGE_RESIZE	After the dialog box is resized.
DIALOG_MESSAGE_CALCSIZE	When the dialog box manager needs to know what the dialog box's size would be if the dialog font height were changed to a specific value.
DIALOG_MESSAGE_FONTCHANGED	When the dialog box is moved between screens using differing font sizes.
DIALOG_MESSAGE_MINIMIZE	To the dialog hook function when the user presses the minimize button in the dialog title bar or the `mdlWindow_minimize` function is called.
DIALOG_MESSAGE_MAXIMIZE	To the dialog hook function when the user presses the maximize button in the dialog title bar or the `mdlWindow_maximize` function is called.

Button Messages (page 17-20)	Sent
DIALOG_MESSAGE_PREBUTTON	When a mouse button event occurs.
DIALOG_MESSAGE_BUTTON	Somewhere in the dialog box.
DIALOG_MESSAGE_ACTIONBUTTON	When the user presses and releases an OK, Cancel, Apply, Reset or Default button.

Open & Close Messages (page 17-24)	Sent
DIALOG_MESSAGE_CHILDDESTROYED	To a dialog box's parent when the dialog is destroyed.
DIALOG_MESSAGE_ANOTHEROPENED	To all dialogs that set the `otherDialogs` field of `DialogHookInterests` to equal `TRUE` when any dialog is opened.
DIALOG_MESSAGE_ANOTHERCLOSED	To all dialogs that set the `otherDialogs` field of `DialogHookInterests` to `TRUE` when any dialog is destroyed.

Miscellaneous Messages (page 17-26)	Sent
DIALOG_MESSAGE_UPDATE	After the dialog box manager has drawn the contents of a dialog box.
DIALOG_MESSAGE_SYNCH	When the dialog box is synched by the mdlDialog_itemsSynch function.
DIALOG_MESSAGE_STATECHANGED	To the dialog hook function after the dialog item state has changed.
DIALOG_MESSAGE_USER	To a dialog hook function with the mdlDialog_hookDialogSendUserMsg function.

Non-requested Messages

The following messages, documented in this section, do not have to be requested; they are always sent:

DIALOG_MESSAGE_CREATE

DIALOG_MESSAGE_INIT

DIALOG_MESSAGE_BEFOREDESTROY

DIALOG_MESSAGE_HIDE

DIALOG_MESSAGE_DESTROY

DIALOG_MESSAGE_CREATE message

The DIALOG_MESSAGE_CREATE message is sent before any DITEM_MESSAGE_CREATE messages are sent to the item hook functions. Additional information is included in the create member of the DialogMessage union u as follows:

```
typedef struct dialoghookinterests    /* set member TRUE if
                                            notification wanted */
{
   ULong  updates:1;          /* after an item receives a draw msg */
   ULong  mouses:1;           /* on button msgs */
   ULong  keystrokes:1;       /* on keystroke events */
   ULong  dialogFocuses:1;    /* when dialog gets/loses focus */
   ULong  itemFocuses:1;      /* when item get/loses focus */
   ULong  synchs:1;           /* when item is synchronized */
   ULong  resizes:1;          /* when dialog resized or moved */
   ULong  calcSizes:1;        /* when dialog is moved to 2nd screen &
                                 dialog mgr. needs to know the size */
   ULong  fontChanges:1;      /* when dialog is moved to 2nd screen &
                                 the font size will be affected */
   ULong  nonDataPoints:1;    /* send RESET & TENTATIVE msgs */
   ULong  otherDialogs:1;     /* another dialog open and close msgs */
```

```
        ULong  stateChangeds:1;  /* after item state changes */
        ULong  minimizes:1;      /* when dialog is minimized */
        ULong  maximizes:1;      /* when dialog is maximized */
        ULong  activates:1;      /* after item is activated */
        ULong  preButtons:1;     /* before normal BUTTON message */
        ULong  unused:16;
      } DialogHookInterests;     /* CREATE, INIT, HIDE, BEFOREDESTROY and
                                    DESTROY ALWAYS SENT */

      struct
      {
        boolean createFailed;         /* <= set to TRUE if error */
        DialogHookInterests interests;/* <= */
        void **userDataPP;            /* <= */
        DialogBoxRsc *dialogBoxRP;    /* => */
    } create;
```

Field	Description
createFailed	Indicates whether the dialog hook function's handling of the create message failed. If so, the function sets this field to TRUE. The dialog box will then remain unopened and an alert will display instead.
interests	Contains bitfields that the dialog hook function should set to TRUE to indicate which dialog messages are needed. For example, if update messages should be sent to the dialog hook function, the updates bitfield must be set to TRUE. DIALOG_MESSAGE_CREATE, DIALOG_MESSAGE_INIT, DIALOG_MESSAGE_HIDE, DIALOG_MESSAGE_BEFOREDESTROY, DIALOG_MESSAGE_DESTROY and DIALOG_MESSAGE_USER messages will always be sent to dialog hook functions. If any other messages are required, the appropriates bits must be set in the interests field when the function receives the DIALOG_MESSAGE_CREATE message.
userDataPP	Stores a pointer to a user-defined (and allocated) data area. This pointer will be passed back to the dialog hook function in all subsequent messages in the userDataP field. This pointer points to another pointer. To set the pointer, enter a command similar to the following: *dmP->u.create.userDataPP = &myData;
dialogBoxRP	Points to the dialog box's loaded resource.

Dialog hook functions will usually want to perform any memory allocations for the entire dialog box in their DIALOG_MESSAGE_CREATE message handling,. They will then store a pointer to the allocated data in the variable pointed at by userDataPP. Remember that at this point, no items have been created so it is illegal (and meaningless) to perform any actions on them.

DIALOG_MESSAGE_INIT message

The DIALOG_MESSAGE_INIT message is sent after DITEM_MESSAGE_CREATE messages are sent to all the item hook functions. Additional information is included in the init member of the DialogMessage union u as follows:

```
struct
{
boolean initFailed;    * <= set to TRUE if error occurred */} init;
```

Field	Description
initFailed	Indicates whether the dialog hook function's handling of the init message failed. If so, the function sets this field to TRUE. The dialog box will then remain unopened and an alert will display instead.

Sometimes dialog box-wide initializations cannot be performed until all individual items in the dialog box are created. For this reason, both DIALOG_MESSAGE_CREATE and DIALOG_MESSAGE_INIT messages exist. When the DIALOG_MESSAGE_INIT message is received by a dialog hook function, it serves as a signal that all of the dialog box's items have been created and can be modified.

DIALOG_MESSAGE_BEFOREDESTROY message

The DIALOG_MESSAGE_BEFOREDESTROY message is sent before the dialog items have been sent the DITEM_MESSAGE_DESTROY message and the dialog box has been sent the DIALOG_MESSAGE_HIDE message. This message allows the dialog box hook function to perform additional cleanup prior to the dialog items being destroyed and/or to stop the dialog box from being destroyed. Any dialog item related memory not freed by dialog item hooks should be freed now. Additional information is included in the destroy member of the DialogMessage union u as follows:

```
struct
{
   boolean stopDestroy;      /* <= TRUE means stop destroy */
   boolean userRequested;    /* => TRUE means user initiated destroy
                                   FALSE means program initiated destroy */
} beforeDestroy;
```

Field	Description
stopDestroy	If set to TRUE by the dialog hook function, the destroy processing by MicroStation will be stopped and the dialog box will not be destroyed or hidden.
userRequested	If TRUE, this field indicates that the destroy was initiated by the user rather than by an application program.

DIALOG_MESSAGE_HIDE message

The DIALOG_MESSAGE_HIDE message is sent when the dialog box is about to be hidden. Additional information is included in the hide member of the DialogMessage union u as follows:

```
struct
{
   int reason;    /* => reason for dialog being hidden */
} hide;
```

Field	Description
reason	Contains a code indicating the reason the dialog is being hidden. Possible values are shown below.

.

reason **Value**	**Meaning**
HIDE_HideWindow	The window has been hidden by an application.
HIDE_ExitingMicroStation	The window is being hidden due to MicroStation being terminated. A DESTROY message will follow the HIDE message in this case.
HIDE_WindowClose	The window is being hidden prior to being closed. A DESTROY message will follow the HIDE message in this case.
HIDE_NoDgnFile	The window is being hidden because there is no open design file.
HIDE_UserClose	The window is being hidden as a result of the user closing the window. A DESTROY message will follow the HIDE message in this case.
HIDE_MdlUnload	The window is being closed as a result of the owning MDL application being unloaded from memory. A DESTROY message will follow the HIDE message in this case.

DIALOG_MESSAGE_DESTROY message

The DIALOG_MESSAGE_DESTROY message is sent when the dialog box is about to be destroyed. All item hook functions have received DITEM_MESSAGE_DESTROY messages. Any generic dialog related memory that was allocated in response to the DIALOG_MESSAGE_CREATE or DIALOG_MESSAGE_INIT message should be freed now.

Additional information is included in the destroy member of the DialogMessage union u as follows:

```
struct
{
   int actionType;      /* => */
} destroy;
```

Field	Description
actionType	Contains the value last set by a call to the mdlDialog_lastActionTypeSet function. If the standard push button hook function is attached to a push button, this field contains the push button's itemHookArg field (usually an ACTIONBUTTON_ constant).

Focus Messages

The following messages can be sent when the focus changes:

DIALOG_MESSAGE_FOCUSIN

DIALOG_MESSAGE_FOCUSOUT

DIALOG_MESSAGE_ITEMFOCUSIN

DIALOG_MESSAGE_ITEMFOCUSOUT

DIALOG_MESSAGE_KEYSTROKE

DIALOG_MESSAGE_ACTIVATE

The first four of these messages all use the focusOutType field, which can have the following values:

focusOutType Value	Meaning
FOCUSOUT_KEYSWITCH	Focus has changed as a result of a keyboard action.
FOCUSOUT_BUTTONSWITCH	Focus has changed as a result of a button action.
FOCUSOUT_SETITEM	Focus has changes as a result of a dialog item programmatically being assigned focus.
FOCUSOUT_HIDEITEM	Focus has changed as a result of the previous item being hidden.
FOCUSOUT_DISABLEITEM	Focus has changed as a result of the previous item being disabled.
FOCUSOUT_SWITCHDIALOG	Focus has changed as a result of a new dialog box receiving focus.
FOCUSOUT_APPLYDIALOG	Focus has changed as a result of an apply operation being performed on the dialog box containing the item.

DIALOG_MESSAGE_FOCUSIN message

The DIALOG_MESSAGE_FOCUSIN message is sent when the dialog box gains the input focus. If the item that will gain the input focus has an item hook function, a DITEM_MESSAGE_FOCUSIN message was already sent. Additional information is included in the focusIn member of the DialogMessage union u as follows:

```
struct
{
   int       itemIndex;     /* => item about to get focus */
   int       focusOutType;  /* => prev item focus out reason */
   RawItemHdr *riP;         /* => item about to get focus */
} focusIn;
```

Field	Description
itemIndex	Specifies the item index (0-based) of the item that just gained the input focus.
focusOutType	Specifies the reason the focus changed from the previous item to the one about to get focus. Possible values for this field are listed at the top of this section and are defined in msdefs.h.
riP	Specifies a pointer to the Raw Item Header of the item to receive focus.

DIALOG_MESSAGE_FOCUSOUT message

The DIALOG_MESSAGE_FOCUSOUT message is sent when the dialog box loses the input focus. If the item with the input focus has an item hook function, a DITEM_MESSAGE_FOCUSOUT message was already sent. Additional information is included in the focusOut member of the DialogMessage union u as follows:

```
struct
   {
   boolean   outOfRange;         /* <= if current value out of range */
   int       itemIndex;          /* => child item that lost focus */
   int       focusOutType;       /* => reason for focus out */
   int       moveDirection;      /* => 1: forward, -1: backward */
   boolean   hookHandled;        /* <= TRUE if handled by hook */
   int       nextFocusItemIndex; /* <=> next item to get focus */
   RawItemHdr *nextFocusRiP;     /* <=> next item to get focus */
   RawItemHdr *riP;              /* => item that just lost focus */
   } focusOut;
```

Field	Description
outOfRange	Indicates whether the current input focus item contains a value that is out of range. The dialog hook function sets this field to TRUE for an out-of-range value., in which case the dialog will not lose the input focus unless it is forced to. (Some methods of changing the input focus allow out-of-range focus-out errors to be ignored).
itemIndex	Specifies the item index (0-based) of the item that just lost the input focus.
focusOutType	Specifies the reason the focus changed from the previous item to the one about to get focus. Possible values for this field are listed at the top of this section and are defined in msdefs.h.
moveDirection	This field is not used for this message.
hookHandled	This field is not used for this message.
nextFocusItemIndex	This field is not used for this message.
nextFocusRiP	This field is not used for this message.
riP	Specifies a pointer to the Raw Item Header of the item losing focus.

DIALOG_MESSAGE_ITEMFOCUSIN message

The DIALOG_MESSAGE_ITEMFOCUSIN message is sent when an item in the dialog box gains the input focus. If the item that will gain the input focus has an item hook function, a DITEM_MESSAGE_FOCUSIN message was already sent. Additional information is included in the focusIn member of the DialogMessage union u as follows:

```
struct
{
   int            itemIndex;     /* => item about to get focus */
   int            focusOutType;  /* => prev item focus out reason */
   RawItemHdr     *riP;          /* => item about to get focus */
} focusIn;
```

Field	Description
itemIndex	Specifies the item index (0-based) of the item that just gained the input focus.
focusOutType	Specifies the reason the focus changed from the previous item to the one about to get focus. Possible values for this field are listed at the top of this section and are defined in msdefs.h.
riP	Specifies a pointer to the Raw Item Header of the item to receive focus.

DIALOG_MESSAGE_ITEMFOCUSOUT message

The DIALOG_MESSAGE_ITEMFOCUSOUT message is sent when an item in the dialog box loses the input focus. If the item with the input focus has an item hook function, a DITEM_MESSAGE_FOCUSOUT message was already sent. Additional information is included in the focusOut member of the DialogMessage union u as follows:

```
struct
{
   boolean   outOfRange;    /* <= if current value out of range */
   int       itemIndex;     /* => item just lost focus */
   int       focusOutType;  /* => reason for focus out */
   int       moveDirection; /* => 1: forward, -1: backward */
   boolean   hookHandled;   /* <= TRUE if handled by hook */
   int       nextFocusItemIndex; /* <=> next item to get focus */
   RawItemHdr *nextFocusRiP;/* <=> next item to get focus */
   RawItemHdr     *riP;     /* => item that just lost focus */
} focusOut;
```

Field	Description
outOfRange	Indicates whether the current input focus item contains a value that is out of range. The dialog hook function sets this field to TRUE for an out-of-range value., in which case the dialog item will not lose the input focus unless it is forced to. (Some methods of changing the input focus allow out-of-range focus-out errors to be ignored).
itemIndex	Specifies the item index (0-based) of the item that just lost the input focus.
focusOutType	Specifies the reason the focus changed from the previous item to the one about to get focus. Possible values for this field are listed at the top of this section and are defined in **msdefs.h**.
moveDirection	Specifies the direction from the current item position in which the focus is shifting. A value of 1 means forward and -1 means backward.
hookHandled	Indicates whether the dialog box manager should continue default handling for the focus out event. The dialog hook function should set this field to TRUE to stop the default handling. In this case, the dialog box manager will validate the nextFocusItemIndex and nextFocusRiP fields to ensure that they point at a valid item, and will set the next input focus item to the proper value if so.
nextFocusItemIndex	Specifies the item index (0-based) of the next dialog item to receive input focus. This field is input-only for this message.
nextFocusRiP	Specifies the pointer to the Raw Item Header of the next item to receive input focus.
riP	Specifies a pointer to the Raw Item Header of the item losing focus.

DIALOG_MESSAGE_KEYSTROKE message

The DIALOG_MESSAGE_KEYSTROKE message is sent when a key is pressed while the dialog box has the input focus. This message is sent to the dialog hook function before a DITEM_MESSAGE_KEYSTROKE message is sent to the relevant item hook function to allow the dialog hook function to modify the key. Additional information is included in the keystroke member of the DialogMessage union u as follows:

```
struct
{
   boolean   hookHandled;        /* <= only for hooks,TRUE = handled */
   int       moveDirection;      /* <= -1 or 1 indicate next field */
   int       keystroke;          /* <=> key that was pressed*/
   int       qualifierMask;      /* => shift/ctrl/alt status */
   int       itemIndex;          /* => item key WOULD be send to */
   Inputq_element *iqelP;        /* => source queue element */
   boolean   isAccelerator;      /* <=> TRUE = accelerator keystroke */
   ULong     commandNumber;      /* <=> Command num for accelerator */
   char      *unparsedP;         /* <=> Parm string for cmd function */
   char      *commandTaskIdP;    /* <=> Task containing cmd function */
   RawItemHdr *riP;              /* => Item that would get keystroke */
} keystroke;
```

Field	Description
hookHandled	Indicates whether the dialog box manager should continue default handling for the keystroke. The dialog hook function should set this field to TRUE to stop the default handling. In this case, a DITEM_MESSAGE_KEYSTROKE message will not be sent to the item hook function of the item that currently has the input focus.
moveDirection	Indicates that the key pressed caused the input focus to be moved if hookHandled is TRUE. 1 means that the input focus should be moved forward (later in the dialog item list) to an item that can accept the focus. -1 means that the input focus should be moved backward (earlier in the dialog item list) to an item that can accept the focus.
keystroke	Contains the keystroke that was just pressed. If the dialog hook function changes this field, the new keystroke value will be the value sent to any relevant item hook function.
qualifierMask	Contains the state of the qualifier keys at the time of the keystroke event. The possible qualifier keys are SHIFTKEY, CTRLKEY and ALTKEY. These bitmasks are defined in **keys.h**.
itemIndex	Specifies the item that currently has the input focus.
iqelP	Contains a pointer to the raw input queue element for the keystroke event. See **msinputq.h** for the declaration of the InputQ_element structure.

Field	Description
isAccelerator	A value of TRUE for this field indicates that the current keystroke is being interpreted as a command accelerator. If this field is TRUE, then the three fields describing the command information contain valid data. The hook function may change the value of this field to perform special case handling of keystroke accelerators.
commandNumber	Contains the command number which will be queued to MicroStation after all keystroke processing is completed. This field is only valid of isAccelerator is TRUE.
unparsedP	Contains a pointer to the unparsed character string which will be passed to the command function when the command is executed. This field is only valid of isAccelerator is TRUE.
commandTaskIdP	Contains a pointer to the task ID of the application which implements the command indicated by the commandNumber field. This field is only valid of isAccelerator is TRUE.
riP	Contains a pointer to the Raw Item Header of the dialog item which would/will receive the keystroke.

DIALOG_MESSAGE_ACTIVATE message

The DIALOG_MESSAGE_ACTIVATE message is sent to the dialog hook function when one of the following occurs:

1. The user uses a keyboard accelerator key to activate a specific menu bar menu item.
2. The user uses a keyboard mnemonic to activate a specific dialog item.
3. An application calls `mdlDialog_pushButtonActivate` to activate a push button item.

If the dialog item being activated has a hook function, the DITEM_MESSAGE_ACTIVATE message has already been sent to the dialog item.

Additional information is included in the activate member of the DialogMessage union u as follows:

```
struct
{
   ULong     couldSetState;      /* <= if not modal, could set state */
   RawItemHdr *activatedRiP;     /* => item to be activated */
   int       itemIndex;          /* => index of item to be activated */
   int       keystroke;          /* => virtual keystroke */
   int       rawKeystroke;       /* => raw keystroke */
   int       qualifierMask;      /* => modified key status */
   boolean   isAccelerator;      /* => key is accelerator */
```

```
    ULong     commandNumber;      /* => accelerator key command */
    char      *unparsedP;         /* => accelerator key cmd parms */
    char      *commandTaskIdP;    /*=> accelerator key cmd task id */
} activate;
```

Field	Description
couldSetState	This field is unused in this message.
itemIndex	Specifies the item index (0-based) of the item that just lost the input focus.
activatedRiP	Contains a pointer to the Raw Item Header of the dialog item which is being activated.
keystroke	Contains the platform independent (virtual) keystroke that was just pressed.
rawKeystroke	Contains the raw, native keystroke just pressed.
qualifierMask	Contains the state of the qualifier keys at the time of the keystroke event. The possible qualifier keys are SHIFTKEY, CTRLKEY, and ALTKEY. These bitmasks are defined in **keys.h**.
isAccelerator	A value of TRUE for this field indicates that the current keystroke is being interpreted as a command accelerator. If this field is TRUE, then the three fields describing the command information contain valid data.
commandNumber	Contains the command number which will be queued to MicroStation after all keystroke processing is completed. This field is only valid of isAccelerator is TRUE.
unparsedP	Contains a pointer to the unparsed character string which will be passed to the command function when the command is executed. This field is only valid of isAccelerator is TRUE.
commandTaskIdP	Contains a pointer to the task ID of the application which implements the command indicated by the commandNumber field. This field is only valid of isAccelerator is TRUE.

Size Messages

The following messages, documented in this section, are sent either before or after a dialog or the items in it need to be or are resized.

DIALOG_MESSAGE_RESIZE

DIALOG_MESSAGE_CALCSIZE

DIALOG_MESSAGE_FONTCHANGED

DIALOG_MESSAGE_MINIMIZE

DIALOG_MESSAGE_MAXIMIZE

DIALOG_MESSAGE_RESIZE message

The DIALOG_MESSAGE_RESIZE message is sent after the dialog box is resized. Additional information is included in the resize member of the DialogMessage union u as follows:

```
struct
{
   long      whichCorners;       /* => which corners moved */
   Rectangle oldFrame;           /* => old frame rectangle */
   Rectangle oldContent;         /* => old content rectangle */
   Rectangle newContent;         /* => new content rectangle */
   boolean   forceCompleteRedraw; /* <= force dialog redraw */
} resize;
```

Field	Description
whichCorners	Specifies which of the dialog boxes corners moved. See the **mdlWindow_resize** function for a description of possible values.
oldFrame	Contains the old size of the dialog frame.
oldContent	Contains the old size of the dialog content area.
newContent	Contains the new size of the dialog content area in global coordinates.
forceCompleteRedraw	Indicates that the hook function wants the complete contents of the dialog box redrawn aftrer the resize operation is complete. By default, the dialog box is not redrawn if the visible portions of the dialog box do not change and the left and top borders are not moved.

DIALOG_MESSAGE_CALCSIZE message

The DIALOG_MESSAGE_CALCSIZE message is sent when the dialog box manager needs to know what the dialog box's size would be if the dialog font height were changed to a specific value. Additional information is included in the calcSize member of the DialogMessage union u as follows:

```
struct
{
   boolean   hookHandled;        /* <= TRUE if handled */
   int       newWidth;           /* <= calculated width (in pixels) */
   int       newHeight;          /* <= calculated height (in pixels) */
```

```
    int         dialogWidth;        /* <> in dialog coord units */
    int         dialogHeight;       /* <> in dialog coord units */
    int         newFontHeight;      /* => new font's height */
    int         oldFontHeight;      /* => old font's height */
} calcSize;
```

Field	Description
hookHandled	Indicates whether the dialog box manager should continue default handling for calculating the dialog box size. The dialog hook function should set this field to TRUE to stop the default handling. In this case, the dialog hook function must set the newWidth, newHeight, dialogWidth and dialogHeight fields correctly.
newWidth	Contains the width (in pixels) the dialog box will have when the height of the dialog font becomes newFontHeight. The dialog hook function should set this field if hookHandled is set to TRUE.
newHeight	Contains the height (in pixels) the dialog box will have when the height of the dialog font becomes newFontHeight. The dialog hook function should set this field if hookHandled is set to TRUE.
dialogWidth	Contains the width (in dialog coordinate units) the dialog box will have when the height of the dialog font becomes newFontHeight. The dialog hook function should set this field if hookHandled is set to TRUE. dialogWidth is initially the old width of the dialog in dialog coordinate units.
dialogHeight	Contains the height (in dialog coordinate units) the dialog box will have when the height of the dialog font becomes newFontHeight. The dialog hook function should set this field if hookHandled is set to TRUE. dialogHeight is initially the old height of the dialog in dialog coordinate units.
newFontHeight	Specifies the font height that should be used in calculating the new size of the dialog box.
oldFontHeight	Specifies the height of the dialog box's old font.

DIALOG_MESSAGE_FONTCHANGED message

The DIALOG_MESSAGE_FONTCHANGED message is sent when the dialog box is moved between screens using differing font sizes. DITEM_MESSAGE_FONTCHANGED messages were already sent to any item hook functions attached to items in the dialog box. The dialog box size was changed to correspond to the new size calculated when the DIALOG_MESSAGE_CALCSIZE message was sent.

Most dialog hook functions ignore this message unless they need to accomplish a task (such as repositioning dialog items) when the dialog font changes.

Additional information is included in the `fontChanged` member of the `DialogMessage` union `u` as follows:

```
struct
{
   int  newFontHeight;      /* => new font's height */
   int  oldFontHeight;      /* => old font's height */
} fontChanged;         /* because dialog now on new screen */
```

Field	Description
`newFontHeight`	Specifies the font height that should be used in calculating the new size of the dialog box.
`oldFontHeight`	Specifies the height of the dialog box's old font.

DIALOG_MESSAGE_MINIMIZE message

The `DIALOG_MESSAGE_MINIMIZE` message is sent to the dialog hook function when the user presses the minimize button in the dialog title bar to minimize the dialog box or the `mdlWindow_minimize` function is called. This message is only generated at minimize time, not when the dialog is restored to it's original size. Additional information is included in the `minimize` member of the `DialogMessage` union `u` as follows:

```
struct
{
   boolean   hookHandled;   /* <= TRUE if new size returned */
   int       newWidth;      /* <= width of minimized dialog */
   int       newHeight;     /* <= height of minimized dialog */
} minimize;
```

Field	Description
`hookHandled`	Indicates that the dialog hook function has returned new values to be used as the minimized size of the dialog box in pixels.
`newWidth`	Specifies the minimized width of the dialog box in pixels.
`newHeight`	Specifies the minimized height of the dialog box in pixels.

DIALOG_MESSAGE_MAXIMIZE message

The `DIALOG_MESSAGE_MAXIMIZE` message is sent to the dialog hook function when the user presses the maximize button in the dialog title bar to maximize the dialog box or the `mdlWindow_maximize` function is called. This message is only generated at

maximize time, not when the dialog is restored to it's original size. Additional information is included in the maximize member of the DialogMessage union u as follows:

```
struct
{
   boolean   hookHandled;   /* <= TRUE if new size returned */
   BSIRect   newRect;       /* <= new content rectangle */
} maximize;
```

Field	Description
hookHandled	Indicates that the dialog hook function has returned new values to be used as the maximized size and location of the dialog box.
newRect	Specifies the new content rectangle for the maximized dialog box in global, pixel coordinates.

Button Messages

The following messages, documented in this section, are sent when button events occur:

DIALOG_MESSAGE_PREBUTTON

DIALOG_MESSAGE_BUTTON

DIALOG_MESSAGE_ACTIONBUTTON

DIALOG_MESSAGE_PREBUTTON message, DIALOG_MESSAGE_BUTTON message

The DIALOG_MESSAGE_PREBUTTON and DIALOG_MESSAGE_BUTTON messages are sent when a mouse button event occurs somewhere in the dialog box. A button event occurs when:

- The mouse button is pressed or released.

  or

- A button timeout occurs because the mouse button is pressed for an extended period of time or too much time elapses between presses.

When a button down event occurs, a DIALOG_MESSAGE_PREBUTTON message is sent to the dialog hook function. This message allows the hook function to interpret the event in different ways and alter the way MicroStation handles this event. The hook function may redirect the processing o f the button down event to another item in the dialog by placing a pointer to the desired dialog item's raw item header in the buttonRiP field of the button member structure as defined below. This is typically used for opening popup menu items in the dialog box and causes all button events up to and including the subsequent button up event to be sent to this alternate dialog item. A

DIALOG_MESSAGE_BUTTON message is sent after the PREBUTTON message to either the item pointed to by the buttonRiP value returned by the dialog hook or item originally receiving the PREBUTTON if this field is NULL.

If the button event occurs with the mouse cursor inside of a dialog item that has an item hook function, a DITEM_MESSAGE_BUTTON message has been sent to that function. Additional information is included in the button member of the DialogMessage union u as follows:

```
struct
{
   int       buttonNumber;       /* => button number */
   int       buttonTrans;        /* => type of transition */
   int       qualifierMask;      /* => ctrl/alt/shift keys down */
   int       upNumber;           /* => 1=singleClick,2=dblClick,etc */
   boolean   clicked;            /* => FALSE means press or drag */
   ULong     buttonTime;         /* => time of transition */
   Point2d   pt;                 /* => location in local coords */
   int       itemIndex;          /* => item button already sent to */
   Inputq_element *iqelP;        /* => source queue element */
   void      (*motionFunc)();    /* <= for mouse down only */
   void      *motionMD;          /* <= for mouse down only */
   RawItemHdr *buttonRiP;   /* <= item to rcv followup msgs */
} button;
```

Field	Description
buttonNumber	Contains the mouse button number. This number will be either DATAPNT, TENTPNT or RESET. These values are defined in **msbutton.h**.
buttonTrans	Contains the type of button transition that just occurred. The values, listed below in the buttonTrans table, are defined in **msdefs.h**.
qualifierMask	Contains the state of the qualifier keys when the button event occurred. The possible qualifier keys are SHIFTKEY, CTRLKEY and ALTKEY. These bitmasks are defined in **keys.h**.

<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td>upNumber</td><td>Contains the number of times the button was released within the amount of time specified by the variable doubleClickTime when buttonTrans is either BUTTONTRANS_UP or BUTTONTRANS_DOWN. When buttonTrans is equal to BUTTONTRANS_UP, a value of 1 means a single-click just occurred, 2 means a double-click just occurred, and so on.

In a button press series, the amount of time between presses must be less than the amount specified by the doubleClickTime variable. When buttonTrans is equal to BUTTONTRANS_DOWN, a value of 0 means this is the first button press in the series; a value of 1 means that this is the next press, and so on.

upNumber specifies the type of button timeout event if buttonTrans is BUTTONTRANS_TIMEOUT. In this case, upNumber can have the following values:

<table>
<tr><th>value</th><th>Meaning</th></tr>
<tr><td>BUTTONTIME-OUT_CLICK</td><td>Indicates that an amount of time equal to half the value of doubleClickTime has elapsed between the time the button last went down, and when it came back up.</td></tr>
<tr><td>BUTTONTIME-OUT_DOUBLECLICK</td><td>Indicates that an amount of time equal to doubleClickTime has elapsed since the button last went down.</td></tr>
</table>
</td></tr>
<tr><td>clicked</td><td>Specifies whether the button came back up because of a click. (The mouse came up within half a doubleClickAmount of time and did not move by an appreciable distance.) This field is meaningless for mouse down events.</td></tr>
<tr><td>buttonTime</td><td>Specifies the time of the button event in ticks.</td></tr>
<tr><td>pt</td><td>Specifies the location, in local coordinates, of the mouse cursor at the time of the button event.</td></tr>
<tr><td>itemIndex</td><td>Specifies the index (0-based) of the item the mouse cursor was on at the time of the button event. -1 means the mouse cursor was not inside a mouse-sensitive item.</td></tr>
</table>

Field	Description
iqelP	Contains a pointer to the raw input queue element for the button event. See **msinputq.h** for the declaration of the InputQ_element structure.
motionFunc	Specifies an MDL function to be called by MicroStation to handle cursor motion events while the mouse button is pressed. The motion function will be called every time MicroStation operates on the cursor while the mouse button is down (i.e., every time it moves, the cursor is drawn, or a timeout occurs). The motion function is called with a single parameter, which is a pointer the the structure shown below. typedef struct motionfuncarg { Point2d pt; boolean dragging; } MotionFuncArg;

field	description
pt	Specifies the location, in global coordinates, of the mouse cursor at the time of the function call.
dragging	Specifies whether the cursor has moved while the mouse button is pressed down.

buttonTrans Value	Description
BUTTONTRANS_UP	Indicates that the mouse button was just released.
BUTTONTRANS_DOWN	Indicates that the mouse button was just pressed.
BUTTONTRANS_TIMEOUT	Indicates that a button timeout event just occurred. See upNumber to determine the type of timeout.

DIALOG_MESSAGE_ACTIONBUTTON message

This message is generated by the default MicroStation buttons (OK, Cancel, Apply, Reset and Default) when the user presses and releases the button. Additional information is included in the actionButton member of the DialogMessage union u as follows:

```
struct
{
   boolean   abortAction;   /* <= TRUE = abort button action */
   int       actionType;    /* => OK, CANCEL, APPLY, RESET */
} actionButton;
```

Field	Description
abortAction	Indicates whether the dialog manager should perform the default actions done as a result of the push button being activated. If TRUE, the dialog manager skips all default processing and acts as if the push button was not activated by the user.
actionType	Contains the value last set by a call to the mdlDialog_lastActionTypeSet function. If the standard push button hook function is attached to a push button, this field contains the itemHookArg field of that push button (usually one of the ACTIONBUTTON_ constants).

Open & Close Messages

The following messages, documented in this section, are sent when dialog boxes are created or destroyed:

DIALOG_MESSAGE_CHILDDESTROYED

DIALOG_MESSAGE_ANOTHEROPENED

DIALOG_MESSAGE_ANOTHERCLOSED

DIALOG_MESSAGE_CHILDDESTROYED message

The DIALOG_MESSAGE_CHILDDESTROYED message is sent to a dialog box's parent when the dialog is destroyed. The parent dialog box is specified by the parentDialogId field of a dialog box's resource specification. The parentId of a dialog box can also be set with the mdlDialog_parentIdSet function. Additional information is included in the childDestroyed member of the DialogMessage union u as follows:

```
struct
{
   long childDialogId; /* => child dialog id */
   int  actionType;    /* => only if stdAction button used */
} childDestroyed;
```

Field	Description
childDialogId	Contains the ID of the dialog box that was just destroyed.
actionType	Contains the value last set by a call to the mdlDialog_lastActionTypeSet function. If the standard push button hook function is attached to a push button, this field contains the itemHookArg field of that push button (usually one of the ACTIONBUTTON_ constants) instead.

DIALOG_MESSAGE_ANOTHEROPENED message

The DIALOG_MESSAGE_ANOTHEROPENED message is sent to all dialog boxes that have set the otherDialogs field of DialogHookInterests to equal TRUE when any dialog is opened. Additional information is included in the anotherOpened member of the DialogMessage union u as follows:

```
struct
{
   DialogBox *db;           /* => ptr to dialog opened */
   ULong     dialogType;    /* => type of dialog opened */
   long      dialogId;      /* => id of dialog opened */
} anotherOpened;
```

Field	Description
db	Contains an opaque pointer to the dialog box that was just opened.
dialogType	Contains the type of the dialog box that was just opened.
dialogId	Contains the ID of the dialog box that was just opened.

DIALOG_MESSAGE_ANOTHERCLOSED message

When any dialog is destroyed, the DIALOG_MESSAGE_ANOTHERCLOSED message is sent to all dialog boxes that have set the otherDialogs field of DialogHookInterests to TRUE. Additional information is included in the anotherClosed member of the DialogMessage union u as follows:

```
struct
{
   DialogBox *db;           /* => pointer to dialog closed */
   ULong     dialogType;    /* => type of dialog closed */
   long      dialogId;      /* => id of dialog closed */
   int       actionType;    /* => only if stdAction button used */
} anotherClosed;
```

Field	Description
db	Contains an opaque pointer to the dialog box that was just destroyed.
dialogType	Contains the type of the dialog box that was just destroyed.
dialogId	Contains the ID of the dialog box that was just destroyed.
actionType	Contains the value last set by a call to the mdlDialog_lastActionTypeSet function. If the standard push button hook function is attached to a push button, this field contains the itemHookArg field of that push button (usually one of the ACTIONBUTTON_ constants).

Miscellaneous Messages

The following messages, documented in this section, are sent when various user interface events occur:

DIALOG_MESSAGE_UPDATE

DIALOG_MESSAGE_SYNCH

DIALOG_MESSAGE_STATECHANGED

DIALOG_MESSAGE_USER

DIALOG_MESSAGE_UPDATE message

After the dialog box manager has drawn the contents of a dialog box, it sends the DIALOG_MESSAGE_UPDATE message to the dialog hook function.

DIALOG_MESSAGE_SYNCH message

The DIALOG_MESSAGE_SYNCH message is sent when the dialog box is synched by the mdlDialog_itemsSynch function. DITEM_MESSAGE_SYNCHRONIZE messages have already been sent to all item hook functions attached to items in the dialog box.

DIALOG_MESSAGE_STATECHANGED message

The DIALOG_MESSAGE_STATECHANGED message is sent to the dialog hook function after the dialog item state has changed. If the dialog item has a hook function, the item hook function has already been sent the DITEM_MESSAGE_STATECHANGED message.

Information about the command is included in the stateChanged member of the DialogMessage union u as follows:

```
struct
{
   int       itemIndex;     /* => index of item that changed */
   boolean   reallyChanged; /* => if setState noticed difference */
   RawItemHdr *riP;         /* => item that changed */
} stateChanged;
```

Field	Description
itemIndex	Specifies the item index (0-based) of the dialog item which changed.
reallyChanged	Indicates whether the item handler recognized a difference in the state of the item during DITEM_MESSAGE_SETSTATE processing.
riP	Specifies a pointer to the Raw Item Header of the item which changed.

DIALOG_MESSAGE_USER message

The DIALOG_MESSAGE_USER message can be sent to a dialog hook function with the **mdlDialog_hookDialogSendUserMsg** function. This message is useful for setting up a general item hook used for items in different dialog boxes. The item hook sends user messages to each dialog box's dialog hook function. These dialog hook functions are then set up to handle the user messages.

Additional information is included in the user member of the DialogMessage union u as follows:

```
struct
{
   int  type;          /* => type of user message */
   void *userDataP;    /* => user specified ptr */
} user;
```

Field	Description
type	Contains the type that is specified in the **mdlDialog_hookDialogSendUserMsg** function call.
userDataP	Contains the userDataP field that is specified in the **mdlDialog_hookDialogSendUserMsg** function call.

Item Hook Functions

A basic item hook function should look as follows:

```
Public void dialogBoxName_specificItemtypeHook
(
DialogItemMessage *dimP
)
{
   dimP->msgUnderstood = TRUE;
   switch (dimP->messageType)
   {
        case DITEM_MESSAGE_CREATE:
        {
             ...
             break;
        }
        case DITEM_MESSAGE_DESTROY:
        {
             ...
             break;
        }

        ...
        default:
        {
             dimP->msgUnderstood = FALSE;
             break;
        }
   }
}
```

Item hook functions are set up similarly to dialog hook functions: one switch statement tests for various messages, and then handles them.

DialogItemMessage structure

The `DialogItemMessage` structure that is the only argument to an item hook function is declared as follows (from **dlogitem.h**):

```
typedef struct dialogitemmessage
{
   boolean msgUnderstood;        /* <= item understood the message */
   int messageType;              /* => message type */
   DialogBox *db;                /* => dialog box handle */
   long dialogId;                /* => resource id of dialog box */
   int itemIndex;                /* => index of item msg sent to */
```

```
    DialogItem *dialogItemP;      /* => dialogItem info for item */
    int auxMessageType;           /* => item specific message type */
    void *auxInfoP;               /* => item specific message info */
    union
    {
        ...
        structures that contain information that
        depends on messageType.
    } u;
} DialogItemMessage;
```

Structures that contain information that depends on `messageType` are discussed in the section on the individual dialog item messages.

Field	Description
`msgUnderstood`	Indicates whether the item hook function understands the message. If it understands the message, the function sets `msgUnderstood` to `TRUE`. If the mesage is not understood, the item handlers take specific actions automatically, depending on the value of `messageType`. Some messages also have a `hookHandled` member. This member indicates that the hook function not only understands the message, but it handled it completely and the item handler should not perform any further default processing.
`messageType`	Type of item message. The individual messages are discussed in the next section.
`db`	Opaque pointer to the dialog box intended for the message. The pointer is usually passed on to other dialog box manager functions.
`dialogId`	ID of the dialog box the message is intended for.
`itemIndex`	Index (0-based) of the item the message is intended for.
`dialogItemP`	Pointer to the `DialogItem` field of the item the message is for. The DialogItem declaration is discussed in the next table.
`auxMessageType`	Type of auxiliary message. This field is used internally by the dialog box manager and can be ignored.
`auxInfoP`	Additional information needed for auxiliary messages. The dialog box manager uses this field internally and it can be ignored.

DialogItem structure

The header for a dialog item message contains a `DialogItem` field. Every dialog item in a dialog box can be uniquely identified by a pointer to its `DialogItem` field. The `mdlDialog_itemGetByTypeAndId` or `mdlDialog_itemGetByIndex` functions can be used to obtain a pointer to an item that is different than the one a particular message is directed to.

The `DialogItem` and associated structures have the following declarations (from **dlogitem.h**):

```
typedef struct dialogitemattributes
{
   ULong acceptsKeystrokes:1;    /* keystroke input possible */
   ULong mouseSensitive:1;       /* mouse events possible */
   ULong traversable:1;          /* kbd navigation possible */
   ULong canHaveSynonyms:1;      /* synonym resources refs */
   ULong enabled:1;              /* TRUE = enabled */
   ULong hidden:1;               /* TRUE = hidden */
   ULong hasFocus:1;             /* TRUE = has input focus */
   ULong updateFlag:1;           /* internal use only */
   ULong xPosLocked:1;           /* internal use only */
   ULong yPosLocked:1;           /* internal use only */
   ULong unused:6;
   ULong unused2:8;
   ULong unused3:8;
} DialogItemAttributes;

typedef struct dialogitem
{
   long type;
   long id;
   int  itemIndex;
   int  itemArg;
   Rectangle rect;     /* mouse sensitive/update region of item */
   Sextent   extent;   /* (in dcoord units) for fontChange msgs */
   Point2d   origin;   /* for fontChange msgs */
   DialogItemAttributes attributes;
   RawItemHdr *rawItemP;
} DialogItem;
```

Field	Description
`type`	Specifies the type of the dialog item.
`id`	Specifies the ID of the dialog item.
`itemIndex`	Specifies the index (0-based) of the dialog item. This is the item's position within the dialog item list. This field will change if items are inserted or deleted from the dialog box containing the item.
`itemArg`	Contains the `itemArg` field that is used in the dialog box's `DialogItemRsc` item list specification.
`rect`	Specifies the mouse-sensitive/update region of the item in pixels.
`extent`	Specifies the mouse-sensitive/update region of the item in dialog coordinate units. This field can be used to handle the `DITEM_MESSAGE_FONTCHANGED` message.

Field	Description
origin	Specifies the origin that should be used with the extent field. Items can be located relative to an origin different than the upper left corner of a dialog box. This field stores that origin.
attributes	Contains the item's attributes. The attributes for an item are described by the DialogItemAttributes structure. The application program should never set/clear any of the bits in this structure except as may be later described in this section. Changes to information affects the way the dialog manager handles an item.The fields of this structure are described in the attributes table below.
rawItemP	Contains a pointer to the item's RawItemHdr declaration. The RawItemHdr declaration is discussed in the next table.

.

attributes Value	Meaning
acceptsKeystrokes	Indicates whether the item can accept keystrokes events as input for processing.
mouseSensitive	Indicates whether the item can accept mouse events as input for processing.
traversable	Indicates whether the item can be traversed using keyboard navigation through the dialog box.
canHaveSynonyms	Indicates whether the item can associated with other items through synonym resources.
enabled	Indicates whether the item is ENABLED or DISABLED. ENABLED items are displayed in full intensity and can received user input and focus. DISABLED items are displayed in half intersity and can not be selected by the user.
hidden	Indicates whether the item is displayed in the dialog box. If HIDDEN, an item is not visible to the user and can not receive input focus.
hasFocus	An internal flag not currently used.
updateFlag	An internal flag used to indicate whether the item has been drawn in the dialog box.
xPosLocked	An internal flag used for locking the X position of the item when working with tool palette slamdowns. This attribute should be ignored.
yPosLocked	An internal flag used for locking the Y position of the item when working with tool palette slamdowns. This attribute should be ignored.

RawItemHdr structure

The DialogItem structure contains information specified in the dialog box's DialogItemRsc item list specification. On the other hand, the RawItemHdr structure contains information that is usually specified in an item's resource specification. A field that does not apply to a particular type of item is set to zero.

A pointer to an item's RawItemHdr structure uniquely identifies it, just like a pointer to an item's DialogItem field does. Also, one of these pointers can be obtained from the other. The RawItemHdr structure has the following declaration (from **dlogitem.h**):

```
/* this structure must be at the beginning of all runtime items */
struct rawitemhdr
{
   void      *userDataP;              /* for use by itemHooks */
   char      *accessStrP;             /* access str to app variable */
   char      *labelP;                 /* items label or NULL */
   Point2d   labelPt;                 /* position of label */
   Rectangle itemRect;                /* item specific rectangle */
   ULong     helpInfo;                /* item help */
   ULong     helpSource;              /* help file indicator */
   long      synonymsId;              /* id of synonyms resource */
   UShort    highlightOn:1;           /* highlight state */
   void      *itemHookMD;             /* mdl process descrip for hook */
   long      itemHookOffset;          /* mdl hook offset */
   long      itemHookArg;             /* rsc defined arg to hook func */
   void      *ownerDialogP;           /* ptr to owner dialog box */
   DialogItem *diP;                   /* ptr to dialog item struct */
   DialogItem *parentDiP;             /* ptr to parent dialog item */
   long      itemHookId;              /* mdl hook id */
   int       mnemonic;                /* keyboard mnemonic */
   int       mneIndex;                /* mnemonic index */
   int       nColorPs;                /* Number of colors settings */
   BSIColorDescr **colorsPP;          /* colors for each setting */
   DialogItemHandlerInfo *itemHandlerP;/* item handler info */
   BSIRect    traversalRect;          /* highlight rectangle */
   void      *ownerMD;                /* owner MDL descriptor */
   ULong      commandNumber;          /* command assoc with item */
   char      *commandTaskId;          /* task owning command */
   char      *unparsedP;              /* command parms */
   void      *auxP;                   /* ptr to pulldown menu item */
   void      *popupWindP;             /* ptr to popup window */
   RawItemHdr *childFocusRiP;         /* child item which has focus */
   ArrayObjectHdr *childArrayP;       /* array of child items */
   };
```

Field	Description
`userDataP`	Stores a pointer to user-defined data. The dialog box manager does not store its information in this field.
`accessStrP`	Points to the item's access string. If this field is `NULL`, no access string exists.
`labelP`	Points to the item's label. If this field is `NULL`, no label exists.
`labelPt`	Specifies the position of the item's label in pixels.
`itemRect`	Specifies an item-specific rectangle for the item in pixels. The various item handlers use this field internally, and the field can be ignored.
`helpInfo`	Contains the value of the `helpInfo` field that is specified in the item's resource specification.
`helpSource`	Contains the value of the `helpSource` field that is specified in the item's resource specification.
`synonymsId`	Contains the value of the `synonymsId` field that is specified in the item's resource specification.
`highlightOn`	Specifies whether the item currently displays in a highlighted state. The various item handlers use this field internally, and the field can be ignored.
`itemHookMD`	The various item handlers use this field internally, and it should be ignored.
`itemHookOffset`	The various item handlers use this field internally, and it should be ignored.
`itemHookArg`	Contains the value of the `itemHookArg` field that is specified in the item's resource specification.
`ownerDialogP`	Points to the dialog box that contains the item.
`diP`	Points to the item's `DialogItem` field.
`parentDiP`	Points to the `DialogItem` field of the item's parent item. The various item handlers use this field internally, and it can be ignored.
`itemHookId`	The various item handlers use this field internally, and it should be ignored.
`mnemonic`	The dialog manager uses this field internally, and it should be ignored.
`mneIndex`	The various item handlers use this field internally, and it should be ignored.
`nColorPs`	The number of entries in `colorsPP` (typically set to `DITEM_COLORTYPE_NCOLORS`). This is an internally used field and should not be modified by the application.

Field	Description
colorsPP	The array of color descriptors for the item. This information in this array is set using mdlDialog_itemSetColor and retrieved using mdlDialog_itemGetColor.
itemHandlerP	The various item handlers use this field internally, and it should be ignored.
traveralRect	The various item handlers use this field internally, and it should be ignored.
ownerMD	Specifies the MDL Descriptor for the owning MDL task.
commandNumber	Command number to be queued when the item is activated.
comamndTaskId	Specifies the owner of the command to be queued.
unparsedP	Pointer to the unparsed character string to be queued along with the command.
auxP	The various item handlers use this field internally, and it should be ignored.
popupWindP	The various item handlers use this field internally, and it should be ignored.
childFocusRiP	The dialog manager uses this field internally, and it should be ignored.
childArrayP	The dialog manager uses this field internally, and it should be ignored.

Item Hook Function Messages

The messageType field of a dialog item message determines the message's type. This section discusses each individual dialog item message, including any additional information. The following messages are listed in this section:

General Message	Sent
DITEM_MESSAGE_CREATE	After an item is created.
DITEM_MESSAGE_ALLCREATED	After all the items contained in the parent dialog box have been sent the DITEM_MESSAGE_CREATE message.
DITEM_MESSAGE_INIT	After an item is created, and after its internal value is initialized with the state of an underlying application variable.
DITEM_MESSAGE_DESTROY	When an item is about to be destroyed.
DITEM_MESSAGE_GETSTATE	When an item handler needs to determine an item's state.
DITEM_MESSAGE_SETSTATE	When an item handler needs to set an item's state.

General Message	Sent
DITEM_MESSAGE_QUEUECOMMAND	After the DITEM_MESSAGE_SETSTATE message if an item's internal value differed from its state.
DITEM_MESSAGE_SYNCHRONIZE	When an item's visible display and internal value is forced to match the item's external state.
DITEM_MESSAGE_STATECHANGED	When an item's state changes.
DITEM_MESSAGE_USER	To an item hook function with the mdlDialog_hookItemSendUserMsg function to send message from one item hook to another.
DITEM_MESSAGE_BUTTON	When a mouse button event occurs within a mouse-sensitive item.
DITEM_MESSAGE_JOURNALSTATE	When CAD input journaling is active and the internal state of an item has changed.

Input focusable Message	Sent
DITEM_MESSAGE_FOCUSIN	When an item gains the input focus.
DITEM_MESSAGE_FOCUSOUT	When an item will soon lose the input focus.
DITEM_MESSAGE_KEYSTROKE	When a key is pressed while the item has the input focus.
DITEM_MESSAGE_POSTKEYSTROKE	After the item handler's default keystroke handling processes a key.

Generic Message	Sent
DITEM_MESSAGE_SETLABEL	When the item's label needs to be changed.
DITEM_MESSAGE_GETVALUE	When the item's internal value is needed.
DITEM_MESSAGE_SETVALUE	When the item's internal value will be set.
DITEM_MESSAGE_DRAW	When the item should draw itself.
DITEM_MESSAGE_HIGHLIGHT	When the item needs to change its highlight state.
DITEM_MESSAGE_MOVE	When the item needs to be moved.
DITEM_MESSAGE_SETEXTENT	When the item size needs to be changed.
DITEM_MESSAGE_FONTCHANGED	When the dialog box is moved to a screen that uses a different font size than the previous screen.
DITEM_MESSAGE_SETENABLEDSTATE	When the item's enabled state is changed.

Generic Message	Sent
DITEM_MESSAGE_ACTIVATE	When the item is activated.
DITEM_MESSAGE_JOURNALSTATE	When CAD input journaling is active and the internal state of an item has changed.

General item hook function messages

A dialog item message's messageType field determines the message's type. Based on this field's value, the DialogItemMessage structure's u field can have additional information. This section discusses each dialog item message, including any additional information.

The dialog item message structures are not used only for item hook functions. In addition, the item handler functions that actually implement the standard dialog items use them. In fact, many of the fields are appropriate for item handlers, and these fields can be safely ignored for item hook functions.

DITEM_MESSAGE_CREATE message

The DITEM_MESSAGE_CREATE message is sent after an item is created. All item handlers will send this message. Additional information is included in the create member of the DialogItemMessage union u as follows:

```
struct
{
   boolean    createFailed;       /* <= set TRUE if error */
   void       *resourceDataP;     /* => */
   Sextent    *itemListExtentP;   /* => */
   char       *labelP;            /* => */
   long       itemArg;            /* => */
   void       *ownerMD;           /* => */
   DialogItem *parentDiP;         /* => NULL = owner is dialog */
   boolean    dontDisplayAlert;   /* <= TRUE = no alert msg */
} create;
```

Field	Description
createFailed	Indicates whether the item hook function's handling of the create message failed. If so, the function sets this field to TRUE. The dialog box will then remain unopened and an alert will display instead.
resourceDataP	Points to the item's loaded resource specification.
itemListExtentP	Points to the extent (in pixels) that is specified in the item's item list specification.
labelP	Points to the dialog item list override label. The various item handlers use this field internally, and it can be ignored.

Field	Description
itemArg	Contains the value of the itemArg field that is specified in the item's item list specification.
ownerMD	The various item handlers use this field internally, and it should be ignored.
parentDiP	Points to the DialogItem field of the item's parent item.The various item handlers use this field internally, and it can be ignored.
dontDisplayAlert	Indicates whether or not an alert box should be displayed if the create fails for any reason. A value of TRUE indicates that no alert box should be displayed.

Item hook functions usually need to allocate memory for an item in their DITEM_MESSAGE_CREATE message handling and store a pointer to the data in the allocated memory in dimP->dialogItemP->rawItemP->userDataP. At this point, items later in the dialog item list are not yet created, so manipulating them is illegal (and meaningless). Instead, multi-item initializations should be handled in a dialog hook function when the DIALOG_MESSAGE_INIT message is received.

DITEM_MESSAGE_ALLCREATED message

The DITEM_MESSAGE_ALLCREATED message is sent after the all items contained in the parent dialog box have been sent the DITEM_MESSAGE_CREATE message. Additional information is included in the allCreated member of the DialogItemMessage union u as follows:

```
struct
{
   boolean createFailed;    /* <= set TRUE if error */
} allCreated;
```

Field	Description
createFailed	Indicates whether the item hook function's handling of this message failed. If so, the function sets this field to TRUE. The dialog box will then remain unopened and an alert will display instead.

DITEM_MESSAGE_INIT message

The DITEM_MESSAGE_INIT message is sent after the item is created, and after its internal value is initialized with the state of an underlying application variable. Additional information is included in the init member of the DialogItemMessage union u as follows:

```
struct
{
   boolean initFailed; /* <= set TRUE if error */
} init;                /* sent after initial GetState */
```

Field	Description
initFailed	Indicates whether the item hook function's handling of the init message failed. If so, the function sets this field to TRUE. The dialog box will then remain unopened and an alert will display instead.

DITEM_MESSAGE_DESTROY message

The DITEM_MESSAGE_DESTROY message is sent when the item is about to be destroyed. Memory that was allocated in response to the DITEM_MESSAGE_CREATE or DITEM_MESSAGE_INIT message should be freed at this time.

DITEM_MESSAGE_GETSTATE message

The DITEM_MESSAGE_GETSTATE message is sent when an item handler needs to determine an item's state (the value of the item's underlying application variable). This state is then typically used to set the item's internal value. Additional information is included in the value member of the DialogItemMessage union u as follows:

```
struct  /* used by GETSTATE, SETSTATE, GETVALUE, SETVALUE */
{
   boolean   hookHandled;        /* <= only for hooks,TRUE = handled */
   boolean   valueChanged;       /* <= on SET's if value changed */
   int       formatType;         /* defined in vartypes.h */
   ValueUnion value;
   char      *stringValueP;      /* only if string */
   char      *formatStrP;        /* only if string */
   int       maxStringSize;      /* only if string & GETs */
} value;
```

Field	Description
hookHandled	Indicates whether the dialog box manager should continue its default handling for obtaining the item state. The item hook function should set this field to TRUE to stop the default handling.
valueChanged	Is ignored for the DITEM_MESSAGE_GETSTATE message.

Field	Description
formatType	Returns the item state's format if the item hook function is handling the DITEM_MESSAGE_GETSTATE message. formatType is FMT_LONG (defined in **vartypes.h**) for all items with an internal value that is an integer. In this case, the item state should be returned in: dimP->u.value.value.sLongFormat Currently, all items except text items specify FMT_LONG for formatType. formatType is FMT_STRING (defined in **vartypes.h**) for all items whose internal value is stored as a string. Currently, only text items meet this requirement. In this case, the item hook function returns the item state by setting the string pointed to by dimP->u.value.stringValueP.
value	Returns the item's state for all items except text items. This field is a union of the standard C types. Currently, all items except text items use the signed long member dimP->u.value.value.sLongFormat to return the item's state. Text items expect the string pointed to by the stringValueP field to be set to the item's state.
stringValueP	Returns the item's state if the item's internal value is contained in a string. (Currently, only text items store their internal values as strings.) The string pointed to by stringValueP should be set to the item's state.
formatStrP	Points to the formatToDisplay string that is part of an item's resource specification. This pointer, if non-NULL, can be used with the **sprintf** function to convert an application variable to a string. This field is used only if the item stores its internal value as a string. (Currently, only text items store their internal values as strings).
maxStringSize	Specifies the size of the buffer pointed to by stringValueP. The size of the string copied in the buffer should not exceed maxStringSize. This field is used only if a string will be returned.

Sometimes the "state" of an application variable is not a simple thing to determine, or the "variable" is really a set of variables. In those cases, the item hook function can trap the DITEM_MESSAGE_GETSTATE and DITEM_MESSAGE_SETSTATE messages to come up with a composite state that can be used to set an item's internal value.

DITEM_MESSAGE_SETSTATE message

The DITEM_MESSAGE_SETSTATE message is sent when an item handler needs to set an item's state (the value of its underlying application variable) using the item's internal value. Remember that with modal dialogs, unless the DIALOGATTR_ALWAYSSETSTATE attribute is set, SETSTATE messages will not be sent until the user OK's the modal dialog box. If the user cancels the dialog box, SETSTATE messages will never be sent. This leaves the dialog box items' underlying application variables unchanged, which is what is desired when canceling a dialog.

See "Modeless and modal dialog boxes" on page 15-10 for more information on modal dialog boxes.

Additional information is included in the `value` member of the `DialogItemMessage` union `u` as follows:

```
struct  /* used by GETSTATE, SETSTATE, GETVALUE, SETVALUE */
{
   boolean   hookHandled;        /* <= only for hooks,TRUE = handled */
   boolean   valueChanged;       /* <= on SET's if value changed */
   int       formatType;         /* defined in vartypes.h */
   ValueUnion value;
   char      *stringValueP;      /* only if string */
   char      *formatStrP;        /* only if string */
   int       maxStringSize;      /* only if string & GETs */
} value;
```

Field	Description
`hookHandled`	Indicates whether the dialog box manager should continue default handling for setting the item state. The item hook function should set this field to `TRUE` to stop the default handling.
`valueChanged`	Contains an indication of whether the item's internal value differed from the item's external state. The item hook function should set this field if `hookHandled` is `TRUE`.
`formatType`	Specifies the type of the item's internal value. If `formatType` is `FMT_LONG`, the item's internal value is stored as a signed long and can be found in the `value.sLongFormat` field. If `formatType` is `FMT_STRING` the item's internal value is stored as a string that is pointed to by `stringValueP`. Currently, `formatType` will be `FMT_LONG` (defined in **vartypes.h**) for all items except text items. For text items, it will be `FMT_STRING`.
`value`	Specifies the item's internal value for all items except text items. This field is a union of the standard C types. Currently all items except text items use the signed `long` member `dimP->u.value.value.sLongFormat` to contain the item's internal value. Text items will set the `stringValueP` field to point to the item's internal string value.
`stringValueP`	Points to the item's internal value if the value is contained in a string. (Currently, only text items store their internal values as strings).
`formatStrP`	Points to the `formatToInternal` string that is part of an item's resource specification. This pointer, if non-`NULL`, can be used with the sscanf function to convert an item's internal string to an application variable. This field is used only if the item stores its internal value as a string. (Currently, only text items store their internal values as strings.)
`maxStringSize`	Is ignored for the `DITEM_MESSAGE_SETSTATE` message.

DITEM_MESSAGE_QUEUECOMMAND message

The DITEM_MESSAGE_QUEUECOMMAND message is sent after the DITEM_MESSAGE_SETSTATE message if the item's internal value differed from its state. It can be used to dynamically determine the command to queue based on the item's or dialog box's state. Additional information is included in the queueCommand member of the DialogItemMessage union u as follows:

```
struct
{
   boolean   hookHandled;   /* <= only for hooks,TRUE = handled */
   int       subItemIndex;  /* => */
} queueCommand;
```

Field	Description
hookHandled	Indicates whether the dialog box manager should continue default handling for queuing a command. After it queues a command, the item hook function should set this field to TRUE to prevent the item handler from queuing another command.
subItemIndex	Contains the current subitem index for the item. For certain items, such as option buttons, knowing the item's current value is useful in determining the command to queue. (An option button's current value is also the index of the subitem currently selected. For this reason, this field is called subItemIndex).

DITEM_MESSAGE_SYNCHRONIZE message

The DITEM_MESSAGE_SYNCHRONIZE message is sent when an item's visible display and internal value is forced to match the item's external state (the value of its underlying application variable). Additional information is included in the synchronize member of the DialogItemMessage union u as follows:

```
struct
{
   boolean   redrawn;  /* => item had to be redrawn */
} synchronize;
```

Field	Description
redrawn	TRUE if the item had to be redrawn in order to match its external state.

DITEM_MESSAGE_STATECHANGED message

The DITEM_MESSAGE_STATECHANGED message is sent when an item's state has changed. Information about the command is included in the stateChanged member of the DialogItemMessage union u as follows:

```
struct
{
   boolean    reallyChanged;     /* => TRUE=item's value changed */
} stateChanged;
```

Field	Description
reallyChanged	Indicates whether, during DITEM_MESSAGE_SETSTATE processing, an actual change in the value of the dialog item was detected. If the user selected the dialog item but entered or selected the same value as was there before, the item's state will have changed but reallyChanged will return FALSE.

DITEM_MESSAGE_USER message

The DITEM_MESSAGE_USER message can be sent to an item hook function with the **mdlDialog_hookItemSendUserMsg** function. This function can be used to send messages from one item hook function to another. Additional information is included in the user member of the DialogItemMessage union u as follows:

```
struct
{
   int  type;          /* => type of user message */
   void *userDataP;    /* => */
} user;
```

Field	Description
type	Contains the type that is specified in the **mdlDialog_hookItemSendUserMsg** function call.
userDataP	Contains the userDataP that is specified in the **mdlDialog_hookItemSendUserMsg** function call.

DITEM_MESSAGE_BUTTON message

The DITEM_MESSAGE_BUTTON message can be sent only for items that are mouse-sensitive. See the individual item descriptions in "Standard Dialog Box Items" on page 16-1 to determine if this message will be sent to an item's hook function.

This message is sent when a mouse button event occurs within a mouse-sensitive item. The screen extent of an item is specified in its DialogItem rect field. (See "DialogItemRsc Structure" on page 16-6 for more information). A button event happens when the mouse button is pressed or released, or when a button timeout occurs because the mouse button is pressed for an extended period of time or too much time elapses between presses. Additional information is included in the button member of the DialogItemMessage union u as follows:

```
struct
{
   boolean   couldSetState;      /* <= if not modal, could set State */
   void      (*motionFunc)();    /* <= for mouse down only */
   void      *motionMD;          /* <= for mouse down only
                                    (only for item handlers) */
   int       buttonNumber;       /* => button number */
   int       buttonTrans;        /* => type of transition */
   int       qualifierMask;      /* => ctrl/alt/shift keys down */
   int       upNumber;           /* => 1=singleClick, 2=dblClick,etc */
   boolean   clicked;            /* => FALSE = press or drag */
   ULong     buttonTime;         /* => time of transition */
   Point2d   pt;                 /* => point in local coords */
   Inputq_element *iqelP;
} button;
```

field	Description
couldSetState	Specifies whether the item state should be changed as a result of the button event. The item hook function should set this field to TRUE if the item state should be changed. The various item handlers use this field internally, and it can be ignored except for generic items.
motionFunc	Specifies the address of a function that should be called while the mouse moves with the button pressed. This field is used only if buttonTrans is BUTTONTRANS_DOWN, and only for generic items. The motion function specified should accept a single argument: a pointer to a MotionFuncArg structure (defined in **dlogitem.h**).
motionMD	(This field is used internally by the various item handlers and can be ignored.)
buttonNumber	Contains the mouse button number. This number will be either DATAPNT, TENTPNT or RESET.
buttonTrans	Contains the type of button transition that just occurred.

value	description
BUTTONTRANS_UP	Indicates that the mouse button was just released.
BUTTONTRANS_DOWN	Indicates that the mouse button was just pressed.
BUTTONTRANS_TIMEOUT	Indicates that a button timeout event just occurred. See upNumber to determine the type of timeout.

field	Description
qualifierMask	Contains the state of the qualifier keys when the button event occurred. The possible qualifier keys are SHIFTKEY, CTRLKEY and ALTKEY. These bitmasks are defined in **keys.h**.
upNumber	Contains the number of times the button was released within the amount of time specified by the variable doubleClickTime when buttonTrans is either BUTTONTRANS_UP or BUTTONTRANS_DOWN. When buttonTrans is equal to BUTTONTRANS_UP, a value of 1 means a single-click just occurred, 2 means a double-click just occurred, and so on. In a button press series, the amount of time between presses must be less than the amount specified by the doubleClickTime variable. When buttonTrans is equal to BUTTONTRANS_DOWN, a value of 0 means this is the first button press in the series; a value of 1 means that this is the next press, and so on. upNumber specifies the type of button timeout event if buttonTrans is BUTTONTRANS_TIMEOUT. In this case, upNumber can have the following values:
clicked	Specifies whether the button came back up because of a click. (The mouse came up within half a doubleClickAmount of time and did not move by an appreciable distance). This field is meaningless for mouse down events.
buttonTime	Specifies the time of the button event in ticks.
pt	Specifies the location, in local coordinates, of the mouse cursor at the time of the button event.
iqelP	Contains a pointer to the raw input queue element for the button event. See **msinputq.h** for the declaration of the InputQ_element structure.

value	description
BUTTONTIMEOUT_CLICK	Indicates that more than half a doubleClickTime amount of time elapsed between the time the button last went down and when it came back up.
BUTTONTIMEOUT_DOUBLECLICK	Indicates that more than a doubleClickTime amount of time elapsed since the button last went down.

DITEM_MESSAGE_JOURNALSTATE message

When CAD input journaling is active, the DITEM_MESSAGE_JOURNALSTATE message is sent after the DITEM_MESSAGE_SETSTATE message if the item's internal value differed from it's state and before the DITEM_MESSAGE_QUEUECOMMAND is sent. This message can be used to journal a command equivalent to setting the item's access string or to prevent any journaling at all. At this time, CAD input journaling is used to generate macros.

Additional information is included in the value member of the DialogItemMessage union u as follows:

```
struct  /* used by GETSTATE, SETSTATE, GETVALUE, SETVALUE */
{
   boolean   hookHandled;        /* <= only for hooks,TRUE = handled */
   boolean   valueChanged;       /* <= on SET's if value changed */
   int       formatType;         /* defined in vartypes.h */
   ValueUnion value;
   char      *stringValueP;      /* only if string */
   char      *formatStrP;        /* only if string */
   int       maxStringSize;      /* only if string & GETs */
} value;
```

field	Description
hookHandled	Indicates whether the dialog box manager should continue default handling for journaling the item state. The item hook function should set this field to TRUE to stop default handling.
valueChanged	Is ignored for the DITEM_MESSAGE_JOURNALSTATE message.
formatType	Specifies the type of the item's internal value. If formatType is FMT_LONG, the item's internal value is stored as a signed long and can be found in the value.sLongFormat field. If formatType is FMT_DPFP, the item's internal value is stored as a double precision floating point value and can be found in value.doubleFormat field. If formatType is FMT_STRING, the item's internal value is stored as a string that is pointed to by stringValueP.
value	Specifies the item's internal value for all items except text items. This field is a union of the standard C types. Text items will set the stringValueP field to point to the item's internal string value.
stringValueP	Points to the item's internal value if the value is contained in a string.

field	Description
formatStrP	Points to the formatToInternal string that is part of an item's resource specification. This pointer, if non-NULL, can be used with the sscanf function to convert an item's internal string to an application variable. This field is used only if the item stores its internal value as a string.
maxStringSize	Is ignored for the DITEM_MESSAGE_JOURNALSTATE message.

Input focusable item hook function messages

Input focusable item hook function messages can be sent only for items that can accept the input focus. See the individual item descriptions in "The Standard Dialog Box Items" to determine whether a message will be sent to an item's hook function.

The first two of these messages use the focusOutType field, which can have the following values:

focusOutType Value	Meaning
FOCUSOUT_KEYSWITCH	Focus has changed as a result of a keyboard action.
FOCUSOUT_BUTTONSWITCH	Focus has changed as a result of a button action.
FOCUSOUT_SETITEM	Focus has changes as a result of a dialog item programmatically being assigned focus.
FOCUSOUT_HIDEITEM	Focus has changed as a result of the previous item being hidden.
FOCUSOUT_DISABLEITEM	Focus has changed as a result of the previous item being disabled.
FOCUSOUT_SWITCHDIALOG	Focus has changed as a result of a new dialog box receiving focus.
FOCUSOUT_APPLYDIALOG	Focus has changed as a result of an apply operation being performed on the dialog box containing the item.

DITEM_MESSAGE_FOCUSIN message

The DITEM_MESSAGE_FOCUSIN message is sent when an item gains the input focus. Additional information is included in the focusIn member of the DialogItemMessage union u.

```
struct
{
   int  focusOutType;       /* => reason focus out of prev item */
} focusIn;
```

Field	Description
focusOutType	Specifies the reason the focus changed from the previous item to the one about to get focus. Possible values for this field are listed at the top of this section and are defined in msdefs.h.

DITEM_MESSAGE_FOCUSOUT message

The DITEM_MESSAGE_FOCUSOUT message is sent when an item will soon lose the input focus. When receiving the DITEM_MESSAGE_FOCUSOUT message, the item hook function can optionally determine whether the item's internal value is out of range. Additional information is included in the focusOut member of the DialogItemMessage union u as follows:

```
struct
{
   boolean    hookHandled;       /* <= only for hooks,TRUE = handled */
   boolean    outOfRange;        /* <= if current value out of range */
   int        nextFocusItemIndex; /* <=> index of next focus item */
   char       *stringValueP;     /* => hooks only, string value */
   char       *formatStrP;       /* => hooks only, to internal format */
   char       *minStrP;          /* => hooks only, max string */
   char       *maxStrP;          /* => hooks only, min string */
   int        focusOutType;      /* => reason for focus out */
   int        moveDirection;     /* <=> 1: forward, -1: backward */
   RawItemHdr *nextFocusRiP;     /* <=> next item to get focus */
} focusOut;
```

Field	Description
hookHandled	Indicates whether the dialog box manager should continue its default handling for the focus out message. The item hook function should set this field to TRUE to stop the default handling.
outOfRange	Indicates whether the current input focus item contains a value that is out of range. The dialog hook function sets this field to TRUE for an out-of-range value. In this case, the dialog will not lose the input focus unless it is forced to. (Some methods of changing the input focus allow out-of-range focus-out errors to be ignored).
nextFocusItemIndex	Contains the index of the next item to receive focus as a result of this focus out message. A value of -1 means search for the next focusable item in the direction indicated by the moveDirection field.

Field	Description
stringValueP	Contains the item's current string value. This field will be non-NULL only for items that store their internal values as strings. Currently, only text items do this. The string value can be checked against the strings pointed to by minStrP and maxStrP to see if it is in range.
formatStrP	Points to the formatToInternal string that is part of an item's resource specification. This pointer, if non-NULL, can be used with the sscanf function to convert stringValueP, minStrP and maxStrP to integers or doubles to range-check stringValueP. This field is used only if the item that is losing the focus stores its internal value as a string. Currently, only text items do this.
minStrP	Points to a minimum string that is a part of an item's resource specification. (Only text items currently have this field).
maxStrP	Points to a maximum string that is a part of an item's resource specification. (Only text items currently have this field).
focusOutType	Specifies the reason the focus changed from the previous item to the one about to get focus. Possible values for this field are listed at the top of this section and are defined in msdefs.h.
moveDirection	Specifies the direction in which focus is shifting to the next item receiving focus. A value of 1 means forward and -1 means backward.
nextFocusRiP	Pointer to the Raw Item Header of the item to receive focus as a result of this focus out event.

DITEM_MESSAGE_KEYSTROKE message

The DITEM_MESSAGE_KEYSTROKE message is sent when a key is pressed while the item has the input focus. Additional information is included in the keystroke member of the DialogItemMessage union u as follows:

```
struct
{
   boolean   hookHandled;        /* <= only for hooks,TRUE = handled */
   int       moveDirection;      /* <= -1 or 1 indicate next field */
   int       keyStroke;          /* <=> */
```

```
    int       rawKeyStroke;       /* <=> */
    int       qualifierMask;      /* => */
    Inputq_element *iqelP;        /* => */
    int       cursorIndex;        /* => hook only, charIndex
                                     curs before */
    boolean   isAccelerator;      /* <=> TRUE = accelerator keystroke */
    ULong     commandNumber;      /* <=> Command num for accelerator */
    char      *unparsedP;         /* <=> Parm string for cmd function */
    char      *commandTaskIdP;    /* <=> Task containing cmd function */
} keystroke;                  /* also for POSTKEYSTROKE msgs */
```

Field	Description
hookHandled	Indicates whether the dialog box manager should continue default handling for the keystroke. The item hook function should set this field to TRUE to stop the default handling.
moveDirection	Indicates that the key that was pressed caused the input focus to be moved if hookHandled is TRUE. 1 means that the input focus should be moved forward (later in the dialog item list) to an item that can accept the focus. -1 means that the input focus should be moved backward (earlier in the dialog item list) to an item that can accept the focus.
keystroke	Contains the keystroke that was just pressed. If the item hook function changes this field, the new value of keystroke will be the value used in the item handler's default keystroke processing (unless hookHandled is set to TRUE).
qualifierMask	Contains the state of the qualifier keys when the keystroke event occurred. The possible qualifier keys are SHIFTKEY, CTRLKEY and ALTKEY. These bitmasks are defined in **keys.h**.
iqelP	Contains a pointer to the raw input queue element for the keystroke event. See **msinputq.h** for the declaration of the InputQ_element structure.
cursorIndex	Contains the current cursor position. 0 means that the cursor is in front of the first character.
isAccelerator	A value of TRUE for this field indicates that the current keystroke is being interpreted as a command accelerator. If this field is TRUE, then the three fields describing the command information contain valid data. The hook function may change the value of this field to perform special case handling of keystroke accelerators.
commandNumber	Contains the command number which will be queued to MicroStation after all keystroke processing is completed. This field is only valid of isAccelerator is TRUE.

Field	Description
unparsedP	Contains a pointer to the unparsed character string which will be passed to the command function when the command is executed. This field is only valid of isAccelerator is TRUE.
commandTaskIdP	Contains a pointer to the task ID of the application which implements the command indicated by the commandNumber field. This field is only valid of isAccelerator is TRUE.

DITEM_MESSAGE_POSTKEYSTROKE message

The DITEM_MESSAGE_POSTKEYSTROKE message is sent after the item handler's default keystroke handling processes a key. Additional information is included in the keystroke member of the DialogItemMessage union u. See the discussion of the DITEM_MESSAGE_KEYSTROKE message for more information.

Generic item hook function messages

Generic item hook function messages are only sent to item hook functions attached to generic items. See"Generic item functions" on page 16-92 for more information on generic items.

DITEM_MESSAGE_SETLABEL message

The DITEM_MESSAGE_SETLABEL message is sent when the item's label needs to be changed. If the item doesn't have a visible label, this message can be ignored. Additional information is included in the setLabel member of the DialogItemMessage union u as follows:

```
struct
{
   char *newLabelP;     /* => */
} setLabel;
```

Field	Description
newLabelP	Points to the string that should be used as the item's new label.

DITEM_MESSAGE_GETVALUE message

The DITEM_MESSAGE_GETVALUE message is sent when the item's internal value is needed. If the item does not have an internal value, this message can be ignored. Additional information is included in the value member of the DialogItemMessage union u. The interpretation of all fields in the value member is the same as the interpretation of fields for the DITEM_MESSAGE_GETSTATE message except that instead of returning the item's state, the item hook function returns the item's internal value.

DITEM_MESSAGE_SETVALUE message

The DITEM_MESSAGE_SETVALUE message is sent when the item's internal value will be set. If the item does not have an internal value, this message can be ignored. Additional information is included in the value member of the DialogItemMessage union u. The interpretation of all fields in the value member is the same as the interpretation of the DITEM_MESSAGE_SETSTATE message except that instead of setting the item's state, the item hook function sets the item's internal value.

DITEM_MESSAGE_DRAW message

The DITEM_MESSAGE_DRAW message is sent when the item should draw itself. Additional information is included in the draw member of the DialogItemMessage union u as follows:

```
struct
{
   boolean   eraseFirst;    /* => erase before drawing */
   int       nRects;        /* => # of update rects */
   Rectangle *rectList;     /* => update rects (local coords) */
} draw;
```

Field	Description
eraseFirst	Contains TRUE if the item should be erased first before it is drawn. Use the item's DialogItem rect field to determine where to erase.
nRects	Contains the number of rectangles in the next field.
rectList	Contains an array of rectangles. To optimize drawing (rarely necessary) draw only the item's parts that intersect with a rectangle in this array.

DITEM_MESSAGE_HIGHLIGHT message

The DITEM_MESSAGE_HIGHLIGHT message is sent when the item needs to change its highlight state. This message is sent only if a motion function was not specified in response to a mouse down event. While the user keeps the mouse down, a highlight-on message will be sent when the cursor moves into the item's mouse-sensitive area, and a highlight-off message will be sent when it moves out. Additional information is included in the highlight member of the DialogItemMessage union u as follows:

```
struct
{
   boolean        highlightOn;
} highlight;
```

Field	Description
highlightOn	Indicates whether the item should be drawn in its highlighted state. If so, this field is TRUE. Otherwise, the item should be drawn in its unhighlighted state.

DITEM_MESSAGE_MOVE message

The DITEM_MESSAGE_MOVE message is sent when the item needs to be moved. Additional information is included in the move member of the DialogItemMessage union u as follows:

```
struct
{
   Point2d   newPt;    /* => new origin of rawItemHdr.itemRect */
   boolean   redraw;   /* => redraw at new position? */
} move;
```

Field	Description
newPt	Contains the point where the item's RawItemHdr itemRect.origin should be moved to.
redraw	Indicates whether the item should be erased before it is moved and then drawn in its new location. If so, this field is TRUE.

DITEM_MESSAGE_SETEXTENT message

The DITEM_MESSAGE_SETEXTENT message is sent when the item size needs to be changed. Additional information is included in the setExtent member of the DialogItemMessage union u as follows:

```
struct
{
   /* used as if resource extent was respecified (except in pixels) */
   Sextent   sextent;  /* => */
   boolean   redraw;   /* => redraw at new position? */
} setExtent;
```

Field	Description
sextent	Contains the item's new extent (in pixels).
redraw	Indicates whether the item should be erased and then drawn in its new size and at its possibly new location. If so, this field is TRUE.

DITEM_MESSAGE_FONTCHANGED message

The DITEM_MESSAGE_FONTCHANGED message is sent when the dialog box is moved to a screen that uses a different font size than the screen it was previously on. By default, the dialog box manager will scale the size of each item. If necessary, an item hook function can perform a different resizing operation on the item in response to this message.

Additional information is included in the fontChanged member of the DialogItemMessage union u as follows:

```
struct
{
   boolean   hookHandled;        /* <= TRUE if handled */
   int       newFontHeight;      /* => new font's height */
   int       oldFontHeight;      /* => old font's height */
} fontChanged;         /* because dialog now on new screen */
```

Field	Description
hookHandled	Indicates whether the dialog box manager should continue default handling for the font changed message. The item hook function should set this field to TRUE to stop the default handling.
newFontHeight	Specifies the font height that should be used in calculating the new item size.
oldFontHeight	Specifies the height of the dialog box's old font.

DITEM_MESSAGE_SETENABLEDSTATE message

The DITEM_MESSAGE_SETENABLEDSTATE message is sent when the item's enabled state is changed. Additional information is included in the setEnabledState member of the DialogItemMessage union u as follows:

```
struct
{
   boolean   enabled;  /* => TRUE = enabled */
} setEnabledState;
```

Field	Description
enabled	Indicates whether the item has been enabled. If so, this field is TRUE.

DITEM_MESSAGE_ACTIVATE message

The DITEM_MESSAGE_ACTIVATE message is sent to the dialog item hook function when one of the following occurs:

- The user uses a keyboard accelerator key to activate a specific menu bar menu item.
- The user uses a keyboard mnemonic to activate a specific dialog item.
- An application calls the mdlDialog_pushButtonActivate function to activate a push button item.

If the dialog box containing the item being activated has a hook function, the DIALOG_MESSAGE_ACTIVATE message is sent after this message has been sent to the dialog item.

Additional information is included in the activate member of the DialogItemMessage union u as follows:

```
struct
{
   ULong     couldSetState;      /* <= if not modal, could set state */
   int       keystroke;          /* => virtual keystroke */
   int       rawKeystroke;       /* => raw keystroke */
   int       qualifierMask;      /* => modified key status */
   boolean   isAccelerator;      /* => key is accelerator */
   ULong     commandNumber;      /* => accelerator key command */
   char      *unparsedP;         /* => accelerator key cmd parms */
   char      *commandTaskIdP;    /*=> accelerator key cmd task id */
} activate;
```

Field	Description
couldSetState	This field is unused in this message.
keystroke	Contains the platform independent (virtual) keystroke that was just pressed.
rawKeystroke	Contains the raw, native keystroke just pressed.
qualifierMask	Contains the state of the qualifier keys at the time of the keystroke event. The possible qualifier keys are SHIFTKEY, CTRLKEY and ALTKEY. These bitmasks are defined in **keys.h**.
isAccelerator	A value of TRUE for this field indicates that the current keystroke is being interpreted as a command accelerator. If this field is TRUE, then the three fields describing the command information contain valid data.
commandNumber	Contains the command number which will be queued to MicroStation after all keystroke processing is completed. This field is only valid of isAccelerator is TRUE.

Field	Description
unparsedP	Contains a pointer to the unparsed character string which will be passed to the command function when the command is executed. This field is only valid of isAccelerator is TRUE.
commandTaskIdP	Contains a pointer to the task ID of the application which implements the command indicated by the commandNumber field. This field is only valid of isAccelerator is TRUE.

18 A Complete Example

This chapter describes an MDL application called **basic.ma** that opens a simple, modeless dialog box. While the application is basic, as the name implies, it does illustrate all of the steps necessary to create an MDL application that uses the dialog box manager.

MDL Applications

It is assumed that the reader is familiar with the contents of the preceding three chapters. However, this section can also be read to *capture the flavor* of MDL dialog box applications.

The **basic** MDL application is also featured in "Building Applications" on page 7-1, which you may want to review concurrently with this one.

All the source files needed to create the application are here along with detailed explanations. These files are also found in the **mdl\examples\basic** directory. To run this example, first make the **basic** application by going to the **mdl\examples\basic** directory and entering: **BMAKE BASIC.**

To load the application, start MicroStation and select the "MDL Applications" menu item from MicroStation's "User" pull-down menu. Then select the **basic** application using the dialog box that appears. If the **basic** application does not appear in the list of "Available Applications" make sure that the `MS_MDL` MicroStation environment includes the **\ustation\mdlapps** directory.

The basic.ma application

When the **basic.ma** application is loaded, a modeless dialog box entitled "Basic Dialog Box" appears on the screen. This modeless dialog box contains three items: a text item, an option button item and a push button item.

Both the text and option button item controls let the user view the same application variable `parameter1`. Using a synonym resource, whenever either of these items changes the variable `parameter1`, the other item's appearance also changes to match `parameter1`'s new value.

When the user clicks on the push button item labeled "Open Modal," a modal dialog box is displayed. Its title is "Basic Modal Dialog Box". As long as this modal dialog box is present on the screen, the user cannot interact with any other window's contents.

The "Basic Modal Dialog Box" also contains three items: a toggle button, a push button labeled "OK," and a push button labeled "Cancel." This modal dialog box lets the user increment the application variable `parameter1`.

If the user clicks the OK button after turning the toggle button "on," the modal dialog box will be dismissed and `parameter1` will be incremented. The appearance of the text and option button items in the "Basic Dialog Box" is also changed to reflect the new value of `parameter1`. If the toggle button is "off" when the OK button is clicked or the Cancel button is clicked, the modal dialog box is simply dismissed and `parameter1` remains unchanged.

When the user closes the initial "Basic Dialog Box," the **basic.ma** application is automatically unloaded.

The **basic** application consists of the following files:

file	description
basic.h	Defines the constants and data structures used by the application.
basic.r	Contains the definition of the application's resources.
basic.mc	Contains the application's source code.
basiccmd.r	Contains the definition of the application's command table resources.
basiccmd.h	Contains the application's command numbers (generated by **rcomp** when **basiccmd.r** is compiled).
basictyp.mt	Used to generate type declaration resources for the application.
basicmsg.r	Contains language-specific messages (located in the **english** subdirectory).
basictxt.h	Contains language-specific text strings (located in the **english** subdirectory).
basic.mke	Contains information used by the **bmake** utility to create the **basic.ma** application from the above files.
basicrsc.mki	Contains rules and dependencies used by **basic.mke**.

The application header file: basic.h

```
/*----------------------------------------------------------------------+
|                                                                       |
|    Function -                                                         |
|                                                                       |
|    Constants & types used in basic dialog example                     |
|                                                                       |
+----------------------------------------------------------------------*/
#ifndef    __basicH__
#define    __basicH__

/*----------------------------------------------------------------------+
|                                                                       |
|    Resource ID's                                                      |
|                                                                       |
+----------------------------------------------------------------------*/
#define DIALOGID_Basic 1           /* dialog id for Basic Dialog */
#define DIALOGID_BasicModal 2 /* dialog id for Basic Modal Dialog */
#define OPTIONBUTTONID_Basic 1/* id for parameter 1 option button */
#define PUSHBUTTONID_OModal 1 /* id for "Open Modal" push button */
#define TEXTID_Basic 1        /* id for "parameter 1" text item */
#define TOGGLEID_Basic 1      /* id for "Inc parameter 1?" toggle */
#define SYNONYMID_Basic 1          /* id for synonym resource */
#define MESSAGELISTID_BasicErrors 1/* id for error message list */
/*----------------------------------------------------------------------+
|                                                                       |
|    Error Message ID Definitions                                       |
|                                                                       |
+----------------------------------------------------------------------*/
#define ERRID_CommandTable 1
#define ERRID_BasicDialog 2
#define ERRID_ModalDialog 3

/*----------------------------------------------------------------------+
|                                                                       |
|    Hook Function ID's                                                 |
|                                                                       |
+----------------------------------------------------------------------*/
#define HOOKITEMID_ToggleButton_Basic 1 /* toggle for item hook id */
#define HOOKDIALOGID_Basic 2            /* id for dialog hook func */
/*----------------------------------------------------------------------+
|                                                                       |
|    Typedefs                                                           |
|                                                                       |
+----------------------------------------------------------------------*/
typedef struct basicglobals
{
```

```
67 int parameter1;    /* used by text & option button item */
68 int parameter2;    /* used by toggle button item */
69 } BasicGlobals;
70 #endif
```

The header file **basic.h** is included in **basic.r**, **basic.mc** and **basictyp.mt**.

Lines 37-44 define symbolic constants that are used as resource ID numbers. Remember that each instance of a particular resource type needs to have its own unique, positive resource ID number. That is why `DIALOGID_BasicModal` is defined to be 2 and not 1, which is already assigned to `DIALOGID_Basic`. Both of these resource IDs are used to create a `DialogBoxRsc` resource (with a resource type of `RTYPE_DialogBox`) and therefore must be different numbers. As a further illustration, if another `DItem_OptionButtonRsc` resource was needed for this application its resource ID could not be 1. That number is already used by `OPTIONBUTTONID_Basic`.

See the "Resources" section in the "MicroStation Dialog Box Manager Overview" chapter for more information on creating resource IDs.

Lines 50-52 define symbolic constants which are used to reference entries in the Message List resource defined by `MESSAGELISTID_BasicErrors`.

Lines 58-59 define symbolic constants used as hook function ID numbers. Remember that each hook function in an application requires its own unique, positive ID number. This is true whether the hook function is a dialog hook function or a dialog item hook function. Dialog hook functions and item hook functions share the same number space. This is why `HOOKDIALOGID_Basic` is 2 and not 1, which is already used by `HOOKITEMID_ToggleButton_Basic`.

See the "Hook function IDs" section in the "MicroStation Dialog Box Manager Overview" chapter for more information on creating hook function IDs.

Finally, lines 65-69 declare the structure `BasicGlobals`. The dialog items in the basic application's dialog boxes control members of a variable of this type (see lines 104, 115 and 143 in **basic.r**).

The `BasicGlobals` structure is created to hold the variables that these dialog box items modify. This was done because any application variable that a dialog box item refers to in a C expression string resource must be published in a symbol set for use by the C expression handling functions. When the variables are part of a structure, only one `mdlDialog_publishComplexVariable` or `mdlDialog_publishComplexPtr` function call must be done for that structure. Otherwise, an `mdlDialog_publishÉ` function call is required for each referenced variable.

See "Referencing Application Variables from Resource Files" on page 15-10 for more information on variable references within C expression strings.

The resource file: basic.r

```
/*----------------------------------------------------------------------+
|                                                                       |
|    Function -                                                         |
|                                                                       |
|    Basic Dialog Example Resources                                     |
|                                                                       |
+----------------------------------------------------------------------*/
/*----------------------------------------------------------------------+
|                                                                       |
|    Include Files                                                      |
|                                                                       |
+----------------------------------------------------------------------*/
#include <dlogbox.h>/* dlog box manager resource consts & structs */
#include <dlogids.h>/* MicroStation resource IDs */
#include "basic.h"  /* basic dialog box example consts & structs */
#include "basiccmd.h"    /* basic dialog box command numbers */
#include "basictxt.h"    /* basic dialog box static text defines */
/*----------------------------------------------------------------------+
|                                                                       |
|    Basic Dialog Box                                                   |
|                                                                       |
+----------------------------------------------------------------------*/
#define X1 (14*XC)/* text & option button x position */
#define X2 (7*XC)/* push button x position */
#define XW (9*XC)/* text & option button width */
#define BTN_WIDTH (12*XC)/* push button width */

DialogBoxRsc DIALOGID_Basic=
{
    DIALOGATTR_DEFAULT | DIALOGATTR_SINKABLE,
    25*XC, 7*YC,
    NOHELP, MHELP, HOOKDIALOGID_Basic, NOPARENTID,
    TXT_BasicDialogBox,
{
{{X1,GENY(1),XW,0}, Text, TEXTID_Basic, ON, 0, "", ""},
{{X1,GENY(2),XW,0}, OptionButton,OPTIONBUTTONID_Basic, ON, 0,"",""},
{{X2,GENY(4),BTN_WIDTH,0},PushButton,PUSHBUTTONID_OModal,ON,0,"",""}
}
};
#undef X1      /* undef symbols so they can be reused */
#undef X2
#undef XW
#undef BTN_WIDTH

/*----------------------------------------------------------------------+
|                                                                       |
```

```
67 |    Modal Sub-Dialog Box                                              |
68 |    (opened when PUSHBUTTONID_OModal is activated)                    |
69 |                                                                      |
70 +----------------------------------------------------------------------*/
71 #define X1 (1*XC)        /* toggle button x position */
72 #define X2 (3*XC)        /* OK button x position */
73 #define X3 (14*XC)       /* Cancel button x position */

74 DialogBoxRsc DIALOGID_BasicModal=
75 {
76 DIALOGATTR_DEFAULT | DIALOGATTR_MODAL,
77 25*XC, 6*YC,
78 NOHELP, MHELP, HOOKDIALOGID_Basic, NOPARENTID,
79 TXT_BasicModalDialogBox,
80 {
81 {{X1,GENY(1),0,0}, ToggleButton, TOGGLEID_Basic, ON, 0, "", ""},
82{{X2,GENY(3),BUTTON_STDWIDTH,0},PushButton,PUSHBUTTONID_OK,ON,0,"","
},
83 {{X3,GENY(3),BUTTON_STDWIDTH,0},PushButton,PUSHBUTTONID_Cancel,ON,
0,"",""},
84 }
85 };
86 #undef X1      /* undef symbols so they can be reused */
87 #undef X2
88 #undef X3

89 /*----------------------------------------------------------------------+
90 |                                                                      |
91 |    Item Resource Specifications                                      |
92 |                                                                      |
93 +----------------------------------------------------------------------*/
94 /*----------------------------------------------------------------------+
95 |                                                                      |
96 |    Text Item Resources                                               |
97 |                                                                      |
98 +----------------------------------------------------------------------*/
99 DItem_TextRsc TEXTID_Basic=
100{
101     NOCMD, LCMD, SYNONYMID_Basic, NOHELP, MHELP, NOHOOK, NOARG,
102     4, "%ld", "%ld", "1", "3", NOMASK, NOCONCAT,
103     TXT_Parameter1,
104     "basicGlobals.parameter1"
105};

106/*----------------------------------------------------------------------+
107|                                                                      |
108|    Option Button Item Resources                                      |
109|                                                                      |
110+----------------------------------------------------------------------*/
```

```
111DItem_OptionButtonRsc OPTIONBUTTONID_Basic=
112{
113     SYNONYMID_Basic, NOHELP, MHELP, NOHOOK, NOARG,
114     TXT_Parameter1,
115     "basicGlobals.parameter1",
116     {
117          {NOTYPE, NOICON, NOCMD, LCMD, 1, NOMASK, ON, TXT_Value1},
118          {NOTYPE, NOICON, NOCMD, LCMD, 2, NOMASK, ON, TXT_Value2},
119          {NOTYPE, NOICON, NOCMD, LCMD, 3, NOMASK, ON, TXT_Value3},
120     }
121};

122/*----------------------------------------------------------------------+
123|                                                                       |
124|    PushButton Item Resources                                          |
125|                                                                       |
126+----------------------------------------------------------------------*/
127DItem_PushButtonRsc PUSHBUTTONID_OModal=
128{
129     NOT_DEFAULT_BUTTON, NOHELP, MHELP, NOHOOK, 0,
130     CMD_OPENMODAL, LCMD, "",
131     TXT_OpenModal
132};

133/*----------------------------------------------------------------------+
134|                                                                       |
135|    Toggle Button Item Resources                                       |
136|                                                                       |
137+----------------------------------------------------------------------*/
138DItem_ToggleButtonRsc TOGGLEID_Basic=
139{
140     NOCMD, LCMD, NOSYNONYM, NOHELP, MHELP,
141     HOOKITEMID_ToggleButton_Basic, NOARG, NOMASK, NOINVERT,
142     TXT_IncrementParameter1,
143     "basicGlobals.parameter2"
144};

145/*----------------------------------------------------------------------+
146|                                                                       |
147|    Synonym List Resources                                             |
148|                                                                       |
149+----------------------------------------------------------------------*/
150DItem_SynonymsRsc SYNONYMID_Basic=
151{
152     {
153          {Text, TEXTID_Basic},
154          {OptionButton,OPTIONBUTTONID_Basic},
155     }
156};
```

The resource source file **basic.r** contains all of the dialog and item resources for the **basic** application.

The header file **dlogbox.h** is included on line 35. **dlogbox.h** contains all of the MicroStation dialog box manager resource related constants and structures. It must be included in any resource source file that defines dialog box manager related resources.

The header file **dlogids.h** is included on line 36. **dlogids.h** contains the resource IDs of all of the dialog box manager related resources present in the MicroStation resource file. This file is needed because the "Basic Modal Dialog Box" uses the standard MicroStation "OK" and "Cancel" push button item resources.

The header file **basic.h** is included on line 37; it contains the application's resource ID and hook function ID numbers.

The header file **basiccmd.h** is included on line 38; it contains the command number for the **basic** application's Openmodal command. This is used in the definition of the push button whose resource ID is `PUSHBUTTONID_OModal` (see line 130).

The header file **basictxt.h** is included on line 39; it contains the static text label definitions referenced by the dialog resources contained in the basic application dialog boxes as shown in lines 54, 79, 103, 114, 131 and 142.

The modeless dialog box entitled "Basic Dialog Box" is defined on lines 45-64. This dialog box is 7 characters tall and approximately 25 characters wide. It will have a "sink" icon on the right side of its title bar. It has a dialog hook function, which in this case is the `basic_dialogHook` function (see line 71 in **basic.mc**). The dialog box has three entries in its dialog item list: a text item, an option button item, and a push button. The x position, width of the text and option button items are the same and are defined by the symbolic constant `X1` and `XW` respectively. This ensures that if it is necessary to change their X position or size, their horizontal relationship will remain correct. The `GENY` macro is used to position the text item on "row" 1, and the option button on "row" 2. Since push buttons are taller than the other items, it is placed on "row" 4 to allow more space between it and the option button item. The height of all three items is specified to be 0. This indicates that the associated item handlers should calculate an appropriate default height.

The modal dialog box entitled, "Basic Modal Dialog Box" is defined on lines 71-88. This dialog box is 6 characters tall and approximately 25 characters wide. Since it is modal, the `DIALOGATTR_MODAL` bit is set in its attributes member. It has a dialog hook function, which is the same `basic_dialogHook` function that the "Basic Dialog Box" uses (see line 71 in **basic.mc**). This hook function looks at the `dialogId` member of any dialog messages received to determine which dialog box the message is for. The dialog box has three entries in its dialog box item list: a toggle button item, and two push button items. Note that the resources for the two push button items are not defined in **basic.r**; instead, these standard resources will be loaded from MicroStation's

resource file. This ensures that the "OK" and "Cancel" modal dialog box behavior is correctly implemented.

See "DialogBoxRsc Structure" on page 16-2 and + for more information on creating dialog box resources.

The next section in **basic.r** defines the actual dialog item resources that were referenced in the above dialog boxes. See the "Standard Dialog Box Items" chapter for more information on creating dialog item resources.

Lines 99-105 define the text item referred to in line 56. The text item can contain up to 4 characters, is displayed as an integer, and must be between 1 and 3. It's label is "Parameter 1": and it controls the `basicGlobals.parameter1` application variable.

This text item has a synonym resource associated with it whose ID is `SYNONYMID_Basic`. Therefore, whenever the user modifies the text item to change `basicGlobals.parameter1`, the appearance of all of the other items that are listed in the `SYNONYMID_Basic` synonym resource will be forced to match the state of their application variables. In this example, the appearance of the `OPTIONBUTTONID_Basic` option button item will change to reflect the new value of `basicGlobals.parameter1`.

Lines 111-121 define the option button item referred to in line 57. Its hook function is the `basic_dummyItemHook` function. It displays the text strings "Value 1," "Value 2" or "Value 3" depending on the value of the `basicGlobals.parameter1` application variable. The option button item also has the `SYNONYMID_Basic` synonym resource associated with it. This means that when the user modifies `basicGlobals.parameter1` with the option button item, the appearance of the `TEXTID_Basic` text item will also be changed.

Lines 127-132 define the "Open Modal" push button item referred to in line 58. Its hook function is the `basic_dummyItemHook` function. The **basic** application's Openmodal command will be placed in the input queue when the user clicks on this button.

Lines 138-144 define the toggle button item referred to in line 81. Its hook function is the `basic_toggleButtonHook` function (see line 50 in **basic.mc**). It controls the `basicGlobals.parameter2` application variable.

Lines 150-156 define the synonym resource that is associated with both the `TEXTID_Basic` text item and the `OPTIONBUTTONID_Basic` option button item. See the "Synonym resources" section in the "MicroStation Dialog Box Manager Overview" chapter for more information on synonym resources.

The source code file: basic.mc

```
/*----------------------------------------------------------------------+
|                                                                       |
|   Function -                                                          |
|                                                                       |
|   MDL example to show creation & use of basic dialog box              |
|                                                                       |
|    - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -        |
|                                                                       |
|   Public Routine Summary -                                            |
|                                                                       |
|    main - Main entry point and initialization logic                   |
|    basic_dialogHook - Dialog box hook function                        |
|    basic_toggleButtonHook - Toggle button item hook function          |
|    basic_openModal - Application command function to open modal|
|        dialog box                                                     |
|    basic_errorPrint - Function to display an error message in         |
|        messages dialog box                                            |
|                                                                       |
+----------------------------------------------------------------------*/
/*----------------------------------------------------------------------+
|                                                                       |
|   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 <dlogman.fdf>   /* dialog box manager function prototypes */
#include "basic.h"/* basic dialog box example constants & structs */
#include "basiccmd.h"    /* basic dialog box command nums */

/*----------------------------------------------------------------------+
|                                                                       |
|    Private Global variables                                           |
|                                                                       |
+----------------------------------------------------------------------*/
/* The following variable is referenced in C expression strings used
by the text, option button, and toggle button items defined in
basic.r. The initial external state of the text item and the option
button (they are both looking at the same application variable) is 1.
The initial external state of the toggle button in the modal dialog
box is 0 (the toggle is "out" or OFF) */

Private BasicGlobals basicGlobals={1, 0};
void basic_toggleButtonHook();
```

```
67 void basic_dialogHook();
68 Private DialogHookInfo uHooks[]=
69 {
70      {HOOKITEMID_ToggleButton_Basic, basic_toggleButtonHook},
71      {HOOKDIALOGID_Basic, basic_dialogHook},
72 };
73 /*----------------------------------------------------------------------+
74 |                                                                       |
75 |    name main                                                          |
76 |                                                                       |
77 |    author BSI     12/90                                               |
78 |                                                                       |
79 +----------------------------------------------------------------------*/
80 Public main
81 (
82 int argc,       /* => number of args in next array */
83 char *argv[]    /* => array of cmd line arguments */
84 )
85 {
86    char *setP;           /* a ptr to a "C expression symbol set" */
87    RscFileHandle rscFileH;/* a resource file handle */
88    DialogBox *dbP;       /* a ptr to a dialog box */

89    /* open the resource file that we came out of */
90    mdlResource_openFile (&rscFileH, NULL, 0);

91    /* load the command table */
92    if (mdlParse_loadCommandTable(NULL) == NULL)
93          basic_errorPrint(ERRID_CommandTable);

94    /* set up variables to be evaluated w/i C expression strings */
95    setP=mdlCExpression_initializeSet(VISIBILITY_DIALOG_BOX,0,FALSE);
96    mdlDialog_publishComplexVariable(setP, "basicglobals",
97                                "basicGlobals", &basicGlobals);

98    /* publish our hook functions */
99 mdlDialog_hookPublish(sizeof(uHooks)/sizeof(DialogHookInfo),uHooks);
100    /* open the "Basic" dialog box */
101    if ((dbP = mdlDialog_open(NULL, DIALOGID_Basic)) == NULL)
102          basic_errorPrint(ERRID_BasicDialog);
103 }
104/*----------------------------------------------------------------------+
105|                                                                       |
106|    Hook Functions                                                     |
107|                                                                       |
108+----------------------------------------------------------------------*/
109/*----------------------------------------------------------------------+
110|                                                                       |
```

```
|    name basic_dialogHook                                          |
|                                                                   |
|    author BSI          12/90                                      |
|                                                                   |
+-------------------------------------------------------------------*/
Private void basic_dialogHook
(
DialogMessage *dmP    /* => a ptr to a dialog message */
)
{
    /* ignore any messages being sent to modal dialog hook */
    if (dmP->dialogId != DIALOGID_Basic)
         return;
    dmP->msgUnderstood = TRUE;

    switch (dmP->messageType)
    {
         case DIALOG_MESSAGE_DESTROY:
         {
              /* unload this mdl task when Basic Dialog is closed */
              mdlDialog_cmdNumberQueue(FALSE, CMD_MDL_UNLOAD,
                             mdlSystem_getCurrTaskID(), TRUE);
              /*  mdlSystem_getCurrTaskID was erroneously omitted
                  from the MDL documentation. It returns a character
                  pointer pointing to the current task's task ID. */
              break;
         };
         default:
              dmP->msgUnderstood = FALSE;
              break;
    }
}

/*-------------------------------------------------------------------+
|                                                                    |
| name    basic_toggleButtonHook                                     |
|                                                                    |
| author BSI             12/90                                       |
|                                                                    |
+-------------------------------------------------------------------*/
Private void basic_toggleButtonHook
(
DialogItemMessage *dimP  /* => a ptr to a dialog item message */
)
{
    dimP->msgUnderstood = TRUE;
    switch (dimP->messageType)
    {
```

```
157          case DITEM_MESSAGE_CREATE:
158          {
159               basicGlobals.parameter2 = 0;
160               break;
161          }
162          default:
163               dimP->msgUnderstood = FALSE;
164               break;
165     }
166}
167/*----------------------------------------------------------------------+
168|                                                                       |
169|   Command Handling routines                                           |
170|                                                                       |
171+----------------------------------------------------------------------*/
172/*----------------------------------------------------------------------+
173|                                                                       |
174|   name basic_openModal                                                |
175|                                                                       |
176|   author BSI          12/90                                           |
177|                                                                       |
178+----------------------------------------------------------------------*/
179Public cmdName void basic_openModal
180(
181char *unparsedP     /* => unparsed part of command */
182)
183cmdNumber CMD_OPENMODAL
184{
185     int lastAction;

186     /* open child modal dialog box */
187     if (mdlDialog_openModal(&lastAction, NULL, DIALOGID_BasicModal))
188     {
189          basic_errorPrint (ERRID_ModalDialog);
190          return;
191     }

192     /* if the user clicked in OK push button, and the toggle button
193        is currently "in", then increment parameter1 and update the
194        appearance of both the text item and the option button item
195        in the Basic dialog box */

196     if (lastAction == ACTIONBUTTON_OK && basicGlobals.parameter2)
197     {
198          basicGlobals.parameter1++;
199          if (basicGlobals.parameter1 > 3)
200               basicGlobals.parameter1 = 1;
201          mdlDialog_synonymsSynch(NULL, SYNONYMID_Basic, NULL);
```

```
202     }
203}
204/*---------------------------------------------------------------------+
205|                                                                      |
206|    Utility routines                                                  |
207|                                                                      |
208+---------------------------------------------------------------------*/
209/*---------------------------------------------------------------------+
210|                                                                      |
211|    name basic_errorPrint -- print an error message into Dialog      |
212|                              Box Manager Messages dialog box         |
213|                                                                      |
214|    author BSI          12/90                                         |
215|                                                                      |
216+---------------------------------------------------------------------*/
217Private void basic_errorPrint
218(
219int errorNumber    /* => number of error to print */
220)
221{
222     char errorMsg[80];
223     if (mdlResource_loadFromStringList(errorMsg, NULL,
224               MESSAGELISTID_BasicErrors, errorNumber))
225          return;   /* unable to find msg. w/ num. "errorNumber" */
226     mdlDialog_dmsgsPrint(errorMsg);
227}
```

The `main` function is defined on lines 80-103. It starts by opening the resource file from which its application was loaded by calling `mdlResource_openFile` with `NULL` as its second argument. In this case, the resource file **basic.ma** is opened in read-only mode. This resource file needs to be opened before any resources that reside in it can be accessed.

The `mdlParse_loadCommandTable` function call on line 92 makes the contents of the command table present in **basic.ma** available as MicroStation key-ins. In this case, the user can key in **OPENMODAL** to display the "Basic Modal Dialog Box", instead of clicking on the "Open Modal" push button.

The text, option button and toggle button item resources defined in **basic.r** refer to the basicGlobals variable that is defined on line 65. Lines 96 and 97 publish this variable and its structure declaration to the C expression handling functions. The dialog box manager can then evaluate the strings in the item resources that refer to the `basicGlobals` variable (see lines 104, 115 and 143 in **basic.r**). See "Referencing application variables from resource files" in the "MicroStation Dialog Box Manager Overview" chapter for more information.

The `DialogHookInfo` array defined on lines 68-72 establishes the connection between hook function ID numbers and MDL function addresses. The `mdlDialog_hookPublish` function call on line 99 informs the dialog box manager of these connections.

Line 101 opens the Basic Dialog Box.

There are two hook functions used in this application: `basic_dialogHook` and `basic_toggleButtonHook`.

The `basic_dialogHook` function is attached to the "Basic Dialog Box." When it receives the `DIALOG_MESSAGE_DESTROY` message for this dialog box, it forces the **basic** application to be unloaded by queueing up the MicroStation MDL Unload command (see line 130).

The `basic_toggleButtonHook` function is the item hook function that is attached to the toggle button item in the "Basic Modal Dialog Box." Every time that a modal dialog box is opened, all of its items are sent `DITEM_MESSAGE_CREATE` messages. When the `basic_toggleButtonHook` function receives this message, it sets the application variable that the toggle button controls to 0, `basicGlobals.parameter2` (see line 159). This ensures that the toggle button will always be "off" when the "Basic Modal Dialog Box" is opened. Note that this only works for the Create message - the `DITEM_MESSAGE_INIT` message cannot be used. The Init message is sent after the toggle button has already been initialized from the value of its underlying application variable; setting `basicGlobals.parameter2` to 0 at this time would thus be incorrect.

The dialog box manager can be told to print out information on all messages sent to dialog boxes and dialog box items that have hook functions attached to them. The `basic_dummyItemHook` function is attached to items that normally would not require an item hook function so that the dialog box manager will print out information on the messages sent to those items. The item hook function does not however perform any useful action. The `basic_dialogHook` function is attached to the "Basic Modal Dialog Box" so that the dialog box manager will print out messages sent to the dialog box. The dialog hook function returns immediately if it receives any messages for that dialog box.

To see the flow of the dialog and item messages that the dialog box manager sends, enter the following commands before starting the **basic** application:

```
DMSG DIALOGDEBUG ON
DMSG ITEMDEBUG ON
DMSG VERBOSEDEBUG ON
```

See the "Tracking hook function messages" section in this chapter for more information on debugging dialog box manager messages.

The `basic_openModal` function is called whenever the user keys in the **basic** application's Openmodal command or clicks the "Open Modal" push button.

`basic_openModal` calls `mdlDialog_openModal` with a dialog box resource ID of `DIALOGID_BasicModal` to open the "Basic Modal Dialog Box" (see line 187). `mdlDialog_openModal` doesn't return to its caller until the user has dismissed the open modal dialog box. At that time, the lastAction variable is set to indicate which of the two standard push buttons the user clicked on to dismiss the modal dialog box.

`basic_openModal` checks to make sure that the user has clicked the "OK" push button and that the toggle button has been turned on. If both of these tests are passed, it increments `basicGlobals.parameter1` (setting it back to 1 if it becomes greater than 3). It then forces the text and option button items to be redrawn to reflect the new value of `basicGlobals.parameter1` by calling `mdlDialog_synonymsSynch`. This function "synchronizes" all of the items listed in the specified synonym resource. In other words, the items' appearances are forced to match the value of their underlying application variables.

The `basic_errorPrint` function loads a string from the `MESSAGELISTID_BasicErrors` message list and displays it in the "Dialog Box Manager Messages" dialog box. The `errorNumber` parameter indicates which string to display.

The command table file: basiccmd.r

```
23/*----------------------------------------------------------------------+
24|                                                                       |
25|   Function -                                                          |
26|                                                                       |
27|   "Basic Dialog Example" Command Table Resources                      |
28|                                                                       |
29+----------------------------------------------------------------------*/
30/*----------------------------------------------------------------------+
31|                                                                       |
32|   Include Files                                                       |
33|                                                                       |
34+----------------------------------------------------------------------*/
35#include <rscdefs.h>
36#include <cmdclass.h>

37/*----------------------------------------------------------------------+
38|                                                                       |
39|  Local Defines                                                        |
40|                                                                       |
41+----------------------------------------------------------------------*/
42#define CT_NONE 0
43#define CT_BASIC 1

44/*----------------------------------------------------------------------+
45|                                                                       |
```

```
46|   "Basic dialog example" commands                                |
47|                                                                  |
48+------------------------------------------------------------------*/
49Table CT_BASIC=
50{
51    {1, CT_NONE, INPUT, NONE, "OPENMODAL"},
52};
```

The **basiccmd.r** file is a resource source file that contains the command tables for the **basic** application. In this case there is only the main command table, which is identified by the constant `CT_BASIC`. Within the main table there is only one command: Openmodal. The Openmodal command has only one command word, "OPENMODAL," has no subtables or options, and is of the INPUT command class.

The MicroStation resource compiler, **rcomp**, is used to generate the header file **basiccmd.h** and the resource file **basiccmd.rsc** from **basiccmd.r** (see lines 42 and 43 of the makefile **basic.mke**). The header file **basiccmd.h**, which is shown in the next section, defines the command number `CMD_OPENMODAL` that is associated with the Openmodal command.

The Openmodal command is automatically placed in MicroStation's input queue when user clicks on the push button labeled "Open Modal" (see line 130 of **basic.r**). When MicroStation processes the Openmodal command it will call the `basic_openModal` function to perform whatever actions are required (see lines 179-183 of **basic.mc**).

See "Compiling an Application Command Table" on page 7-11 for more information about command tables.

The command number header file: basiccmd.h

```
1 #define CMD_OPENMODAL    0x01000000  /* INPUT */
```

The **basiccmd.h** file is created from the **basiccmd.r** file by the **rcomp** resource compiler. It contains the command number associated with the **basic** application command Openmodal.

The **basiccmd.h** header file is included in the files **basic.r** and **basic.mc**. The push button resource defined in lines 127-132 of **basic.r** uses the symbolic constant `CMD_OPENMODAL` to specify the command to place in the input queue when the user clicks on the push button. The `basic_openModal` function definition on lines 179-183 in **basic.mc** uses `CMD_OPENMODAL` to indicate that `basic_openModal` should be called whenever the Openmodal command is seen on the input queue.

The type definition file: basictyp.mt

```
30    #include "basic.h"
31    publishStructures(basicglobals);
```

This type of definition file is used to generate type declaration resources for the `BasicGlobals` structure. This is necessary because a variable of type `BasicGlobals` is referenced by C expression strings in the resource file **basic.r** (lines 104, 115 and 143).

The header file **basic.h** contains the declaration of the `BasicGlobals` structure and thus must be included in **basictyp.mt**. The `publishStructures` statement indicates that type declaration resources should be built for the structure whose tag is `basicGlobals`. The **rsctype** utility program, under the direction of line 48 of the makefile **basic.mke**, generates these resources and places them in the resource source file **basictyp.r**.

See "Generating Resource Files from C Type Definitions" on page 7-17 and "Referencing Application Variables from Resource Files" on page 15-10 for more information on type definition files.

The message file: basicmsg.r

The message file is located in the **english** directory, and contains the messages used by the application. It is separated from the other files so that it can easily be customized for any language the developer wishes.

```
23/*----------------------------------------------------------------------+
24|                                                                       |
25|   Function -                                                          |
26|                                                                       |
27|   Basic application message string resources                          |
28|                                                                       |
29+----------------------------------------------------------------------*/
30#include <dlogbox.h>/* dlog box manager resource consts & structs */
31#include <dlogids.h>/* MicroStation resource IDs */
32#include "basic.h" /* basic dialog box example consts & structs */
33/*----------------------------------------------------------------------+
34|                                                                       |
35|   Error Messages                                                      |
36|                                                                       |
37+----------------------------------------------------------------------*/
38MessageList MESSAGELISTID_BasicErrors=
```

```
39{
40    {
41          {ERRID_CommandTable, "Unable to load command table."},
42          {ERRID_BasicDialog, "Unable to open Basic dialog box."},
43          {ERRID_ModalDialog, "Unable to open modal dialog box."},
44    }
45};
```

Lines 38-45 define a message list resource that contains error messages. If all of the text strings of an application are defined in a resource file, those strings will be much easier to translate into a different language. In fact, internationalization of this example can be done without access to the **basic.mc** source file. The steps required to internationalize **basic** are as follows:

1. Create the intermediate file **basic.mi** by combining the program file (**basic.mp**) with the command table and type resources (**basiccmd.rsc** and **basictyp.rsc**).
2. Give copies of **basictxt.h** and **basicmsg.r** to the translators for internationalization. In addition, give them **basic.r** in case they need to resize any dialog items to accomodate longer strings, and give them the **basic.mi** file.
3. When the translators have finished their internationalization, they compile **basicmsg.r** into **basicmsg.rsc** and then compile **basic.r** into **basic.rsc**. Note that **basic.r** must be compiled whether it was changed or not because it includes **basictxt.h**.
4. Once **basic.rsc** and **basicmsg.rsc** are ready, they can be combined with **basic.mi** to create an internationalized version of **basic.ma**.

The text file: basictxt.h

The text file is located in the **english** directory, and contains the text strings referenced by dialog boxes as labels for the dialog box, dialog items and items subentities. It is separated from the other files so that it can easily be customized for any language the developer wishes.

```
/*----------------------------------------------------------------------+
|                                                                       |
|    Function -                                                         |
|                                                                       |
|    Static text defines for the basic application                      |
|    dialog resources                                                   |
+----------------------------------------------------------------------*/
#if !defined (__basictxtH__)
     #define __basictxtH__
     #define TXT_BasicDialogBox "Basic Dialog Box"
     #define TXT_BasicModalDialogBox "Basic Modal Dialog Box"
     #define TXT_Parameter1 "Parameter 1:"
     #define TXT_OpenModal "Open Modal"
     #define TXT_IncrementParameter1 "Increment parameter 1?"
     #define TXT_Value1 "Value 1"
     #define TXT_Value2 "Value 2"
     #define TXT_Value3 "Value 3"
#endif
```

The application makefile: basic.mke

```
#----------------------------------------------
# Define macros specific to this example
#----------------------------------------------
BaseDir = $(MS)/mdl/examples/basic/
privateInc = $(BaseDir)

#----------------------------------------------
# mdl.mki contains default rules for creating .rsc, .mo, etc files
# mdlexmpl.mki contains output directory overrides used by examples
#----------------------------------------------
%include mdl.mki
%include mdlexmpl.mki

#----------------------------------------------------------------------
# Define macros for files included in our link and resource merge
#----------------------------------------------------------------------
basicObjs = \
    $(o)basic.mo \
```

```
34      $(mdlLibs)ditemlib.ml
35  basicRscs = \
36      $(o)basic.mp \
37      $(o)basiccmd.rsc \
38      $(o)basictyp.rsc

39  #--------------------------------------------
40  #    Generate command table include & resource file using rcomp
41  #--------------------------------------------
42  $(genSrc)basiccmd.h: $(BaseDir)basiccmd.r
43  $(o)basiccmd.rsc: $(BaseDir)basiccmd.r

44  #--------------------------------------------
45  #    Create & compile the application's type resource file using
46  #    rsctype and rcomp
47  #--------------------------------------------
48  $(o)basictyp.r : $(BaseDir)basictyp.mt $(BaseDir)basic.h
49  $(o)basictyp.rsc : $(o)basictyp.r $(BaseDir)basic.h

50  #--------------------------------------------
51  #    Compile the MDL source file using mcomp
52  #--------------------------------------------
53  $(o)basic.mo : $(BaseDir)basic.mc $(BaseDir)basic.h

54  #--------------------------------------------
55  #    Link MDL program file from basic.mo & ditemlib.ml using rlink
56  #--------------------------------------------
57  $(o)basic.mp : $(basicObjs)
58      $(msg)
59      > $(o)make.opt
60      $(linkOpts)
61      -a$@
62      $(basicObjs)
63      <
64      $(linkCmd) @$(o)make.opt
65      ~time

66  #--------------------------------------------
67  #    Merge the dialog resources & MDL program file using rlib
68  #--------------------------------------------
69  $(reqdObjs)basic.mi : $(basicRscs)
70      $(msg)
71      > $(o)make.opt
72      -o$@
73      $(basicRscs)
74      <
75      $(rscLibCmd) @$(o)make.opt
76      ~time

77  # complete construction of the .ma by getting the last resource.
```

```
78 %include $(BaseDir)basicrsc.mki
```

The **bmake** utility uses the makefile **basic.mke** to automatically build the **basic.ma** application file from its component parts.

Lines 21 and 22 define some macros to specify directories where various files can be found or placed. In particular, `BaseDir` is set to be the directory where the source files for the **basic** application will be found.

Line 27 includes the file **mdl.mki** (the **.mki** extension indicates that the file is a makefile header file). **mdl.mki** is a handy header file that defines the most useful rules to use when creating a file with one extension from a file of another extension. Most of the dependency lines in **basic.mke** use the rules from **mdl.mki** and do not have to specify any explicit build commands.

Line 28 includes the file **mdlexmpl.mki** (the **.mki** extension indicates that the file is a makefile header file). **mdlexmpl.mki** is a header file that overrides various settings made in **mdl.mki** and provides new rules and macros which are specific to the building of the MDL example applications distruibuted with MicroStation. It is recommended that new applications and/or projects have their own **.mki** file which defines information specific to the building of that application - **mdl.mki** should never be modified by the user.

Lines 32-34 define the macro `basicObjs` as the two files **basic.mo** and **ditemlib.ml**. This macro is used when creating the MDL program file **basic.mp**.

Lines 35-38 define the macro `basicRscs` to be the files **basic.mp**, **basiccmd.rsc** and **basictyp.rsc**. This macro is used when creating the MDL intermediate file **basic.mi**.

Line 42 specifies that the command number header file **basiccmd.h** should be generated from the command table file **basiccmd.r**. A rule from **mdl.mki** indicates that the **rcomp** resource compiler performs this operation.

Line 43 specifies that the resource file **basiccmd.rsc** should be generated from the command table file **basiccmd.r**. A rule from **mdl.mki** indicates that the **rcomp** resource compiler performs this operation.

Line 48 specifies that the resource source file **basictyp.r** should be generated from the type definition file **basictyp.mt**. A rule from **mdl.mki** indicates that the **rsctype** utility performs this operation.

Line 49 specifies that the resource file **basictyp.rsc** should be generated from the resource source file **basictyp.r**, which was itself generated in line 41. A rule from **mdl.mki** indicates that the **rcomp** resource compiler performs this operation.

Line 53 specifies that the MDL object file **basic.mo** should be generated from the MDL source code file **basic.mc**. A rule from **mdl.mki** indicates that the **mcomp** MDL compiler performs this operation.

In lines 57-65, the MDL linker mlink is used to link the MDL object file **basic.mo** with the dialog box manager library file **ditemlib.ml**. The output of this linkage is the MDL program file **basic.mp**. Lines 59-63 create the temporary response file **make.opt** that is then read by the MDL linker on line 64. This avoids exceeding DOS command line length limits. Notice that the `basicObjs` macro is used to specify both the dependency list and the list of files that need to be linked.

In lines 69-76, the separate resource files **basic.mp**, **basiccmd.rsc** and **basictyp.rsc** are combined by the **rlib** resource librarian into the intermediate file **basic.mi**. Lines 71-74 create the temporary response file **make.opt** that is then read by the resource librarian on line 75. This also avoids exceeding the limits on the length of a DOS command line. Notice that the `basicRscs` macro specifies both the dependency list and the list of files that need to be merged.

At this point, all the language generic components of the basic application have been built.

Finally, in line 78, the **basicrsc.mki** file is included (the **.mki** extension indicates that the file is a makefile header file). **basicrsc.mki** is a header file which defines the final steps necessary for building the languaging specific portions of the basic application.

The makefile include file: basicrsc.mki

```
18basicRscs = \
19    $(reqdObjs)basic.mi \
20    $(rscObjects)basic.rsc \
21    $(rscObjects)basicmsg.rsc

22$(rscObjects)basic.rsc : $(BaseDir)basic.r $(langSpec)basictxt.h \
23                        $(privateInc)basic.h

24$(rscObjects)basicmsg.rsc:$(langSpec)basicmsg.r $(privateInc)basic.h
25$(mdlapps)basic.ma : $(basicRscs)
26     $(msg)
27     > $(rscObjects)make.opt
28     -o$@
29     $(basicRscs)
30     <
31     $(rscLibCmd) @$(rscObjects)make.opt
32     ~time
```

Lines 18-21 define the macro `basicRscs` to be the files **basic.mi**, **basic.rsc** and **basicmsg.rsc**. This macro is used when creating the MDL application file **basic.ma**.

Lines 22 and 23 specify that the resource file **basic.rsc** should be generated from the main application resource source file **basic.r**. Notice that this file is dependent upon the **basictxt.h** header file which contains the dialog label definitions for all dialog box related resources. As described earlier, the rule from **mdl.mki** indicates that the rcomp resource compiler performs this operation.

Line 24 specifies that the resource file **basicmsg.rsc** should be generated from the main application resource source file **basicmsg.r**. As before, the rule from **mdl.mki** indicates that the **rcomp** resource compiler performs this operation.

Finally, in lines 25-32, the separate resource files **basic.mi**, **basic.rsc** and **basicmsg.rsc** are combined by the **rlib** resource librarian into the application file **basic.ma**. Lines 27-30 create the temporary response file **make.opt** that is then read by the resource librarian on line 31. This also avoids exceeding the limits on the length of a DOS command line. Notice that the `basicRscs` macro specifies both the dependency list and the list of files that need to be merged.

19 Dialog Box Style Guidelines

The MicroStation dialog box manager implements a user interface style that follows the emerging OSF/Motif user interface standard. When designing dialog boxes, the OSF/Motif Style Guide should be used as the basic reference for style considerations. See "Dialog Box Manager Overview" on page 15-1 for more information on the OSF/Motif Style Guide.

Design Small Dialog Boxes

Try to keep dialog boxes as small as possible. It's better to have multiple small dialog boxes than one huge dialog box that does everything.

Remember that monitors come in various sizes and resolutions. If a dialog box is too large it may not fit on the smaller screens. Monochrome screens are typically 720 pixels wide and only 348 pixels tall. The most common color screens that will be used for MicroStation are 640 pixels wide by 480 pixels tall. The size of dialog boxes is directly related to the size of each screen's default dialog font. MicroStation allows the user to specify the font height to use for each screen, making it difficult to set exact limits on dialog box size. As a general rule, however, do not design dialog boxes that are more than 25 characters tall and 80 characters wide. Try to make them much smaller than this maximum size.

Large dialog boxes obscure more of the windows underneath them, thus making it difficult for the user to interact with their design file views - which are generally underneath all dialog boxes. Non-modal dialog boxes should be *sinkable*, allowing users, if they desire, to *sink* them behind view windows.

If an application will be translated to a different language, remember that the width of some dialog boxes may have to be increased to make room for longer item labels. If the dialog boxes are wide initially, after language translation they may become so large they no longer meet the guidelines stated above.

Other alphabets may be wider than the Roman alphabet. For example, Japanese Kanji characters are typically twice as wide as Roman characters. If a dialog box is designed with the Roman alphabet and then converted to use Kanji, it may double the width.

For these reasons, try to keep dialog boxes small.

Emphasize modeless rather than modal dialog boxes

A major characteristic of a well-designed graphical user interface application is that it is very *user-driven*. The user has a wide number of operations available which can be performed in any order desired. All of the application's possible operations are activated by visible graphical entities. The user has several options on the screen and does not have to remember the command names. Operations that are invalid at a particular time are displayed in a visually distinct manner, usually the graphical entities are drawn *grayed out* or *dim*. The user knows immediately that an option is unavailable and does not have to attempt it to find that it does not work. All of this gives users the feeling that they control the application, not the other way around.

Modal dialog boxes tend to disrupt this image, forcing the user to pause and react to the dialog box before performing any other operations.

Modeless dialog boxes, on the other hand, can remain on the screen while the user interacts with other parts of an application. Modeless dialog boxes can display application status information and also let the user modify application variables when desired - even in the middle of another sequence of operations.

For these reasons, dialog boxes should be modeless by default. A dialog box should be modal only if the situation requires it. Imagine how much less useful the "Element Attributes" or the "Toggles" dialog boxes would be if they were modal. Every time the user needed to change an item in these dialog boxes it would be necessary to bring the dialog box up, make the changes, and then dismiss it. These dialog boxes could not remain on the screen, where they show the current state of MicroStation's locks or the active element attributes.

Try to indicate when a modal dialog box will be opened as a result of choosing an item. Pull-down text menu item labels and push button item labels should end with an ellipsis if picking them opens a modal dialog box.

Limit the number of fonts

Placing too many text fonts in a dialog box can make it look *busy* or cluttered. By default the dialog box manager creates dialog boxes with only two fonts. The normal dialog font is used most of the time, and the bold font is only used for push button item labels and within menu bar items.

Limit the number of colors

Too many colors in a dialog box make it distracting. The dialog box manager only uses white, black, light gray and medium gray when drawing all of the dialog items. Any other colors should be used sparingly. Remember that not all users have color screens.

Use the standard order for push buttons

The OSF/Motif Style Guide recommends that the standard push buttons OK, Apply, Reset, Cancel and Help be laid out horizontally along the bottom of dialog boxes and in the order just listed. It is obviously not necessary to use all of the push buttons in every dialog box - just be sure that for the push buttons that are used, the above order holds.

Match the dialog item type to the intended operation

Toggle button items are used to indicate an on/off choice.

Option button items are used to select a single choice from a small group of options.

List box items are used to select one or more choices from a large list of options.

Push buttons are used to invoke an action.

Tool palettes are used to change the operational state of an application in a relatively major way. For example, MicroStation uses tool palettes to set the current element drawing or viewing operation.

Use MicroStation as an example

Become familiar with MicroStation's dialog boxes before starting the design of a new application. MicroStation has many good examples of dialog box design and the kinds of user interactions that are possible.

Debugging Hook Functions

The Dialog Box Manager Message dialog

The dialog box manager provides the "Dialog Box Manager Messages" dialog box for displaying debugging or other messages. It contains a list box item in which message lines are stored as rows in the list. These messages can be browsed through by scrolling through the list in the usual manner. The "Messages" dialog box is very handy when debugging with only one screen or when running MicroStation in a dual-screen configuration.

Messages are always added to the end of the list and up to 100 text lines can be stored at one time. When the 101st text line is added, the first 50 text lines are deleted and the new text line becomes the 51st row.

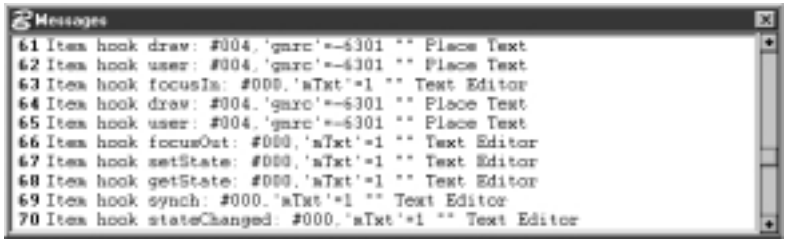
Displaying strings

A message can be displayed in the “Messages” dialog box by calling `mdlDialog_dmsgsPrint(char *messageString)`. If the “Messages” dialog box is not visible, this function will also open it. *messageString* can contain embedded ‘`\n`’ characters that will cause the next list row to be used.

`mdlDialog_dmsgsClear(void)` clears the “Messages” dialog box. It takes no arguments.

Tracking hook function messages

One of the more difficult parts of learning to use the MicroStation dialog box manager is understanding the type, order, and contents of the many messages that can be sent to hook functions. To ease this task, the dialog box manager, upon request, can send information about every message that is sent to dialog hook functions, item hook functions, or item handlers to the Messages window. To get this information about dialog hook function messages or item hook function messages, a hook must be attached to the appropriate dialog or item.



MicroStation’s Message Window.

The following MicroStation key-ins control the display of hook function message information. They each open the Messages window if it has not been already:

Key-in	Description
DMSG ITEMDEBUG [on/off]	Turns on/off the display of all dialog box item messages that have been sent to item hook functions. If an item does not have an item hook function no debugging messages will be displayed for it.
DMSG HANDLERDEBUG [on/off]	Turns on/off the display of all dialog box item messages that have been sent to item handlers. These messages are displayed whether or not items have attached hook functions. Remember, messages that are sent to item handlers do not necessarily get passed on to item hook functions. It is up to the handler to decide if a message should be sent to any attached hook functions. This command is less useful than DMSG ITEMDEBUG.
DMSG DIALOGDEBUG [on/off]	Turns on/off the display of all dialog box messages that have been sent to dialog hook functions. If a dialog box does not have a dialog hook function attached to it, no debugging messages will be displayed for it.
DMSG VERBOSEDEBUG [on/off]	Turns on/off the display of detailed debugging information for all of the above commands.
DMSG ACTION SHOWMNEMONIC	Displays an alphabetized list of a dialog's keyboard mnemonics. Useful for guarding against key conflicts. To obtain info on dialogs other than the Command Window, you must assign this key-in to a function key or access it via the shortcut <Control><Alt><Shift>-M.
DMSG CLEARDEBUG	Clears the "Messages" dialog box.

Closing the "Messages" dialog box will turn off all message debugging. A DMSG xxxDEBUG command must be reentered to continue tracking the dialog box manager messages.

Sometimes it is useful to display dialog box manager message information under program control. The following library functions are available to perform this task:

`mdlDialog_dmsgsSet(boolean dialogMessages, boolean itemHookMessages, boolean itemHandlerMessages, boolean verbose)` controls which types of messages for which information will be displayed. Set the *dialogMessages*, *itemHookMessages* or *itemHandlerMessages* parameter to `TRUE` to display information for those types of

messages. Set *verbose* to TRUE to turn on detailed information for any displayed messages.

mdlDialog_dmsgsPrintItemMsg(DialogItemMessage **dimP*, boolean *filterCommandWindow*, boolean *fromHookFunction*) displays information on the dialog item message *dimP* in the "Messages" dialog box. It can be used to display information on messages that are being sent to a particular item hook function. The last two parameters should always be set to TRUE.

mdlDialog_dmsgsPrintDialogMsg(DialogMessage **dmP*, boolean *filterCommandWindow*) displays information on the dialog message *dmP* in the "Messages" dialog box. It can be used to display information on messages that are being sent to a particular dialog hook function. The last parameter should always be TRUE.

20 Dynamic Link Modules

Dynamic Link Modules (DLMs) are portions of code that can be loaded and linked by MicroStation at runtime. These modules are compiled into native machine code. Dynamic Link Modules are created using the host operating system's tools such as the compiler and linker. The MDL tools are not used to create Dynamic Link Modules.

Overview

Dynamic linking addresses a number of developer concerns:

- Performance of MDL.
- Shared libraries for MDL applications to use. Developers often have multiple applications with common code.
- Access to object-code libraries (with some limitations).
- Ability to use the MDL built-in functions from native machine code.
- Access to C++.
- Access to 32-bit Windows NT DLLs.

Following are definitions of some terms relating to Dynamic Link Modules:

- **Built-ins** - variables and functions that are actually part of MicroStation, but are accessible to MDL applications. They are also available to Dynamic Link Modules.
- **Dynamic Link Specification (DLS)** - used to specify what symbols can be resolved from a specific Dynamic Link Module. A Dynamic Link Specification source file is compiled using the program dlmspec to create a Dynamic Link Specification object file. A Dynamic Link Specification object file is used when linking an MDL application to tell mlink what symbols will be resolved from the Dynamic Link Module.
- **Native code** - code created by using standard compilers and linkers. The keyword nativeCode in a declaration tells the MDL compiler that the function or variable being declared is part of MicroStation or a DLM. The function or variable being declared is not part of the MDL application but is accessible from the MDL application.

- **Pseudo-code instructions** - instructions generated by the MDL compiler and understood by MicroStation's MDL interpreter.
- **Task ID** - the name used for an MDL or MicroCSL application. This name appears in MicroStation input queue elements. It also is used in MicroStation commands that refer to MDL applications. For example, in the command MDL UNLOAD QDIM, QDIM is the task ID. It is also displayed in the MDL Applications dialog box.
- **Resources** - anything a DLM or MicroStation may allocate and release for an MDL application. In this document, resource does not refer to any of the resources that are managed by the MicroStation Resource Manager. Examples of DLM resources are memory and file handles.

Developers should keep the following points in mind when working with Dynamic Link Modules:

- The features of dynamic linking are not guaranteed to be available on all platforms.
- Applications share the stack with MicroStation. On the PC, the applications are limited by the size of MicroStation's stack. This is not a problem on any of the UNIX platforms or under Windows NT.
- On some platforms, there are problems when different Dynamic Link Modules define variables of the same name and the variables are not initialized at compile time. The second Dynamic Link Module may not get a new copy of the variable. Even worse, if the first Dynamic Link Module is removed, then the second may get a fault if it accesses that variable. To avoid this, use as few uninitialized globals as possible. This problem does not occur on the following versions of MicroStation: Clipper, PC DOS and PC Windows NT.

Please refer to the file **devtools.txt**, installed at the root of the **/mdl** directory, for the latest information on compilers and linkers for each platform.

Changes to MDL Source

The MDL compiler generates special instructions for accessing functions and variables that are not part of the MDL program. Therefore, it is necessary to identify for the compiler the symbols that refer to functions and variables that are not in the MDL application. To indicate that these symbols refer to code or data implemented in native code rather than MDL, declare them using a storage class of `nativeCode`. The following declarations tell the MDL compiler that `exampleFunc` is a function that was compiled to machine code, and that `exampleVar` is a variable defined in a native module.

```
nativeCode int exampleFunc(int firstParam);
nativeCode long exampleVar;
```

When the MDL compiler compiles code that uses a function pointer, it must know whether that pointer points to an MDL function or a `nativeCode` function. By default, function pointers in MDL refer to MDL functions. A `typedef` or variable can be designated as a pointer to a native-code function through use of the `pointerToNative` pragma. The following lines would be used to create a `typedef` for pointing to a native code function.

```
typedef int (*FuncP)();
#pragma pointerToNative FuncP
```

The syntax for the `pointerToNative` pragma is:

```
#pragma pointerToNative <object_name> [,] ....
```

`object_name` refers to a variable or `typedef` name. The `pointerToNative` pragma accepts a comma-separated list of object names. Each object name must be a `typedef` name or variable name. The object must have type pointer-to-function.

Access to MicroStation's Built-ins From DLMs

On some platforms, DLM source files must include the file **dloadlib.h** to be able to access built-in functions using the same names MDL programs use. **dloadlib.h** contains define statements that cause the C preprocessor to replace references to MDL built-ins with references to the internal names for the built-in functions and variables.

dloadlib.h is platform specific. On the HP and SPARC, it contains a lot of defines because there are many names to translate. However, on the Clipper workstation, it does not do anything because dynamic linking is implemented in such a way that translation of the names is unnecessary.

If a source file includes **dloadlib.h**, then it should be possible to use that source file unchanged on all platforms both as MDL source and as DLM source. All of the differences should be handled by **dloadlib.h**.

Linking an MDL Program With a DLM

The MDL linker mlink must know how to resolve the symbols for the DLM. It must know what symbols can be resolved at runtime. Since the DLM itself cannot be linked with the MDL program, a Dynamic Link Specification (DLS) object file is used. The DLS object file specifies the name of the dynamic link module, the list of symbols to be resolved from the dynamic link module, and the version number of the library. All this information can be linked with the MDL program so that MicroStation can use it at runtime to dynamically load and link the DLM.

The syntax for Dynamic Link Specification files is described in the "Dynamic Link Specification Source Files" section.

mlink treats DLS object files like libraries. It does not make the DLS object file part of the program unless the DLS object file resolves symbols required by the mdl object files. Since it is treated like a library, the DLS object file must appear in the link step after any MDL object files that use functions or variables from the corresponding DLM.

Runtime Concerns

MicroStation's DLMs are modeled after standard dynamically linked shared libraries. MicroStation does not create a process (does not allocate a process descriptor or MDL descriptor) for a DLM. A DLM is primarily a set of functions that can be called by MDL programs. A DLM is loaded only if it is required by an MDL program. This happens if mlink resolved references for the program from a corresponding DLS object file.

Generally, a DLM executes only when a function in the DLM is called by an MDL program or if MicroStation calls one of the DLM's hook functions.

If a DLM is called by an MDL program, then that MDL program is the current process while the DLM executes. If the DLM calls any functions that allocate resources that must belong to a process, then the resources will belong to the MDL program.

It is difficult to predict what the current process will be when MicroStation calls a DLM hook function. For example, a DLM can have a hook that is called when an MDL program is unloaded. When MicroStation calls this hook, the current process is random. It may be the MDL program affected by what is going on at the time, or it may be MicroStation. If it is important for the DLM to know what MDL program is affected, then a pointer to the MDL descriptor is provided as a parameter.

MicroStation never loads more than one copy of a DLM. If there are multiple requests to load the DLM, then MicroStation keeps track of the number of requests. Each time there is a request to unload the DLM, MicroStation decrements the counter of opens. If the counter goes to 0, the DLM is physically unloaded if possible. On some systems, it is impossible to unload the DLM.

Function Pointers as Parameters to Built-in Functions

DLM function pointers and MDL function pointers cannot be used interchangeably. Using an MDL function pointer where a real function pointer is expected would cause a fault. Also, using a real function pointer where an MDL function pointer is expected would cause a fault.

There are many MDL functions for establishing asynchronous functions. These functions generally have a form similar to `mdlSystem_setFunction (<function_type>, <function_pointer>)`. `<function_pointer>` must be an offset into the current MDL application. It cannot be a pointer into the DLM. Therefore, a DLM can call one of these functions to set up a hook for an MDL application, but it cannot use these to set up a hook for itself.

In MicroStation's **.fdf** files, all function parameters that are MDL function pointers are declared as type `MdlFunctionP`. `MdlFunctionP` is declared in **mdl.h** as:

```
#if defined (mdl)
     typedef int (*MdlFunctionP)();
#else
     typedef unsigned long MdlFunctionP;
#endif
```

If a source file includes the **.fdf** files, then it should be impossible to pass real function pointers where MDL function offsets are expected. Declarations such as the one for `mdlSystem_setFunction` should cause the compiler to catch all invalid uses of these functions.

```
void mdlSystem_setFunction
(
int type,
MdlFunctionP function
);
```

Identifying MDL Applications

It may be important for some DLMs to keep track of certain MDL applications, particularly if the DLM is used by more than one MDL application, and the DLM allocates and frees resources for these applications. The DLM should be able to detect when the application is removed. It should then free any of the application's resources that the application failed to free. MicroStation provides built-in functions to let a DLM track an MDL application. MicroStation can notify a DLM whenever an MDL application is unloaded.

Internally, MicroStation manages data structures called MDL descriptors to track the status of MDL applications. Internally, most of the functions used to manage MDL functions require a pointer to an MDL descriptor as a parameter. So, while the MDL descriptor format is not published, a DLM can call various functions to obtain pointers to important MDL descriptors.

The function `mdlSystem_getCurrMdlDesc` returns a pointer to the MDL descriptor of the MDL application that is active when the function is called. The task ID associated with an MDL descriptor can be determined with the function `mdlSystem_getMdlTaskID(mdlDescP)`. The MDL descriptor for a task can be determined with `mdlSystem_findMdlDesc(taskIdP)` where `taskIdP` points to the application's task ID. This task ID is not case sensitive.

The MDL descriptor is essentially the internal name, and the task ID is the external name. The task ID is also used in MicroStation input queue elements.

A DLM can have an unload hook that is called when an MDL application is unloaded. Using this hook, the DLM can free any resources allocated for the MDL application.

Calling Custom MDL Functions

A DLM may need to call functions in the MDL application. The mechanism described here to accomplish this works regardless of whether or not the native code was called by an MDL function. The mechanism works even if the native code is called by one MDL application and calls a function in another MDL application.

A DLM cannot call MDL functions directly. The native code must use `dlmSystem_callMdlFunction` to call MDL functions. In addition, the native code must know the offset for that function, and the address of the MDL descriptor for the application that contains that function. As usual, 0 can be used to represent a `NULL` MDL function pointer.

MicroStation needs a pointer to the MDL descriptor to determine the start of the code segment so that it can determine the absolute address of the function. When MicroStation calls an MDL function, its saves the current value of `currMdlDesc`, and sets

it to the valued passed into `dlmSystem_callMdlFunction`. When `dlmSystem_callMdlFunction` returns, MicroStation restores `currMdlDesc`.

See the section "DLM Functions" for more information on `dlmSystem_callMdlFunction`.

Determining When an MDL Program is Unloaded

It is often important to know when an MDL application is being unloaded. For example, if a DLM calls an MDL application's user hook after the application has been unloaded, the results will be unpredictable, and a crash may occur. Also, if the DLM is allocating and tracking resources for an MDL application, then it must know to free those resources when the application is unloaded.

The DLM can use an unload hook to learn when an MDL application is unloaded. To install an unload hook, the DLM must call `dlmSystem_setFunction(DLM_SYSTEM_MDL_UNLOAD, <dlmID>, <unloadFunctionP>)`. When an MDL application is unloaded, MicroStation will call the unload hook passing the application's MDL descriptor as a parameter.

Each DLM may have an unload hook, but a given DLM may have only one hook. To remove an unload hook, call `dlmSystem_setFunction(DLM_SYSTEM_MDL_UNLOAD, <dlmID>, NULL)`. See the description of `userHook_mdlUnload` for more information on this user hook.

Application-Specific Resources versus System Resources

There are a number of functions that allocate resources for an MDL program. These resources then belong to the MDL program. MicroStation frees these resources when it unloads the MDL application. Examples are `dlmSystem_mdlMalloc` and `dlmSystem_mdlFopen`. These handle `malloc` and `fopen` calls from MDL programs.

`dlmSystem_mdlFopen` is merely a layer on top of `fopen`. `dlmSystem_mdlFopen` performs an `fopen` and associates the `FILE` pointer with the currently active MDL application, and then returns the same `FILE` pointer as `fopen`. To avoid problems, files opened with `dlmSystem_mdlFopen` must always be closed with `dlmSystem_mdlFclose`. If `fclose` is used to close a file opened with `dlmSystem_mdlFopen`, MicroStation will attempt to close the file itself when it exits, the results of which will be unpredictable, and a crash may occur. If DLM needs to open a file that will not be associated with any MDL application, the DLM should call `fopen` directly.

Memory allocated by `dlmSystem_mdlMalloc`, `dlmSystem_mdlCalloc` or `dlmSystem_mdlRealloc` must be freed by `dlmSystem_mdlFree`. Memory allocated by

`malloc` must be freed by calling `free`. An MDL application cannot free memory that a built-in or DLM function allocated by calling `malloc` directly. Memory allocated by calling `dlmSystem_mdlMalloc` is automatically freed by MicroStation when the MDL application is unloaded.

For more information, see the "Managing Application Resources" section of "A Sample DLM" below.

Dynamic Link Specification Source Files

This section describes the syntax used for Dynamic Link Specification source files. The **dlmspec** utility is used to compile a Dynamic Link Specification source file producing a Dynamic Link Specification object file. The object file is used at link time to specify information on symbols that are to be resolved at runtime. Normal MDL object files (**.mo** files) that need to have symbols resolved from a given Dynamic Link Specification object file must appear in **mlink**'s command line prior to the Dynamic Link Specification file.

The statements in a Dynamic Link Specification file can be categorized as preprocessor directives and commands. Comments are also supported. Comments are bracketed with `/*` and `*/`.

All preprocessor directives contain # at the start of a line. `#include` can be used to include other source files. `#if`, `#ifdef`, `#else`, `#elif` and `#endif` can be used for conditional compilation. `#define` can be used to define values used for conditional compilation. It cannot be used to define macros. The Dynamic Link Specification compiler **dlmspec** defines standard constants that can be used for conditional compilation. The defines that are built in for the appropriate platforms are `msdos`, `pm386`, `IP32`, `clipper`, `unix`, `sparc`, `hp700`, `BIG_ENDIAN`, `EnvironV`, `XWindow`, `winNT`, `win32`, `rs6000` and `sgimips`.

DLM source file commands all begin with `%`. The commands are `%Version`, `%Functions`, `%EndFunctions`, `%Variables`, `%EndVariables`, `%ModuleName` and `%End`.

`%Version` is followed by a version number. This version number is saved in the Dynamic Link Specification object file. It is also saved with the application at link time. At load time, this version number is provided to the DLM's initialization function.

`%Functions` tells **dlmspec** to start processing the subsequent symbols as function names. It is used to specify the list of functions that can be resolved at runtime. `%EndFunctions` is used to signal the end of the list. Any number of pairs of `%Functions` `%EndFunctions` can be used, but they cannot be embedded.

`%Variables` tells **dlmspec** to start processing the subsequent symbols as variable names. It is used to specify the list of variables that can be resolved at runtime.

`%EndVariables` is used to signal the end of the list. Any number of pairs of `%Variables` `%EndVariables` can be used.

`%ModuleName` is followed by the name of a file. For DLMs, this is used at runtime to determine the name of the file containing the DLM. The file name suffix usually is not specified. It is system-specific. It is provided by MicroStation at runtime. The defaults are **.out** (Clipper workstation), **.rex** (PC), **.so** (SPARC), **.dll** (Windows NT) and **.sl** (HP700). No suffix is used on the RS/6000.

The final command in a Dynamic Link Module specification source file must be `%End`.

Additional Include Files

The following files are provided:

- **dloadlib.h** - contains #defines that map the MDL names into the real MicroStation names if the MicroStation names are needed to load the DLM. This file is system-specific. On platforms where the operating system resolves dynamic links, it has a lot of entries.
- **dlmfuncs.h** - contains declarations and definitions used only by DLMs.
- **.fdf** files for all of the MDL built-ins that are not in the standard C library. Most of these built-ins are available to both DLMs and MDL programs. Both MDL source and DLM source can use the **.fdf** files. The files are provided in the **mdl/include** directory along with all of the MDL header files. The file **dlmsys.fdf** declares functions that are available only to DLMs. The other **.fdf** files declare functions that are available to both DLMs and MDL programs.

A Sample DLM

The mdl/examples/dlink directory contains an example that illustrates most of the important concepts needed to implement a DLM. This example implements the UNIX file I/O functions `read`, `write`, `open`, `close`, etc. To compile and link this example, just run bmake -I$MS/mdl/include/fileio. Doing this will build the application including both the MDL program and the DLM. Then load the MDL application testio1, testio2 or both. MicroStation will see that the applications need the DLM and will automatically load it.

The concepts illustrated in the example are:

- Dynamic Link Specification Source file.
- Using two names for a given function - the name used within the DLM and the name used for access from an MDL program.
- Associating resources (file handles) with a specific MDL application.
- Calling custom MDL functions from the DLM.
- DLM hook functions.
- Use of an initialization function.
- Use of the error-reporting function.

Although the makefile is not described in this chapter, it is well-commented.

Dynamic Link Specification Source File

The Dynamic Link Specification source file is fileio.dls. It has sections specifying the version number, module name and function names. The `%Version` command specifies a number that is stored with the Dynamic Link Specification object file. It is also stored with any MDL application created with this object file. When the DLM is loaded into memory, MicroStation calls the DLMs initialization function. The version number is passed as one of the parameters. In this example, the initialization function initialize in filemain.c verifies that the version number is less than `0x500`. If it is `0x500` or greater, then it rejects the load request and displays a message saying that the versions are incompatible.

The `%ModuleName` command specifies the name of the file that contains the DLM. In this example it provides the name fileio to specify that the DLM is in the file fileio.out (Clipper workstation), fileio.rex (PC), fileio.so (SPARC), fileio.dll (Windows NT), fileio (RS/6000) or fileio.sl (HP700).

The `%Functions` section specifies what functions can be accessed from an MDL program that is linked with this DLM. For each function, if only one function name is given then that name is used both within the MDL program and within the DLM. If a specification also contains a name in parentheses, then the name preceding the

parentheses is used by the MDL program and the name in the parentheses is the name used by the DLM. Consider `open (fileio_open)`. When the MDL program calls the function `open`, it is calling the DLMs function `fileio_open`.

This example contains an empty Variables section. This section would be used for listing the variables that can be accessed from the MDL application. The format is the same as for functions.

Managing Application Resources

When MicroStation allocates resources for an MDL task, it associates those resources with that MDL task. If the MDL task is unloaded, then MicroStation must `free` those resources. Since a DLM should appear to be part of MDL, it must behave the same way. When an MDL application is unloaded, a DLM must also release all resources it allocated for that application.

In this example, file handles are allocated by `open`, `create` and `dup`. Therefore, the DLM must intercept all calls to these functions and record what MDL application called the function.

The first step in the procedure is to establish different internal and external names for these functions. This is done in the Dynamic Link Specification file.

The specification for open is `open(fileio_open)`. This specifies that the name "open" can be used in an MDL program. When the MDL program calls `open`, MDL will call the DLM's function `fileio_open`.

The function `fileio_open` must associate the current MDL application and the file handle returned by the real open. To get an identifier for the MDL application, it calls `mdlSystem_getCurrMdlDesc`. Then it finds a free entry in the array `fileList`. In that entry, it records both the file handle and the MDL descriptor.

The DLM also intercepts calls to `close`. When the MDL application calls `close`, the DLM's function `fileio_close` intercepts the call. It clears the data structure where it had established that the handle was associated with this MDL application.

The final step in controlling the file handles is to guarantee that when the MDL application is unloaded, all of its open files will be closed. To do this, the DLM sets up an MDL unload function, `mdlUnloadHook`, to be called whenever an MDL application is unloaded. MicroStation passes a pointer to the MDL descriptor when it calls the unload hooks. The DLM uses this pointer to determine what files are owned by the MDL application that is being unloaded. It closes all of these files.

Emulation of an MDL Asynchronous Function

This application supports a user hook to be called whenever an MDL application opens, creates or duplicates a file.

MDL applications set up the user hook by calling `fileio_setFunction`. This function records the offset of the function and the current MDL descriptor. Whenever an MDL application opens, creates or duplicates a file the DLM calls all of the hooks using the function `dlmSystem_callMdlFunction`. The first two parameters for this function are a pointer to an MDL descriptor and the offset to the MDL function. The remaining parameters are variable. These parameters are passed to the MDL function.

DLM Hook Functions

The DLM `fileio` sets up a hook function to be called whenever an MDL application is unloaded. It does this by calling `dlmSystem_setFunction`. Notice that this function is similar to the `mdlSystem_setFunction` except that the second parameter identifies the DLM. When an MDL application sets a user hook it does not have to identify itself because MicroStation always knows what MDL application is active. However, MicroStation does not have any concept of the active DLM. Therefore, a DLM must identify itself to the `dlmSystem_setFunction` call. When MicroStation calls the DLM's initialization function, MicroStation provides the DLM's identifier as one of the parameters. The DLM uses this identifier as the second argument in calls to `dlmSystem_setFunction`.

Initialization Function; initialize

The DLM fileio contains an initialization function. This function is called immediately after the DLM is loaded. Typically, this is used to verify that the versions are compatible and to set up any hook functions that are required.

The initialization function must always be called `initialize`.

If the initialization function returns anything other than `SUCCESS` then the load fails and the MDL application is unloaded.

Error Reporting Function; dlmSystem_displayError

When a load fails, many error messages are typically required to completely explain what happened. The message fields in the Command Window are insufficient for displaying all the error messages. Therefore, load errors are reported with the function `dlmSystem_displayError`. The format for this is the same as for `printf`, except that the messages should not have a newline. The function `dlmSystem_displayError` appends a newline to every message. The `dlmSystem_displayError` function is also available to DLMs.

C++ DLMS

Dynamic Link Modules containing C++ object code require additional work on certain platforms. This additional work mainly pertains to "static constructors and destructor" routines. When compiling C++ code which contains static objects, the compiler automatically inserts functions to create and destroy these static objects. When the DLM is loaded, it must be ensured that those constructor routines get called. Similarly when the DLM is unloaded, the destructor routines must be invoked. The work required to accomplish these goals is platform specific. Further, this work can also vary between different C++ compiler versions.

The list of platform specific compiler versions for each platform is documented in **devtools.txt** installed in the root of the **/mdl** directory.

Platform Specifics

When this document was written, the example had been tested on a SPARCstation, a Clipper workstation running an EnvironV version of MicroStation, a PC running DOS, a PC running Windows NT, an RS/6000 workstation, an HP workstation, a DEC Alpha running Windows NT, an SGI Indigo and a SPARCstation running Solaris v2.2. This section describes some of the problems that were encountered.

In general, it is dangerous to have a DLM that needs to be linked with any system libraries. When linking with a system library, it is very difficult to link in the desired modules from the library without bringing in extra modules. For example, the link step may bring in a new copy of heap management but may not have any of the heap management data structures initialized.

The example violates this rule to illustrate how to handle some of the difficult problems that arise in linking DLMs.

Since nearly all standard C functions are available as built-in functions, a DLM should not have to use any of the shared libraries. In fact, many DLMs will not be linked with any libraries. All of the references will be resolved at runtime from MicroStation's built-ins. A DLM should be able to have all of the symbols resolved from MicroStation's built-ins at runtime.

Clipper-specific notes

A DLM is a relocatable object file. To create a relocatable object file, use the UNIX utility **ld** to combine object files. Specify `-r` in the command line to specify that the output is a relocatable object file, not an executable file. **ld** must be used to start the link; neither **acc** or **cc** pass the `-r` flag to **ld**.

MicroStation can resolve symbols from a shared library only if MicroStation is linked with that shared library. Currently, only the tools and C shared libraries are supported. In the XWindows version of ustation32, programs will be able to use the XWindows shared library.

In this example, the link is performed in two steps. First **fileresl.out** is created. This resolves references to symbols that are to be resolved from the system libraries rather than from MicroStation. This step resolves as few symbols as possible. Then **fileresl.out** is linked with the other object files to create **fileio.out**. There are many references unresolved in **fileio.out**. These are all resolved at runtime when **fileio.out** is loaded. If the link step only contained one step, and the link step that builds **fileio.out** used the libraries, then these references would be resolved from the libraries rather than from MicroStation.

On the Clipper, it is particularly important that the DLM not link in another copy of `malloc`, `fopen` or `assert`. The DLM must use the copies of these functions that are linked with MicroStation. When a DLM is linked with system libraries, the system libraries typically cause the link to link in the these functions from the standard library. To avoid this, use the files `dlmalloc`, `dlmfopen` and `dlmassert` that are provided with MicroStation. If a DLM contains one of these functions, and it is resolved from a file other than these, then MicroStation displays a message similar to "`malloc cannot be redefined`". This example illustrates how to use **dlmalloc.o** in a link step.

C++ DLM Platform Specifics

On the Intergraph Clipper platform you have to:

- explicitly search the C++ object files for the static constructor and destructor routiness. The static constructor and destructor functions are generated by the compiler. The static constructors have a prefix of '`__sti__`' and the destructors have a prefix of '`__std__`'. Typically a command like:

```
% nm foo.o | grep '__sti__'
% nm foo.o | grep '__std__'
```

will generate a list of the required functions.

- build a list of those function pointers.

```
extern 'C'
{
   extern void __sti__*******();
   extern void __sti__*******();
   ...
   extern void __std__*******();
   ...
};
```

- manually execute them in `initialize`.

```
typedef void(*pfv)();

static pfv static_initializers[]=
{
   __sti__**********(),
   ...
   0
};

static pfv static_initializers[]=
{
   __std__**********(),
   ...
   0
};

extern 'C' void run_static_ctors()
{
   int i;
   pfv *ifnp;

   for (ifnp=static_initializers,i=0; *ifnp; ifnp++,i++)
   {
        (*ifnp)();
   }
};

extern 'C' void run_static_dtors()
{
   pfv *dfnp;

   for (dfnp=static_destructors; *dfnp; dfnp++)
   {
        (*dfnp)();
   }
}
```

Then run them from `initialize`.

```
   static int initialized=0;
   int initialize
   (
        char *filename,
        char *taskid,
        void *dlmId,
        unsigned long initparam
   )
   {
        if (!initialized)
```

```
        {
                ...
                /* Initialize the C++ portion of the DLM */
#if defined (unix) && defined (clipper)
                run_static_ctors();
#endif
                ...
                initialized=1;
        }
    }
```

You will need to setup a DLM unload handler function with the following function call `initialize`.

```
    typedef int (*pfi)();
    dlmSystem_setFunction(DLM_SYSTEM_DLM_UNLOAD, dlmId,(pfi)
                          dlmUnloadHandler);
```

The DLM unload handler function will then call:

```
    static int dlmUnloadHandler()
    {
        /* Finalize the C++ portion of the DLM */
#if defined (unix) && defined (clipper)
        run_static_dtors();
#endif
        return 0;
    }
```

SPARCstation-specific Notes

The DLM is a sharable object file. This is done automatically. There is nothing unusual in the compilation or link steps, except that **.so** is specified as the file suffix for the sharable object the link step creates.

The DLM must provide intermediate functions to call functions that are in the shared library. MDL applications cannot directly call functions from a system shared library. The MDL application must call a function in the DLM that in turn calls the function in the shared library. The runtime load step would fail because MicroStation cannot get the address of symbol from the shared library. The system call used for that would fail and would terminate MicroStation.

On other systems, MDL functions can call library functions directly.

MicroStation never unloads a DLM. There is a bug in the SunOS that causes MicroStation to crash when a DLM is reloaded. To get around this problem,

MicroStation always keeps the DLM loaded. This is most significant to developers who must restart MicroStation to use a new version of a DLM.

Solaris Notes

The link step for a Solaris DLM should contain `-B symbolic`. This tells ld to resolve any symbols that can be resolved from the DLM at link time. The remaining symbols are then resolved when MicroStation loads the DLM.

If `-B symbolic` is not specified, nothing is resolved at link time. When MicroStation loads the DLM, the dynamic loader tries to resolve each symbol from MicroStation, then from the DLM. Therefore, if the DLM has a function that has the same name as a MicroStation function, the DLM uses MicroStation's copy of the function. The DLM's copy of the function is never called. To prevent this from happening, include `-B symbolic` in the link step.

If a link step contains `-B symbolic` then the linker displays warnings about every reference that cannot be resolved at link time. Often this is useless information because the DLM has many references that are to be resolved from MicroStation at load time. If your are using csh, you can redirect standard out by appending the following to the end of the link step:

```
>&! /dev/null
```

If you are using ksh, you can redirect standard out by appending the following to the end of the link step:

```
2> /dev/null
```

C++ DLM Platform Specifics

This platform requires DLMs to be linked with CC, the C++ compiler, not ld, the linker. This ensures the appropriate initializations. The remaining arguments are the same as for C DLMs. No further changes are necessary for this platform.

Debugging Solaris

To debug a DLM under Solaris, first start MicroStation and load the DLM. Then start the debugger using the `-pid` option to make the debugger attach to MicroStation. The following command provides an easy way to do this:

```
dbx - `ps -ae | grep ustation | awk '{print $1'}'`
```

Place this in a shell script or create an alias to make this easy to use. The following command establishes this an an alias for ksh:

```
alias dbxdlm="dbx - \`ps -ae | grep ustation | awk '{print $1}'\`"
```

HP700-Specific Notes

The DLM is a shared library. All of the object files must contain position-independent code. This is achieved by specifying `+z` in the compilation step. Include `-b` in the command line of the load step to specify that the output file is a shared library.

Prior to MicroStation 5.5, DLM's on the HP were always loaded with the `BIND_IMMEDIATE` option. That means that as soon as the DLM is loaded, all references are resolved. That has the advantage that all unresolved references are detected immediately. It also protects the DLM from having symbols resolved from other DLM's that are loaded later. It has the disadvantage that load time may be slower.

Starting with the MicroStation 5.5, MicroStation specifies `BIND_DEFERRED` when loading a DLM. This gives the DLM developer the ability to choose whether the DLMs are loaded with `BIND_IMMEDIATE` or `BIND_DEFERRED`. The operating system loads the shared libraries with `BIND_IMMEDIATE` if the application specifies `BIND_IMMEDIATE`, or if the application is linked with the `-B` immediate option. Although this change should not affect most DLM's, it does give the developer the ability to control how the DLM is loaded.

C++ DLM Platform Specifics

This platform requires DLMs to be linked with **CC**, the C++ compiler, not **ld**, the linker. This ensures the appropriate initializations. The remaining arguments are the same as for C DLMs. No further changes are necessary for this platform.

Debugging on HP

To debug on the HP700 using **xdb**, use the `-l` and `-s` option in the **xdb** command line. The `-s` tells xdb that you want to debug shared libraries. Use the `-l` option to specify the libraries you want to debug. For example, to debug the DLM from the **fileio** example start **xdb** with the command line:

```
xdb ustation32 -s -l $MDLAPPS/fileio.sl
```

From **xdb** enter the command lsl to list the shared libraries. You should see your shared library in the list.

You will need a special version of **ustation32** to debug shared libraries. See the file **mdl/devtools.txt** for information on how to obtain the debugging version of ustation32.

PC Protected Mode Notes

The DLM is a REX file that is created by the Phar Lap linker and then processed by the BSI-supplied utility **linkdlm** to create a DLM-specific symbol table.

A REX file is a standard Phar Lap relocatable executable that must be linked with a number of standard BSI-supplied object files and libraries. Both **linkdlm** and MicroStation's runtime verify that these files were included in the link step.

The object file **ustnfrst.obj** must be the first input file specified in the link step. The library file **dlmfuncs.lib** must be specified after all of the object files. This contains definitions for all of the symbols that can be resolved from MicroStation. **dlmdata.lib** should follow **dlmfuncs.lib** but should appear before the MetaWare libraries.

After creating the REX file, run it through **linkdlm**. The command to invoke **linkdlm** is:

```
"linkdlm -r<rex file> -o<output file>"
```

`<rex file>` specifies a REX file that was created by the Phar Lap linker and `<output file>` specifies the name of the output file. The output will be used as a DLM.

If you find that structure members such as fields in the `Tcb` are not correct, then your DLM is probably not compiled with the correct alignment information. By default, the MetaWare 3.x compilers do not pack structures. For compatibility with MicroStation 4.x, MicroStation needs the strctures packed. Therefore, code that will access MicroStation's structures, such as the `Tcb`, must be compiled with the following pragma:

```
pragma Align_members (1)
```

If your use a makefile similar to the one provided with the **dlink** example and you use the default compilation rule, then this pragma is included in the compilation. The default compilation rule specifies that the compiler must use the profile **extg.pro**, and that **extg.pro** contains that pragma.

If your program has random floating point errors or causes MicroStation to have random floating point errors, then the errors are probably floating point stack overflows. These can occur if a function calls a function that returns a floating point value, but the source file of the calling function does not contain a declaration of the called function. A function that returns a floating point value pushes that value onto the floating point stack. The calling function must contain code to remove that value from the floating point stack, even if it does not use the return value.

Therefore, the compiler must know what functions return floating point values so it can generate the code to clean up the floating point stack. If the source file that contains a call does not contain a declaration of the called function, then the compiler cannot know that the floating point value has to be popped. Consequently, some

values are left on the stack. This eventually causes a floating point exception due to an overflow of the floating point stack.

This is tremendously difficult to debug because the symptom often shows up in a place other than the cause of the problem.

C++ DLMS Platform Specifics

The DLM needs to be linked with an **initcxx.obj** file which is provided in the **$***** directory. The initialize subroutine needs to make the following function call:

```
static int initialized=0;
int initialize
(
     char           *filename,
     char           *taskid,
     void           *dlmId,
     unsigned ling  initparam
)
{
     if (!initialized)
     {
          ...
          /* Initialize the C++ portion of the DLM */
#if defined (msdos)
          init_cxx_globals();
#endif
          ...
          initialized=1;
     }
}
```

Windows NT Notes

DLMs are implemented as runtime Windows NT Dynamic Link Libraries. Application DLMs are DLLs. To make MicroStation built-in functions visible to DLMs, MicroStation itself consists of a small executable (**ustation.exe**) and a large DLL (**ustation.dll**).

The **dlmspec** program, which processes a Dynamic Link Specification (**.dls**) file, has been modified to create a Windows Module Definition automatically if desired. To take advantage of this feature, use the command line switch `-w<module def file name>`. If you use **bmake** and include **mdl.mki**, you can define a **bmake** variable called `moduleDef` and the command line argument to **dlmspec** is automatically generated. The Module definition file is processed by the Windows NT **lib** program to produce both an import library **.lib** file and an exports specification **.exp** file. The **.exp** file is linked with the DLL and defines the functions and variables that are visible from the

DLL. The **.lib** file is linked with executables that use the DLL as a "load time" DLL. Usually the **.lib** file is not used when creating a DLM, but the **.exp** file is.

Application DLLs must link with the MicroStation built in import library if they use any built in functions or variables. This import library is in the **mde\mdl\library** directory, and is called **mdlbltin.lib**.

Every Windows NT DLL must have an entry point defined as part of the link step. This is usually called `LibMain`. This function is called when the DLL is loaded or unloaded, and can perform initialization or unload chores at those times. Minimally, it must return a value of 1 (the Windows convention for successful completion) to indicate successful initiation.

On the link step, Window NT DLLs are told where to load. Code sharing between different invocations of executables that use DLLs is only possible if they can load the code at the same address. If they can't, the DLLs still work, but there are multiple copies of the code. Most MicroStation DLLs are not large, and thus code sharing is much less important that it would otherwise be. Thus the strategy for picking a starting address in the DLM example is to pick the same load address as the MicroStation DLL uses (`0x1c000000`). This guarantees that the suggested load address will be unavailable, and the system will pick another available address.

The example includes a source file called **dllentry.c** that implements a minimal LibMain. The **dllentry.c** file can be used in other DLMs if desired.

If you take advantage of the automatic creation of the Windows module definition file mentioned above, you will have to make sure there is an `initialize` function in your DLM, since the **.def** file reated by **dlmspec** will expect one. A minimal `initialize` function should simply return 0 (the MicroStation convention for success), indicating successful initialization. On other platforms, or if you generate your own module definition file, the `initialize` function is not required, but it never hurts to have one either.

Unfortunately, Windows NT DLLs cannot export data items, only pointers to data. This complicates a DLL's use of MicroStation built-in variables. An include file **mde\mdl\include\dloadlib.h** is provided that minimizes the effort required to make this adjustment and maintain source-code compatibility across platforms. This include file contains definitions of the following type:

```
#define mgds_modes (*mgds_modesP)
#define tcb (*tcbP)
```

and the **mdlbltin.lib** import library resolves `mgds_modesP`. Thus expressions such as:

```
if (mgds_modes.three_d)
```

and

```
tcb->actangle = PI;
```

are changed by the preprocessor to:

```
if ((*mgds_modesP).three_d)
```

and

```
(*tcbP)->actangle = PI;
```

which are acceptable and correct expressions.

C++ DLM Platform Specifics

Nothing special needs to be done on this platform for C++ DLMs.

Silicon Graphics Notes

The DLM is a sharable object file. To make a sharable object, specify "-shared" in the link step. The link step must also contain `-B symbolic`.

The following text is directly from SGI's ld documentation:

When `-B symbolic` is specified, ld inserts the element `DT_SYMBOLIC` into the ".dynamic" section of a shared object library and this alters the dynamic linker's symbol resolution algorithm for references within the library. Instead of starting a symbol search with the executable file, the dynamic linker starts from the shared object itself. If the shared object fails to supply the referenced symbol, the dynamic linker then searches the executable file and other shared objects as well.

Consider what this means for a DLM. If both the DLM and MicroStation contain a function named `convertFormat` and the DLM was linked without `-B symbolic`, then the DLM uses MicroStation's `convertFormat`. That occurs because the dynamic linker first searches for symbols in the executable file, ustation32. However, if `-B symbolic` is specified, then the DLM uses its own version of `convertFormat` because the dynamic linker starts the search for symbols with the shared object itself.

C++ DLM Platform Specifics

This platform requires DLMs to be linked with CC, the C++ compiler, not ld, the linker. This ensures the appropriate initializations. The remaining arguments are the same as for C DLMs. No further changes are necessary for this platform.

Debugging SGI DLMs

To debug a DLM on the SGI Unix platform, just start up the debugger after the DLM is loaded. You can start **dbx** and have it attach to MicroStation by using the command:

dbx -P ustation32

IBM RS/6000 Notes

On the RS/6000, MicroStation Dynamic Link Modules are implemented as shared objects. A link step for an RS/6000 DLM must specify a symbol file, an entry point and a binder-control import list. All three of these are described in the following paragraphs.

The link step specifies the binder-control import list with the command line option:

-bI:$(mdlLibs)ustnexport.lst

This file specifies a list of functions and variables available from MicroStation. The **ustnexport.lst** distributed with MicroStation has `#!` as the first line. This marks the file as an import list, but does not specify what file to use to resolve the symbols. Because this does not specify a file, the DLM will not contain any information on what file to use to resolve the symbols. When MicroStation loads the DLM, the loader resolves all the symbols from MicroStation whenever possible, but it does not report unresolved symbols. To make the loader report unresolved symbols, change the first line to completely specify the name of the MicroStation executable. For example, you may change it to:

```
#! /usr/ingr/mstation/ustation32
```

The linker then stores information in the DLM indicating that all of the symbols must be loaded from **/usr/ingr/mstation/ustation32**. Now when the loader loads the DLM, it will report errors for each symbol that cannot be resolved from MicroStation. Although this technique is useful when checking for unresolved symbols, it does require that the MicroStation name and location not be changed once the DLM is linked (i.e., created by **ld**.) Therefore, you should use this technique only for debugging and should remove the explicit path name from the import list prior to creating a usable version of your DLM.

The link step specifies the entry point using the `-e` command line option. If the `-e` command line option is not specified, then the linker assumes that the entry point is `main`. That will cause the linker to display an error message saying that main is unresolved.

If the makefile uses the standard approach for creating the symbol file, then the link step should contain the command line option `-edlmEntryFunction` to specify the entry point. The entry point function is in the symbol file created by **dlmspec**. Immediately

after loading the DLM, MicroStation calls the entry point function specifying the DLM id as the only parameter. MicroStation calls the entry point function even before it calls the `initialize` function.

The symbol file provides information that MicroStation uses to determine the addresses of functions and variables in the DLM. If a makefile uses the default rule for compiling a **.dls** file, and the **bmake** variable `moduleDef` is defined, then **dlmspec** creates the symbol file. The **fileio** example uses **fileisym.c** as the symbol file. Following is part of the symbol file:

```
#include <basedefs.h>
#include <dlmfuncs.h>

extern int initialize();
extern int fileio_open();
extern int fileio_creat();
extern int fileio_dup();
extern int fileio_close();
extern int fileio_setFunction();
extern int fileio_read();
extern int fileio_write();
extern int fileio_chmod();
extern int fileio_lseek();

Private DlmFunctionAddress dlmFunctionAddresses[]=
{
   {10, "initialize", initialize},
   {11, "fileio_open", fileio_open},
   {12, "fileio_creat", fileio_creat},
   {10, "fileio_dup", fileio_dup},
   {12, "fileio_close", fileio_close},
   {18, "fileio_setFunction", fileio_setFunction},
   {11, "fileio_read", fileio_read},
   {12, "fileio_write", fileio_write},
   {12, "fileio_chmod", fileio_chmod},
   {12, "fileio_lseek", fileio_lseek},
   NULL
};

Private DlmDataAddressdlmVariableAddresses []=
{
   NULL
};

int dlmEntryFunction
(
void    *dlmP
)
{
```

```
    dlmSystem_publishSymbolNames(dlmP, 1, dlmFunctionAddresses,
                                 dlmVariableAddresses);
    return 0;
}
```

This file is created entirely by **dlmspec**, although you may create a similar file manually. If you do not need to create this manually, this you can ignore the rest of this description. You should not need to create this manually unless you are using multiple **.dls** files.

C++ DLM Platform Specifics

The DLM must be linked with **makeC++SharedLib**, which is delivered with **xlC**, the IBM C++ compiler. The file that contains the initialize code must also include the following code:

```
#if defined (rs6000)
   typedef void (*PFV)();
   typedef struct PFVSTR
   {
        PFV  init, term;
   } PFVSTR;
   extern PFVSTR __cdtors[];
#endif

   ...

   int i;
   PFV tmpfunc;
   for (tmpfunc=__cdtors[1].init, i=1; tmpfunc&&tmpfunc!=(PFV)-1;
        tmpfunc=__cdtors[++i].init)
   {
        tmpfunc();
   }
```

You will need to setup a DLM unload handler function with the following function call in initialize.

```
typedef int (*pfi)()
dlmSystem_setFunction(DLM_SYSTEM_DLM_UNLOAD, dlmId, (pfi)
                  dlmUnloadHandler);
```

The DLM Unload Handler function will then call:

```
int i;
PFV tmpfunc;
```

```
for (tmpfunc=__cdtors[1].term, i=1; tmpfunc&&tmpfunc!=(PFV)-1;
     tmpfunc=__cdtors[++i].term)
{
   tmpfunc();
}
```

dlmSystem_publishSymbolNames (RS/6000 only)

The function `dlmSystem_publishSymbolNames` is a MicroStation function that is available only on the RS/6000. The declaration is:

```
void dlmSystem_publishSymbolNames
(
void                *dlmID,
int                 infoFormat,
DlmFunctionAddress  functionAddrs[],
DlmDataAddress      dataAddrs[]
);
```

dlmID is the DLM id that uniquely identifies the DLM. It is provided as a parameter to the entry function. It is the same value that is passed into the `initialize` function.

infoFormat specifies the format of the information provided in the call to `dlmSystem_publishSymbolNames`. Currently, 1 is the only valid value.

functionAddrs and *dataAddrs* provide information MicroStation needs to find all of the functions and variables to be resolved from the DLM.

Debugging on RS/6000

On the RS/6000, it is possible to debug DLMs using either **xde** or **dbx**. First start MicroStation and load the DLM. Then start the debugger specifying the `-a` command line option and MicroStation's process id to attach to MicroStation. After that, you can debug as normal. The following command provides an easy way to start the debugger and attach to MicroStation:

```
dbx -a `ps -ae | grep ustation32 | awk '{print $1}'`
```

Place this in a shell script or create an alias to make this easy to use. The following command establishes this an alias for the ksh:

```
alias dbxdlm="dbx -a \`ps -ae | grep ustation32 | awk '{print \$1}'\`"
```

Calling Across DLMs

This section describes how DLMs can cooperate with 1 DLM calling into another. There are really 2 issues to address:

- all DLMs are in the same address space. Therefore, one DLM can access functions and variables in other DLM's via pointers.
- DLMs can symbolically access symbols in other DLMs on some but not all platforms. A DLM cannot have symbols resolved to another DLM on the DOS, Clipper and Macintosh versions of MicroStation. It can have the symbols resolved on other platforms.

The techniques to have the symbols resolved vary from platform to platform. Generally, the client DLM must be linked with the server DLM or with some sort of symbol definition file for the server DLM.

For the Windows platforms, the list of symbols in the module definition file (**.def**) of the server DLM must contain all of the symbols that will be available to other DLMs. This Module Definition File is used to generate an import library (**.lib**) file that is linked with client DLMs. The **.lib** file must appear in the link step after the object files and libraries that reference symbols it resolves.

On the Unix platforms other than the Clipper, it is possible to resolve symbols from a server DLM by actually specifying the server DLM in the client's link step. On most of the Unix platforms, if the link step specifies a path name and a file name instead of just a file name for the server DLM, then at runtime the library must be in the location specified by the absolute path name. To resolve this problem, just specify the file name. However, doing so causes a new problem. You need to be concerned about how the operating system will find the server DLM. One clean technique is to force MicroStation to load the server DLM prior to loading the client DLM. Then when the operating system needs the server DLM for the client DLM, it will find that it already has it loaded. The operating system will not have to search for the library. To guarantee that the server library is always preloaded, any MDL application that has symbols resolved from the client DLM must also have symbols resolved from the server DLM, and the server DLM must appear in the MDL application's link step prior to the client DLM. That will cause MicroStation to load the server DLM first using either the `MS_MDL` environment variable or the host application's path to find the DLM.

Debugging DLMs

If your specific platform is not discussed in this section, review the general information below and then refer to the platform-specific section for debugging details.

To debug a DLM on a Clipper workstation, just start MicroStation under dbg. After the DLM has been loaded, press <Ctrl-C> to get into the debugger. Then you can set breakpoints in the DLM. To invoke the debugger immediately after the DLM is loaded but before the `initialize` function is called, set a breakpoint on the function `dlink_debugHook`. This function is called immediately after a DLM is loaded.

MicroStation DOS PC DLMs can be debugged using the 386|SRCBug debugger from Phar Lap Software. Because the executable code for these modules is loaded after MicroStation starts, the debugger symbol table must besynchronized with the actual location of the executable code when the code isloaded. Once the symbol tables are synchronized, 386|SRCBug will allow source level debugging of the DLM.

➤ **To debug a DOS DLM, use the following procedure**

1. Link the program with symbols and start MicroStation with the debugger using the following command line: SB386 -SYMFILE <PATH> MGDS.EXE
   `<path>` specifies the .rex file that was used to create the DLM.
2. Set the MDL global variable `debugLevel` to 5 to cause the debugger to break with an `int 3` instruction when the DLM is loaded. This can be done from the MicroStation command prompt with the set debug 5 command. When the breakpoint occurs, the debugger will display a message like:
   ```
   Breakpoint at ssss:oooooooo ...
   ```

Synchnronize the symbols with the debugger command: XO EAX

`eax` is the value stored in the `eax` register which is used here to avoid having to re-type the value in the `eax` register.

Step past the `int 3` instruction with the debugger command: R EIP EIP+1

Breakpoints can now be set at symbols within the DLM and execution can be resumed.

You may want to set `MS_TRAP=none`. If `MS_TRAP` is not defined, MicroStation tries too catch all faults and recover or at least report the source. If your DLM casues a fault, MicroStation intercepts it and reports what line in the MDL program caused the DLM to be called. If `MS_TRAP` is set to `none`, the fault is passed to the debugger. You may define `MS_TRAP` by:

- Defining it as a DOS environment variable using the DOS command set MS_TRAP=none.
- Setting it in a MicroStation configuration file using the statement MS_TRAP=none.
- Modifying the user configuration by selecting the Modify User Configuration menu selected from the Workspace option from the User pulldown menu. You must restart MicroStation before it takes effect.

`MS_TRAP` is only needed under DOS. Under the other operating systems, the debugger intercepts the fault and MicroStation does not know about it. DOS is the only platform where MicroStation processes the fault before the debugger does.

DLM Functions

This section documents the following built-in functions:

Function	Used to
`dlmSystem_getDLMPath`	Get the name of the file that contains the DLM.
`dlmSystem_mdlMalloc`	`malloc` and associate the memory with the current MDL app.
`dlmSystem_mdlCalloc`	`calloc` and associate the memory with the current MDL app.
`dlmSystem_mdlRealloc`	`realloc` the memory with the current MDL app.
`dlmSystem_mdlFree`	Free memory associated with with the current MDL app.
`dlmSystem_mdlFopen`	`fopen` and associate the file pointer with the current MDL app.
`dlmSystem_mdlFclose`	`fclose` and associate the file pointer with the current MDL app.
`dlmSystem_mdlTmpfile`	tmpfile and associate the file pointer with the current MDL app.
`dlmSystem_mdlFreopen`	`freopen` and associate the file pointer with the current MDL app.
`dlmSystem_callMdlFunction`	Call an MDL function.
`dlmSystem_setFunction`	Designate a DLM user hook.

Function	Used to
initialize	Called by MicroStation as the final step of loading an MDL application that requires DLM.
userHook_mdlUnload	Called by MicroStation whenever an MDL application is unloaded.
userHook_dlmUnload	Called by MicroStation prior to unloading the DLM that designated that hook.

dlmSystem_getDLMPath

```
#include <dlmsys.fdf>

char *dlmSystem_getDLMPath
(
void    *dlmID
);
```

Description The dlmSystem_getDLMPath function returns a pointer to a string that contains the name file that contains the DLM.

The *dlmID* parameter is the DLM identifier provided on the call to the initialize user function.

Returns dlmSystem_getDLMPath returns a pointer to a string.

DLM Memory Functions

dlmSystem_mdlMalloc, dlmSystem_mdlCalloc, dlmSystem_mdlRealloc and dlmSystem_mdlFree have the same parameters and return values as malloc, calloc, realloc and free. These functions associate the memory with the current MDL application. When that application is unloaded, the memory is automatically freed. To allocate and free memory without associating it with the current MDL application, use malloc, calloc, realloc and free.

DLM File Functions

dlmSystem_mdlFopen, dlmSystem_mdlFclose, dlmSystem_mdlTmpfile and dlmSystem_mdlFreopen have the same parameters and return values as fopen, fclose, tmpfile and freopen. Except for dlmSystem_mdlFclose, which disassociates the pointer, these functions associate the file pointer with the current MDL application. When the application is unloaded, the file is automatically closed. To manipulate files without associating them with the MDL application, use the standard functions fopen, fclose, tmpfile and freopen.

dlmSystem_callMdlFunction

```
#include "dlmsys.fdf"

long dlmSystem_callMdlFunction
(
void              *mdlDescP,
MdlFunctionP      functionOffset,
...               /* Parameters for the MDL function */
);
```

Description The `dlmSystem_callMdlFunction` function is used to call an MDL function. *mdlDescP* points to the MDL descriptor for the function to be called.

functionOffset is an offset to the function to be called. This is an MDL function pointer.

The variable argument list represents the arguments to be passed to the MDL function. Up to 15 arguments may be passed. MicroStation automatically determines the number and types of arguments expected by the function. (It does this by examining the function's header. This header is actually part of the MDL program loaded by MDL. It is created automatically by the MDL compiler.) Using this information, MicroStation moves the arguments from MicroStation's stack to the MDL application's stack and dispatches the function.

Returns `dlmSystem_callMdlFunction` returns the value returned by the MDL application. It only returns integer values. If `dlmSystem_callMdlFunction` is not able to call the function, then it returns 0. This only happens if the offset is 0, or the pointer to the MDL descriptor is `NULL`.

dlmSystem_setFunction

```
#include "dlmsys.fdf"

int dlmSystem_setFunction
(
int     functionType,
void    *dlmID,
int     (*funcP)()
);
```

Description The `dlmSystem_setFunction` function is used to designate a DLM user hook. The *functionType* parameter specifies the type of hook. Valid values are `DLM_SYSTEM_MDL_UNLOAD` and `DLM_SYSTEM_DLM_UNLOAD`.

dlmID identifies the DLM setting up the hook. This value was supplied by MicroStation in the call to the DLMs initialization function.

funcP specifies the function to be called when the designated event occurs. A value of `NULL` cancels the outstanding hook.

Returns `dlmSystem_setFunction` returns `SUCCESS` if it was able to set up the hook. Otherwise, it returns `ERROR`.

initialize

```
int initialize
(
char         *fileNameP,
char         *taskIdP,
void         *dlmID,
unsigned long initParameter
);
```

Description MicroStation calls `initialize` as the final step of loading an MDL application that requires DLM. If the DLM is required by more than one MDL program, then the DLM is only loaded once but the initialize function is called for each MDL program.

The name "initialize" must be used for the initialization function. MicroStation uses the DLMs symbol table to find the address of this function.

fileNameP points to the name of the file that contains the MDL application that is being loaded. The DLM is being loaded to resolve symbols for the MDL application. Use the function `dlmSystem_getDLMPath` to get the name of the file containing the DLM.

taskIdP points to the task ID of the MDL program being loaded.

dlmID is a pointer that uniquely identifies the DLM. The DLM should use this value if it calls any functions that take a DLM ID as a parameter. *initParameter* is the value specified with the `%Version` command in the Dynamic Link Specification source file.

If the `initialize` function aborts the load by returning a non-zero value, it should aso display a meaningful error message. MicroStation does not display an error message because it could not display anything more meaningful than "initialize failed."

Returns `initialize` may return 0 to allow the load to continue. If `initialize` returns a non-zero value then the load of the MDL program is aborted. If no other MDL programs have the DLM loaded, then the DLM is also unloaded.

userHook_mdlUnload, userHook_dlmUnload

```
void userHook_mdlUnload
(
void              *mdlDescP,
unsigned char     *taskIdP,
unsigned char     *fileNameP
);
void userHook_dlmUnload(void);
```

Description MicroStation calls the `userHook_mdlUnload` user hook functions whenever an MDL application is unloaded. Both the *mdlDescP* and the *taskIdP* identify the MDL application being unloaded. Either parameter is enough to uniquely identify the application. Both parameters are supplied just as a matter of convenience. *mdlDescP* may or may not be the same as the current MDL descriptor. Never depend on the value of the current MDL descriptor when this unload hook is called.

The DLM does not need to use the names `userHook_mdlUnload` and `userHook_dlmUnload`. These hook functions are designated to MicroStation by address only using `dlmSystem_setFunction`.

MicroStation calls the `userHook_dlmUnload` function prior to unloading the DLM that designated that hook.

Returns Both of these hook functions are of type `void`. MicroStation ignores the return values of both of these functions.

See Also `dlmSystem_setFunction`.

21 MDL Shared Libraries

Do you have an MDL function library which you would like other applications to use without the overhead of using statically linked libraries and the memory they require? Or maybe you have an existing DLM which could be converted to an MDL application for use in MicroStation PowerDraft and really need a non-native shared library. MDL Shared Libraries can provide the capabilities you need without a lot of extra tools and overhead.

Difference

An MDL Shared Library is an MDL application that can be dynamically linked with other MDL applications at runtime. The MDL shared library has symbols (functions and variables) which are exported and can be used by other MDL applications in much the same way that MDL applications can use symbols exported by Dynamic Link Modules (DLM).

The ability to use MDL shared libraries is available to a limited extent in MicroStation 5.0 and PowerDraft and completely in MicroStation.

Overview

MDL shared libraries provide a mechanism that faciliates the sharing of functions and variables between different MDL applications in situations (i.e., while running under MicroStation PowerDraft) where it is not possible to use DLMs.

This discussion references portions of a new MDL example called MDLSHARE which demonstrates the use of MDL shared libraries. The source code for this example is available on the Bentley BBS and FTP node and is also delivered with MicroStation as an MDL example.

The following terms and phrases are referenced throughout this section:

System Resource - generic resources such as memory, file handles and symbol sets.

Resource Manager - resources allocated by the Resource Manager.

MDL task - the set of system resources corresponding to a loaded MDL application. Every task has an MDL descriptor used for tracking system resources that belong to the

application. Every MDL application also has a task ID. The task ID is an external name for the application.

Current MDL task - while MicroStation is executing there is always a current MDL task. By default, all system resources allocated are associated with the current application.

Client task - an MDL task which requires an MDL shared library.

Library task - the task associated with an MDL shared library. An MDL shared library is also an MDL application and therefore has a task associated with it.

MDL context switch - occurs when MicroStation dispatches an MDL task. This process includes selecting an MDL stack, setting the current MDL descriptor, and calling the MDL interpreter to start executing the MDL task.

Switched and **non-switched function call** - a call to an MDL function is a switched call if MicroStation's context switching logic is involved in making the call. A call to an MDL function is non-switched if MicroStation's context switching logic is not involved in making the call. When an MDL application calls a function using a normal symbolic reference or using a function pointer, the call is a non-switched. When an MDL application calls a function using `mdlDialog_callFunction`, the call is switched.

An MDL shared library is created by the MDL linker and is a standard MDL application that also has a set of symbols that can be accessed by other MDL applications. When an MDL shared library is loaded, an MDL task is created that corresponds to the application. An MDL shared library can be loaded as an MDL application even when it is not being used as a library.

An MDL application that uses a shared library can symbolically reference functions and variables in the shared library, however, the shared library cannot reference any symbols from the application that loads it. A client application can call a shared library function via a symbolic reference, via a function pointer using normal function pointer syntax, or via `mdlDialog_callFunction`. For both the symbolic reference and the normal function pointer syntax, the call behaves as if the library was statically linked with the application. MicroStation does not perform an MDL context switch. If the shared library does anything that causes MicroStation to allocate a system resource, then that system resource belongs to the MDL task that called the library. It does not belong to the library's task. If the call is made via `mdlDialog_callFunction`, then an MDL task switch does occur. Any system resources allocated by MicroStation belong to the MDL shared library's task, not the task that called it.

Creating an MDL Shared Library

An MDL shared library is just an MDL application with symbols exported. A Dynamic Link Specification (DLS) object file is included in the link step to specify the symbols that are to be exported. Use the `-e` option to specify that the DLS object file specifies symbols that are being exported. A DLS object file specified without `-e` is treated as an object file. It specifies either an MDL shared library or DLM that is to be used by the application being linked.

Linking an MDL Program with an MDL Shared Library

The MDL linker **mlink** must know how to resolve the symbols for the shared library. It must know what symbols can be resolved at runtime. Since the shared library itself is not linked with the MDL program, a DLS object file is used. The DLS object file specifies the name of the dynamic link module, the list of symbols to be resolved from the dynamic link module, and the version number of the library. All this information can be linked with the MDL program so that MicroStation can use it at runtime to dynamically load and link the shared library.

The syntax for Dynamic Link Specification files is described in the *Dynamic Link Specification Source Files* section.

mlink treats DLS object files like libraries. It does not make the DLS object file part of the program unless the DLS object file resolves symbols required by the MDL object files. Since it is treated like a library, the DLS object file must appear in the link step after any MDL object files that use functions or variables from the corresponding shared library.

The Dynamic Link Specification object file must be specified for every link step that may resolve symbols from that file. To simplify this, **mlink** uses the environment variable `MLINK_STDLIB`. `MLINK_STDLIB` specifies a list of object files to be included in a link step. The entries in the list are blank-separated. After processing the command line, **mlink** gets the value of `MLINK_STDLIB`. If the environment variable `MLINK_STDLIB` is defined, **mlink** interprets it as a list of files. This list is appended to the list of input files specified in the command line.

Sample 1 below shows portions of the MDLSHARE application makefile illustrate the MDL shared library build steps:

```
...
baseDir = $(_MakeFilePath)
privateInc = $(baseDir)
%include mdl.mki
%include mdlexmpl.mki
appName = mdlshare
```

```
sAppName = mdlsh
appName1 = mdltest1
sAppName1 = mdlt1
appName2 = mdltest2
sAppName2 = mdlt2
mdlshareObjs = $(o)$(appName).mo $(mdlLibs)ditemlib.ml
mdlshareRscs = $(o)$(appName).mp $(o)$(sAppName)cmd.rsc
mdltest1Objs = $(o)$(appName1).mo $(o)$(appName).dlo
mdltest1Rscs = $(o)$(appName1).mp $(o)$(sAppName1)cmd.rsc
mdltest2Objs = $(o)$(appName2).mo $(o)$(appName).dlo \
            $(mdlLibs)ditemlib.ml
mdltest2Rscs = $(o)$(appName2).mp $(o)$(sAppName2)cmd.rsc

#--------------------------------------------------------------------
# Rule to generate a dlo from the dls source file
# -------------------------------------------------------------------
#  The default rule is being overriden to allow the definition of a
#  preprocessor symbol to control the exporting of runtime/symbolic
#  names via the $(dlsOpts) make symbol
# -------------------------------------------------------------------
.dls.dlo:
   $(msg)
   > $(o)temp.cmd
   -o$@
   $(dlsOpts)
%if defined (winNT) || defined (rs6000) || defined (macintosh)
   ||defined (os2)
   -w$(moduleDef)
%endif
   $%$*.dls
   <
   $(dlmspecCmd) @$(o)temp.cmd
   ~time

...


#--------------------------------------------------------------------
# Compile the Dynamic Load Specification
#--------------------------------------------------------------------
$(o)$(appName).dlo: $(baseDir)$(appName).dls

...


#--------------------------------------------------------------------
# Link the MDL Applications
```

```
#--------------------------------------------------------------------------
#  Link the shared library making sure to mark the exported
#  files using the -e option
#--------------------------------------------------------------------------
$(o)$(appName).mp : $(mdlshareObjs) $(o)$(appName).dlo
   $(msg)
   > $(o)make.opt
   $(linkOpts)
   -e$(o)$(appName).dlo
   -a$@
   $(mdlshareObjs)
   <
   $(linkCmd) @$(o)make.opt
   ~time
#--------------------------------------------------------------------------
# Link the client MDL Applications including the shared library DLO file
#--------------------------------------------------------------------------
$(o)$(appName1).mp : $(mdlTest1Objs)
   $(msg)
   > $(o)make.opt
   $(linkOpts)
   $(mdlTest1Objs)
   -a$@
   <
   $(linkCmd) @$(o)make.opt
   ~time
$(o)$(appName2).mp : $(mdlTest2Objs)
   $(msg)
   > $(o)make.opt
   $(linkOpts)
   $(mdlTest2Objs)
   -a$@
   <
   $(linkCmd) @$(o)make.opt
   ~time

...
```

Sample 1. Portions of MDLSHARE application makefile.

Loading an MDL Shared Library

An MDL shared library is loaded automatically if it is required by a loaded application. Since an MDL shared library is like any other MDL program, it can also be loaded in any manner other MDL applications may be loaded.

When MDL loads an application, the final step is to load any DLMs and MDL shared libraries required by that application. This is a recursive process. Since an MDL shared library is an MDL application, the MDL shared library may require other MDL shared libraries. However, cyclical dependencies are not allowed. If MDL application **app1** requires MDL application **app2** as a shared library, then **app2** can not also require **app1** as a shared library. MDL checks for this when loading the applications. If it detects a cyclical redundancy, it aborts the load.

If an MDL application requires a shared library that is already loaded, then MicroStation does not load the shared library again. MicroStation establishes a link between the application being loaded and the previously loaded copy of the library.

After all of the required libraries are loaded for an application, MicroStation calls the `initialize` function of each shared library used by the client application. (In the current versions PowerDraft and MicroStation, MicroStation does not call the `initialize` function. Client MDL applications must call the shared library's `initialize` function directly and the `initialize` function must be exported in the shared library's DLS. If any of the `initialize` functions return anything other than `SUCCESS`, MicroStation aborts the load process. For each of the libraries, MicroStation calls the `initialize` function regardless of whether the library was already loaded. That is, it calls the `initialize` function each time there is a new attachment to the library, not just when the library is loaded for the first time. To ensure that the shared library's `initialize` function is called for each client task as it is loaded, the `initialize` function must be specified in the %Functions section of the shared library Dynamic Link Specification file (see *Defining the Shared Functions and Variables* below).

Sample 2 below contains the **MDLSHARE** shared library `initialize` function.

```
/*----------------------------------------------------------------------+
 * name initialize
 * author BSI          5/95
 +----------------------------------------------------------------------*/
Public int initialize
(
char       *fileNameP,      /* <= Client MDL Application */
char       *taskIdP,        /* <= Client MDL Task ID */
void       *libDescP,       /* <= Shared Library Descriptor pointer */
unsigned long initParameter /* <= Initialization parameters */
)
{
  int rscFileH;
```

```
char outName[MAXFILELENGTH];
TaskInfo *taskInfoP, *taskP;
/* Initalize the environment for the client application:
          1) Allocate a task block for the application in the
               shared library tasks memory space
          2) If not initialized, set up header pointer for task
               list and set up the asynch function for handling all
               client tasks terminating, otherwise hook new block
               into list
          3) Initialize task specific data including access to
               shared library resources and access to the dialog
               hook functions via the local dialog box interface */

share_printMsg(MSGID_Initialize, mdlSystem_getCurrTaskID());

/* Allocate new task block in the shared library's memory space */
if (!(taskInfoP =
               (void*)mdlDialog_callFunction(libTaskInfo.mdlDescrP,
                                    share_allocTaskBlock, NULL)))
     return -1;

share_printMsg(MSGID_TaskBlockP, taskInfoP);
/* Increment client task count and connect new task block into out
   task list. */
taskCount++;
if (!initialized)
{
     share_printMsg(MSGID_InitLibrary);
     taskListP = taskInfoP;
     initialized++;

     /* Set up asynch function to handle notification of all clients
        being unloaded */
     mdlShare_setFunction(SHARED_LIB_NO_MORE_CLIENTS, libDescP,
                          share_noMoreClients);
}
else
{
     share_printMsg (MSGID_LinkTaskInfo);
     taskP = taskListP;
     while (taskP->next)
          taskP = taskP->next;
     taskP->next = taskInfoP;
     taskInfoP->prev = taskP;
}
/* Initialize task specific information including access to the
   library's resources and dialog item hook function */
```

```
    strcpy(taskInfoP->taskId, taskIdP);
    taskInfoP->mdlDescrP = mdlSystem_findMdlDesc (taskIdP);
    taskInfoP->passwordP = &localPassword[0];

    /* The fully qualified path name of this shared library application
       is used to open the resource files which are linked into this
       library. These resources are available to any application which
       uses this shared library. */

    mdlResource_openFile(&taskInfoP->rscFileH, fullPathName, RSC_READ);
    mdlDialog_hookPublish(sizeof(uHooks)/sizeof(DialogHookInfo),uHooks);
    return SUCCESS;
}
```

Sample 2. MDLSHARE shared library initialize function.

If a library has a main function, that function is called when the shared library is loaded. The main function is called before the `initialize` function is called. MDL always uses the `argv[1]` argument to `main` to signal to an application why it was loaded. When MicroStation loads an application as an MDL shared library, it sets `argv[1]` to `SHARED_LIB`. The MDLSHARE shared library `main` function is shown in Sample 3.

```
/*--------------------------------------------------------------------
 * name main
 * authorBSI                    5/95
 *-------------------------------------------------------------------*/
Public void main
(
int argc,    /* => number of args in next array */
char *argv[] /* => array of cmd line arguments */
)
{
   /* Save the fully qualified path name of the shared library
    * for use in mdlResource_openFile during initialize() */
   strcpy(fullPathName, argv[0]);

   /* Open the resource file that we came out of */
   mdlResource_openFile(&libTaskInfo.rscFileH, NULL, 0);

   /* Load the command table */
   mdlParse_loadCommandTable(NULL);

   /* Publish our hook functions */
   mdlDialog_hookPublish(sizeof(uHooks)/sizeof(DialogHookInfo),uHooks);
   strcpy(libTaskInfo.taskId, mdlSystem_getCurrTaskID());
```

```
    libTaskInfo.mdlDescrP = mdlSystem_getCurrMdlDesc();
    strcpy(libTaskInfo.password, "mdlshare");
    libTaskInfo.passwordP = &localPassword[0];

    /* Init client application unload asynch function */
    mdlSystem_setFunction(SYSTEM_ALL_MDL_UNLOADS,
                          share_clientTaskUnload);

    /* Set up our own unload function */
    mdlSystem_setFunction(SYSTEM_UNLOAD_PROGRAM,
                          share_sharedLibraryUnload);
}
```

Sample 3. MDLSHARE shared library main functions.

Unloading an MDL Shared Library

MicroStation keeps an MDL shared library loaded as long as it is being used as a library. If MicroStation must unload a shared, then it also unloads all MDL tasks that require that library.

MicroStation does not automatically unload an MDL shared library when the last client task is unloaded. In order to have a shared library unload after the last MDL client task terminates, a shared library asynch function must be initialized using the `mdlShare_setFunction` function specifying the `SHARED_LIB_NO_MORE_CLIENTS` event type (this asynch function will be available for use in beta 2 of MicroStation) and the library must explicitly unload itself from the asynch function. If this is not done, the user or the shared library application itself must cause the unload of the library in some other way.

When unloading an shared library, MicroStation first calls the library task's unload hook. If the unload hook rejects the request to unload it, then there is no additional processing. The fact that there was an aborted attempt to unload the shared library does not affect any of the tasks that use the library. However, if the request to unload the library continues then all MDL tasks that use the library must be unloaded. MicroStation runs through the normal process for unloading these tasks. If an MDL task has an unload hook, MicroStation calls the hook specifying `SYSTEM_TERMINATED_LIBRARY_UNLOADED` as the parameter that specifies why the application is being unloaded. An MDL task that is being unloaded for this reason cannot abort the unload. That is, if an task is being unloaded because one of the required libraries is being unloaded then MicroStation ignores the return value of the task unload hook. Every client task is unloaded regardless of the hook's return value.

Determining When an MDL Program is Unloaded

It is often important to know when an MDL application is being unloaded. For example, if an MDL shared library calls an MDL application's user hook after the application has been unloaded, the results will be unpredictable, and a crash may occur. Also, if the MDL library is allocating and tracking resources for an MDL application, then it must know to free those resources when the application is unloaded.

The MDL shared library can use an unload hook to learn when an MDL application is unloaded. To install an unload hook, the library must call `mdlSystem_setFunction` with function type `SYSTEM_ALL_MDL_UNLOADS`. When an MDL application is unloaded, MicroStation will call the unload hook passing the application's MDL descriptor as a parameter. Prior to calling `mdlSystem_setFunction`, the library should ensure that it is running in the library task's context, instead of the client task's context. The easiest way to do this is to call `mdlSystem_setFunction` from the library's main entry point as shown in Sample 3 above.

Sample 4 below shows the MDLSHARE unload processing:

```
/*----------------------------------------------------------------------
 * name share_clientTaskUnload
 * authorBSI                    05/95
 *----------------------------------------------------------------------*/
Private void share_clientTaskUnload
(
void *mdlDescP,
long exitReason,
unsigned char *taskIdP
)
{
   TaskInfo *taskP=share_getTaskInfoP(mdlDescP);
   /* Free task block for unloading MDL application */
   if (taskP)
   {
        share_printMsg(MSGID_UnloadTask, taskP->taskId, taskP);
        if (taskP->prev)
             taskP->prev->next = taskP->next;
        if (taskP->next)
             taskP->next->prev = taskP->prev;
        if (taskListP == taskP)
             taskListP = taskP->next;
        free(taskP);
        taskCount--;
        /* If no more client tasks, unload the library */
        if (!taskCount)
        {
```

```
            taskListP = NULL;
        }
    }
}

/*----------------------------------------------------------------------
 * name share_sharedLibraryUnload
 * authorBSI                    05/95
 *----------------------------------------------------------------------*/
Private int share_sharedLibraryUnload
(
void *mdlDescP,
long exitReason,
unsigned char *taskIdP
)
{
   TaskInfo *taskP;
   share_printMsg(MSGID_UnloadLibrary);
   /* Free all client specific memory if still allocated */
   if (taskListP)
   {
        while (taskP=taskListP)
             share_clientTaskUnload(taskP->mdlDescrP,0L,taskP->taskId);
        taskListP=NULL;
   }
   return SUCCESS;
}
/* ---------------------------------------------------------------------
 * name share_noMoreClientsHook
 * authorBSI                5/95
 ---------------------------------------------------------------------*/
Private void share_noMoreClients
(
void
)
{
   share_printMsg(MSGID_NoMoreClients);
   mdlDialog_cmdNumberQueue(FALSE, CMD_MDL_SILENTUNLOAD,
                            libTaskInfo.taskId, TRUE);
}
```

Sample 4. MDLSHARE unload process.

Defining the Shared Functions and Variables

This section describes the extensions made to the syntax used for Dynamic Link Specification source files to accommodate MDL shared libraries (additional information on the DLS can be found in the section entitled *Dynamic Link Specification Source Files* in Chapter 20 of the MDL Programmer's Guide). The **dlmspec** utility is used to compile a Dynamic Link Specification source file producing a Dynamic Link Specification object file. The object file is used at link time to specify information on symbols that are to be resolved at runtime. The symbols can be resolved at runtime either from a DLM or from an MDL shared library. Normal MDL object files (.mo files) that need to have symbols resolved from a given Dynamic Link Specification object file must appear in **mlink**'s command line prior to the Dynamic Link Specification file.

The `%InterfaceType` command defines the type of shared library interface being used. The command is followed by `Mdllib` or `DLM` which identifies the shared library as being an MDL application or DLM respectively. The case is not significant. `Mdllib`, `mdllib` and `MDLLIB` are equivalent. The default is DLM. If `%InterfaceType` is not specified, **dlmspec** assumes that the library is a DLM.

The `%ModuleName` command has been extended as well. The name of the file specified after the command is used at runtime to determine the name of the file containing the DLM or MDL shared library. The file name suffix usually is not specified. For DLMs, it is system-specific. It is provided by MicroStation at runtime. The defaults are .out (Intergraph workstation), .rex (PC), .so (SPARC) and .sl (HP700). For MDL shared libraries, MicroStation uses the module name with .ma appended to look for the file. Starting with MicroStation 95, MicroStation will also look for MDL shared libraries with the extension .msl.

The DLS for the MDLSHARE shared library is presented in Sample 5:

```
/* Specify the version number */
%Version 0x551
%InterfaceType Mdllib
%ModuleName mdlshare
%Functions
   initialize
   sharelib_passwordHook(share_passwordHook)
   sharelib_passwordDialog(share_passwordDialog)
   sharelib_setPassword(share_setPassword)
   sharelib_setPasswordVarP(share_setPasswordVarP)
%EndFunctions
%Variables
   taskCount
%EndVariables
%End
```

Sample 5. DLS for MDLSHARE shared library.

Runtime Concerns

The developer of an MDL shared library must understand the concepts of switched and non-switched calls. This affects the operation of a lot of the MDL built-in functions.

For many types of hook functions, each MDL task may have one instance of the hook specified. When the task specifies another value for that hook, the first value is replaced. If an MDL library specifies a hook function, then it may be overriding the hook function of the client task. Also, the library's hook function will be disabled if the client task is disabled. In some cases, that may be the desired behavior. In other cases, that behavior is unacceptable. There are a few solutions.

One solution is for the library to do a task switching prior to establishing the hook function. To do so, the library must call one of its own functions using `mdlDialog_callFunction`. When the library function calls one of it's own functions using `mdlDialog_callFunction`, it causes MicroStation to do an MDL context switch, switching from the client task's context to the MDL library's context. Then when the library sets up the user hook, it is done using the library's context instead of the client task's context. The context at the time the hook is set up also determines the context when the hook is dispatched.

A second solution is for the library to save the call the client task's hook function as part of the processing in its hook function. Most of the `mdl..._setFunction` functions return the value of the old hook function. When the library sets it own hook function, it can save the value returned by the `mdl..._setFunction` call used to establish the hook. When MicroStation calls the library's hook function, the library's hook function can use the saved function pointer to call the client task's hook function.

Sample 2 above shows an example of the use of the `mdlDialog_callFunction` function in the `initialize` function. In this case, the shared library calls the function `mdlshare_allocTaskBlock` to allocate memory in its own address space instead of the client task's address space.

initialize

```
int initialize
(
char         *fileNameP,
char         *taskIdP,
void         *libDescP,
unsigned long     initParameter
);
```

Description MicroStation calls the initialize function as the final step of loading an MDL application that requires an MDL shared library. If the MDL shared library is

required by more than one MDL program, then the shared library is only loaded once but the initialize function is called for each MDL program.

MicroStation uses the client's MDL context when it calls the MDL shared library.

fileNameP points to the name of the file that contains the client MDL application. The name includes the entire path.

taskIdP points to the task ID of the client MDL task.

libDescP points to the library descriptor for the library's task. This field should be used as the second parameter in the `mdlShare_setFunction` call to set the asynch function to handle no more shared library client tasks.

initParameter is the value specified for `%Version` in the Dynamic Link Specification object file that was linked with the client application.

Returns `initialize` may return 0 to allow the load to continue. If `initialize` returns a non-zero value then the load of the MDL program is aborted.

userHook_mdlUnload, userHook_sharedLibNoMoreClients, userHook_allMdlUnload

```
void userHook_mdlUnload
(
void        *mdlDescP,
long        exitReason,
unsigned char    *taskIdP
);

void userHook_sharedLibNoMoreClients(void);

void userHook_allMdlUnload
(
void        *mdlDescP,
long        exitReason,
unsigned char    *taskIdP
);
```

Description MicroStation calls the `userHook_mdlUnload` user hook functions whenever an MDL application is unloaded. Both the *mdlDescP* and the *taskIdP* identify the MDL application being unloaded. Either parameter is enough to uniquely identify the application. Both parameters are supplied just as a matter of convenience.

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 `userHook_sharedLibNoMoreClients` hook function for each library that no does not have any other clients.

MicroStation calls the `userHook_allMdlUnload` 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.

MicroStation sets the MDL context to that of the library's task prior to calling the hook functions.

Returns Both of these hook functions are of type `void`. MicroStation ignores the return values of both of these functions.

22 Database Manipulation

The two sections in this chapter, “SQL Database Interface Toolkit” and “Using MicroStation Database Servers from Initapps” tell you how to use MDL to interface MicroStation to databases.

SQL Database Interface Toolkit

The SQL database interface toolkit is a set of tools which enable a developer to build standard MicroStation interfaces to SQL databases. The SQL toolkit minimizes the development necessary to support a new database.

Many of MicroStation’s capabilities utilize the database interface. Graphic functions, interactive requests and MDL database run-time library function calls can generate database service requests. By using the SQL toolkit in combination with tools from the target database vendor, a developer will produce a “module” that will receive the database service requests, complete the necessary processing and return any resulting data to MicroStation. Building the module is simplified with tools that have a SQL Application Programmers Interface, but the SQL Server can be developed without a SQL API if the developer writes a parser for the vendor supplied tools.

Architecture

The developer supplied module that processes SQL requests has the same requirements for all databases. Access is needed to:

- Embedded SQL/C tools provided by the database vendors.
- MDL for string resource management, error handling, user interface.

The developer supplied module consists of an MDL component and an external program component.

The “SQL Toolkit” diskette is available by calling BSI Technical Support. The following is the directory structure supplied on the “SQL Toolkit” diskette:

```
/MDL
     dbload.mke
     loader.mc
```

```
        loadipc.mc
        loadutil.mc
        dbload.fdf
    /SERVER
        extmain.c
        extipc.c
        extutil.c
        extmain.fdf
    /RIS
        usntris.mke
        rissql.rc
```

The toolkit provides code for the mdl application **dbload.ma** (under the MDL heading), code for the external components that are not database dependent (under the SERVER heading), and an example of the implementation of the database dependent portion using the RIS SQL utilities.

SQL Requests

MicroStation generates over 70 database (DB) requests that are handled by the MDL application **server.ma**. The mdl application processes these requests and in turn can generate one or more of the 18 database service (DS) requests. The requests are as follows:

```
DS_DatabaseProfile      DS_ProcessStatement
DS_OpenDatabase         DS_OpenCursor
DS_CloseDatabase        DS_FetchRow
DS_Terminate            DS_CloseCursor
DS_DescribeDatabase     DS_ProcessForm
DS_DescribeTable        DS_GetErrorInformation
DS_DescribeColumn       DS_AdditionalRequest
DS_CommitTransaction    DS_DebugMode
DS_RollbackTransaction  DS_CopyTable
```

The requests are defined in the **database.h** include file. In the discussion of each individual request, the symbol => defines data that is sent from **dbload.ma** to **ustnris** and the symbol <= defines data that is sent from **ustnris** to **dbload.ma**.

DatabaseService Structure

The `DatabaseService` structure consists of a header and a union of the database service request structures. For example the `DS_OpenDatabase` request has a header which is of type `ServiceHeader`, and is followed by the structure for an open request, aptly named `OpenDatabase`.

ServiceHeader

The service header consists of the following variables:

`long status`

=> *status* is initialized to zero.

<= *status* contains `SUCCESS` if the request was executed without errors or an `SQL_LOOKUP` if the request resulted in an SQL error.

If the request resulted in any error, the header is followed by an `ErrorInformation` structure which contains the error code and the text of the error message.

`ULong type`

<=> The type of request. This is the `ULong` that the `DS` requests are defined to equal.

`ULong length`

<=> The length in bytes of the service structure.

`boolean isCallocd`

<=> A boolean value indicating whether a pointer in the message structure has been dynamically allocated (`TRUE`) or whether it has not (`FALSE`).

`long callocdLength`

<=> If *isCallocd* is `TRUE` then this will be the number of bytes allocated to the pointer in the message structure.

DS_DatabaseProfile

The database profile is a structure whose information is set by the developer when the toolkit is originally built. It should not need to change unless the design or underlying database changes. A `DS_DatabaseProfile` message is issued by MicroStation when it starts the server. This information is used by **server.ma** to tailor requests to the specific database being used.

=> There is no data in a `DS_DatabaseProfile` message when it is sent from **dbload.ma** to **ustnris**. The message consists of a header with length set to zero.

<= The `DatabaseProfile` structure is filled in with the appropriate information for the underlying database.

The database profile consists of the following structures. These structures are defined in database.h:

`BrandProfile`, `AccessProfile`, `DataTypeProfile`, `SQLProfile`, `FunctionProfile`, `TransactionProfile`, `FormsProfile` and `GraphicsProfile`.

DS_OpenDatabase

This request tells the SQL Server which database to open.

```
char text[512]
```

=> Contains the text following a `DB=` statement. This data is not parsed by server.ma. The developer should decide the format of the connect statement.

<= Currently there is no data returned from the SQL Server in the text variable. This is left for future use.

DS_CloseDatabase

This request tells the SQL Server to close the database. This request has no data.

DS_Terminate

This request tells the SQL Server to do its cleanup and halt execution. This request has no data.

DS_DescribeDatabase

This request gets a description of the database.

```
long descriptionLength
```

=> *DescriptionLength* is zero.

<= *DescriptionLength* is the number of characters in the description, including the `NULL` character that terminates the buffer.

```
char *description
```

=> Description is a `NULL` pointer.

<= Description points to the `NULL` terminated character buffer of table names. Each character string is `NULL` terminated. Note that there are two `NULL` characters at the end of the buffer, one ending the last string, and one signifying the end of the entire buffer.

```
char databaseName[DB_DATABASE_NAME_SIZE]
```

<=> Not used. This is a place holder for future advancement.

DS_DescribeTable

This request gets a description of a table.

```
long descriptionLength
```
=> *DescriptionLength* is zero.

<= *DescriptionLength* is the number of bytes in the description.

```
char *description
```
=> *Description* is a `NULL` pointer.

<= *Description* points to a descriptor message structure. The structure of a descriptor message is described after the request descriptions.

```
char databaseName[DB_DATABASE_NAME_SIZE]
```
=> Not used. This is a place holder for future advancement.

```
char tableName[DB_TABLE_NAME_SIZE]
```
=> This is the name of the table to describe.

DS_DescribeColumn

This request gets a description of a column.

```
long descriptionLength
```
=> *DescriptionLength* is zero.

<= *DescriptionLength* is the number of characters in the description, including the `NULL` character.

```
char *description
```
=> *Description* is a `NULL` pointer.

<= *Description* points to a one dimensional descriptor message structure. It contains a descriptor message header, one column name, one type and one length.

```
char databaseName[DB_DATABASE_NAME_SIZE]
```
=> Not used. This is a place holder for future advancement.

```
char tableName[DB_TABLE_NAME_SIZE]
```
=> This is the name of the table the column is in.

```
char columnName[DB_COLUMN_NAME_SIZE]
```
=> This is the name of the column to describe.

DS_CommitTransaction

This request tells the SQL Server to commit transactions processed since the last commit. This request has no data.

DS_RollbackTransaction

This request tells the SQL Server to rollback the transactions processed since the last commit. This request has no data.

DS_ProcessStatement

This request processes SQL statements that are not select statements.

```
long statementLength
```

=> *statementLength* is the length of the SQL statement including the `NULL` character.

```
char *statement
```

=> *statement* points to a character buffer containing the SQL statement to process. It is a `NULL` terminated character string.

DS_OpenCursor

This request opens a cursor for SQL statements that are selects.

```
long descriptionLength
```

=> *descriptionLength* is the length of the select statement.

<= *descriptionLength* is the length of the descriptor message.

```
char *description
```

=> *description* points to a `NULL` terminated character string with the select statement for open cursor.

<= *description* points to a descriptor message structure. This is the result of the describe on the select statement. It is sent to **server.ma** so that **server.ma** knows the format of the data that will be returned.

```
long cursorID
```

=> Initialized to -1.

<= A number between [`0..MAX_OPEN_CURSORS`]. This indicates the cursor that was opened. The `fetch` and `close` cursor operations should use this number as the *cursorID*.

DS_FetchRow

This request fetches a row from the cursor identified by *cursorID*.

```
long rowLength
```
=> *rowLength* is 0.

<= *rowLength* is the number of bytes in row.

```
char *row
```
=> *row* is a `NULL` pointer.

<= *row* points to a buffer of values. One for each column selected.

```
long cursorID
```
=> The *cursorID* to fetch from.

```
short returnDataType
```
<=> Not Used.

DS_CloseCursor

This request closes the cursor identified by *cursorID*.

```
long cursorID
```
=> The *cursorID* to fetch from.

DS_ProcessForm

This request processes a form.

```
char arguments[256]
```
=> The arguments to the process form executable, executed from server.

```
char formName[256]
```
=> The name of the form to be executed.

```
long textMode
```
=> On DOS platforms, this will switch from a text monitor to a graphics monitor when the screen form is executed.

```
long noStdOut
```
=> Redirects the IBM PCs standard out.

DS_GetErrorInformation

This request gets the last SQL error code and text of the error message from the server. This request is not currently used. When the external server encounters an error, it returns this structure to **dbload.ma**. It is not necessary to go back to the external server for this information.

```
long code
```
<= *code* is the error code that occurred in the server.

```
char message[MAX_ERROR_LENGTH]
```
<= *message* is a `NULL` terminated character string that contains the error message.

DS_AdditionalRequest

This request is produced by a call to `mdlDB_additionalRequest`. It allows a developer to add requests to the toolkit. `MaxMSLink` in **rissql.rc** is an example of the an additional request. `MaxMSLink` does not need to be supported by any developer supplied SQL Servers, since in future revisions the processing done in the SQL server, will be moved into **server.ma**.

```
long requestLength
```
<=> *requestLength* is the number of bytes in the request.

```
char *request
```
<=> *request* is a pointer to a developer defined request.

DS_DebugMode

This request changes the debug mode of the toolkit.

```
long mode
```
=> Indicates the state of debug.

DS_CopyTable

This request copies the structure, not the contents, of one table to another.

```
char fromTableName
```
=> The name of the table to copy from.

```
char toTableName
```
=> The name of the table to copy to.

Descriptor Message Structure

This structure is sent from **ustnris** to **dbload.ma** in an `DS_OpenCursor` request, a `DS_DescribeTable` and a `DS_DescribeColumn`.

`short columns`
<= The number of columns in a returned row.

`short rows`
<= The number of rows in the packed buffer

`short items`
<= The total number of items packed.

`short continued`
<= This variable is no longer used, but remains for compatibility to Version 4 MicroStation. It is always zero.

`char data[1]`
<= The packed buffer. The first items that are packed are the column names, the second are the column types, and the third are the column lengths.

Communications

server.ma <=> dbload.ma

The SQL module is invoked by **server.ma**. When **server.ma** is started up, either from the command window or at MicroStation startup as a DGN application, it looks for the name of the mdl loader module in the environment variable `MS_SERVER` and starts up, in the case of the delivered examples, **dbload.ma**. During startup, **dbload.ma** uses an `mdlDialog_...` function to publish a pointer to its message handling function `handleDBRequest`. By publishing the function, **server.ma** can execute `handleDBRequest` with an `mdlDialog_...` function, passing in the published pointer and the generated `DS` request as a parameter. Thus from the view of **server.ma**, all that is necessary to execute a `DS` request is a call to `handleDBRequest`.

dbload.ma <=> ustnris

When **dbload.ma** is started by **server.ma**, it looks for the name of the external server in the environment variable `MS_DBEXT`, sets up a message queue, and starts the external server, passing the message queue identifier as a parameter. All communication between **dbload.ma** and **ustnris** is through this message queue utilizing the external program communication utilities provided by MDL. On DOS platforms this method is only useful if the external program can be made into a PHARLap executable. If this is not possible, then you may want to make the external program a Dynamically Linked Module (DLM) or use shared memory.

Session Debug

With the new toolkit there are several new debug commands. Here is a breakdown of all of the debug commands and their effects:

Command	Effect
session debug on	Turns debug on for server.ma.
session debug off	Turns debug off for both **server.ma** and the SQL server.
session debug toggle	Toggles the current **server.ma** debug session setting.
session debug ipc on	Turns debug on for the packets being sent between **dbload.ma** and the SQL server.
session debug ipc off	Turns ipc debug off for the SQL server.
session debug ipc toggle	Toggles ipc debug for the SQL server.
session debug external on	Turns debug on for the SQL server.
session debug external off	Turns debug off for the SQL server.
session debug external toggle	Toggles debug for the SQL server.

Environment Variables

MS_SERVER

This is the task name of your mdl loader application. The task name for the example mdl loader application provided in the SQL toolkit is defined as **dbload** in **dbload.mke**.

MS_DBEXT

This is the name of your C executable. The name for the example RIS SQL / C executable is **ustnris**.

MS_DBASE

This is the path to your C executable in `MS_DBEXT`.

MS_DBFLOAT

Since character data is returned to **server.ma** from the sql server, any `float` data read from the database must be translated into character data. This variable is the `sprintf` format string used to translate data read as `floats`. If the database server that you are implementing returns character data or you choose to coerce it to character data, this variable is not needed. For the provided RIS SQL server, "%.7g" is the default setting.

MS_DBDOUBLE

This variable is the `sprintf` format string used to translate data read as doubles. If the database server that you are implementing returns character data or you choose to coerce it to character data, this variable is not needed. For the provided RIS SQL server, "%.16lg" is the default setting.

MS_LINKTYPE

This is the same environment variable that was needed in earlier versions of MicroStation. It specifies the type of links that are understood by the server. The first in the list is the type that will be written.

DBTYPE Resource

In the include directory there is a file called **dbtype.r**. If you are building a new database interface to MicroStation using the SQL Toolkit you will need to add a line in this file that defines the level, id, link type and the ascii id for your database. The file explains these four variables.

You will then need to add this resource file to the **server.ma** mdl application. To do so, first compile **dbtype.r** into a resource file using the **rcomp** utility. To create the compiled resource file **dbtype.rsc**, a command such as the following should be used:

```
RCOMP -I<YOUR INCLUDE DIRS> DBTYPE.R
```

Then the **rlib** command can be used to add **dbtype.rsc** to **server.ma**. The command should resemble:

```
RLIB -OSERVER.MA DBTYPE.RSC
```

If you need the old **server.ma** you should make a copy of it since **rlib** does not backup the old version when it creates the new one.

The new **server.ma** will now have knowledge of the new database type and be able to manipulate the new database's linkages.

Using MicroStation Database Servers from Initapps

This section describes the procedures for using MicroStation database servers from within a startup application (**initapp**). At the end of the section is a code sample illustrating this process.

Overview

MicroStation database servers can be started for use by an initapp. Typically MicroStation is configured for use with a database by designating the "server" MDL application as a design file application (**dgnapp**). The server gets the value of the `MS_SERVER` environment variable and starts the mdl loader. The MDL loader will determine the name of the external server by getting the `MS_DBEXT` environment variable and start the external server. When a MicroStation initapp is started, no design file has been attached, so the **dgnapp**s, including the server application, have not yet been started.

Therefore, before using one of the database servers, `mdlSystem_loadMdlProgram` must be called to load the MDL server **server.ma**. When `mdlSystem_loadMdlProgram` is called it should be given the argument `FRONTEND`. This will notify the loader that it is being loaded from an **initapp**. A loader must know that it is being started as an initapp for the following reasons:

- The INFORMIX, RIS and XBASE servers write a type 66 element to the design file with state information about the database session as a result of the **DB=** keyin or the `mdlDB_activeDatabase` function. This action should not be taken when the server is started as an **initapp** since no design file has yet been attached.
- MicroStation sends several startup messages to the server when the server is launched as a **dgnapp**. The `FRONTEND` argument will ensure that the loader simulates this "handshaking."

To complete the process of making the database server available from an initapp, log on to the database by issuing a connect statement with `mdlDB_activeDatabase`.

If the **initapp** will put MicroStation in graphics mode, further considerations must be made. If control will return to the **initapp** after MicroStation exits graphics, the server connection must be reestablished as above from the reload hook if the database is to be accessed from the **initapp**. This is necessary because MicroStation unloads all MDL applications including the loader when exiting graphics and returning to the initapp. When the reload function is finished with the server, it must unload the loader so the server is properly terminated.

The MicroStation database model has been revised in Version 5. The Version 5 model makes the database system that is being used transparent to a user or application writer. In previous versions an **initapp** would load

the value of the MS_SERVER environment variable, oloader, ixloader, urloader or dbloader depending on what database was being used. In Version 5, an **initapp** loads an MDL server application, "server", that will take care of loading the appropriate mdl loader application. To upgrade an initapp from an earlier version of MicroStation, in the call to **mdlSystem_loadMdlProgram** replace the value of MS_SERVER or a hard coded mdl loader with the MDL server application "server."

Sample Database Initapp

The following code is presented as a sample skeleton initapp that uses a database server. The main routine first initializes the database server and does any necessary pre-processing of the database. The **initapp** then enters graphics, sets a reload function and attaches a design file. The **reload** function is called when MicroStation exits graphics. The database server is reloaded and any post-processing of the database is done here. Finally, the server is terminated and MicroStation is exited.

```
/*----------------------------------------------------------------------+
|                                                                       |
| Copyright (C) 1992   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.                                                         |
|                                                                       |
+----------------------------------------------------------------------*/
/*------------------------------- -------------------------------------+
|  initapp.mc                                                           |
|                                                                       |
+----------------------------------------------------------------------*/
/*----------------------------------------------------------------------+
|  Include Files                                                        |
|                                                                       |
+----------------------------------------------------------------------*/
#include <mdl.h>
#include <cexpr.h>
#include <system.h>
#include <global.h>
#include <dbdefs.h>
#include <dberrs.h>
```

```
#include <userfnc.h>
/*----------------------------------------------------------------------+
|  Private Global Variables                                             |
|                                                                       |
+----------------------------------------------------------------------*/
/*----------------------------------------------------------------------+
|  Major Public Code Section                                            |
|                                                                       |
+----------------------------------------------------------------------*/
 /*----------------------------------------------------------------------+
| name startupDatabaseServer                                            |
|                                                                       |
| Startup the database server from the initapp                          |
|                                                                       |
| author BSI 6/93                                                       |
|                                                                       |
+----------------------------------------------------------------------*/
startupDatabaseServer(void)
{
   int status;
   status=mdlSystem_loadMdlProgram("server", "server", "FRONTEND");
   if (status != SUCCESS)
        return(status);
   /* connect to the database */
   status=mdlDB_activeDatabase("gis");
   if (status != SUCCESS)
        return(status);
}
/*----------------------------------------------------------------------+
| name reloadFunction                                                   |
|                                                                       |
| This function is called by MicroStation after the                     |
| user closes the design file opened in main.                           |
|                                                                       |
| author BSI 6/93                                                       |
|                                                                       |
+----------------------------------------------------------------------*/
reloadFunction
(
int argc,
char *argv[]
)
{
   char buffer[256];
   int status;

   /* The database server is unloaded by MicroStation when closing the
      design file.  If more work needs to be done with the database, the
```

```
      server must be reactivated. */
   startupDatabaseServer();

   /* do any necessary "cleanup" work with the database */
   status=mdlDB_sqlQuery(buffer, “select owner from parcel
                         where mslink=1”);

   /* unload the loader here so the server is properly terminated */
   mdlSystem_unloadMdlProgram(“server”);

   /* exit the MDL initapp and MicroStation will exit */
   mdlSystem_exit(0, 1);
}
/*----------------------------------------------------------------------+
|                                                                       |
| name main                                                             |
|                                                                       |
| author BSI 6/93                                                       |
|                                                                       |
+----------------------------------------------------------------------*/
cmdName main()
{
   char buffer[256];
   int status;
   startupDatabaseServer();

   /* do any necessary setup work with the database */
   status=mdlDB_sqlQuery(buffer, “select owner from parcel
                         where mslink=1”);
   printf(“query result <%s>\n”, buffer);
   mdlSystem_enterGraphics();

   /* Set the function MDL should call when it needs to reload this
      MDL application. This function will be called, for example, when
      the user chooses to close the current design file. */
   mdlSystem_setFunction(SYSTEM_RELOAD_PROGRAM, reloadFunction);
   mdlSystem_newDesignFile(“cd9”);
}
```

23 Low Level DOS Interface Functions (PC only)

This section describes MDL functions that let you access real mode memory and perform software interrupts. The functions described in this section are available only for MicroStation PC.

MDL and DOS

MDL provides the standard DOS interrupt functions, `int86`, `intdos` and `bdos`, to provide access to standard DOS functions through software interrupts. It also provides `int86PassThrough` for access to custom interrupt handlers.

The `int86`, `intdos` and `bdos` functions cannot be used for access to custom interrupt handlers, because the Phar Lap DOS extender modifies the registers for these calls. For DOS functions requiring addresses, the DOS extender uses a buffer in the real mode address space. It copies the data into the buffer, modifies the parameters to point to the buffer, reissues the interrupt, and then copies the data from the buffer after the interrupt has been processed.

Phar Lap provides extensions to the standard DOS functions, which are available through `int 21h`. These functions should be accessed with `int86` or `intdos`.

The `int86PassThrough` function issues a software interrupt, and all specified register values are passed to the interrupt handler unmodified.

The `copyToReal` function copies from protected mode memory to real mode memory, and the `copyFromReal` function copies from real mode memory to protected mode memory.

Please read **devtools.txt**, installed in the MDL directory, for the latest version information on PC development tools.

Function	Used to
`int86`	Load registers with the specified values and perform the interrupt.
`intdos`	Load registers with the specified values and perform the interrupt.

Function	Used to
int86PassThrough	Load registers with the specified values and perform the interrupt.
bdos	Place *number* in the AH register, *dxreg* in the DX register and *alreg* in the AL register. It then performs an int 21h.
segread	Place current value of the segment registers into the structure that *segRegsP* points to.
copyToReal	Copy memory between the protected mode.
copyFromReal	Address space and the computer's real mode address space.

Example

See dosfunc.mc.

int86, intdos

```
#include <dos.h>

int int86
(
union REGS   number,        /*  => interrupt number */
union REGS   *beforeP,      /*  => input values */
union REGS   *afterP        /* <=> output values */
);

int intdos
(
union REGS   *beforeP,      /*  => input values */
union REGS   *afterP        /* <=> output values */
);
```

Description int86 and intdos load registers with the specified values and perform the interrupt. After the interrupt returns, the hardware registers are copied into the union that afterP points to. The carry flag is expanded and stored in the union's cflag field.

intdos always uses int 21h. int86 uses the interrupt number specified in number. Otherwise, these functions are the same.

These functions cannot be used to invoke interrupt handlers that return the modified DS register. If the DS register is modified to point to more information, use the built-in function int86PassThrough.

Returns The int86 and intdos functions return the value that the interrupt returns in EAX.

See Also int86PassThrough, bdos.

int86PassThrough

```
#include <dos.h>

int int86PassThrough
(
int           number,       /*  => interrupt number */
int          mode,          /*  => PROTECTED_MODE_INTERRUPT */
union REGS   *dataRegsP,    /* <=> AX, BX */
struct SREGS *segRegsP,     /* <=> SS, DS */
Flags86      *flagsP        /* <= flag register */
);
```

Description `int86PassThrough` loads registers with the specified values and performs the interrupt. After the interrupt returns, the hardware registers are copied into the areas that *dataRegsP* and *segRegsP* point to. The flags are stored in the word *flagsP* points to.

If *mode* is `PROTECTED_MODE_INTERRUPT`, the interrupt is issued in protected mode and control is given to a protected mode interrupt handler. If mode is `REAL_MODE_INTERRUPT`, the interrupt is issued in real mode and control is given to a real mode interrupt handler.

`NULL` can be passed for *segRegsP* or *flagsP*. If either of these registers is `NULL`, the corresponding parameter is ignored.

`int86PassThrough` differs from `int86` because the register values specified in *dataRegsP* and *segRegsP* are not modified. They are passed directly to the interrupt handler.

Returns The `int86PassThrough` function returns the value that the interrupt returns in `EAX`.

See Also `intdos`, `int86`, `bdos`, `segread`.

bdos

```
#include <dos.h>

int bdos
(
int     number,    /* => DOS function number */
int     dxreg,     /* => DX register */
int     alreg      /* => AL register */
);
```

Description The `bdos` function places number in the `AH` register, *dxreg* in the `DX`, and *alreg* in the `AL` register. It then performs an `int 21h`.

Returns The `bdos` function returns the value that the operating system returns in `AX` to the caller of `bdos`.

See Also `int86`, `intdos`, `int86PassThrough`.

segread

```
#include <dos.h>

void segread
(
struct SREGS *segRegsP
);
```

Description The `segread` function places the current value of the segment registers into the structure that *segRegsP* points to.

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

See Also `int86PassThrough`.

copyToReal, copyFromReal

```
int copyToReal
(
int     realSegment,    /* => real mode segment */
int     realOffset,     /* => real mode offset */
void    *protectedP,    /* => protected mode address */
int     count           /* => bytes to copy */
);

int copyFromReal
(
void    *protectedP,    /* <=> protected mode address */
int     realSegment,    /* => real mode segment */
int     realOffset,     /* => real mode offset */
int     count;          /* => bytes to copy */
);
```

Description The `copyToReal` and `copyFromReal` functions copy memory between the protected mode address space and the computer's real mode address space.

Returns The `copyToReal` and `copyFromReal` functions return a count of bytes copied.

Appendix A: Internationalization

More than one-half of MicroStation users live in countries where English is not the primary language. The number of such users is growing. However, most third-party applications for MicroStation are designed for English-speaking users and cannot properly handle foreign language text and country specific information. Acceptance of English-only applications is reduced where native language versions of competing software are available.

Enhancement of software for worldwide use is done in two stages: internationalization and localization. Internationalization is done when an application is designed and developed. The main objective of internationalization is modifying source code so that a single executable can handle various languages. Language and culture-specific information and assumptions are separated from the main program code and put into external resource files and data tables. This allows for easy modification for different languages. Localization is the subsequent process of translating the resource files and customizing data files. It is usually performed by translators.

Many confuse internationalization with localization and misunderstand that it is just a matter of translating menus and labels in dialog boxes. This is true to some extent, if target languages are limited to Western European languages which share many characters with English. But, it is untrue for many other languages that require characters not found in standard English ASCII. For instance, some Asian languages require several thousand characters to display text. Source code has to be designed to handle these large numbers of characters. In many countries, culture-specific data, such as date and time, are written or arranged in different orders and formats. Further, different engineering standards exist for measurement and symbols.

The primary purpose of this chapter is to describe the necessary steps for internationalizing MDL applications. It also explains how to configure MicroStation for use with various languages. This document does not discuss how to translate resource files and documentation. The subject of how to design applications to adhere to local design and industrial standards is beyond the scope of this chapter. This is because standards differ between fields.

Overview of Internationalization Issues

An internationalized application lets users enter, store, retrieve, manipulate and display data in native languages, according to local customs and standards. MicroStation handles most of the work for the entry of user data and display of characters for different languages. An application developer's primary responsibility is string processing, text element manipulation and messaging. Adapting to local standards and translation of documentation are the responsibility of developers.

Target Languages

The amount of work required for internationalization depends on the target language. Languages in the world can be grouped into three categories: European, Far Eastern and Middle Eastern.

Many European languages (German, Spanish and Romanian) use Latin alphabets containing fewer than 256 characters. In many cases, users' needs can be met by separating text strings from program code into resource files and translating them. Nevertheless, special attention must be paid to special characters, symbols, sorting order and local standards.

Far Eastern Asian languages (Chinese, Japanese and Korean) use character sets consisting of several thousand characters. It usually takes two bytes of data to represent each of these characters internally; the standard assumption of “1 byte = 1 character” for English text does not work for processing Asian text strings. In general, it takes more work to develop applications capable of processing Asian languages.

Middle Eastern languages (Hebrew and Arabic) are bidirectional. The basic text is written right-to-left. When mixed with numbers and English text, the direction of writing changes within the same sentence. In addition, Arabic is context-sensitive, meaning that the form of a character changes depending on the value of nearby characters. Because MicroStation handles directionality and the number of characters is generally less than 256 characters, applications designed for European languages also generally work.

Recommendation

MicroStation's internationalization interface is continuously under development. This is because international programming standards are still evolving. The extent of MicroStation's support depends on the underlying operating system. Some platforms, such as Windows NT, have extensive tools for foreign languages which MicroStation can utilize. However, MicroStation's functions and applications are designed to be portable across platforms so the same level of support cannot be extended to all platforms.

For example, some of the new wide character functions documented in the section *String Processing*, are not supported on all platforms. The wide character functions are designed to be usable with any language, but their functionality is currently limited to Asian languages.

If you are already familiar with using wide character functions in another developing environment, you should not convert all existing code to wide character format. MicroStation itself has not been fully converted to the wide character format, and still uses both multi-byte and wide character formats for processing foreign languages. You should start with the conventional coding method and change only language-sensitive parts using available tools. If you are not considering Asian markets, it might be a good idea to wait for the finalization of the wide character functions explained in *String Processing*.

Character Encoding Methods

Character encoding is the mapping of a character to a numeric value. The same character value may represent different letters, ideographs, digits, symbols or control functions depending on the selected encoding method. Encoding methods differ between languages.

DEC Multinational ASCII Character Set

The ASCII character set was originally represented by 7 data bits, describing 128 characters. Extended ASCII code, represented by 8 bits, results in 256 characters and contains the accented characters required by many Western European Latin-derived languages. There are many variations of Extended ASCII encoding methods. Mapping of alphabetical and numeric characters in the first 128 characters is identical in all versions of Extended ASCII. However, mapping of accented characters and symbols differ depending on the hardware platform and language of choice.

By default, MicroStation expects text displayed in dialogs, message fields and text elements to be in DEC Multinational ASCII code, (Figure 2.1). This is because DEC ASCII already contains characters for English and many European languages. The code range, (0xC0 - 0xFF) encompasses accented characters and other special symbols. Raster and vector fonts (international) delivered with MicroStation are encoded in DEC Multinational ASCII. This works with many European languages (German, French, Italian or Spanish).

One significant shortcoming of DEC Multinational ASCII is that accented characters for all European languages are not contained, (Icelandic and Nordic). Further, characters for non-Roman scripts (Russian, Greek, Hebrew and Czech) are not included. For these languages, alternative coding schemes are selected with raster and vector fonts encoded accordingly.

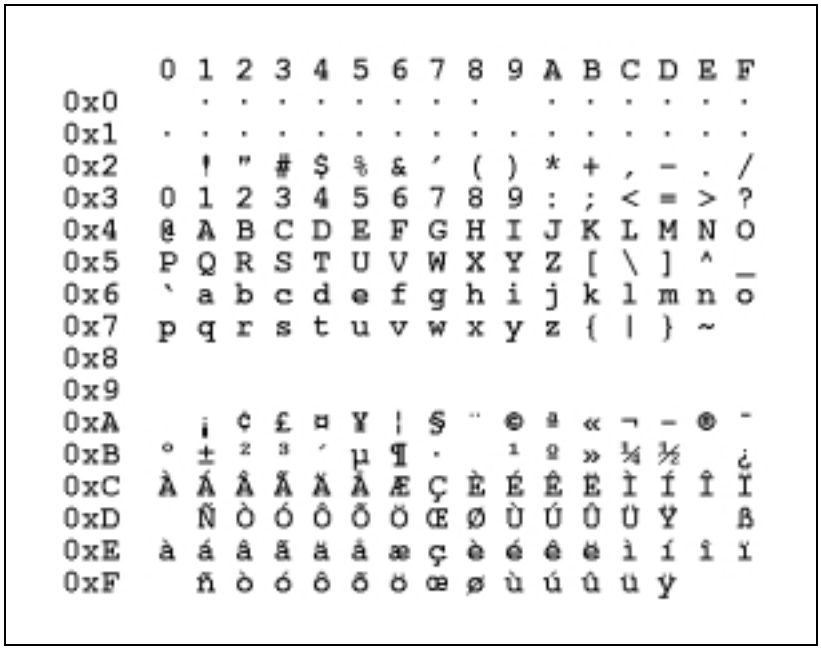
Character Translation Table

On different hardware platforms or operating systems, the same language can have different character encoding methods. To keep compatibility of design files across platforms, MicroStation uses one standard encoding method for a given language. When text is imported or exported in and out of MicroStation, character values need to be translated according to the standard encoding method.

For Western European languages, DEC Multinational ASCII is used for internal encoding by MicroStation. On the PC, the English version of MS-DOS uses code page 437, (see ref. I. for more information on code page). Macintosh has its own encoding method. Mapping accented characters in the range 128(0x80) - 255(0xff) is different for each encoding method. For instance, the value of ü is 129(0x81) in DOS, 159(0x9f) in Mac and 252(0xfc) in DEC. When a user enters characters from a keyboard, MicroStation automatically translates the character values using a character mapping table called the character translation table.

Default character translation tables are delivered in /CHARTRAN. Users can select character translation tables in the user preference settings dialog box. If keyboard layout or input language is changed on the operating system, the corresponding character translation table needs to be selected. For some languages and platforms, character translation is not required. For Western European languages, no translation is

necessary on Unix and Microsoft Windows because their character encoding is very similar to DEC Multinational ASCII.



	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0x0		·	·	·	·	·	·	·	·		·	·	·	·	·	·
0x1	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·
0x2		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
0x3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
0x4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
0x5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
0x6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
0x7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	
0x8																
0x9																
0xA		¡	¢	£	¤	¥	¦	§	¨	©	ª	«	¬	-	®	¯
0xB	°	±	²	³	´	µ	¶	·	¸	¹	º	»	¼	½	¾	¿
0xC	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
0xD		Ñ	Ò	Ó	Ô	Õ	Ö	Œ	Ø	Ù	Ú	Û	Ü	Ÿ		ß
0xE	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
0xF		ñ	ò	ó	ô	õ	ö	œ	ø	ù	ú	û	ü	ÿ		

Figure 2.1. DEC multinational characters.

Asian Character Set

Asian languages, (Chinese, Korean and Japanese) typically have many characters. Approximate numbers of characters are as follows:

- Japanese: 6,400 characters
- Korean: 8,000
- Simplified Chinese (Mainland China): 6,800
- Traditional Chinese (Taiwan): 14,000

Due to the large numbers, at least 2 bytes or 16 bits are needed to represent a single character.

In Japan, multiple encoding methods are used depending on platforms. For instance, Shift-JIS coding is commonly used on PC platforms, while EUC (Extended UNIX Code) is used on UNIX platforms. Internal processing of MicroStation is performed in Shift-

JIS. Therefore character values need to be translated by MicroStation from EUC to Shift-JIS whenever text is displayed.

Figure 2.2. Chinese characters.

Codeset Handler

The codeset handler is part of the dialog box manager and handles foreign characters primarily in Far Eastern languages. It accepts multi-byte strings and distinguishes single-byte characters from double-byte characters based on a runtime configuration resource file. The codeset handler also controls cursor movement as well as character insertion and deletion of bidirectional text in Middle Eastern and Asian languages.

Both the codeset handler and the runtime configuration resource are usually included in the MicroStation resource file **ustation.rsc**. The codeset handler can also be loaded as a separate MDL program by setting the `MS_CODESET` environment variable.

The codeset handler is disabled for English and Western European languages. By disabling the codeset handler, MicroStation will not display Asian text properly. For development and testing of applications for Asian markets, it is necessary to obtain a localized version of MicroStation and install the correct fonts. If you work on localized operating systems (Chinese Windows NT and Japanese Macintosh OS), it is also possible to enable the codeset handler with the `SETLANG` utility available from Bentley.

Unicode

The Unicode standard includes over 40,000 characters from the world's scripts in one 16-bit encoding range. This allows simultaneous display of multiple languages in one design session. Also, string processing is simplified because all characters in the world have unique code values.

MicroStation Version 5 has limited Unicode support; only text elements in design files and font symbology resource files can have Unicode characters at this point. Because the dialog box manager does not support Unicode, text displayed in dialog boxes is limited to only one language.

Unicode fonts can be created by importing TrueType fonts, AutoCAD Release 13 shape files and MicroStation cell libraries. TrueType fonts included in Microsoft Windows typically include accented characters used in both West and East Europe. Some fonts (Lucida Sans Unicode) have over 2000 characters including Hebrew, Cyrillic, Greek and other various symbols.

Unicode text elements can be created using the wide character versions of `mdlText_xxx` functions.

Data Input

Internationalized programs must be capable of processing and displaying all the characters that users enter in their native language. A keyboard is most commonly used for entering text. External files, including ASCII text files and resource files, are another form of text input. Concerning keyboard input, no special processing is required in applications because the Dialog Box Manager handles all the work. However, the existing code may be modified if external data files are extensively used.

Keyboard Input

English characters are entered by striking keys on a keyboard. The shift-key distinguishes between upper and lowercase letters. For typing the English alphabet, one keystroke or the combination of a keystroke and the shift-key is sufficient. However, many foreign languages require multiple keystrokes and/or special applications to translate keystrokes for generating one character.

Input of European languages

For many European languages, a shifted or unshifted keystroke is insufficient to produce accented characters because most keyboards only have approximately 100 keys. There are a variety of methods to input special characters that involve multiple keystrokes. The first keystroke often represents an accent mark. It is saved by the keyboard hardware or keyboard software and not sent to an application. After the second keystroke, the two keystrokes are combined into a single character. This process is usually performed by an input method application provided by the operating system.

Input of Asian languages

Input of Asian ideographic languages is much more complicated. In the case of European languages, two or three keystrokes form a unique character. Due to the large number of characters, the input of Asian languages require an intermediate application interpreting between the keyboard and MicroStation. One method of text input is to type phonetic characters from the same 100 key keyboard or with a special keyboard having extra keys.

Because there can be many ideographic characters which match the same phonetic characters, the input application requires a large dictionary and intelligence. The input application may have to display a menu or pop-up dialog box so the user can choose among ideograms with the same phonetic representation.

Figure 2.3. Asian input method.

Getting Input Directly from Users

The input of both European and Asian languages are usually supported by the underlying operating system or by a localized version of MicroStation. The output is usually sent to a text field in the dialog boxes. Therefore, the user input is transparent to MDL applications so long as applications obtain input from dialog box items.

Collecting input directly from users by `mdlSystem_getChar` or standard C functions such as `getch` and `getwc` is discouraged. These functions will not work with input method applications for Asian and some European languages.

Writing Direction

Writing direction varies among languages. In Latin-derived writing systems, characters are entered left-to-right. Arabic and Hebrew writing systems have most characters entered right-to-left.

In Asian languages, characters are traditionally written top-to-bottom as well as left-to-right. Input into dialog box text fields is limited to the standard left-to-right direction, while text elements in designs may be drawn top-to-bottom.

Arabic and Hebrew systems are bi-directional rather than purely right-to-left because numbers and commonly intermixed foreign words are written left-to-right. MicroStation

supports bi-directional keyboard input into dialog box text items. It can be enabled by setting the following environment variable:

SET MS_RIGHTLOGICKB=ON

Pressing Ctrl-K is used to toggle between left-to-right and right-to-left directions. In multi-line text fields, Ctrl-Y is used to display text in right-justified mode.

Figure 2.4. Display text in right-justified mode.

Typing Code Values Directly

Special characters and symbols can be entered using their numerical value. This is useful for testing applications when the keyboard input application or a localized operating system is unavailable. To type a character into dialog text items on the PC, use:

ALT + KEY CODE

For instance, 'ç' can be created by typing:

ALT + 193

To type Asian double-byte characters, the ALT key needs to be released after the first byte. For example, 漢字 (two Japanese characters: 漢 = 138, 191, 字 = 142, 154) can be entered with:

ALT + 138, ALT + 191, ALT + 142, ALT + 154

A sequence of "\digit" can be converted into single byte representation using `mdlText_compressString`. The fourth argument of the function needs to be set to "INTERNATIONAL." MicroStation's text editor uses this function and accepts a sequence of back slashes ('\') followed by decimal character values. For example:

\77\105\99\114\111\115\116\97\116\105\111\110

is same as typing "MicroStation".

To enter Asian double-byte text, character values need to be preceded by \253\255 which indicates a wide character string. The 漢字 string is entered as:

\253\255\138\191\142\154

Although the ALT method works only in localized versions of MicroStation, this method works in the English version where the code set handler is disabled.

File Input

While the keyboard is the most obvious source of text input, MicroStation also encounters characters from database files, UCM files and other text files. If these files are encoded differently in different operating systems, character values must be translated into MicroStation's internal processing code, DEC Multinational ASCII or equivalents, using the character translation table. While keystrokes are usually translated automatically by MicroStation, it is the application's responsibility to apply translation whenever a file is read. Character translation is applied by MicroStation if external database files are accessed by MDL functions such as `mdlDB_writeColumn` and `mdlDB_readColumn`.

To translate text from the external text format to DEC Multinational ASCII, use `mdlCharTran_nativeToInternalDisplay`.

To translate a text string in MicroStation to the external text format, use `mdlCharTran_internalDisplayToNative`. Both functions always return `SUCCESS`.

```
int mdlCharTran_nativeToInternalDisplay
(
char *outstring,  /* <= output, may be same a input. */
char *instring,   /* => input string to be translated */
int nBytes        /* => size of output buffer */
)
int mdlCharTran_internalDisplayToNative
(
char *outstring,  /* <= output, may be same a input. */
char *instring,   /* => input string to be translated */
int nBytes        /* => size of output buffer */
)
```

Currently, there is not a good way to identify what encoding method is used. An application can assume the input file was generated using the same character encoding which the operating system uses.

On UNIX platforms, character translation is usually not required, but if text files are edited on PC and brought to UNIX, they must be translated.

String Processing

An internationalized program must operate regardless of the method of encoding characters in the user's language. Identifying and isolating language-sensitive code sections is a major part of the programmer's work when internationalizing applications. Applications designed for the English language fail to work with other languages because of the character encoding, coding range and character size of English. For instance, a program that ignores or truncates the eighth bit of every character will not work in Europe, because the accented characters used in many European languages are represented with numbers greater than 127. An application that assumes every character is 8 bits long does not work in Asia where there are many thousands of ideographic characters. Furthermore, it is not even safe to assume that characters are of a uniform width, as 16-bit Asian characters are often mixed with 8-bit Latin characters within the same string.

Coding Practices to Avoid

Common language sensitive coding practices can be summarized as below:

Avoid making assumptions about the number of characters in a codeset (no mapping tables with 128 or 256 entries).

```
for (ch=0; ch<128; char++) /* BAD */
  display_character(ch);

int table[256]; /* BAD */
```

In this case, you should not assume the maximum number is 128 or 256. If the maximum number is 128, accented characters in Europe will be ignored. If the maximum is set to 256, Asian characters will be ignored. For Asian languages, the maximum is difficult to predict because it depends on the application and target language. Creating an array of size 14,000 (Maximum number for Taiwanese Big-5 encoding) is not recommended because all 14,000 characters are not necessarily used. Making a `for` loop for 14,000 characters will be computationally expensive.

Avoid assumptions about the size of characters. Use `strlen` only for counting bytes in a string, not characters.

```
textWidth=tcb->textwidth * strlen(string); /* BAD */
stringP=(char*) malloc(sizeof(char) * strlen(string));/* GOOD */
```

In English, `strlen` returns both the number of characters and bytes in a string. But in Asian double-byte text strings, `strlen` returns the number of bytes, and the number of characters is usually less. To count the number of characters, `wcslen`, one of wide-character functions, should be used.

Here is another example of a common error. In this case, a back slash, '\', may be part of a double-byte character in Asian text. To compare a character in a string, the string needs to be first converted to a wide-character format.

```
/* BAD */
char *charP, string[MAX_LENGTH];
for (charP=string; *charP; charP++)
  if (*charP == '\')
       printf (“This is a back slash\n”);

/* GOOD */
MSWideChar *wCharP, wString[MAX_LENGTH];
mbstowcs(wString, string, MAX_LENGTH);
for (wCharP=wString; *wCharP; wCharP++)
  if (*wCharP == '\')
       printf (“This is a back slash\n”);
```

Don't make character comparisons to hard-coded numeric values.

```
/* BAD */
if (value>0x20 && value<0x7f)
  print_char=TRUE;
```

Use `iswprint` instead.

Don't apply conversions to a character code using hard-coded values. For instance, you should not use explicit values when converting characters to lowercase or uppercase.

```
lower_case_char=value | 0x20; /* BAD */
upper_case_char=value & 0xdf; /* BAD */
```

The use of `toupper` and `tolower` is also discouraged. `towupper` and `towlower` should be used instead.

Wide Character Processing

Applications and MicroStation normally exchange text data in multi-byte text format. A multi-byte text string is a zero-terminated string where each character can be made up of any number of bytes. An ASCII string is a special case of a multi-byte string where the character size is always one byte. In Asian languages, character size varies from one to two bytes, or occasionally more. A single string may contain a mixture of single and double-byte characters. A multi-byte string is declared as type `char`.

Applications that need to process text should first convert multi-byte strings to wide character strings before doing any locale-sensitive processing. Wide characters have uniform width.

Locale is the definition of a user's environment that depends on language and cultural conventions. A locale is roughly equal to a language, but there can be multiple locales for one language. For instance, French-speaking people in Quebec would display the date and time in a different manner than people in France. Some wide-character functions return different results depending on a selected locale.

By making all characters uniform in width, the resulting string is easier to manipulate in operations such as indexing, traversing (especially backwards) and size prediction based on the number of characters.

Applications that simply receive a multi-byte string and output it without trying to interpret the contents can remain unchanged. Programming code that receives or returns a `char *` text value can remain unchanged if string processing is limited to `strcpy`, `strcat` and `strcmp`.

Wide Characters

Wide characters are declared as type `wchar_t` in ANSI C and ISO C. The size of a wide character depends on the underlying operating system and compiler. On UNIX platforms such as HP and Sun, `wchar_t` is defined as an `unsigned int`, which is 4 bytes long. On Windows NT, it is defined as an `unsigned short`, which is 2 bytes long. Two bytes of data is usually sufficient to represent the most commonly used Asian and European characters.

MicroStation internally uses `MSWideChar` instead of standard `wchar_t`. The definition of `MSWideChar` is found in an include file **widechar.h**, which is delivered with Version 5 of MDL. `wchar_t` is by definition allowed to be one byte long, which is equal to `char` on non-internationalized platforms. The MetaWare High-C compiler in the PC-DOS environment internally defines `wchar_t` as `char`. Because this prevents the processing of Asian characters, a wide character is defined as 2 bytes long and the name `MSWideChar` is used to avoid confusion with `wchar_t`. Another reason `MSWideChar` is used is to ensure compatibility of MDL code across different platforms.

For the convenience of developers who want to convert existing C programs to MicroStation, `wchar_t` can be also used inside MDL. `wchar_t` is defined to be equal to `MSWideChar` in include file **wchar.h**, which is delivered with MDL. In addition to `wchar_t`, ANSI C and ISO C define `wctype_t` and `wint_t`. These two type definitions are redefined as `MSWideType` and `MSWideInt` (see **widechar.h**).

Conversion of String Format

Text generated by MicroStation's dialog box and external sources (e.g., ASCII text files) is normally encoded as a series of multi-byte characters. To process these internally, they must be converted to wide character representation. Even if applications are intended for only single-byte languages, this conversion will be useful because some of the wide character functions (explained in *Wide Character Processing*) provide locale-sensitive processing such as sorting, time, and data functions.

This statement is currently true only under the Windows NT environment.

After all the processing is complete, resulting strings should be converted back to multi-byte representation for output. While converting strings between multi-byte format and wide character format may sound inefficient, it has to be done until the Dialog Box Manager handles strings in wide character format.

MDL has the following built-in functions for converting between multi-byte format and wide-character format. These functions are defined in **stdlib.h**, delivered with MDL.

In the future, all `mdlDialog_xxx` functions will return strings in wide-character format so that no conversion will be required.

mbstowcs

```
size_t mbstowcs(MSWideChar *wcs, char *mbs, size_t n);
```

This function converts a sequence of characters from the array pointed to by *mbs* into a sequence of corresponding wide character codes. The third argument, *n*, is the size of the output array in the number of wide characters. The function returns the number of wide characters stored in *wcs*.

wcstombs

```
size_t wcstombs(char *mbs, MSWideChar *wcs, size_t n);
```

This `wcstombs` function converts a sequence of wide character codes pointed to by *wcs* to its multi-byte character representation and stores the results in the array pointed to by *mbs*. The third argument, *n*, is the size of the output array in the number of bytes. The function returns the number of bytes stored in *mbs*.

mbtowc

```
int mbtowc(MSWideChar *wchar, char *mbchar, size_t maxLen);
```

This function determines the number of bytes that constitute the character pointed to by *mbchar*, converts this to the corresponding wide character representation and stores the result in the wide character pointed to by *wchar*. The third argument is the size of the output buffer, *wchar*. The number of characters converted is returned.

wctomb

```
int wctomb(char *mbchar, MSWideChar *wchar);
```

This function converts the wide character, *wchar* to its multi-byte character representation and stores the result in the array pointed to by *mbchar*. The function returns the number bytes in the output string.

mblen

```
int mblen(char *s, size_t n);
```

This function determines the number of bytes constituting the character pointed to by *s*. The function evaluates the input string up to *n* bytes which is the maximum number of bytes per character.

Wide Character Functions

International organizations (ISO and X/Open) have defined a set of wide-character functions known as the Worldwide Portability Interfaces (WPI). Wide character functions are sensitive to locale.

In other words, they behave differently depending on the language of choice. Although some functions are unique to WPI, a majority of them have cousins in the standard C libraries that have similar functionality except that they are not sensitive to locale. The functionality of wide character functions depend on the underlying operating system. Because not all the operating systems on which MicroStation runs support these functions, some are sensitive only to English ASCII. The most useful and functional among them are those for character classifications, case conversion and string processing. Function prototypes of these functions are defined in **mswchar.fdf**.

Locale Initialization

According to the WPI, an international environment and locale needs to be selected by executing the setlocale function prior to using any of the wide character functions.

setlocale

```
char *setlocale(int category, char *locale);
```

The `setlocale` function is designed to change or query the program's international environment. The first argument, *category*, specifies all or part of the program locale defined in the header file **locale.h**. The value `LC_ALL` sets the entire locale.

The *locale* argument is a pointer to a character string containing the name of locale. The value "`C`" specifies the minimal environment for C language translation. The value "" specifies that the locale should be initialized to the system default locale. `NULL` is used to direct the function to query the current locale setting and return its name.

In Version 5, this function is available for MDL applications. But it currently has effects only on a limited number of platforms, Windows NT and HP700 workstations. On the other platforms, the wide character functions behave as if `setlocale`(`LC_ALL`, "") were called.

If your application is running under the French version of MicroStation, wide character functions are sensitive to French. If it is running with the Chinese version, the functions will behave in the Chinese manner.

Therefore, MDL applications do not need to call this function until all underlying operating systems support this function. When all operating systems fully support this function, an MDL function will be able to handle multiple languages in one session; handling German and Japanese together.

Character Classification

`iswalnum`, `iswalpha`, `iswcntrl`, `iswdigit`, `iswgraph`, `iswlower`, `iswprint`, `iswpunct`, `iswspace`, `iswupper`, `iswxdigit`

These functions are internationalized versions of the `ctype` functions `isalnum`, `isalpha`, `iscntrl`, `isdigit`, `isgraph`, `islower`, `isprint`, `ispunct`, `isspace` and `isxdigit` respectively, except they accept a wide character as input.

These character classification functions are defined as macros in **widechar.h**.

It is important to notice that the result of character classification in foreign languages is not necessarily the same. For instance, in an English language locale supporting the ASCII codeset, only the characters **a-z** and **A-Z** would be classified as alphabetic. In a Japanese environment, alphabetics may also include double-byte characters.

String Functions

`wcscat`, `wcschr`, `wcscmp`, `wcscpy`, `wcscpn`, `wcslen`, `wcsncat`, `wcsncmp`, `wcspbrk`, `wcsrchr`, `wcsspn`, `wcstok`, `wcsstr`

These functions are wide character versions of `strcat`, `strchr`, `strcmp`, `strcpn`, `strlen`, `strncat`, `strncmp`, `strpbrk`, `strrchr`, `strspn`, `strtok` and `strstr` respectively. These are most frequently used by MicroStation internally along with the `mbstowcs` and `wcstombs` functions.

Case Conversion

`towlower`, `towupper`

These case conversion functions are equivalents of `tolower` and `toupper`. Wide character versions of `strupr` and `strlwr` are not defined in WPI. Therefore, if an equivalent of `strupr` is necessary, you can add a simple routine like below:

```
MSWideChar *wcsupr(MSWideChar *wstring)
{
```

```
    MSWideChar *wcP;
    for (wcP=wstring; *wcP; wcP++)
         *wcP=towupper(*wcP);
    return (wstring);
}
```

Character Collation

`wcscoll`, `wcsxfrm`

The `wcscoll` and `wcsxfrm` functions compare strings according to the local custom of the selected locale. The two functions are functional only on the HP700 workstation. On other platforms, `wcscoll` is the same as `wcscmp` and `wcsxfrm`.

wcscoll

```
int wcscoll(MSWideChar *ws1, MSWideChar *ws2);
```

The `wcscoll` compares sorting order of two strings according to the local custom of a selected locale. It is similar to `wcscmp` and `strcmp`. But both `wcscmp` and `strcmp` simply compare the numerical values of characters in two strings. The function `wcscoll` has more intelligence. For instance, consider the following data:

```
abcdef
abc
abcd
hot
cheque
cull
LLLLLL
ABCDE
lot
czech
cult
efgh
chess
llllll
abcde
ábcd
àbcd
```

If `strcmp` or `wcscmp` is used, the result of sorting would be as follows:

```
ABCDE
LLLLLL
abc
abcd
abcde
```

```
abcdef
cheque
chess
cull
cult
czech
efgh
hot
llllll
lot
ábcd
àbcd
```

But this is not the correct sorting order in French and Spanish. The `wcscoll` function would sort the same data as follows if the selected locale is French:

```
abc
abcd
ábcd
àbcd
ABCDE
abcde
abcdef
cheque
chess
cull
cult
czech
efgh
hot
LLLLLL
llllll
lot
```

In Spanish:

```
abc
abcd
ábcd
àbcd
ABCDE
abcde
abcdef
cull
cult
czech
cheque
chess
```

```
efgh
hot
lot
LLLLLL
llllll
```

In both languages, the ordering of accented characters ('à', 'á') is after the base character ('a'). In Spanish, 'ch' and 'll' are treated as one collating symbol and placed at the end of 'c' and 'l', respectively.

wcsxfrm

```
size_t wcsxfrm(MSWideChar *ws1, MSWideChar *ws2, size_t n);
```

The `wcsxfrm` function transforms a wide character string pointed to by *ws2* and places the result in the array pointed to by *ws1*. The transformation is such that if the `wcscmp` function is applied to two transformed strings, the result is the same as if the strings had been compared using `wcscoll`. The third argument, *n*, is the size of the output string, *ws1*, including a terminating null wide character. The function returns the number of characters stored in *ws1*.

Date and Time Functions

`wcsftime`

WPI defines `wcsftime`, which is an internationalized version of `strftime`. The locale affects the display format of date and time. For example, the display format of a date in the English locale is:

Month/Day/Year

For the French locale, it would be:

Day/Month/Year

wcsftime

```
size_t wcsftime(MSWideChar *wcs, size_t maxsize, char *format,
            struct tm *timeptr);
```

The `wcsftime` function is functional only on HP700 workstations. On other platforms, the function is not sensitive to locale or returns the same result in wide character format as `strftime` does. Therefore, it is recommended to instead define a format string in an external resource file and construct date and time using `sprintf` and numbered argument lists as explained in Messages and Resource Files.

Printing and Scanning Functions [not supported]

`fwprintf`, `wprintf`, `swprintf`, `vfwprintf`, `vwprintf`, `vswprintf`, `fwscanf`, `wscanf` and `swscanf`

These printing and scanning functions are wide character versions of `fprintf`, `printf`, `sprintf`, `vfprintf`, `vprintf`, `vsprintf`, `fscanf`, `scanf` and `sscanf` respectively. These functions are currently not supported by MDL.

Number Conversion Functions

`wcstod`, `wcstol`, `wcstoul`

These number conversion functions are wide character versions of `strtod`, `strtol` and `strtoul`.

Input and Output Functions [not supported]

`getwc`, `getwchar`, `getws`, `fgetwc`, `fgetws`, `fputwc`, `fputws`, `putwc`, `putwchar`, `putws`, `ungetwc`

These input and output functions are wide character versions of `getc`, `getchar`, `gets`, `fgetc`, `fgets`, `fputc`, `fputs`, `putc`, `putchar` and `ungetc`. It is not recommended to use these functions from MDL. The functions that perform input operations will not work with foreign languages because input method applications often interact with the Dialog Box Manager. See “Getting Input Directly from Users” on page A-8. The output functions will not work unless the underlying operating system uses the same encoding method. MDL applications should perform character input and output via the dialog boxes.

Limitations

A string constant or a character constant in wide character format is currently not supported by MDL compilers.

In the conventional string processing, string and character constants can be used as:

```
char *string = “regular string”;
char ch = 'A';
sprintf(string, “value = %d”, value);
strcat(string, “32”);
```

In a wide character context, the same lines can be rewritten as:

```
wchar_t *wString = L“wide character string”;/* not supported */
wchar_t wch = L'A';                         /* not supported */
swprintf(wString, L“value = %d”, value);    /* not supported */
```

```
wcscat(wString, L"32");                    /* not supported */
```

Here, the actual strings in quotes are in multi-byte format. The L"....." syntax converts the format to wide character format. Unfortunately, the L "....." syntax is not currently supported.

To work around this problem, all format strings should be saved in an external resource file. Then, format strings should be converted to wide character format using `mbstowcs`. This should not cause too much inconvenience because hard-coded strings are to be generally avoided.

Assignment of wide character constants can be done in the conventional method without the L prefix as `long` as the constant is one-byte long.

```
wchar_t wch = 'A';
```

Character constants of double-byte length are rarely used.

Performance Considerations

There exists an efficiency versus portability issue which should be kept in mind. Currently, the Dialog Box Manager and external data sources use multi-byte text format. Converting multi-byte strings to wide-character format, and then converting back to multi-byte format for output can be inefficient. The processing of Asian characters can be slower than the processing of English and other European languages. Although not desirable, if performance is your priority, it is possible to create two functions. The first function would do standard ASCII processing and the other would handle wide-character processing. An MDL global variable, `multiByteCharsPossible`, tells us if Asian double-byte text is used. For example:

```
if (multiByteCharsPossible)
{
   mbstowcs(wString, string, MAX_STRING_LENGTH);
   processWideString(wString);
   wcstombs(wString, string, MAX_STRING_LENGTH);
}
else
   processNormalString(string);
```

Programming Example

Many examples distributed with MicroStation work with English and European languages, but fail to work with Asian languages. For instance, the CHNGTXT example fails in `compareTextStrings` when `strstr` is applied to Asian double-byte character strings. Suppose there are two Japanese text strings:

1-a) 剣 (0x8c, 0x95) 道 (0x93, 0xb9)

2-a) 白 (0x94,0x92) 血 (0x8c,0x8c) 病 (0x95,0x61)

The first string consists of two double-byte Kanji characters and means “fencing”. The second consists of three double-byte Kanji characters and means “leukemia”. If a user wants to replace the first character, 剣 (0x8c,0x95), with 柔 (0x8f,0x5f), `strstr` returns a correct result for the first string.

1-b) 柔 (0x8f,0x5f) 道 (0x93,0xb9) which means “Jyudo(wrestling)”.

But the `strstr` matches 柔 (0x8f,0x5f) with a wrong part of 白 血 病 and converts the string into a series of meaningless characters although no replacement should take place:

2-b) 白 (0x94,0x92) 件 (0x8c,0x8f), _(0x5f) a(0x61)

The third double-byte character, BYO, was converted to two ASCII characters.

This error happens because `strstr` does not recognize character boundaries and matched a double-byte character to the second byte of 血 and the first byte of 病 .

Another error can occur with the `strupr` function in `compareTextStrings` where part of a double-byte character may be recognized as a single ASCII character. Subsequently it is converted to an incorrect double-byte character. For instance, `strupr` will incorrectly convert 各 (0x8a,0x65) (meaning = “each”) to 界 (0x8a,0x45) (meaning = “world”).

These problems can be avoided by first converting strings into wide character representations. The `changeTextElement` and `compareTextStrings` functions in CHNGTXT.MC have been rewritten using some of the wide-character functions explained earlier.

The original version of `changeTextElement` and `compareTextStrings` in CHNGTXT.MC look like:

```
/*----------------------------------------------------------------+
|                                                                 |
|name   changeTextElement                                         |
|author   BSI     8/89                                            |
|                                                                 |
+----------------------------------------------------------------*/
Private int changeTextElement
(
MSElementUnion *element,
int doChange
)
{
```

```
char tempText[256],oldText[256],newText[256],message[512];
char *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;
nChars=strlen(oldText);
strlenNewString=strlen(chTextInfo.newString);
strcpy(newText, oldText);
strP=newText;
while (!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_openAlert(message);
     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 (doChange)
        mdlText_create(element, element, newText, NULL,
                       NULL, NULL, NULL, NULL);
    return retval;
}
/*----------------------------------------------------------------+
|                                                                 |
|name   changeTextStrings                                         |
|author   BSI     8/90                                            |
|                                                                 |
+----------------------------------------------------------------*/
Private int compareTextStrings
(
char  *searchStr,
char  *patternStr,
char  **start,
char  **end
)
{
    static char *space = "";
    char searchBuf [256], patternBuf [256];
/*----------------------------------------------------------------+
 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;
}
```

Wide character versions of the same functions are listed below. `changeTextElementW` is the wide character version of `changeTextElement`. `compareTextStringsW` is the wide character version of `compareTextStrings`.

```
/*---------------------------------------------------------------+
|                                                               |
|name   changeTextElementW                                      |
|author BSI            8/89                                     |
|                                                               |
+---------------------------------------------------------------*/
Private int changeTextElementW
(
MSElementUnion *element,
int doChange
)
{
    MSWideChar tempTextW[256];
    MSWideChar oldTextW[256];     /* old string of text element */
    MSWideChar newTextW[256];     /* new string of text element */
    MSWideChar oldStringW[128];   /* old string in dialog box */
    MSWideChar newStringW[128];   /* new string in dialog box */
    MSWideChar *wcsP, *start, *end;
    char newTextA[256];           /* ASCII version of newTextW */
    char message[512];
    int status, stringChanged=FALSE, strlenNewString;
    int retval=SUCCESS, nChars, nEDFs;
    TextEDParam edParam;
    if (mdlElement_getType(element) != TEXT_ELM
        || mdlText_extractString(oldTextW, element))
        return ERROR;
    if (mdlText_extractWide(oldTextW, NULL, NULL, NULL, NULL, NULL,
        &edParam, element) != SUCCESS)
        return ERROR;
    if (*chTextInfo.oldString==0)
        return ERROR;
    mbstowcs(oldStringW, chTextInfo.oldString, 128);
```

```
    mbstowcs(newStringW, chTextInfo.newString, 128);
    nChars=wcslen(oldTextW);
    strlenNewString=wcslen(newStringW);
    wcscpy(newTextW, oldTextW);
    wcsP=newTextW;
    while (!compareTextStringsW(wcsP, oldStringW, &start, &end))
    {
         if ((wcslen(newTextW)+strlenNewString+wcslen(end))
             *sizeof(MSWideChar) >= sizeof(newTextW)/sizeof(MSWideChar))
         {
              retval=ERROR;
              mbstowcs(newTextA, newTextW, 256);
              sprintf(message, "Unable to insert '%s' after '%s'",
                      chTextInfo.newString, newTextA);
              mdlDialog_openAlert(message);
              break;
         }
         wcscpy(tempTextW, end);
         wcscpy(start, newStringW);
         if (tempTextW[0])
              wcscat(wcsP, tempTextW);
         wcsP = start+wcslen(newStringW);
         stringChanged=TRUE;
    }
    if (!stringChanged)
         return ERROR;
    if (edParam.numEDFields && wcslen(newTextW) < nChars)
    {
         int nNewLen=wcslen(newTextW);
         wcsP=newTextW+nNewLen;
         while (nNewLen<nChars)
         {
              *wcsP = ' ';             /* can't use L' ' here */
              ++wcsP;
              ++nNewLen;
         }
         *wcsP = '\0';
    }
    if (doChange)
    mdlText_createWide(element, element, newTextW, NULL, NULL,
                       NULL, NULL, NULL);
    return retval;
}
/*----------------------------------------------------------------+
|                                                                 |
|name wcsupr                                                      |
|author BSI      8/90                                             |
```

```
|                                                                |
+----------------------------------------------------------------*/
Public MSWideChar *wcsupr
(
MSWideChar *wstring
)
{
   MSWideChar *wcP;
   for (wcP=wstring; *wcP; wcP++)
   {
        *wcP=towupper(*wcP);
   }
   return (wstring);
}
/*----------------------------------------------------------------+
|                                                                |
|name compareTextStringsW                                        |
|author BSI       8/90                                           |
|                                                                |
+----------------------------------------------------------------*/
Private int compareTextStringsW
(
MSWideChar *searchStr,
MSWideChar *patternStr,
MSWideChar **start,
MSWideChar **end
)
{
   static MSWideChar space[2];
   MSWideChar searchBuf [256], patternBuf [256];
/*----------------------------------------------------------------+
 Initialize space string, this is not very good
 MSWideChar *space= L“”; does not work!
 In ASCII version, it was static char *space = “” ; did all.
+----------------------------------------------------------------*/
   space[0]=' ';  /* space */
   space[1]='\0'; /* terminate string */
/*----------------------------------------------------------------+
 Prepare for Whole Word search if necessary
+----------------------------------------------------------------*/
   searchBuf[0]=0;
   patternBuf[0]=0;
   if (chTextInfo.wholeWords)
   {
        wcscat(searchBuf, space);
        wcscat(patternBuf, space);
   }
```

```
    wcscat(searchBuf, searchStr);
    wcscat(patternBuf, patternStr);
    if (chTextInfo.wholeWords)
    {
         wcscat(searchBuf, space);
         wcscat(patternBuf, space);
    }
/*---------------------------------------------------------------+
 Prepare for case insensitive search if necessary
+---------------------------------------------------------------*/
    if (!chTextInfo.matchCase)
    {
         wcsupr(searchBuf);
         wcsupr(patternBuf);
    }
/*---------------------------------------------------------------+
 Attempt to match substrings
+---------------------------------------------------------------*/
    if ((*start=wcsstr(searchBuf, patternBuf)) == NULL)
         return ERROR;
    *start=searchStr + (*start-searchBuf);
    *end=*start + wcslen(patternStr);
    return SUCCESS;
}
```

The wide character version works with Asian text, but is not efficient. This example exposes the limitations of internationalization support by MicroStation very well.

Firstly, the *chTextInfo* variable, published by `mdlDialog_publishComplexVariable` in `main` cannot accept wide character strings because the dialog box manager currently does not support wide character format. Therefore, *chTextInfo.oldString* and *chTextInfo.newString* need to be represented by multi-byte format and converted to wide character format by `mbstowcs` before processing.

The second problem is the `wcsupr` function. Most of `char` variables were replaced by `MSWideChar` variables. The conventional string functions, `strcpy`, `strlen`, `strcat` and `strstr` were replaced by wide character functions, `wcscpy`, `wcslen`, `wcscat` and `wcsstr`. However, `wcsupr` had to be created using `towupper` because the wide character version of `strupr` is not defined in WPI.

The third problem is the `mdlDialog_openAlert` function in `changeTextElementW`. Most MDL functions do not accept wide character strings. Therefore, any wide character strings need to be converted to multi-byte format by `wcstombs` before calling the MDL functions.

The only exceptions are the `mdlText_xxxWide` functions. For instance, `mdlText_extractWide` and `mdlText_createWide` are used in `changeTextElementW`.

The arguments are not exactly the same between `mdlText_create` and `mdlText_createWide`. For details, refer to the "MicroStation MDL Function Reference Manual."

The assignment of character constants in `changeTextElementW` does not use the L prefix as explained in Limitations.

Display and Output

Applications can display messages and prompts in dialog boxes employing the same MDL functions used for English applications. Text elements can also be created in design files using the same MDL functions, although Version 5 has some new functions and parameters useful for foreign languages. Generally, major modifications are unnecessary to existing program code. Applications should not hard code font name and number as they vary from language to language.

To display translated text, correct fonts need to be installed and selected. Font libraries for foreign languages are usually included in localized versions of MicroStation. If there is no localized version of MicroStation, it may be necessary to create a new font for your applications.

Output devices like plotters usually recognize text elements as lines and shapes when a design file is output. Therefore, no special precaution is required to output text in foreign languages.

Fonts

MicroStation uses two different types of font libraries: raster font and vector font libraries. The term *Font libraries* generally refers to the latter, as users cannot change the raster font. In Version 5, the vector font libraries are called font symbology resource files since they are saved in resource file format.

The raster font library contains bit-map images of characters used for displaying text in dialog boxes. The vector font library contains line strokes of the same character sets for displaying and plotting text elements in a design. The two libraries are created in different ways, and often have slightly different character sets. Since they have completely different forms, they cannot be translated or exchanged. For example, the Font Installer does not let you install a TrueType font or a PostScript font as a dialog box font.

Raster Font (X-Window BDF Font)

MicroStation uses a raster font for displaying window titles and dialog items. The DOS and EnvironV versions of MicroStation have their own raster font library rastfont.lib.

rastfont.lib was generated from public domain fonts from the X Window system. On other platforms, MicroStation directly uses the raster font available on the operating systems. For instance, the Macintosh version uses fonts defined in the System File and the X Windows version uses fonts defined in the X Windows Window Manager. In general, users are unable to change raster font names although font sizes can be selected from the User Preference dialog box.

Creating a New Raster Font

If a raster font for a particular language is unavailable, it may be necessary to create a new one. To do so, you need to obtain or create the desired fonts in X Windows BDF (Basic Distribution Format) and then convert them to SNF (Server Natural Format) binary format. The SNF conversion is carried out using a program called bdf2snf which Bentley can provide. The SNF files are then put into rastfont.lib using the MicroStation MDL librarian.

Creating a BDF file

The BDF format is an ASCII text format which defines font characteristics and character definitions as specified by the MIT X Consortium and Adobe Systems, Inc. The documentation "Bitmap Distribution Format 2.1" provides the necessary information describing the format of the BDF files. It is recommended that unless you are familiar with the BDF file format, that you modify an existing font definition.

```
STARTFONT 2.1
FONT -Misc-Fixed-Medium-R-Normal--14-130-75-75-C-140-JISX0208.1983-0
SIZE 14 75 75
FONTBOUNDINGBOX 14 14 0 -2
STARTPROPERTIES 19
FONTNAME_REGISTRY ""
FOUNDRY "Misc"
FAMILY_NAME "Fixed"
WEIGHT_NAME "Medium"
SLANT "R"
SETWIDTH_NAME "Normal"
ADD_STYLE_NAME ""
PIXEL_SIZE 14
POINT_SIZE 130
RESOLUTION_X 75
RESOLUTION_Y 75
SPACING "C"
AVERAGE_WIDTH 140
CHARSET_REGISTRY "JISX0208.1983"
CHARSET_ENCODING "0"
DEFAULT_CHAR 165
FONT_DESCENT 2
```

```
FONT_ASCENT 12
COPYRIGHT "Public domain font. Share and enjoy."
ENDPROPERTIES
CHARS 7052
STARTCHAR 0001
ENCODING 1
SWIDTH 42 0
DWIDTH 7 0
BBX 7 14 0 -2
BITMAP
00
00
18
18
3C
3C
7E
7E
3C
3C
18
18
00
00
ENDCHAR
STARTCHAR 0002
..........
```

Compiling a BDF file into a SNF file

A BDF font definition font can be compiled into the SNF format by:

```
bdftosnf <bdf_file_name> -o <snf_file_name>
```

The output font file names have to be of the format **xxxxxxNN.snf** (x's represent some contraction of the font series name and NN represents the point size of the font). For example, to compile **helvr14.bdf**, execute the following command:

```
bdftosnf helvr14.bdf -o helvr14.snf
```

Adding the SNF file to the font library

To add the SNF font to the library, execute the `mlib` command as shown below:

```
mlib -a rastfont.lib <snf_file_name>
```

If the user wants to retain the ability to select point sizes, he will have to put the whole series of font sizes into **rastfont.lib**.

Changing Default Font Series and Sizes

To make MicroStation use the new font series, certain fields in the user preference structure have to be changed with a simple MDL program or the CHNGFONT application available from Bentley Systems, Incorporated. In the example below, **username.upf** is the name of a user preference file. In Version 5 the user preference files have the file extension **.upf** and are usually found in **CONFIG/USER**. The font series name and font sizes also need to be changed to fit your situation. In this example, **newfont14.snf** is the name of a SNF file.

To change the default font definition for MicroStation:

```
#include <basedefs.h>
#include <rscdefs.h>
#include <userpref.h>
RscFileHandle userPrefsH;
UserPrefs *userRscP=NULL;
int status;
mdlResource_openFile(&userPrefsH, "username.upf", RSC_READWRITE);
/* read MicroStation's current font specification */
userRscP=(UserPrefs *)mdlResource_load(userPrefsH,RTYPE_USERPREFS,-1);
strcpy(userRscP->systemRoot,"newfont");         /* system font */
strcpy(userRscP->titleRoot,"newfont");          /* title font */
strcpy(userRscP->dlogNormRoot,"newfont");       /* dialog normal font */
strcpy(userRscP->dlogBoldRoot,"newfont");       /* dialog bold font */
strcpy(userRscP->dlogFixRoot,"newfont");        /* fixed-width font */
strcpy(userRscP->dlogFixBoldRoot,"newfont");    /* fixed bold font */
strcpy(userRscP->buttonFont,"newfont");         /* dialog button font */
userRscP->systemPointSize[0] = userRscP->systemPointSize[1] = 14;
userRscP->titlePointSize[0] = userRscP->titlePointSize[1] = 14;
userRscP->dialogPointSize[0] = userRscP->dialogPointSize[1] = 14;
userRscP->buttonPointSize[0] = userRscP->buttonPointSize[1] = 14;
/* Write the updated resource */
mdlResource_write(userRscP);
if (userRscP)
  mdlResource_free(userRscP);
if (userPrefsH)
  mdlResource_closeFile(userPrefsH);
```

Common Problems

When dealing with foreign language fonts, you may see one of the following problems related to the raster font library.

- MicroStation fails to start and displays an error message, “Unable to locate font in raster font library”.
  This error message indicates that MicroStation could not load the font specified in the user preference setting.
  For DOS and EnvironV versions, the location and contents of rastfont.lib need to be checked. The location of rastfont.lib is specified by the MS environment variable `MS_RFONT`. Check if rastfont.lib exists at the location to which `MS_RFONT` points. In Version 5 the default location is the root directory of MicroStation on Version 5.

The contents of **rastfont.lib** can be viewed by the following command:

```
mlib -t rastfont.lib
```

On platforms other than DOS and EnvironV, check system settings. For example, the `xlsfonts` command can be used on the X Window System. On Windows NT, check the Control Panel's font setting. On the Macintosh, check the System Folder.
If the raster font library is switched from one to another, deleting old ***.upf** files in **/CONFIG/USER** often resolves the problem. If deleting the old ***.upf** files does not resolve the problem, view the contents of the ***.upf** files by the resource `dump` utility. To do so, type:

```
rdump -t'Uprf' -v username.upf
```

- Rainbow color noise is displayed when typing European accented or Asian characters and MicroStation hangs. This error occurs because the character you typed is missing in the font library. On DOS and EnvironV, the showsnf utility can be used to dump the contents of a SNF file. On other platforms, different tools are available for dumping the contents of system fonts.
- Wrong characters are displayed. This error happens when an incorrect character translation table is applied or glyphs in the font are incorrectly mapped. If the language in use is a European language and its internal encoding method is based on DEC Multinational ASCII, the raster font needs to be encoded in DEC Multinational ASCII. In the case of Asian language text, the codeset handler may be disabled (see the section titled, Codeset Handler). If this is the case, an Asian double-byte character is displayed as two single byte characters. A localized version of **ustation.rsc** needs to be installed to fix the problem.
- Symbols are not correctly displayed. Character values for some symbol characters are defined in the User Preference resource file. The values may not match the values in the font library, (see **userpref.h**).

```
editorEOLChar     /* end of line in text editor */
degreeCharDialog  /* degree symbol used in dialog box */
degreeCharDesign  /* degree symbol used in design text */
plusMinusChar     /* +/- symbol for dimension */
```

These characters cannot be edited from the User Preference dialog box. If necessary, their values need to be evaluated or changed using the method explained in Creating a New Raster Font.

- DOS Window does not display correct characters. The DOS Window (DOS version only) uses raster font available in system ROM instead of **rastfont.lib**. Bitmap images of characters are usually stored in code-page information files, also called font files and have a **.cpi** extension. In order to display foreign characters (non-Asian languages), the correct code page needs to be selected. For Asian double-characters (Chinese, Korean and Japanese), the DOS Window uses the **rastfont.lib** for display.

Vector Font (Font Symbology Resource)

MicroStation's vector font library contains stroke elements to represent text in design files. In Version 5, font libraries are resource files, and typically have the extension **.rsc**. The old and new font files are not interchangeable.

Character Mapping of Vector Fonts

Vector fonts do not necessarily have the same character mapping as raster fonts. Almost all vector fonts have the same characters in the lower half of ASCII, range (0x20 - 0x79). Many fonts have undefined characters in the upper half of ASCII, (0x80 - 0xff). In the standard font symbology resource file, **font.rsc**, only Font #32, 90, 91 and 92 have accented European characters.

Several fonts have fraction symbols in the range, (0x81 - 0xbf). Fractions are used in the United States but not in most foreign countries where metric measurement is utilized. Therefore, this area may be used for foreign characters.

Some common symbols in the standard fonts are mapped to different code values from the ones in the raster font libraries because of the existence of fraction symbols. For instance, the degree symbol is mapped to 0x5e which is a caret in the raster font. But in foreign fonts, which have no fraction symbols, the degree symbol is mapped to 0xb0, the same value as in the raster font. Similarly, the +/- symbol is mapped to 0xc0

in English fonts with fractions, and 0xb1 in foreign fonts. Character mapping of vector fonts can be viewed with the FONTEDIT application.

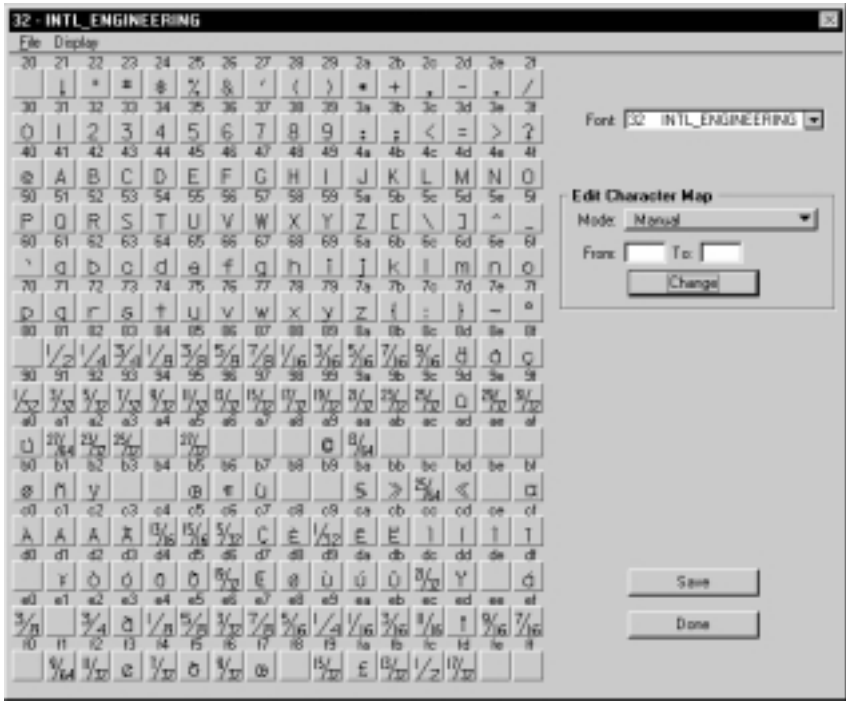


Figure 2.5. Viewing character mapping with FONTEDIT.

Creating Vector Font Libraries for Foreign Languages

In Version 5, industry-standard fonts such as PostScript Type1, TrueType fonts and AutoCAD shape files can be imported into MicroStation. These fonts often have different character mappings. Therefore, character positions may need adjustment using the FONTEDIT utility after fonts are imported.

A vector font can also be created from a font cell library. When creating a foreign font, it is necessary to define /international and /nofractions switches in $FONT$ cell.

Common Problems

When dealing with foreign language version of vector fonts, you may see one of the following problems related to the vector font library.

- Wrong characters are displayed or no character is displayed.
  In Version 4, if a text string contained a character not available in the font, MicroStation displayed an asterisk '*'. If the font did not contain an asterisk, (oct052/dec42), MicroStation would not display a character. Version 5 displays a question mark in a diamond if a character is unavailable. This problem can be corrected by installing or selecting the correct font.
- Characters are overlapped when text is drawn in the vertical direction.
  To display text vertically, a font needs to be designed so each character fits inside the character tile box. Often, glyphs for 'j', 'g', 'p' and 'y' are drawn below the baseline of the tile box. Because MicroStation aligns each character based on tile box height, some characters may overlap with the next character unless inter-character spacing is defined. It is also recommended that each glyph be centered inside the character tile box. Otherwise, each character will not align in the vertical direction.

Text Placement in Design Files

The existing MDL functions can be used to place and manipulate text in any foreign language if the correct font number is selected. Applications should never hard-code font names or font numbers when creating text elements.

In addition, Version 5 offers new text functions for MDL applications. The new functions are extended versions of existing text functions. The new functions accept strings in wide-character format and also support new text attributes such as slant, inter-character spacing, underline and vertical direction. Vertical text is useful in Asian user data. Inter-character spacing is also used for drawing Asian text because there is no word spacing in Asian text writing. To align text, an equal amount of text is inserted between characters.

There are ten new MDL functions for manipulating text elements:

```
mdlText_createWide

mdlText_extractWide

mdlText_extractStringWide

mdlTextNode_createWide

mdlTextNode_createWithStringsWide

mdlTextNode_extractWide

mdlText_extractStringsFromDscrWide

mdlText_addStringsToNodeDscrWide
```

`mdlText_expandStringWide`

`mdlText_compressStringWide`

In these functions, the string format has been changed from `char` to `MSWideChar`. The `TextParam` structure was replaced by the `TextParamWide` structure defined in **mdl.h**. It defines the text element's inter-character spacing direction (horizontal or vertical), slant and underline.

The order of function arguments has been rearranged in some of these functions.

```
typedef struct textParamWide
{
   int       font;
   int       just;
   int       style;
   int       viewIndependent;
   int       nodeNumber;    /* for textnode only */
   double    slant;         /* valid if flags.slant = 1 */
   double    lineSpacing;        /* for textnode only */
   double    characterSpacing;   /* if fixedWidth/interCharSpacing */
   double    underlineSpacing;   /* valid if flags.underline */
   TextDrawFlags  flags;
} TextParamWide;
```

The values for inter-character spacing, slant and underline spacing are valid only if corresponding flags are set in *flags*. Vertical direction is also set by *flags*. For details, see the MDL Function Reference Manual and also the **create.mc** example in the **/MDL/EXAMPLES/DOC** directory.

Plotting and Printing

When design files are sent to output devices such as plotters and printers, text elements are usually translated into stroked elements. Therefore, applications have nothing to worry about.

The Comment line in the Plotting Option does not work with most foreign languages unless the plotting device supports foreign text output.

Messages and Resource Files

The proper treatment of message strings and dialog items is an essential part of localization. All user visible embedded text in an application must be externalized.

Your application and makefile must be set up properly so that it can be translated in a timely and convenient manner. Development of applications and translation of message strings often take place at different locations. Therefore, it is important to separate all text strings in .mc and dialog box resources into message resource files so they can be recompiled by the translators.

Untranslatable Strings

In principle, every text string to which users are exposed, needs to be translated and therefore should be externalized from program code. For example, error messages, prompts, yes/no responses and all labels in dialog boxes (including window titles) should be placed in resource files. Many users prefer to create filenames in their native languages for design files, cell libraries and reference files. This is also true for level names and database table names. There are some cases where strings should not be translated.

Cell names

While a cell library file name can be in a foreign language, names of each cell are restricted to Radix-50 for historical reasons. Radix-50 is a subset of ASCII which includes only upper case letters, (A-Z, 0-9) and just three special characters (space, '$' and '.'). This is probably the most disliked feature of MicroStation by users in foreign countries.

Database column names (dBASE only)

Currently, database column names cannot be displayed in foreign languages. However, column values can be entered and displayed in foreign languages.

MDL Command Names

Command names are usually not translated, although a translated word can be typed as an argument after a command name.

Dialog Box Design

When a string is translated into another language, its length can sometimes shrink, but normally it expands. The extent of expansion depends on the language and the length

of the original text. When there is not enough space, translators are forced to abbreviate the string. This is often confusing to users.

Figure 2.6. Translated text becomes longer than original English text.

Leaving Extra Space for String Expansion

The best solution is to leave enough white space when a dialog box is created. Usually it is adequate if 30 - 50% of extra space is designated for expansion. Developers tend to make push buttons too narrow. There is often not enough space to the left of text field labels. In this case, translated strings may need abbreviation. To avoid this, text items and slider items can be shifted to the right, and the push buttons can be made wider.

Figure 2.7. Not enough space on the left of the two text fields and the buttons are too narrow.

Figure 2.8. Better design of the same dialog box.

Dialog Box Aspect Ratio

Although it is desirable to leave extra space in dialog boxes, this method does not always work. It is difficult to leave white space for a large dialog box with many dialog items.

The only *perfect* way to fit all translated strings without the dialog box editor is to edit the **.r** resource files and change the position and size of individual dialog items. This is difficult and time consuming, because translators are usually not familiar with MDL and resource file format.

An easier (although more approximate) method is to change the aspect ratio for each dialog box individually. The aspect ratio for each dialog box is defined at the top of a dialog resource as:

```
/* myappdlg.r: */
#undef XC
#define XC (DCOORD_RESOLUTION/2)*ASPECT_MYDIALOGBOX
#define DBOX_WIDTH57*XC
#define DBOX_HEIGHT14*YC
#define XW 17*XC
#define BW 12*XC
DialogBoxRsc DIALOGBOXID_MyDialog=
{
   DIALOGATTR_DEFAULT, DBOX_WIDTH, DBOX_HEIGHT, NOHELP, MHELP,
   NOHOOK, NOPARENTID, TXT_DialogBoxTitle,
{
{{4*XC, 9*YC, TW, 1*YC}, Text, TEXTID_MyText, ON, 0x1, "", ""},
{{3*XC, 11*YC, BW, 0}, PushButton, PUSHBUTTONID_, ON, 0, "", ""},
...
```

The aspect ratio value (`ASPECT_MYDIALOGBOX`) is defined in a dialog box text include file under the **/ENGLISH** sub-directory file (**myapptxt.h** in Figure 2.13). The default value of the aspect ratio is 1.0.

```
/* myapptxt.h: */

#define ASPECT_MYDIALOGBOX 1.0
#define TXT_DialogBoxTitle "My Dialog Box"
#define TXT_TextFieldLabel "My Text Field"
#define TXT_PushButtonLabel "My Push Button"
...
```

The value is adjusted by translators, usually to 1.2 - 1.5. Because size and position of each dialog box defined is based on `XC`, the entire dialog box expands, leaving more space for translation.

Whenever you create a new dialog box, always add an aspect ratio. In order for the aspect ratio to work, the size and width of all dialog boxes and dialog items need to be defined using the `XC` macro. Do not use absolute values.

If your program dynamically adjusts the size of a dialog box or dialog items, do not use fixed sizes. It is necessary to measure the actual size of dialog items with `mdlDialog_stringWidth` in the hook function.

If there is more than one dialog box in a resource file, definitions of dialog items should be defined immediately after the dialog box definition to which they belong (Figure 2.12). It is bad practice to define all dialog items at the end (Figure 2.11) because different aspect ratio values might be defined. This results in some dialog items being scaled incorrectly. If two or more dialog boxes have to share the same dialog items, the aspect ratio needs to be shared by the dialog boxes.

The aspect ratio sometimes does not work for a dialog box which has a generic item. This is because the hook function of the generic item often assumes the size of the dialog box does not expand. The size of the generic item should be measured inside its hook function. If a generic item should not expand horizontally, the `YC` macro needs to be used to define the width of the generic item.

Figure 2.9. Aspect Ratio = 1.0.

Figure 2.10. Aspect Ratio = 1.5.

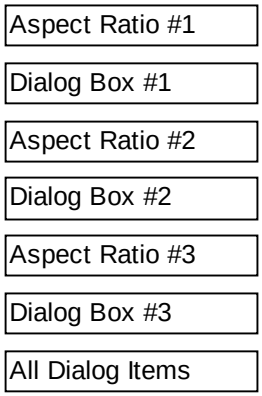


Figure 2.11. All dialog items defined at the end.

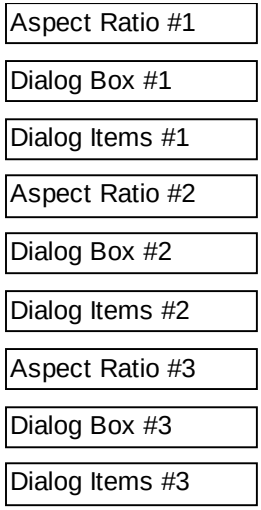


Figure 2.12. Dialog items defined separately.

Source File Organization

Only files with translatable strings and dialog box resource files should be given to translators. The set of such files is called a translation kit. The files should be organized so translators can recompile them without source, **.mc**, files. If source files are not properly organized errors often occur when files are extracted from the development source tree to create a translation kit.

Many developers are aware that hard-coding strings in source code is bad practice for translation. However, many do not know that larger numbers of untranslatable strings are caused by makefile errors or improper file organization. For instance, one message resource file located in a wrong make file can result in 50 untranslatable strings. The damage is far larger than a couple of hard-coded strings.

A makefile is typically divided into two parts. One part that makes the string-free portions of the application **.mke**, and the other **.mki** file that makes the language dependent parts. The **.mki** file is included from the **.mke** file. The **.mki** file makes the final application **.ma** file. All the include files, string list resource files and the makefile (**.mki** only) are given to translators. After translation, translators execute the **.mki** makefile.

A typical source file structure is shown below:

```
        /mdlapps/myapp/ - (non-language dependent source files)
         myapp.mke      - make file for language independent files
         myapprsc.mki   - make file for language dependent files
         myapp1.mc      - program file #1
         myapp2.mc      - program file #2
         myapp.h        - program header for.mc files
         myappdlg.h     - dialog IDs for myappdlg.r
         myappdlg.r     - dialog box definition
         myappcmd.r     - command resource
         mynotran.r     - other non-translatable resource
         mytypdef.mt    - type definition file
    /mdlapps/myapp/english/ - (directory for language dependent files)
         myappmsg.r     - message list used by myapp1.mc and myapp2.mc
         myapptxt.h     - label strings for myappdlg.r
```

Program objects and non-language dependent resource objects are merged into a **.mi** file and given to the translators. A translation kit includes the following files:

```
myapprsc.mki
myapp.mi- string-free object
myappdlg.h
myappdlg.r
myappcmd.h
myappmsg.r
myapptxt.h
```

The relationship of source files and object files is shown in Figure 2.13.

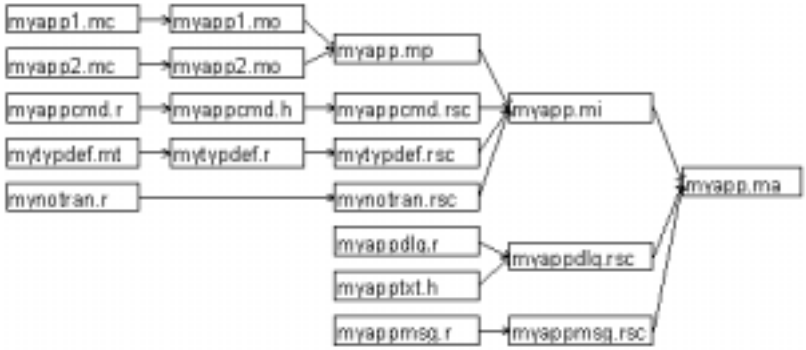


Figure 2.13. Build process of MDL source files.

Dialog box resource files (**myappdlg.r** in this example) and message resource files (**myappmsg.r**) should always be built by a **.mki** file. A typical error developers make is to include these resource files in the **.mke** file.

Another common error is that the text header file (**myapptxt.h**) is directly included in MDL source files, **.mc**. In this case, English strings are built into the **.mi** file even if the

strings are not hard-coded. Because translators are not able to rebuild **.mi** files, it is impossible to translate the strings. Never include a string header file into program source files.

When a new MDL program is written, it is good to start with one of the makefiles for one of the MicroStation MDL samples.

Strings in Program Source (.mc) Files

All strings in **.mc** source files should be stored in separate message resource files, **.r**. Some people use an ASCII text file or a binary file to store strings. But it is not recommended because it might provide difficulties for automatic translation tools.

Program data and other strings which should not be translated must be placed in a separate resource file so they are not translated by mistake. A program often fails because such strings are translated. If you have to add such a string in a message resource file, add a comment "`/* DO NOT TRANSLATE */`". In the example below, a graphic driver failed after `/color` was translated because the string was used as a switch for the configuration program.

```
ConfMenu CM_BIOS_MODE=
{
    {0,
    "Color",
    "Text mode emulates IBM Color Graphics Adapter (CGA)",
    "/color"      /* DO NOT TRANSLATE */
    },
```

A common mistake of inexperienced developers is to define macros for strings in include files for inclusion in source files. Even if the strings are not hard-coded, they are not translatable because translators are not given the source files. No header files (**.h**) which contain translatable strings should be included by source files. All message strings need to be loaded at run-time by functions like `mdlResource_loadFromStringList`. When loading message strings, take into account expansion of translated strings and allocate a large enough string buffer.

Common errors are shown below:

```
/* Bad example #1 */
/* myapp.mc: */

mdlDialog_openAlert("Are you sure?");
mdlOutput_error("This is an error!);

/* Bad example #2 */
/* myapp.h: */

#define ALERT_STRING "Are you sure?"
#define ERROR_STRING "This is an error!"
```

```
/* myapp.mc: */

#include <myapp.h>
mdlDialog_openAlert(ALERT_STRING);
mdlOutput_error(ERROR_STRING);

/* Good example */
/* myapp.mc: */

char alertString[512];/* allocate enough size for expansion */
mdlResource_loadFromStringList(alertString, 0, MESSAGELISTID_Errors,
                           ERRORID_Alert);
mdlDialog_openAlert(alertString);
mdlOutput_rscPrintf(MSG_ERROR, NULL, MESSAGELISTID_Errors,
ERRORID_Error);

/* myappmsg.r: */

MessageList MESSAGELISTID_Errors=
{
   {
   {ERRORID_Error, "This is an error!"},
   {ERRORID_Alert, "Are you sure?"},
   ...
   }
}
```

Strings in Dialog Box Resource (.r) Files

Dialog box resources should not have labels and other text strings hard-coded. Instead they should have text strings defined in an include file **.h** in the language dependent sub-directory (**/english**) of the application. Then translators can work only on the include file and not be concerned with the complicated syntax of dialog box definitions and other resources. Definition for data structure and dialog ids should be defined in a separate include file and should not be mixed with text strings.

An exception is an icon resource. Icon names are usually not visible to users unless they try to edit icons with the icon editor. If an icon name is translated and becomes longer than 14 characters, the resource compiler cannot compile the icon resource file. Translators often do not know how to fix the compiler error. Therefore, it is recommended to leave icon names in English. In the example below, "Place Line" should not be pulled out to an include file under the **/ENGLISH** directory.

```
IconCmdSmallRsc ICONCMDID_PlaceLine =
{
   23, 23, FORMAT_MONOBITMAP, BLACK_INDEX,
   "Place Line",/* DO NOT TRANSLATE */
   {
   0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
```

```
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    ...
    }
};
```

Some common errors are shown below:

```
/* Bad example */
/* dialog.r: */

DialogBoxRsc DIALOGID_Basic=
{
   DIALOGATTR_DEFAULT, 25*XC, 7*YC,
   NOHELP, MHELP, HOOKDIALOGID_Basic, NOPARENTID,
   "Basic Dialog Box", /* Hard-coded dialog box title */
   {
        {{X1,GENY(1),XW,0}, Text, TEXTID_Basic, ON, 0, "", ""},
   }
};
DItem_TextRsc TEXTID_Basic=
{
   NOCMD, LCMD, SYNONYMID_Basic, NOHELP, MHELP,HOOKITEMID_Dummy,
   NOARG, 4, "%ld", "%ld", "1", "3", NOMASK, NOCONCAT,
   "Parameter 1:", /* Hard-coded dialog item label */ ""
};

/* Good example */
/* myapptxt.h - translators will work only on this file */

#define TXT_DIALOGTITLE "Basic Dialog Box"
#define TXT_DLOGITEMLABEL "Parameter 1:"

myappdlg.r - never edited by translators

#include <myapptxt.h>
DialogBoxRsc DIALOGID_Basic=
{
   DIALOGATTR_DEFAULT, 25*XC, 7*YC, NOHELP, MHELP,HOOKDIALOGID_Basic,
   NOPARENTID, TXT_DIALOGTITLE, /* replaced with macro */
   {
        {{X1,GENY(1),XW,0}, Text, TEXTID_Basic, ON, 0, "", ""},
   }
};
DItem_TextRsc TEXTID_Basic=
{
   NOCMD, LCMD, SYNONYMID_Basic, NOHELP, MHELP, HOOKITEMID_Dummy, NOARG,
   4, "%ld", "%ld", "1", "3", NOMASK, NOCONCAT,
   TXT_DLOGITEMLABEL, /* replaced with macro */ ""
};
```

Compiling Translated Resource Files

Compiling ".r" Files with Character Translation Tables

Character translation can be applied to text strings in MDL resource files by inserting the pragma directive shown below.

```
#pragma translate "chartran.rsc"
```

This tells the resource and MDL compiler to get the character translation resource from the resource file **chartran.rsc** and apply it to all quoted strings.

Character translation tables for some European languages are found in MicroStation's **CHARTRAN** directory.

cnfr863.rsc - code page 863 (Canadian-French)
engl437.rsc - code page 437 (United States)
mult850.rsc - code page 850 (Multilingual - Latin I)
nord865.rsc - code page 865 (Nordic)
port860.rsc - code page 860 (Portuguese)

Before compiling, ensure that the `MS` environment variable is set to indicate the base directory (without a trailing "/", for example, `"MS=/ustation"`). The MDL compilers search the `$(MS)`**/chartran** directory for the table. If the table is located elsewhere, the directory for translation resource can be specified by the `-t` flag. For example:

```
rcomp -t $(MS)/chartran message.r
```

This `#pragma` is only necessary if the files have been edited on the PC or Macintosh. If you edit the files on UNIX workstations, do not include this line. This translation method currently does not apply to Asian languages (Chinese, Korean and Japanese).

Compiling Translated Resource Files for Asian Text

When compiling resource files with Asian double-byte text strings, a special precaution is necessary. The MDL resource compiler uses code range information stored in a localized version of the MicroStation resource file **ustation.rsc**. The file **ustation.rsc** contains a run-time configuration for the code set handler. The codeset handler is integrated inside **ustation.rsc** and decides which sequence of two bytes is an Asian double-byte character or two ASCII characters.

To do so, the `MS` environment variable should point to the directory where a localized version of **ustation.rsc** is located, typically MicroStation's root directory. When the resource compiler finds **ustation.rsc**, it displays the message:

"Multi-byte character processing enabled"

If this message is not displayed, the environment variable is not correctly set, or the English version of **ustation.rsc** is being used. As a result, some double-byte characters may be converted to the wrong characters.

Numbered Argument Lists

The ordering of words in a message string may vary in some languages. The standard argument list imposes a fixed ordering on substitutions in the output string.

```
mdlResource_loadFromStringList(formatString,0,MESSAGELISTID,MSGID_OWNER
);
mdlOutput_vprintf(MSG_MESSAGE,formatString,objectString,subjectString);
MessageList MESSAGELISTID_MESSAGES=
{
   {
    ...
   {MSGID_OWNER, "%s owned by %s"}
```

If translation requires reordering of output values, the `%` character is replaced by `%digit$`, where digit is a decimal value giving the position of the required argument in the argument list. For example:

```
MessageList MESSAGELISTID_MESSAGES=
{
   {
         ...
         {MSGID_OWNER, "%2$s owns %1$s"}
```

Similarly,

```
{23, "month: %d, date: %d, year: %d"}
```

can be rearranged to:

```
{23, "date: %2$d, month: %1$d, year: %3$d"}
```

This method can be used with `mdlOutput_printf`, `mdlOutput_vprintf`, `printf`, `sprintf`, `vsprintf`, `vprintf`, `fprintf` and `vfprintf`.

Tools for Development and Testing

Software development for foreign language applications generally can be done using the same tools used for the English version. Translation of resource files and testing may require special hardware or localized versions of the operating system.

Hardware

Some foreign languages may require a special keyboard for typing foreign characters which do not exist on a standard 100-key keyboard.

Asian versions of MicroStation often require additional memory due to large font libraries. The size of both raster font and vector font libraries usually exceeds 1 Mb. A symbology resource file converted from TrueType or PostScript fonts can occupy several megabytes of disk space. Running MicroStation with insufficient memory may be slow due to disk swapping.

The Japanese version of MicroStation runs on the NEC-9800 series computer as well as on Japanese IBM compatibles. The NEC-9800 computer is a proprietary system marketed only in Japan and the architecture is not compatible with IBM PCs. It is recommended to test on the NEC computer if you are writing external programs.

Software

Depending on the target language, a localized version of the operating system may be required. You may also need a localized version of MicroStation. For example, a correct version of the character translation table and **ustation.rsc** may be required. If your application communicates with external applications; database applications such as Oracle and dBASE, localized versions of such software may be necessary.

Recommended Reading

There are several books on internationalization.

1. “Internationalization Made Easy” by The Open Software Foundation, 1990
2. “The Programmer's Supplement for Release 5” of the O'Reilly Definitive guides to the X Window System” by David Flanagan, O'Reilly & Associates, Inc., 1991
3. “Unicode Standard - Worldwide character Encoding” by the Unicode Consortium, Addison-Wesley Publishing Company, Inc., 1991 (Vol. 1), 1992 (Vol. 2)

4. "Guide to Macintosh Software Localization" Addison-Wesley publishing Company, 1992, by Apple Computer, Inc.
5. Internationalisation Guide, X/Open Company, Ltd. 1992
6. Understanding Japanese Information Processing, by Ken Lunde, O'Reilly & Associates, Inc., 1993
7. Software Internationalization and Localization: an Introduction, by Emmanuel Uren, Robert Howard, and Tiziana Perinotti, Van Nostrand Reinhold
8. Programming Language C: Multibyte Support Extensions, ISO/IEC 9899:1990 Amendment 1:1993
9. Developing International Software for Windows 95 and Windows NT, by Nadine Kano, Microsoft Press, 1995
10. Global Software, by Dave Taylor, Springer-Verlag, 1992
11. Solaris International Developer's Guide, by Bill Tuthill, Prentice Hall, 1993

Appendix B: Settings Manager

Reading and writing settings manager files is quite straightforward using the structures and constants defined in **settings.h**. Settings manager files are resource files, so they can be manipulated with the `mdlResource_*` MDL functions. These files have a **.stg** extension by convention.

Creating Settings Manager Files

To create a new settings manager file and open it for writing, the following code fragment may be used:

```
#include <msrsrc.fdf>

int status;
RscFileHandle rFileH;
status = mdlResource_createFile("mysets.stg", "Settings Data", 0);
if (status == SUCCESS)
   status = mdlResource_openFile(&rFileH, "mysets.stg",
                                 RSC_READWRITE);
```

`"Settings Data"` is the resource identifier string used by the settings manager application. This string SHOULD NOT be translated and is used here for consistency.

The resource types (`RTYPE_*`) of the settings manager resources are also included in **settings.h**. They are:

Resource Type	Description
`RTYPE_Base`	Base Units (SI, English)
`RTYPE_Unit`	Units (ft, in, m, cm, mm)
`RTYPE_DwgScale`	Drawing Scale
`RTYPE_WorkUnits`	Working Units
`RTYPE_Linear`	Linear Settings
`RTYPE_Mline`	Multi-line Settings
`RTYPE_Dim`	Dimension Settings
`RTYPE_Cell`	Cell Settings
`RTYPE_Apat`	Area patterning Settings

Resource Type	Description
RTYPE_TextStg	Text Settings
RTYPE_Point	Active Point Settings
RTYPE_SettingsGroup	Settings Group
STYLETYPE_Dimension	Dimension style type
STYLETYPE_Multiline	Multiline style type

The file **settings.h** in the **mdl/include** directory details the structure of resources found in settings manager resource **.stg** files. These structures include:

```
typedef struct settingsGroup
{
   char name[32];
   long rscId[10];
} SettingsGroup;
```

The string, `name`, contains the name of the settings group. The array of `longs`, `rscId`, contains a list of resource identifiers for each setting component. The component resource Ids are organized within the array as follows:

Array Element	Description
rscId[0]	Linear component
rscId[1]	Text component
rscId[2]	Cell component
rscId[3]	Point component
rscId[4]	Area pattern component
rscId[5]	Multiline component
rscId[6]	Dimension component
rscId[7]	reserved for future expansion
rscId[8]	reserved for future expansion
rscId[9]	reserved for future expansion

The component resource Id is set to -1 if the group contains no components of the type corresponding to the array index. If the settings group contains one or more components corresponding to the array index, the resource Id of the first in the list of those components is stored in the `rscId` array.

Creating a Settings Group

To create and initialize a settings group, the following code fragment may be used:

```
SettingsGroup myGroup;
int i;

/* give the settings group a name */
strcpy(myGroup.name, "My Settings");

/* initialize all the component ids to -1 */
for (i=0; i<10; i++)
   myGroup.rscId[i] = -1L;
```

Writing a Settings Group

To write a settings group, the following code fragment may be used:

```
SettingsGroup myGroup;
long rscId;
RscFileHandle rFileH;

/* open the settings manager resource file */
if (SUCCESS==mdlResource_openFile(&rFileH,"myset.stg",RSC_READ_WRITE))
{
   /* find next available settings group resource id in this file */
   mdlResource_queryClass(&rscID, rFileH, RTYPE_SettingsGroup,
                          RSC_QRY_AVAIL_ID, NULL);

   mdlResource_add(rFileH, RTYPE_SettingsGroup, rscId, &myGroup,
                   sizeof(SettingsGroup), NULL);
   mdlResource_closeFile(rFileH);
}
```

Settings components are stored as resources in the settings file. Each component structure contains a common information structure called the “settings header.”

```
typedef struct /* Information common to every setting */
{
   long      nextComponentId;    /* ID of next component */
   int       flag;               /* override flags */
   int       level;              /* level override */
   int       color;              /* color override */
   int       weight;             /* weight override */
   char      style[32];          /* line style override */
   char      name[32];           /* name/description */
   char      command[148];       /* keyin string */
   double    styleScale;         /* line style scale */
   char      reserved[4];
} SettingsHeader;
```

Most members of this structure match the corresponding TCB or element structure members, but two of the members need to be explained.

The `long`, `nextComponentId`, is used to implement a list of components of a particular type in a settings group. If the component is the end of the list, `nextComponentId` is set to -1. If there are more components of its type in the settings group, `nextComponentId` will contain the resource Id of the next component in the list.

Bits in the `int`, `flag` are set to indicate which of the settings in the component are active. For example, if the component should set the active color when it is activated, the value of the color must be placed in the `int`, `color` and the value `COLOR_BIT` (specified as `0x2` in **settings.h**) should be **OR**'d into the `int`, `flag`. This mechanism allows the component to specify ONLY the necessary active settings. In addition to the settings contained in the settings header structure, `flag` is also used to indicate overrides for some component-specific settings as documented in each component structure.

The string, `name`, contains the name/description of the component.

The string, `command`, contains an optional command to be executed when the component is activated.

For example, to add a linear settings component to an existing settings group structure (refer to the `LinearSettings` structure in **settings.h**):

```
SettingsGroup *myGroupP;
LinearSettings myLinear;
RscFileHandle rFileH;

/* open the settings resource file */
if (SUCCESS!=mdlResource_openFile(&rFileH, "myset.stg", RSC_READWRITE))
   return;

/* load settings group resource with id=1 */
/* Note: see the function ascGroup_findResourceByName() in ascgroup.mc
   to load a settings group by name */

myGroupP = (SettingsGroup *)mdlResource_load(rFileH,
                                RTYPE_SettingsGroup, 1L);
if (myGroupP == NULL)
   return;

/* set the active color to 3 */
myLinear.hdr.color = 3;
myLinear.hdr.flag |= COLOR_BIT
/* name the component "My Linear Setting" */
strcpy(myLinear.hdr.name, "My Linear Setting");

/* activate the place smartline command */
strcpy(myLinear.hdr.command, "place smartline");

/* place this linear component at the head of the list for this group */
myLinear.hdr.nextComponentId=myGroupP->rscId[COMPONENT_Linear];
```

```
/* get the next available LinearSettings resource id */
mdlResource_queryClass(&myGroupP->rscId[COMPONENT_Linear],
                  rFileH,RTYPE_Linear, RSC_QRY_AVAIL_ID, NULL);

/* add the LinearSettings resource */
mdlResource_add(rFileH, RTYPE_Linear,myGroup->rscId[COMPONENT_Linear],
                &myLinear,
sizeof(LinearSettings), NULL);

/* update the SettingsGroup resource */
mdlResource_write(myGroupP);
mdlResource_closeFile(rFileH);
```

An example MDL application called **ascgroup** has been provided to further illustrate manipulation of settings manager files using `mdlResource_` functions. This example creates and edits settings groups as well as Linear, Cell and Text components based on specifications the user provides in an ASCII file, and is available in the **mdl/examples/ascgroup** directory.

Adding Functionality to ascgroup.ma

Adding Keywords to the Keyword Table

ascgroup stores the keywords it parses for in a string list that has three info fields. The first info field (at offset 0) contains the keyword, `level`. Keyword levels defined so far are primary (keyword operates at the settings group level), group (keyword operates on a component contained in a group), or component (keyword operates on a component).

The second info field is currently used only for component keywords, and indicates the type of settings component to which the keyword applies. Some keywords (`SETTINGS_HDR`) apply to all components.

The last info field identifies the action taken when that keyword is found.

Since info field 0 is used, this string list cannot be used in a list box.

`mdlStringList_search` is used to search for the keyword in the list.

Adding Keywords to the List

1. Add the necessary defines to **ascgroup.h**. For example, if you are implementing a new component, the component's key value (`SETTINGS_LINEAR`) as well as all the individual command values (`FILLCOLOR_SETTING`) must be added.
2. Add the keyword members to the string list `STRINGID_KeywordList` in **ascgroup.r**.

Coding the Functionality for new Keywords

ascgroup.mc will search the keyword list and return the string list index matching your keyword, but you must add the code to process that keyword.

1. If you have added a new keyword level, a case must be added for the new level in `ascGroup_processKeywords`.
2. If you have added a new primary keyword, a case must be added for the new primary keyword in `ascGroup_doPrimary`.
3. If you have added a new group keyword, a case must be added for the new group keyword in `ascGroup_doGroup`.
4. If you have added a new component keyword, a case must be added for the new component keyword in the corresponding component's processing function (`ascGroup_setLinearField`).
5. If you have added processing for a component type that was not previously supported, the functions: `ascGroup_loadComponent`, `ascGroup_newComponent`, `ascGroup_writeComponent` and `ascGroup_getComponentHeader` must be updated to support the new component type. You must also add a case to `ascGroup_doComponent` for the newly supported component type.

By convention, each supported component has a separate function to process its keywords. The settings header also has a separate function for settings header keywords.

Formatting the ASCII Input File

This section contains information on the rules which apply to the formatting of the ASCII input file.

Rules/Format

1. Any text following a '*#*' character is considered a comment.
2. Keywords are case sensitive.

3. Each keyword must be placed with its value on a separate line (there cannot be a line feed between the keyword and its value).
4. Maximum line length is 255.
5. Each "add" or "set" group or component keyword must have a matching "end" keyword. Keywords in between the "add" and "end" group or component keyword apply only to that group or component.
6. Key-in: MDL KEYIN ASGROUP ASCGROUP CONVERT <ascii name> <settings file name>. If filenames are not supplied, the user will choose them via the file open/create dialog boxes.
7. Supported keywords include those found in the following tables.

   Values listed in quotations indicate that only the listed values apply. In the input file, the quotations around the values should be omitted.

Keyword	Value	Description
`setGroup`	group name	Modify an existing settings group.
`addGroup`	group name	Add a new settings group.
`endGroup`	none	End specifications/modifications to the group named by the last "`setGroup`" or "`addGroup`" statement.
`setLinear`	component name	Modify an existing linear settings component.
`addLinear`	component name	Add a new linear settings component (must be inside a "`setGroup`" or "`addGroup`" block).
`endLinear`	none	End specifications/modifications to the linear settings component named in by the last "`setLinear`" or "`addLinear`" statement.
`setText`	component name	Modify an existing text settings component.
`addText`	component name	Add a new text settings component (must be inside a "`setGroup`" or "`addGroup`" block).
`endText`	none	End specifications/modifications to the text settings component named by the last "`setText`" or "`addText`" statement.
`setCell`	component name	Modify an existing cell settings component.
`addCell`	component name	Add a new cell setting component (must be inside a "`setGroup`" or "`addGroup`" block).
`endCell`	none	End specifications/modifications to the cell settings component named in the last "`setCell`" or "`addCell`" statement.

Generic Settings

These apply to all components and must be contained within a "`set<X>`" or "`add<X>`" component block.

Keyword	Value	Description
`setLevel`	integer	Sets active level for the component.
`setColor`	integer	Sets active color for the component.
`setWeight`	integer	Sets active weight for the component.
`setStyle`	style name	Sets the active line style for the component.
`setName`	component name	Changes the component's name/description.
`setCommand`	key-in string	Sets the active command for the component.
`setStyleScale`	double	Sets the active line style scale for the component.

Linear Settings

These apply only to linear settings components and must be contained within a "`setLinear`" or "`addLinear`" component block.

Keyword	Value	Description
`setArea`	"solid", "hole"	Sets the active area for the linear component.
`setFill`	"none", "filled", "boundary"	Sets the active fill type for the linear component.
`setFillColor`	integer	Sets the active fill color for the linear component.

Text Settings

These apply only to text settings components and must be contained within a "`setText`" or "`addText`" component block.

Keyword	Value	Description
`setTextHeight`	double	Sets the active text height for the component.
`setTextWidth`	double	Sets the active text with for the component.
`setLineSpacing`	double	Sets the active line spacing for the component.
`setCharSpacing`	double	Sets the active character spacing for the component.

Keyword	Value	Description
setUnits	double	Sets the active units name ("mm" or "") for the component.
setTextAngle	double	Sets the active angle for the component.
setTextJust	*values below	Sets the single-line text justification for the component. Possible values are: "LT" - left top "LC" - left center "LB" - left bottom "CT" - center top "CC" - center center "CB" - center bottom "RT" - right top "RC" - right center "RB" - right bottom.
setNodeJust	*values below	Sets the multi-line text justification for the component. Possible values are: "LT" - left top "LC" - left center "LB" - left bottom "LMT" - left margin top "LMC" - left margin center "LMB" - left margin bottom "CT" - center top "CC" - center center "CB" - center bottom "RT" - right top "RC" - right center "RB" - right bottom "RMT" - right margin top "RMC" - right margin center "RMB" - right margin bottom.
setFont	font name	Sets the active font for the component.
setFractions	“on”, “off”	Sets fractions on or off for the component.
setLineLength	integer	Sets the active line length for the component.
setSlant	double	Sets the amount of slant for the component.
setVertical	“on”, “off”	Sets vertical text setting on or off for the component.
setUnderline	“on”, “off”	Sets the underline text setting on or off for the component.
setUsePaperSize	“on”, “off”	Sets the user paper size setting on or off for the component.

Cell Settings

These apply only to cell settings components and must be contained within a "setCell" or "addCell" component block.

Keyword	Value	Description
setCellXScale	double	Sets the active scale in the X direction for the component.
setCellYScale	double	Sets the active scale in the Y direction for the component.
setCellZScale	double	Sets the active scale in the Z direction for the component.
setApplyDrawingScale	“on”, “off”	Sets the apply drawing scale setting on or off for the component.
setAllowSharedCells	“on”, “off”	Sets the use shared cells setting on or off for the component.
setCellLibrary	library name	Sets the active cell library for the component.
setCellName	cell name	Sets the active cell name for the component.
setCellAngle	double	Sets the active angle in degrees for the component.
setCellType	“placement”	Indicates whether the component refers to a placement “terminator” or terminator cell.
setSharedCell	“enabled”, “disabled”	Indicates whether cells should be placed as shared for the component.

Example, Creating a Settings Manager Resource

```
# Example file to create a Settings Manager resource

1. Any text following a ‘#’ character is considered a comment.

2. Keywords are case sensitive.

3. Each keyword must be placed with its value on a separate line (there
cannot be a line feed between the keyword and its value).

4. Maximum line length is 255.

5. Each "add" or "set" group or component keyword must have a matching
"end" keyword. Keywords in between the "add" and "end" group or
component keyword apply only to that group or component.

addGroup     Example             # create a new group
  addLinear      Linear Example # Create linear settings component
  setCommand     place smartline# place a smartline
  setLevel       1              # on level 1
```

```
  setColor       5              # color 5
  endLinear                     # end linear component

  addText        Text Example   # Create text settings component
  setLevel       3              # set active level = 3
  setColor       3              # set active color = 1
  setHeight      2.0            # set active text height = 2
  setLineLength  50             # set text line length = 50
  setFont        FANCY          # set active font FANCY font (2)
  setTextJust    LT             # set text just to left-top
  setNodeJust    CC             # set text node just to center-center
  endText                       # end text settings component

  addCell        Cell Example   # create a cell settings component
  setCommand     place cell     # place a cell
  setCellName    CHAIR2         # cell to place is CHAIR2
  setCellLibrary remodel.cel    # from the remodel library
  setCellAngle   90             # at angle 90 degrees
  setSharedCell  enabled        # place as a shared cell
  endCell                       # end cell settings component
endGroup                        # end Example group

# edit an existing Linear component
setLinear    Linear Example# modify example linear settings component
  setLevel       2    # changed my mind, place smartline on level 2
  setStyle       { Batten }     # place smartline in Batten style
endLinear                       # end modifications
```

Appendix C: MicroStation 95

Bentley has made an effort to introduce enhancements to MicroStation in a manner that will impact the MicroStation community as minimally as possible. Our primary concern is to ensure that existing MDL applications and MicroCSL programs are upwardly compatible with this new version of MicroStation.

What follows is a brief, yet comprehensive, list of things that developers should take into account regarding existing or new applications that will run on MicroStation.

To ease the burden on third-party MDL developers, MicroStation will be backward compatible with MDL applications from V5 or MicroStation PowerDraft and should run without recompiling on MicroStation. However, new features and optimizations will prevent applications compiled on MicroStation from running on V5.

Developers are encouraged to enhance their applications to keep them consistent with the new user interface and features of MicroStation. The dialog manager section that follows, points out some cases where recompiling may be desired.

No new vector-based element types are introduced to the MicroStation design file format. Type 90 raster elements associated with the reference raster file capability are to be incorporated into MicroStation. Raster Reference was first introduced into Review. It was subsequently released for MicroStation on the Vault.

Platform Changes

MicroStation will use Microsoft's Win32s subsystem instead of Windows Connection to run under Windows 3.1x. Win32s lets 32-bit programs run in a 16-bit environment. One installation set will cover all Windows platforms: Windows 3.1, Windows NT and Windows 95. This is contrary to V5, where Windows Connection used a DOS delivery to run under Windows 3.1x. To develop new MDL applications to run on any Windows platform (or to recompile existing applications), at least one copy of Windows 95 or Windows NT is required. This is because it is not feasible to provide MicroStation MDL development tools that run under Windows 3.1x.

Windows NT and Windows 95 are preemptive multi-tasking operating systems and support similar APIs while Windows 3.1x only supports cooperative multi-tasking and a subset of the Win32 API (Win32s). V5 NT applications should run without

recompiling on all MicroStation Windows platforms as long as the application does not rely on multi-tasking or API services that are not included in Win32s.

MDL Compiler

The MDL Compiler has been *tightened up* with respect to the generation of compiler warnings. As a result, applications that compiled warning-free with the MicroStation V5 MDL compiler might now generate warnings when compiled with the MicroStation MDL compiler. The most commonly seen new warning will probably be:

```
"Function is missing a return statement."
```

The most common occurrence of this warning will take place with functions that implement commands. For example:

```
cmdName myFunction
{
   (code to implement a command, with no return statement)
}
```

Functions that implement commands do not have return values, so they should be defined as `void`. The function shown above is interpreted by the compiler as returning an `int` value. To eliminate the warning in this case, redefine the command as:

```
cmdName void myFunction{(code…)}
```

Additional type checking

The MicroStation MDL compiler contains some type information about each MDL built-in function. However, much better type checking can be obtained by using the delivered function prototype files (.fdf). Bentley highly recommends the use of function prototype files in all .mc source modules.

To enforce the use of .fdf files, the MDL compiler recognizes a pragma: `noAnsiDeclOn`. This will enable strict ANSI function prototyping. This can also be effected from the MDL compiler command line using the `-r` option as follows: `-rnoAnsiDeclOn`. The pragma is used by default if a makefile includes mdl.mki. When ANSI function prototyping is enforced, the following warning will result when a function is called with no previous prototype:

```
"(function name) does not have an ANSI function declaration".
```

The offending function call will need to be prototyped by the inclusion of a function prototype file (.fdf) at the top of the source modules.

Resource Files

No changes were made to the format of resource files. However, there is one minor backwards compatibility issue with regard to using MicroStation resource files with V5 on a different platform. MicroStation resource files contain new information in their data definition resources. This new information is necessary to properly translate resource files when read on different platforms (different from where the file was created). The fixes release version of MicroStation V5 *understands* this new information, but older versions do not. Therefore, MicroStation resource files will not work across multiple platforms with versions of MicroStation prior to V5.0.95 (V5.0.70 on the RS/6000). This does not apply when using the MicroStation resource file on a single platform; in that case, they will work. The symptom of using a MicroStation resource file on a platform where it is not supported is easily recognized; the resource will fail to load.

If all your MicroStation seats are V5.0.95 or newer, then you can ignore this section on Resource Files. Also, if your MicroStation resource files are going to be used on the same platform as the one on which they were created, you can ignore this section. The only feature that is not supported is using MicroStation resources with versions of MicroStation older than V5.0.95 on a different platform from where the resource file was created.

Resource Editing Tools

In V5.0 of MicroStation, the Dialog Builder was used to modify binary (.rsc) files. Another application, Sourcer, was launched separately to convert those .rsc files into source code (.r files). The V5 Sourcer would generate source files from scratch every time, regardless of any existing source files.

In the new Resource Development Environment (RDE), Builder and Sourcer have been joined together as part of a new *source compatible* resource editing system. *Source compatible* means that existing source files will be updated. Source files can be opened within RDE. RDE compiles the .r file and generates a list of resources, each of which can be edited by a resource editor, such as the Dialog Builder.

Other Development Facilities

This section briefly describes the impact that MicroStation has on MicroStation BASIC, MicroCSL and user commands.

Compatibility with macro customization

MicroStation includes a new macro capability, macro record/playback, with which end users can automate common sequences and tool usage. A macro is created by monitoring the user's actions and writing a description of those actions to a BASIC source file. The BASIC file can then be edited and executed many times to recreate the same or similar results. These macros are BASIC programs, an emerging industry standard.

Using these same macro tools, users should be able to customize third party applications as well. To this end, application developers are urged to follow these guidelines:

All commands soliciting user input should be organized as state machines. This, since the new macro facility automatically records commands and variable access string changes. For further control, MDL programmers will be able to publish additional dialog hook results to the macro recorder.

Dialog items that queue commands or update access string variables when manipulated are handled automatically by the macro tools. Therefore, this style of dialog item implementation is encouraged.

Dialog items that make use of hook function processing will be least compatible with macro customization. New item handler “journaling” messages are available in this case. This allows an application to tell the macro tools how to save, and later execute, the events. This journaling API will be documented at a later date.

MicroCSL

MDL is still the preferred development language. However, MicroCSL will continue to be supported on all platforms. On DOS, only protected mode MicroCSL will be supported. As forewarned in April 1994, DOS real mode MicroCSL is no longer supported in versions of MicroStation beyond V5 (including MicroStation).

Like MDL applications, external protected mode and MicroCSL programs compiled for DOS will work only with MicroStation for DOS. To use these programs under Windows 3.1x, they must be compiled as Windows NT executables.

Note that both Windows NT and Windows 95 are preemptive multi-tasking operating systems and support similar APIs. Windows 3.1x supports only cooperative multi-tasking and a subset of the Win32 API (Win32s). The differences are irrelevant for most

external/MicroCSL programs, so these programs should run without recompiling on all Windows platforms. However, programs that rely on multi-tasking or API services that are not included in Win32s will be limited to running on Windows NT or Windows 95.

No enhancements will be made to MicroCSL for MicroStation. MicroCSL programs created for V5 will continue to work; neither re-compilation or relinking should be necessary.

User commands

No enhancements will be made to user commands for MicroStation. All current user commands should continue to work. However, BASIC macros should be considered an eventual successor to user commands.

File Open Dialogs

The file open dialogs for MicroStation and PowerDraft are modeled after the Windows standard. The benefits include a visual directory hierarchy and descriptive filter information. The calls to `mdlDialog_fileOpen` and `mdlDialog_fileCreate` are unaffected unless the dialog has been customized. In this case, the dialog resource will need to be updated to reflect the new dialog items and layout.

MicroStation PowerDraft and MicroStation utilize a different style of file open dialog. It's API and resource definition is very similar. Conversion will be straightforward. MicroStation PowerDraft's and MicroStation's file open dialogs have the following advantages:

Visual directory hierarchy

Descriptive filter information

Consistent with MicroSoft's Office products

Although V5 (and V4) file open dialogs will be supported, they will be inconsistent with MicroStation PowerDraft and MicroStation.

Other Trends

Tool consolidation is one method of reducing complexity without reducing functionality. Utilization of the Tool Settings dialog is one of the most effective ways of consolidating tools. *Smart* tools that perform context-sensitive operations are also effective. The tool consolidations that occurred between V4 and V5 has been carried even further for MicroStation and MicroStation PowerDraft. Application programmers are encouraged to follow suit.

While key-ins are the hallmark of the power user (and should remain available), there are more intuitive command entry methods for the new or occasional user. In order to de-emphasize key-ins for this type of user, all common commands should be accessible through icons and all common settings should be available through dialog boxes.

AccuDraw

MicroStation and MicroStation PowerDraft incorporate a new drawing tool called AccuDraw. It allows precision geometry to be created without having to know distances or having to create construction geometry. AccuDraw effectively replaces many V5 settings (alignments, axis lock, unit lock, precision key-ins and the precision input dialog box) by consolidating their functionality into a single entity. MDL programmers will be able to optimize their application commands for use with AccuDraw.

MicroStation vs. PowerDraft

MicroStation PowerDraft enhances the task of production drafting by adding several new subsystems, however, it's functionality remains a subset of MicroStation. As such, PowerDraft MDL routines are contained in the MicroStation domain. The following functions are outside of the PowerDraft subset and therefore not accessible to PowerDraft MDL applications:

```
mdlBspline_xxx
mdlMaterial_xxx
mdlExternal_xxx
mdlDB_xxx
mdlTutorial_xxx
mdlImage_movieXXX
```

MDL applications not utilizing the above functions will execute without recompiling on both PowerDraft and MicroStation.

Also external to the PowerDraft subset is support for DLMs and external programs. It should be noted that PowerDraft is initially targeted for DOS and Windows (NT, 95 and 3.1) only. Naturally, MicroStation has no MDL, DLM or platform restrictions and will support all new functionality described in this text.

Dialog Manager

Many of the dialog item handler functions formerly implemented in ditemlib.ml have been repackaged as MDL built-in functions in MicroStation. Therefore, any optimizations or fixes that may have been implemented in these functions will not be realized by existing V5 MDL applications until those applications are re-compiled using the MicroStation development tools.

The following classes of DITEMLIB functions are affected:

```
mdlDialog_icFrame...
mdlDialog_icPalette...
mdlDialog_listBox...
mdlDialog_menuBar...
mdlDialog_textPDMItem...
mdlDialog_optionPDMItem...
mdlDialog_optionButton...
mdlDialog_pushButton...
mdlDialog_colorPicker...
mdlDialog_toggleButton...
mdlDialog_levelMap...
mdlDialog_radioButton...
```

and the single function, `mdlDialog_labelSetAttributes`

Command window vs. status bar

Version 5 allowed for a single user interface style, the command window. Version 95 allows for an additional interface style, the status bar. This is a Windows standard interface introduced with PowerDraft. If applications adhere to established conventions they will run without problem under either user interface style. The majority of existing V5 applications need not be concerned by the user's choice of interface style and should run right away. However, applications that directly reference the command window's dialog pointer may need to make some minor updates. To understand how

to handle each of the interface styles, it is probably best to think about the (V5 and V95) user interface in terms of 4 distinct *regions* and their associated routines.

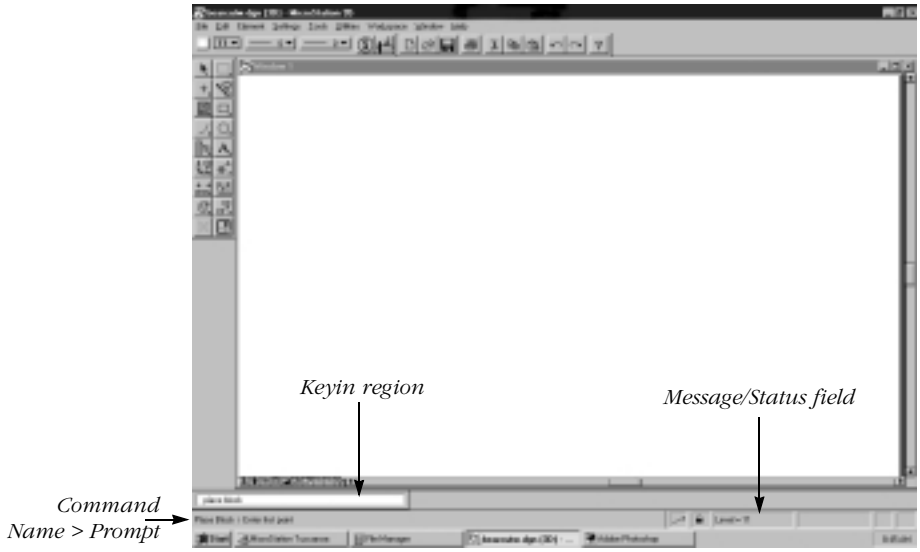


Figure C.1 Status Bar interface - Error messages displayed where Command Name > Prompt displayed.

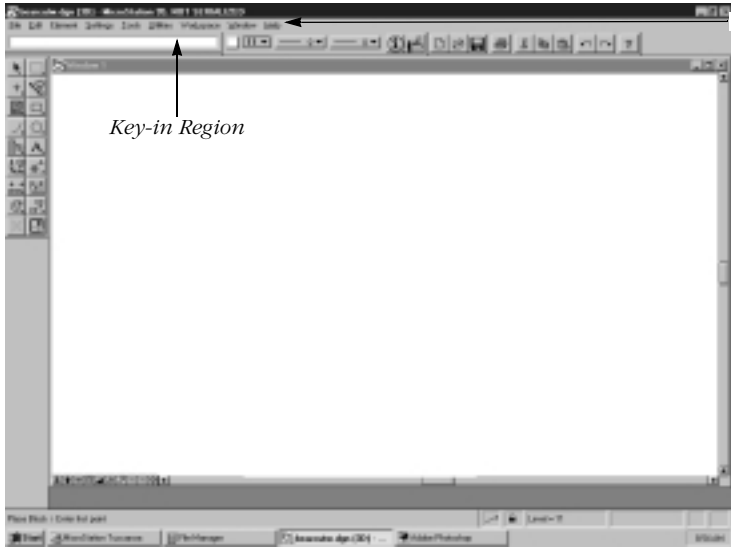


Figure C.2Command Window interface.

The Prompt Region is defined as the dialog box that displays messages and prompts. The Prompt Region contains 5 fields - command name, prompt, message, status and error. To output message strings to these fields, use the `mdlOutput_XXX` routines. The only real impact with using the `mdlOutput_XXX` calls is that in the status bar interface MESSAGE and STATUS fields overlay each other.

The Key-in Region is defined as the dialog box that contains the main MicroStation key-in area. The Key-in Region applies to both interface styles, but is only required to be open using the command window interface. As such, the associated routine `mdlDialog_keyinWindowGet`, will never return `NULL` using the command window. However, this function may return `NULL` when using the status bar interface if the Key-in dialog is not open.

The Overall Title Bar Region is defined as the dialog box that displays the application title (the name of the active design file) and houses MicroStation's main menu bar. The associated routines with this region are `mdlDialog_overallTitleBarGet` and `mdlDialog_menuBarGetCmdWinP`. `mdlDialog_menuBarGetCmdWinP` is still the correct way for applications to find MicroStation's menu bar. Previous usage of `mdlDialog_menuBarAddCmdWinMenu` to manipulate the main MicroStation menu bar is unaffected no matter which user interface style is selected.

The fourth region, the Status Area Region, is defined as the region (on the right side of the status bar) that displays the current snap, current level, etc. When using the command window, this region does not exist. The associated routine with this region is `mdlDialog_statusAreaGet`.

Prompts and Messages

One of the most noticeable differences between the interfaces of MicroStation V5 and PowerDraft is message handling. While V5 has the command window, MicroStation PowerDraft sends all messages to the status area located at the bottom of its application window. This status area contains no key-in field, and consolidates the prompts and messages into three areas. For this reason, it is important to examine the messaging practices of each application that will run on MicroStation PowerDraft to ensure there are no conflicts. Some things to consider:

The command and prompt messages display in the form [command]>[prompt]. Sending strings other than command names and prompts to these fields may produce confusing messages.

Message, status and error strings display in the same field will conflict. Sending the most important message last is a good idea, since it will overwrite the preceding ones. If a status message must display at the same time as an error message, consider opening a message box.

Only the display of prompts and messages has changed. The API (`mdlOutput_error`, etc.) is unchanged.

Message Boxes

Message boxes are the new means of displaying important information and include an icon indicating message type in addition to actual message text. A new MDL routine, `mdlDialog_openMessageBox`, obsoletes `mdlDialog_openInfoBox` and `mdlDialog_openAlert`.

```
int mdlDialog_openMessageBox/* <= ACTIONBUTTON_xxx */
(
long dialogId     /* => dialog ID of message box */
char *stringP     /* => message to display */
long iconId       /* => which icon to show */
);
```

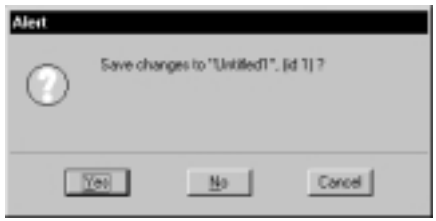


Figure C.3 Yes/No/ Cancel Message Box.

In order to easily match button layout (Yes/No, OK/Cancel, etc.) with the message text, the following choices are available:

```
DIALOGID_MsgBoxOK
DIALOGID_MsgBoxOKCancel
DIALOGID_MsgBoxYesNo
DIALOGID_MsgBoxYesNoCancel
```

In order to easily identify type or severity of the message, the following icons are available:

```
MSGBOX_ICON_INFORMATION
MSGBOX_ICON_WARNING
MSGBOX_ICON_QUESTION
MSGBOX_ICON_CRITICAL
```

MicroStation PowerDraft uses message boxes instead of info boxes and alerts. Message boxes have the following advantages:

Icon indicates type of message
Buttons more closely match messages
Consistent with MicroSoft Office products

The `mdlDialog_openMessageBox` function opens a modal dialog box which contains one to three push buttons, a read-only multi-line text 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 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 `dialogId`'s are available for use with `mdlDialog_openMessageBox`:

dialogId	description
`DIALOGID_MsgBoxOK`	The message box contains one button: `OK`.
`DIALOGID_MsgBoxOKCancel`	The message box contains two buttons: `OK` and `Cancel`.
`DIALOGID_MsgBoxYesNo`	The message box contains two buttons: `Yes` and `No`.
`DIALOGID_MsgBoxYesNoCancel`	The message box contains three buttons: `Yes`, `No` and `Cancel`.

`mdlDialog_openMessageBox` can also be used to display Icons in the following, (V4/V5) 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.

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 `MsgBoxOK` dialog, for example:

```
mdlDialog_openMessageBox(DIALOGID_MsgBoxOK, stringP,
MSGBOX_ICON_INFORMATION);
```

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, “File exists. Overwrite brake.dgn?”.

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 brake.dgn?”. Some applications may find the question mark symbol more appropriate if the alert is in the form of a question. However, note 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.

`mdlDialog_openMessageBox` returns `ACTIONBUTTON_OK`, `ACTIONBUTTON_Cancel`, etc. or returns `-1` if failure.

Although calls to `mdlDialog_openInfoBox` and `mdlDialog_openAlert` will be supported, the resulting dialog will be inconsistent with MicroStation PowerDraft.

Index

A

AccuDraw C-6
Adobe Systems, Inc. A-30
alias pragma *see pragmas, alias*
Alphabets
  see Character Sets
ANSI (American National Standards Institute) 12-1
application architecture 3-7
  primitive commands 3-7
  user interface 3-7
  utility commands 3-7
application files 8-1
application task identifier 1-8
  overview 1-8
  task ID 1-8
application variables 15-2, 15-10
  referencing 15-10
  synonym resources 15-3
ASCII (American Standard Code for Information Interchange) 10-10
ASCII Character Set A-3
Asian Character Set
  see Character Sets
AutoCAD Shape Files A-35

B

Balloon Help
  see Tool Boxes, tool tips
BDF (Basic Distribution Format) A-30
bdf2snf A-30
bdos 23-1, 23-3
Bidirectionality
  see Special Considerations
big endian format 6-29
binary portability
  big endian format 6-29
  bitfield representation 6-29
  byte ordering 6-29
  defined 6-27
  floating point standards 6-29
  internal data representations 6-28
  little endian format 6-29
  storage of data types 6-28
bitfield handling 5-14
bmake 7-1
  conditionals 7-7
  dependency format 7-6
  include files 7-8
  macro definition 7-4
  macro substitution 7-4
  overview 7-2
  predefined macros 7-5
  reserved macros 7-4
  rules format 7-6
  starting the bmake utility 7-9
bmake utility 15-21
building applications 7-1
  bmake 7-1
  makefiles 7-1
built-in functions 1-2
built-in variables 1-2
built-ins *see DLM, specific terms*
button group item *see dialog box items, button group item*

C

C expression handling 15-10
  rsctype utility 15-10
  strings 15-10
C Programming Language 5-1
  contrasted to MDL 5-1
C Programming Language and MDL *see MDL and the C Programming Language*
Case Conversion
  see Wide Character Functions
Cell Names A-38
Character Encoding A-3
  definition A-3
  Macintosh A-4
  mapping A-3, A-4
Character Sets A-2
  Asian Character Set A-5
  bytes required per character A-2, A-5
Character Translation Table A-4
  CHARTRAN A-4
CHARTRAN
  see Character Translation Table
check box *see toggle button*
check button *see toggle button*
CmdItemListRsc 16-75
cmdName keyword 1-10, 5-2, 8-4
cmdNumber keyword 1-10, 5-2, 7-17, 8-4
Codeset Handler A-6
  MS_CODESET environment variable A-6
  SETLANG utilitiy A-6
color picker item *see dialog box items*
color picker pull-down menu *see dialog box items, color picker pull-down menu*
command functions
  cmdName keyword 1-10
  cmdNumber keyword 1-10
command parsing 6-2, 6-14
  mdlParse_load CommandTable 6-15
  mdlParse_loadKeywordTable 6-15
  mdlParse_unloadTable 6-15
command processing 3-6
  immediate commands 3-6
  primitive commands 3-6
  utility commands 3-6
command table editor 2-1, 4-1, 4-2
  edit menu 4-4

editing existing command tables 4-2
file menu 4-2
overview 4-2
command tables 1-6, 1-10, 2-1, 6-2
command class 7-13
compiliation of 7-11
creation of using command table editor 2-1
example of 7-12, 18-16
Command Window C-7
common item resource fields 16-8
accessStr 16-8
commandNumber 16-8
commandSource 16-8
helpInfo 16-8
helpSource 16-8
itemHookArg 16-8
itemHookId 16-8
label 16-8
synonymsId 16-8
unparsed 16-8
Compiler
see MDL Compiler
Compiling Translated Resource Files A-47
complex elements 11-1
headers 11-4
conditionals *see bmake*
Context Sensitive Characters *see Special Considerations*
copyFromReal 23-4
copyToReal 23-4

D

Data Input A-7
file input A-10
keyboard input A-7
obtaining from users A-8
special characters A-9
writing direction A-8
database column names A-38
database manipulation 22-1
overview 22-1
SQL database interface *see SQL database interface*
database servers *see MicroStation database servers*
dBase A-38
DBTYPE resource 22-11
debugger *see MDL debugger* 9-1
DEC Multinational ASCII *see ASCII Character Set*
Default Header Files dialog box 4-20
dependencies *see bmake*
design file elements 11-1
complex elements 11-1, 11-4
element range 11-4
Developing and Testing A-49
hardware A-49
software A-49
dialog box builder 2-2, 4-1, 4-4
alignment menu 4-13
edit menu 4-7
overview 2-2, 4-4
tools palette 4-14
window menu 4-14
dialog box coordinate system 15-14
DCOORD_RESOLUTION 15-14
dialog coordinate units 15-14
pixel units 15-14
XC 15-14
YC 15-14
Dialog Box Design A-38
string expansion A-39
dialog box item 16-1
composition of 15-8
coordinates of within dialog box 15-14
keyboard focus 15-13
list specification 15-8
resource specification 15-8
state of 15-9
synchronization 15-12
synonym resources 15-12
dialog box items 15-1
application variables 15-1
button group item 16-93
color picker item 15-3, 16-38
color picker pull-down menu 16-2, 16-55
common fields *see common item resource fields*
creation of 15-1
example of 18-1
generic item 16-2, 16-90
group box item 16-1, 16-11
hook functions 15-1, 15-4
icon command 16-2
icon command frames 16-2
item list specification 15-3
label item 16-1, 16-9
level map item 16-41
list box item 16-2, 16-77
menu bar item 16-43
multi-line text item 16-1, 16-33
option button item 16-1, 16-21
option pull-down menu 16-52
overview 15-1
popup menu item 16-2, 16-104
radio button item 16-106
sash item 16-2, 16-98
scale item 16-100
scroll bar item 16-1, 16-26
separator item 16-1, 16-12
text item 15-3, 16-1, 16-28
text pull-down menu 16-47
toggle button 15-2
toggle button item 16-1, 16-13
tool palette 15-3, 16-2, 16-57
dialog box manager 6-16
application variables 15-2
contrasted to other GUIs 15-1, 15-2
creating MDL applications 15-1
creation of dialog boxes 15-2
dialog box items 15-2
dialog item handler 15-17
dialog items 15-1
example dialog box application 15-2, 15-21
example of use 18-1
guidelines 15-2
hook functions 15-1, 15-17
input loop 15-4
internal architecture 15-16
internationalization concerns 15-5
item handlers 15-2, 15-17

item list specification 15-3
mdlDialog_open 6-16
MicroStation windows 15-6
modification of application state variables 15-1
overview 6-16, 15-1
portability concerns 15-5, 15-7
recommended reading 15-22
related MDL functions 15-2
resource manager 15-7
resources 15-7
reusing dialog items 15-3
synchronization 15-9
synonym resources 15-3
user interaction
explanation 15-16
dialog box manager functions 15-16
dialog box manager header files 15-19
dlogbox.h 15-19
dlogids.h 15-19
dlogitem.h 15-19
dlogman.fdf 15-19
dialog box manager hook functions 17-1
button messages 17-20
dialog hook functions 17-1
focus messages 17-10
item hook functions 17-1
miscellaneous messages 17-26
non-requested messages 17-6
open & close messages 17-24
overview 17-1
size messages 17-16
types of 17-1
dialog box manager internal architecture 15-16
dialog box manager functions 15-16
hook functions 15-16, 15-17
item handler functions 15-16
messages 15-16
sub-systems of 15-16
dialog box manager library 15-19
dialog box style guidelines 19-1
appearance concerns 19-2
helpful suggestions 19-1
internationalization concerns 19-1
modeless vs. modal dialog boxes 19-2
overview 19-1
push buttons 19-3
dialog boxes 1-6, 1-10
color 15-16
coordinates 15-14
creation of 15-2, 15-20
creation of using dialog box builder 2-2
designing dialog boxes 3-5
example dialog box application 15-2
example of modeless dialog box 18-1
fonts 15-16
hook functions 15-4
mdlDialog_hookPublish 15-20
modal dialog boxes 15-9
modeless dialog boxes 15-9
modification concerns 15-7
overview 15-1, 15-6
portability concerns 15-5
sinking 15-13
synonym resources 15-3
dialog button items
push button item 16-16
dialog coordinate units *see dialog box coordinate system*
dialog hook function messages 17-4
DIALOG_MESSAGE_ACTION BUTTON 17-23
DIALOG_MESSAGE_ ACTIVATE 17-15
DIALOG_MESSAGE_ANOTHE RCLOSED 17-25
DIALOG_MESSAGE_ANOTHE ROPENED 17-25
DIALOG_MESSAGE_BEFORE DESTROY 17-8
DIALOG_MESSAGE_BUTTON 17-20
DIALOG_MESSAGE_ CALCSIZE 17-17
DIALOG_MESSAGE_CHILDD ESTROYED 17-24
DIALOG_MESSAGE_CREATE 17-6
DIALOG_MESSAGE_ DESTROY 17-9
DIALOG_MESSAGE_ FOCUSIN 17-11
DIALOG_MESSAGE_ FOCUSOUT 17-11
DIALOG_MESSAGE_ FONTCHANGED 17-18
DIALOG_MESSAGE_ HIDE 17-9
DIALOG_MESSAGE_INIT 17-8
DIALOG_MESSAGE_ ITEMFOCUSIN 17-12
DIALOG_MESSAGE_ ITEMFOCUSOUT 17-13
DIALOG_MESSAGE_ KEYSTROKE 17-14
DIALOG_MESSAGE_ MAXIMIZE 17-19
DIALOG_MESSAGE_ MINIMIZE 17-19
DIALOG_MESSAGE_ PREBUTTON 17-20
DIALOG_MESSAGE_RESIZE 17-17
DIALOG_MESSAGE_ STATECHANGED 17-26
DIALOG_MESSAGE_SYNCH 17-26
DIALOG_MESSAGE_UPDATE 17-26
DIALOG_MESSAGE_USER 17-27
dialog hook functions 15-17, 17-1, 17-2
DialogMessage structure 17-2
example of 17-2
ID numbers 15-18
message handling 15-17
messages *see dialog hook function messages*
overview 15-17, 17-1

dialog item handler 15-17
passing messages 15-17
purposes of 15-17
dialog item list specification 15-8
dialog item resource specification 15-8
overview 15-8
Dialog Manager C-6
command window C-7
key-in region C-9
message handling C-9
overall title bar region C-9
prompt region C-9
status area region C-9
status bar C-7
dialog manager hook functions 1-10
DialogBoxRsc structure 16-2
DialogItem structure 17-29
DialogItemMessage structure 17-28
DialogItemRsc structure 16-6
DialogMessage structure 17-2
Direction of Writing A-8
MS_RIGHTLOGICKB A-8
directory structure 3-8
DItem_IconCmdFrameXRsc 16-67
DItem_IconCmdRsc 16-67
DITEMLIB
mdlDialog_colorPicker C-7
mdlDialog_icFrame C-7
mdlDialog_icPalette C-7
mdlDialog_labelSetAttributes C-7
mdlDialog_levelMap C-7
mdlDialog_listBox C-7
mdlDialog_menuBar C-7
mdlDialog_optionButton C-7
mdlDialog_optionPDMItem C-7
mdlDialog_pushButton C-7
mdlDialog_radioButton C-7
mdlDialog_textPDMItem C-7
mdlDialog_toggleButton C-7
DITEMLIB Functions C-7
DLM
see Dynamic Link Modules
DLM (Dynamic Link Module) 1-6, 20-1
accessing MicroStation built-ins 20-3
application resources 20-11
calling into other DLM's 20-27
Clipper specifics *see DLM Clipper specifics*
debugging 20-28
DLM functions 20-29
DLM hook functions *see DLM hook functions*
DLM memory functions *see DLM memory functions*
dlmSystem_displayError 20-12
dynamic link source file 20-10
error reporting function 20-12
example DLM 20-10
file functions *see DLM file functions*
function pointers 20-5
HP700 specifics *see DLM HP700 specifics*
IBM RS/6000 specifics *see DLM IBM RS/6000 specifics*
linking with MDL applications 20-4
MDL asynchronous functions 20-12
MDL compiler 20-3
overview 20-1
PC protected mode specifics *see DLM PC protected mode specifics*
platform specifics overview 20-13
programming concerns 20-2
runtime concerns 20-4
Silicon Graphics specifics *see DLM Silicon Graphics specifics*
SPARCstation specifics *see DLM SPARCstation specifics*
specific terms 20-1
use with shared libraries 20-1
uses of 20-1
uses with C++ 20-1
Windows NT specifics *see DLM Windows NT specifics*
DLM Clipper specifics 20-13
C++ DLM specifics 20-14
overview 20-13
static constructor functions 20-14
static destructor functions 20-14
unload handler function 20-16
DLM file functions 20-30
dlmSystem_callMdlFunction 20-31
dlmSystem_setFunction 20-31
initialize 20-32
overview 20-30
userHook_dlmUnload 20-33
userHook_mdlUnload 20-33
DLM hook functions 20-12
dlmSystem_setFunction 20-12
initialize function 20-12
overview 20-12
DLM HP700 specifics 20-18
C++ DLM specifics 20-19
debugging 20-18
overview 20-18
DLM IBM RS/6000 specifics
C++ DLM specifics 20-25
debugging 20-26
dlmSystem_publishSymbolNames 20-26
overview 20-23
DLM memory functions
dlmSystem_mdlCalloc 20-30
dlmSystem_mdlFree 20-30
dlmSystem_mdlMalloc 20-30
dlmSystem_mdlRealloc 20-30
DLM PC protected mode specifics
C++ DLM specifics 20-20
overview 20-19
DLM Silicon Graphics specifics
C++ DLM specifics 20-23
debugging 20-23
overview 20-22
DLM SPARCstation specifics
C++ DLM specifics 20-17
debugging 20-17

overview 20-16
solaris 20-17
SunOS 20-16
DLM Windows NT specifics
C++ DLM specifics 20-22
dlmspec Utility 20-8, 21-12
dlmspec utility
commands 20-8
dlmSystem_callMdlFunction 20-12, 20-31
dlmSystem_displayError 20-12
dlmSystem_mdlCalloc 20-30
dlmSystem_mdlFree 20-30
dlmSystem_mdlMalloc 20-30
dlmSystem_mdlRealloc 20-30
dlmSystem_setFunction 20-12, 20-31
DLS
see Dynamic Link Specification
DLS (Dynamic Link Specification) *see DLM (Dynamic Link Module), specific terms*
documentation 10-1
glossary of terms 10-9
help creation tools 10-1
hints 10-7, 10-9
methods of 10-1
overview 10-1
principles 10-7
tagging of character strings 10-1
tagging of paragraphs 10-1
DOS interface functions 23-1
bdos 23-1, 23-3
copyFromReal 23-4
copyToReal 23-4
example of 23-2
int86 23-1, 23-2
int86PassThrough 23-1, 23-3
intdos 23-1, 23-2
overview 23-1
Phar Lap DOS extender 23-1
segread 23-4
DS (database service) requests
DS_AdditionalRequest 22-8
DS_CloseCursor 22-7
DS_CloseDatabase 22-4
DS_CommitTransaction 22-6
DS_CopyTable 22-8
DS_DatabaseProfile 22-3
DS_DebugMode 22-8
DS_DescribeColumn 22-5
DS_DescribeDatabase 22-4
DS_DescribeTable 22-5
DS_FetchRow 22-7
DS_GetErrorInformation 22-8
DS_OpenCursor 22-6
DS_OpenDatabase 22-4
DS_ProcessForm 22-7
DS_ProcessStatement 22-6
DS_RollbackTransaction 22-6
DS_Terminate 22-4
Dynamic Link Modules 21-1
Dynamic Link Specification 21-3
accomodating MDL shared libraries 21-12
dynamic link specification source files 20-8
defines 20-8
dlmspec utility 20-8
dynamic resources 6-7

E

element descriptors 11-1
design file elements 11-1
functions using 11-4
header information 11-1
memory allocation 11-1
memory concerns 11-5
msElementDescr structure 11-1
overview 11-1
recursive programming 11-4
recursive programming example 11-4
uses of 11-1
validation of header data 11-4
Engineering Standards A-1
EUC
see Extended UNIX Code
event driven programming 1-5, 1-13
input handlers 1-14
input loop 1-13
input queue 1-14
processing events 1-14
task ID dispatcher 1-14
example DLM application
dlink 20-10
example MDL applications 1-14
basic.ma 18-1
building example applications 14-2
calculat 14-2
chngtxt 14-1, 14-4
dlogdemo 14-3, 14-4, 15-21
dosfunc.mc 23-2
iconedit 15-21
MS environment variable 14-2
newitems 16-103
plashape 14-1
rasticon 14-1, 16-90
running example applications 14-2
trumpet 14-1
exception handling 1-8
faults detected by MDL 1-8
faults detected by operating system 1-8
Extended ASCII
see ASCII Character Set
Extended UNIX Code A-5
external state *see state, dialog box item*

F

FDF (Function Definition File) *see MDL and the C Programming Language*
fgetwc A-20
fgetws A-20
File Input A-10
database files A-10
text files A-10
UCM files A-10
UNIX platforms A-10
File Open Dialogs C-5
file types 1-9
C source files 1-9
command functions 1-9
extensions 1-11

main function 1-9
resource source files 1-9, 1-10
state functions 1-10
fillers 6-4, 6-6
Flyover Help
see Tool Boxes, tool descriptions
focus *see keyboard focus*
FONTEDIT A-35
Fonts A-29
fonts *see dialog boxes*
fputwc A-20
fputws A-20
FrameMaker 10-2
fwprintf A-20
fwscanf A-20

G

generic item *see dialog box items, generic item*
getch A-8
getwc A-8, A-20
getwchar A-20
getws A-20
group box item *see dialog box items, group box item*
GUI (Graphical User Interface)
1-3, 6-2, 15-1
Macintosh 15-5
MicroSoft Windows 15-5
Motif 15-5
X-Windows 15-5

H

help text 1-6
help tools *see on-line help*
high-level MDL application development 3-2
design tools 3-2
element placement 3-2
manipulation tools 3-2
settings boxes 3-2
user input 3-2
hook function messages 19-4
mdlDialog_dmsgsPrintDialogMsg 19-6
mdlDialog_dmsgsPrintItemMsg 19-6
mdlDialog_dmsgsSet 19-5
tracking 19-4
hook functions 1-2, 15-1, 15-16, 15-17
debugging 19-3
dialog box manager message dialog 19-3
dialog hook functions 17-1
dialog hook functions *see dialog hook functions*
dialog manager hook functions 1-10
ID numbers 15-18
item hook functions *see item hook functions*
mdlDialog_hookPublish 15-19
number space 15-18
return type 17-1
uses of 15-1, 15-4
when required 15-4

I

I/Help file format (Intergraph Help) 10-2
creation of 10-2
portability 10-2
Icon Command Resources 16-75
CmdItemListRsc 16-75
IconCmdRscs 16-75
icon editor 2-2, 4-1, 4-14
edit menu 4-16
file menu 4-15
import menu 4-16
IconCmdRscs 16-75
error in 16-75
recommendation when using CmdItemListRsc 16-75
icons 2-2
creation of using icon editor 2-2
Ident pragma *see pragmas, Ident*
If A-30
immediate commands 3-6
Include Files 4-19
default files 4-20
incompatiblePointerParameters *see pragmas, incompatiblePointerParameters*
incompatiblePointers *see pragmas, incompatiblePointers*
incompatibleReturn *see pragmas, incompatibleReturn*
initialize 20-12, 20-32, 21-13
input handlers 1-14
input queue 1-14
int86 23-1, 23-2
int86PassThrough 23-1, 23-3
intdos 23-1, 23-2
internal state *see state, dialog box item*
International Standards Organization A-15
Internationalization A-1
definition of A-1
introduction A-1
misconceptions A-1
overview A-2
recommendations A-2
requirements A-7
internationalization concerns 15-5
isalnum A-16
isalpha A-16
iscntrl A-16
isdigit A-16
isgraph A-16
islower A-16
isprint A-16
ispunct A-16
isspace A-16
iswalnum A-16
iswalpha A-16
iswcntrl A-16
iswdigit A-16
iswgraph A-16
iswlower A-16
iswprint A-16
iswpunct A-16
iswspace A-16
iswupper A-16
iswxdigit A-16
isxdigit A-16

item handler *see dialog item handler*
item handler functions 15-16
item hook function messages 17-34
DITEM_MESSAGE_ACTIVATE 17-54
DITEM_MESSAGE_ALLCREATED 17-37
DITEM_MESSAGE_BUTTON 17-42
DITEM_MESSAGE_CREATE 17-36
DITEM_MESSAGE_DESTROY 17-38
DITEM_MESSAGE_DRAW 17-51
DITEM_MESSAGE_FOCUSIN 17-46
DITEM_MESSAGE_FOCUSOUT 17-47
DITEM_MESSAGE_FONTCHANGED 17-53
DITEM_MESSAGE_GETSTATE 17-38
DITEM_MESSAGE_GETVALUE 17-50
DITEM_MESSAGE_HIGHLIGHT 17-51
DITEM_MESSAGE_INIT 17-37
DITEM_MESSAGE_JOURNALSTATE 17-45
DITEM_MESSAGE_KEYSTROKE 17-48
DITEM_MESSAGE_MOVE 17-52
DITEM_MESSAGE_POSTKEYSTROKE 17-50
DITEM_MESSAGE_QUEUECOMMAND 17-41
DITEM_MESSAGE_SETENABLEDSTATE 17-53
DITEM_MESSAGE_SETEXTENT 17-52
DITEM_MESSAGE_SETLABEL 17-50
DITEM_MESSAGE_SETSTATE 17-39
DITEM_MESSAGE_SETVALUE 17-51
DITEM_MESSAGE_STATECHANGED 17-41
DITEM_MESSAGE_SYNCHRONIZE 17-41
DITEM_MESSAGE_USER 17-42
item hook functions 15-17, 17-1, 17-28
DialogItem structure 17-29
DialogItemMessage structure 17-28
example of 17-28
general item messages 17-36
generic item messages 17-50
ID numbers 15-18
input focusable messages 17-46
item hook function messages 17-34
message handling 15-17
overview 15-17, 17-1
RawItemHdr structure 17-32
item list specification 15-3

K

keyboard focus 15-13
Keyboard Input A-7
Asian languages A-8
European languages A-7

L

label item *see dialog box items, label item*
Languages
categories A-2
level map item *see dialog box items, level map item*
library files (.ml) 7-24
linking MDL applications 7-24
mdlInput_waitForMessage 7-26
list box item *see dialog box items, list box item*
List Boxes 16-84
modifying to include icons 16-85
little endian format 6-29, 6-31
Locale A-13
Locale Initialization Functions
setlocale A-15
Locale Initization Functions A-15
Localization A-1

M

Macintosh Encoding *see Character Encoding*
Macros 4-16
macros *see bmake*
main function 1-9
makefile record types 7-3
conditionals 7-3
dependencies 7-3
macros 7-3, 7-4
rule 7-3
makefiles 7-1, A-42
creation of 7-2
example of 7-3, 18-24
format of 7-3
mblen A-15
mbstowcs A-14, A-16
mbtowc A-14
mcomp 5-1, 7-21
action upon encountering char 5-1
automatic variable memory limitation 5-2
bitfields 5-2, 5-14
cmdName keyword 5-2
cmdNumber keyword 5-2
command line syntax 7-21
enumerated types 5-2
float data type 5-2
MDL_COMP environment variable 7-22
options 7-21
overview 7-21
preprocessor directives 5-2
special considerations when using preprocessor directives 5-2

storage class specification 5-2
structures as function arguments 5-1
structures as function return values 5-1
upon encountering short data type 5-1
MDE editing tools 4-1
command table editor 4-1
dialog box builder 4-1
icon editor 4-1
string list editor 4-1
MDE on-line help 2-2
MDE workspace 2-1
overview 2-1
tools menu 2-1
MDL (MicroStation Developer's Language) 5-1
bitfield handling *see bitfield handling*
contrasted to C Programming Language 5-1
structures *see structure handling*
MDL and the C Programming Language
analogous MDL functions 12-1
formatting strings 12-2
Function Definition Files 12-3
standard header files 12-1
stderr 12-1
stdin 12-1
stdout 12-1
supported C functions 12-1
MDL application development 1-12, 2-1, 3-1
consideration of user 3-1
database considerations 3-2
functional specification 3-1
getting started 3-1
object attributes 3-2
object definition 3-1
object manipulation 3-2
object placement 3-2
project task list 3-3
tools 2-1
MDL application development *see high-level MDL application development*
MDL application development approaches 1-12
advantages of each 1-12
MDL application organization 1-12
approaches to MDL development 1-12
MDL applications 1-5
debugging 9-1
documenting of 10-1
dynamic link modules 1-6
external programs 1-7
loading and unloading of 8-1, 8-3
MDL application defined 1-5
MDL program defined 1-5
MDL task defined 1-5
on-line help creation 10-2
parts of 3-7
primitive commands 3-7
running of 8-1
user interface 3-7
utility commands 3-7
MDL built-in variables 13-1
defining of non-base type built-in variables 13-1
global variables 13-1
read-only variables 13-1
MDL Command Names A-38
MDL Compiler C-2
changes to C-2
compiler warnings C-2
MDL debugger 9-1
first time debugging 9-1
input to 9-2
output 9-2
overview 9-1
PC concerns 9-2
preparing input to 9-4
setup 9-2
source code searching 9-3
unix concerns 9-2
MDL development utilities 1-9
MDL function pointers 1-8
MDL fundamentals 1-2
advantages over MicroCSL 1-3
built-in functions 1-2
C programming language 1-2
event driven programming 1-5
GUI 1-3
hook functions 1-2
overview 1-2
portability 1-3
resource manager 1-3
MDL librarian *see mlib utility*
MDL overview 1-1
application task identifier 1-1
file types 1-1, 1-9
MDL application organization 1-1, 1-12
MDL applications 1-1
MDL development utilities 1-1, 1-9
MDL fundamentals 1-1
MDL runtime environment 1-1
MDL runtime environment 1-7
accessing files 1-7
dynamic memory concerns 1-7
exception handling 1-8
MDL function pointers 1-8
overview 1-7
MDL Shared Libraries
applications 21-1
creation of 21-2
determining when MDL program is unloaded 21-10
DLM comparison 21-1
hook functions 21-13
linking MDL programs to 21-3
loading 21-6
mdlDialog_callFunction 21-2, 21-13
non-switch function calls 21-13
non-switched function calls 21-2
overview 21-1
runtime concerns 21-13
switched function calls 21-2, 21-13
unloading 21-9

mdlCExpression_symbolPublish 15-10
mdlCharTran_ internalDisplayToNative A-10
mdlCharTran_ nativeToInternalDisplay A-10
mdlCnv_bufferFromFileFormat

6-32, 6-33
mdlCnv_toScanFormat 6-32
mdlDB_activeDatabase 22-12
mdlDB_readColumn A-10
mdlDB_writeColumn A-10
mdlDialog_callFunction 21-2
mdlDialog_colorPickerGetInfo 16-41
mdlDialog_colorPickerSetInfo 16-41
mdlDialog_dmsgsClear 19-4
mdlDialog_dmsgsPrint 19-4
mdlDialog_dmsgsPrintDialogMsg 19-6
mdlDialog_dmsgsPrintItemMsg 19-6
mdlDialog_dmsgsSet 19-5
mdlDialog_fileCreate C-5
mdlDialog_fileOpen C-5
mdlDialog_hookPublish 15-19, 15-20
mdlDialog_icFrameGetItemInfo 16-60
mdlDialog_icFrameGetNItems 16-60
mdlDialog_icFrameGetNSubItems 16-60
mdlDialog_icFrameSelectIcon 16-60
mdlDialog_icFrameSetItemInfo 16-60
mdlDialog_icPaletteGetItemInfo 16-62
mdlDialog_icPaletteGetNItems 16-62
mdlDialog_icPaletteSelectIcon 16-62
mdlDialog_icPaletteSetItemInfo 16-62
mdlDialog_itemGetByTypeAndId 16-10, 16-11, 16-12
mdlDialog_itemGetValue 16-37
mdlDialog_itemSetLabel 16-9, 16-11
mdlDialog_itemSetValue 16-37
mdlDialog_keyinWindowGet C-9
mdlDialog_levelMapGetInfo 16-43
mdlDialog_levelMapSetInfo 16-43
mdlDialog_listBoxDeleteAll 16-83
mdlDialog_listBoxDeleteColumn 16-83
mdlDialog_listBoxDrawContents 16-83
mdlDialog_listBoxEnableCells 16-83
mdlDialog_listBoxGetColInfo 16-83
mdlDialog_ listBoxGetDisplayRange 16-83
mdlDialog_listBoxGetInfo 16-83
mdlDialog_ listBoxGetLocationCursor 16-83
mdlDialog_listBoxGetNColumns 16-83
mdlDialog_listBoxGetNextSelection 16-83
mdlDialog_listBoxGetSelections 16-84
mdlDialog_listBoxGetSelectRange 16-84
mdlDialog_listBoxGetStrListP 16-84
mdlDialog_listBoxInsertColumn 16-84
mdlDialog_listBoxIsCellEnabled 16-84
mdlDialog_listBoxIsCellSelected 16-84
mdlDialog_listBoxLastCellClicked 16-84
mdlDialog_listBoxNRowsChanged 16-84
mdlDialog_listBoxSelectCells 16-84
mdlDialog_listBoxSetColInfo 16-83
mdlDialog_listBoxSetInfo 16-83
mdlDialog_listBoxSetLocationCursor 16-83
mdlDialog_listBoxSetSelections 16-84
mdlDialog_listBoxSetStrListP 16-77, 16-84
mdlDialog_listBoxSetTopRow 16-84
mdlDialog_ listBoxSetTopRowRedraw 16-84
mdlDialog_ listRowNRowsChanged 16-78
mdlDialog_menuBarAddAppMenu 16-45
mdlDialog_menuBarAddCmdWin Menu 16-45, C-9
mdlDialog_menuBarAttachMenu 16-46
mdlDialog_ menuBarDeleteAllItems 16-46
mdlDialog_ menuBarDeleteAllMenus 16-46
mdlDialog_ menuBarDeleteCmdWinMenu 16-46
mdlDialog_menuBarDeleteItem 16-46
mdlDialog_menuBarDeleteMenu 16-46
mdlDialog_menuBarDetachMenu 16-46
mdlDialog_menuBarFind 16-46
mdlDialog_ menuBarFindAppMenu 16-46
mdlDialog_menuBarFindItem 16-46
mdlDialog_menuBarFindMenu 16-46
mdlDialog_menuBarGetCmdWinP 16-46, C-9
mdlDialog_menuBarGetItem 16-46
mdlDialog_menuBarGetMenu 16-46
mdlDialog_menuBarGetNItems 16-46
mdlDialog_menuBarGetNMenus 16-46
mdlDialog_menuBarGetSelection 16-46

mdlDialog_menuBarInsertMenu 16-46
mdlDialog_menuBarInsMenu 16-46
mdlDialog_menuBarMenuGetEnabled 16-46
mdlDialog_menuBarMenuGetTitle 16-47
mdlDialog_menuBarMenuSetEnabled 16-47
mdlDialog_menuBarMenuSetTitle 16-47
mdlDialog_mlTextGetCursor 16-38
mdlDialog_mlTextGetInfo 16-38
mdlDialog_mlTextGetLineCoords 16-38
mdlDialog_mlTextGetLineRange 16-38
mdlDialog_mlTextInsertString 16-38
mdlDialog_mlTextSetCursor 16-38
mdlDialog_mlTextSetInfo 16-38
mdlDialog_mlTextTopRowNumber 16-38
mdlDialog_open 6-16, 15-11, 15-21
mdlDialog_openAlert C-10, C-12
mdlDialog_openInfoBox C-10, C-12
mdlDialog_openMessageBox C-10, C-12
mdlDialog_openModal 15-11
mdlDialog_openPalette 16-60
mdlDialog_optionButtonDeleteAll 16-25
mdlDialog_optionButtonDeleteItem 16-25
mdlDialog_optionButtonGetItemInfo 16-25
mdlDialog_optionButtonGetNItems 16-25
mdlDialog_optionButtonInsertItem 16-26
mdlDialog_optionButtonSetEnabled 16-26
mdlDialog_optionButtonSetItemInfo 16-25
mdlDialog_optionPDMItemGetInfo 16-55
mdlDialog_optionPDMItemInsert 16-55
mdlDialog_optionPDMItemSetEnabled 16-55
mdlDialog_optionPDMItemSetInfo 16-55
mdlDialog_overallTitleBarGet C-9
mdlDialog_publishComplexVariable A-28
mdlDialog_pushButtonActivate 16-19
mdlDialog_pushButtonGetInfo 16-20
mdlDialog_pushButtonSetCancel 16-20
mdlDialog_pushButtonSetDefault 16-20
mdlDialog_pushButtonSetInfo 16-20
mdlDialog_scrollArrowDraw 16-28
mdlDialog_scrollBarGetInfo 16-28
mdlDialog_scrollBarSetInfo 16-28
mdlDialog_scrollBarSetRange 16-28
mdlDialog_selectIconsByCmd 16-65
mdlDialog_selectIconsByCmdNoMsg 16-65
mdlDialog_selectIconsById 16-65
mdlDialog_selectIconsByIdNoMsg 16-65
mdlDialog_stringWidth A-41
mdlDialog_synonymsSynch 18-16
mdlDialog_textGetInfo 16-33
mdlDialog_textGetRange 16-33
mdlDialog_textPDMItemGetInfo 16-51
mdlDialog_textPDMItemIns 16-52
mdlDialog_textPDMItemInsert 16-52
mdlDialog_textPDMItemSetEnabled 16-52
mdlDialog_textPDMItemSetInfo 16-52
mdlDialog_textPDMItemSetLabel 16-52
mdlDialog_textPDMItemSetMark 16-52
mdlDialog_textSetInfo 16-33
mdlDialog_textSetRange 16-33
mdlDialog_toggleButtonGetInfo 16-16
mdlDialog_toggleButtonSetInfo 16-16
mdlElmdscr_freeAll 8-4
mdlFile_setDefaultShare 12-2
mdlInput_waitForMessage 7-26
mdlOutput_error C-9
mdlOutput_printf A-48
mdlOutput_vprintf 12-2, A-48
mdlOutput_XXX Routines C-9
mdlParse_loadCommandTable 6-15
mdlParse_loadKeywordTable 6-15
mdlParse_unloadTable 6-15
mdlResource_add 6-22, 6-35
mdlResource_closeFile 6-17
mdlResource_createFile 6-17
mdlResource_delete 6-12
mdlResource_directAdd 6-12
mdlResource_directAddComplete 6-12
mdlResource_load 6-12, 6-33, 6-34
mdlResource_loadFromStringList A-44
mdlResource_openFile 6-16, 6-33, 6-34, 15-20
mdlResource_queryFileHandle 6-33
mdlResource_resize 6-12
mdlResource_write 6-12
mdlshare_allocTaskBlock 21-13
mdlStringList_create 16-77
mdlStringList_deleteMember 16-78
mdlStringList_insertMember 16-78
mdlStringList_setMember 16-78
mdlSystem_closeDesignFile 8-7
mdlSystem_createStartupElement 8-3
mdlSystem_enterGraphics 8-6
mdlSystem_exit 8-3
mdlSystem_getChar A-8
mdlSystem_getCurrMdlDesc 20-11
mdlSystem_getenv 12-2

mdlSystem_loadMdlProgram 8-2, 22-12
mdlSystem_newDesignFile 8-6, 8-7
mdlSystem_setFunction 8-6, 12-1, 21-10
mdlText_ addStringsToModeDscrWide A-36
mdlText_compressStringWide A-36
mdlText_create A-29
mdlText_createWide A-28, A-29, A-36
mdlText_expandStringWide A-36
mdlText_extractStringWide A-36
mdlText_extractWide A-28, A-36
mdlTextMode_createWide A-36
mdlTextMode_ createWithStringsWide A-36
mdlTextMode_ extractStringFromDscrWide A-36
mdlTextMode_extractWide A-36
mdlWindow_close 8-6
mdlWindow_lineStyleSet 15-16
menu bar item *see dialog box items, menu bar item*
Message Box Icons C-10
  MSGBOX_ICON_CRITICAL C-10
  MSGBOX_ICON_ INFORMATION C-10
  MSGBOX_ICON_QUESTION C-10
  MSGBOX_ICON_WARNING C-10
Message Boxes C-10
  DIALOGID_MsgBoxOK C-10
  DIALOGID_MsgBoxOKCancel C-10
  DIALOGID_ MsgBoxYesNo C-10
  DIALOGID_MsgBoxYesNoCancel C-10
  mdlDialog_openAlert C-10
  mdlDialog_openInfoBox C-10
  mdlDialog_openMessageBox C-10
Message Files
  see Messaging and Resource Files
Message Handling C-9
message list resources 6-8
message lists 1-6, 1-10
message subsystem 6-14
MessageList 6-8
Messaging and Resource Files A-37
MicroCSL 1-3, C-4
  data sharing concerns 1-3
Microsoft Word 10-2
MicroStation 95
  AccuDraw C-6
  macro capabilities C-4
MicroStation database server
  Initapp example 22-13
MicroStation database servers
  initapp, startup application 22-12
  mdlDB_activeDatabase 22-12
  mdlSystem_loadMdlProgram 22-12
  overview 22-12
MicroStation input loop 1-13
  input queue 1-14
MicroStation Resource Manager 6-16
MicroStation windows 15-6
MIF (Maker Interchange Format) 10-2
mif2s utility *see on-line help*
MIT X Consortium A-30
mlib utility 7-24
  command line syntax 7-24
  library file (.ml) 7-24
  object files (.mo) 7-24
  overview 7-24
mlink 1-11, 6-6, 7-24, 21-3
  command line syntax 7-25
  executable files 1-11
  library files (.ml) 7-24
  object files (.mo) 7-24
  program files (.mp) 7-24
modal dialog boxes 15-9, 15-10
  invoking 15-11
  mdlDialog_open 15-11
  mdlDialog_openModal 15-11
modeless dialog boxes 15-9, 15-10
MS_CODESET Environment Variable A-6
MS_INITAPPS
  stdin concerns 12-1
MS_INITAPPS applications 8-5
  batch processing 8-7
  overview 8-5
MS_INITAPPS environment variable 8-5
MSGBOX_ICON_CRITICAL C-10, C-12
MSGBOX_ICON_INFORMATION C-10, C-12
MSGBOX_ICON_QUESTION C-10, C-12
MSGBOX_ICON_WARNING C-10, C-12
mswchar.fdf A-15
MSWideInt A-13
MSWideType A-13
multi-line text item *see dialog box items, multi-line text item*

N

native code *see DLM (Dynamic Link Module), specific terms*
noAnsiDeclaration *see pragmas, noAnsiDeclaration*
noAnsiDeclOn
  see Pragma
Non-switched Function Calls 21-13
noReturnStatement *see pragmas, noReturnStatement* 5-4
number space *see hook functions*
Numbered Argument Lists A-48

O

object files (.mo) 7-24
on-line help 2-2, 10-2
  contents of 2-2
  context-sensitive help tagging 10-9
  creation of 10-2
  creation of I/Help .s files 10-5
  I/Help compiler 10-2

I/Help source files 10-2
MIF (Maker Interchange Format) 10-2
mif2s utility 10-5
mif2s utility, command line syntax 10-5
reasons for 10-9
RTF (Rich Text Format) 10-2
rtf2s utility 10-5
rtf2s, command line syntax 10-6
style sheets 10-2
templates, FrameMaker 10-2, 10-9
templates, Microsoft Word 10-2, 10-9
opaque pointers 17-3
option button item *see dialog box items, option button item*
option pull-down menu *see dialog box items, option pull-down menu*
options pragma *see pragmas, options*
Output and Display A-29
Fonts A-29
messages A-29
output devices A-29
plotting and printing A-37
prompts A-29
text placement in design files A-36
translated text A-29

P

packedLittleEndianData pragma *see pragmas, packedLittleEndianData*
Palettes 16-66
comparison to tool boxes 16-66
conversion to tool boxes 16-74
Paths for Include Files dialog box 4-19
P-Code *see pseudocode*
Phar Lap DOS extender *see DOS interface functions*
pixel units *see dialog box coordinate system*
pointerToNative pragma *see pragmas pointerToNative*
poput menu item *see dialog box items, popup menu item*
portability 6-27
binary portability 6-27
PostScript Fonts A-35
Pragma C-2
noAnsiDeclOn C-2
pragmas 5-2
alias 5-2
Ident 5-10
incompatiblePointerParameters 5-4
incompatiblePointers 5-4
incompatibleReturn 5-4
noAnsiDeclaration 5-4
noReturnStatement 5-4
options 5-7
packedLittleEndianData 5-8
packLittleEndianData 6-31
pointerToNative 5-9
resourceID 5-9
suppressREQCmds 5-11
translate 5-12
undeclaredFunction 5-4
Version 5-13
primitive commands 1-10, 1-12, 3-6, 3-7
input to 1-12
interaction with view commands 1-10
printf 12-2
program file (.mp) 7-24
programs 8-1
project task list 3-3
application initialization logic 3-3
pseudocode 1-6
pseudo-code instructions *see DLM (Dynamic Link Module), specific terms*
publishing 15-20
publishStructures 15-20
push button item *see dialog box item, push button item*
putwc A-20
putwchar A-20
putws A-20

R

radio button item *see dialog box items, radio button item*
Radix-50 A-38
Raster Font (X-Window BDF Font) A-29
creating new raster fonts A-30
RawItemHdr structure 17-32
rcomp 6-3, 6-6
command line syntax 7-16, 7-20
error detection 7-16
options 7-20
overview 6-6
undefined length structures 6-3
RDE (Resource Development Environment) 4-1
recursive programming 11-4
debugging concerns 11-4
element descriptor example 11-4
element descriptors 11-4
recursive functions 11-4
stack usage 11-4
Resource C-3
resource class 6-5
message list resources 6-8
table resources 6-8
resource compiler *see rcomp*
resource declaration 6-3
Resource Development Environment
creation of example 4-20
limitations 4-22
modification of example 4-21
Tab Page item 16-110
Resource Development

Environment dialog box 4-17
File Menu 4-17
Options Menu 4-19
Resource Menu 4-18
Resource Files A-47, C-3
Asian text A-47
backwards compatibility C-3
compiling A-47
see Messaging and Resource Files
resource ID 6-5
resource librarian 7-27, 15-7
overview 7-27
rlib utility 7-27
resource manager 1-3, 6-11, 15-7
creating new resources 6-11
deleting existing resources 6-11, 6-12
mdlResource_openFile 15-20
updating existing resources 6-11, 6-12
resource manager functions 6-11
resource source files 1-10
basic.r example 18-5
command tables 1-10, 2-1
dialog boxes 1-10, 2-2
icons 2-2
message lists 1-10
string lists 2-2
supportability of pointers 6-4, 6-9
syntax of 6-3
uses of 1-10
variable sized arrays 6-5, 6-9
resource source generator 2-2
overview 2-2
resourceclass statement 6-3
syntax of 6-3
resourceID pragma *see pragmas resourceID*
resources 6-1
compiling 7-19
creating 6-1, 6-6
dynamic resources *see dynamic resources*
example of 6-1, 6-4, 6-22
managing 6-1, 6-16
modifying 6-1, 6-9
overview 6-1
portability concerns 6-1, 6-27
resource file 15-7
resource IDS 15-7
resource IDs 15-7
resource manager 6-1
resource source generator 6-1
resource type 15-7
static resources
see static resources
uses of 15-7
utilities 6-1
resources overview 6-1
advantages of 6-2
application specific resources 6-3
identification of resources 6-1
resources defined 6-1
syntax of resource source files 6-3
uses of 6-1
resources, applications accessing of 6-16
resources, creation of 6-6
dynamic resource example 6-7
dynamic resources 6-6
static resource example 6-7
static resources 6-6
resources, management of 6-13, 6-16
command parser 6-14
dialog box manager 6-16
message subsystem 6-14
resources, modification of 6-9
navigating about resources 6-9
rlib utility 6-6, 7-27
command line syntax 7-27
RscDirectAccess structure 6-12
rsctype utility 7-17, 15-10, 15-20
command line syntax 7-19
options 7-19
overview 7-17
predefined macros 7-18
RTF (Rich Text Format) 10-2
rtf2s utility *see on-line help*
RTYPE_DialogBox resource type 15-7
rules *see bmake*

S

sash item *see dialog box items, sash item*
scale item *see dialog box items, scale item*
scanf 12-2
scroll bar item *see dialog box items, scroll bar item*
Scroll Bars 16-28
segread 23-4
separator item *see dialog box items, separator item*
Server Natural Format A-30
SETLANG Utility
see Codeset Handler
setlocale A-15
showsnf A-33
sinkable dialog boxes 15-13
SNF
see Server Natural Format
Software Enhancement Stages
internationalization A-1
localization A-1
Source Files A-42
Special Characters A-9
Special Considerations
context sensitive characters A-2
direction of writing A-2
SQL database interface 22-1
architecture 22-1
communications 22-9
DatabaseService structure 22-2
DBTYPE resource *see DBTYPE resource*
descriptor message structure 22-9
environment variables *see SQL environment variables*
session debug 22-10
SQL requests 22-2
toolkit 22-1

SQL environment variables
MS_DBASE 22-10
MS_DBDOUBLE 22-11
MS_DBEXT 22-10
MS_DBFLOAT 22-10
MS_LINKTYPE 22-11
MS_SERVER 22-10
state functions 1-10
events 1-10
primitive commands 1-10
viewing commands 1-10
state handlers *see event driven programming*
state, dialog box item 15-9
external state 15-9, 15-12
internal value 15-9, 15-12
obtaining 15-9
setting 15-9
synchronization 15-9
static resources 6-7
defined 6-7
Status Bar C-7
stderr 12-1
stdin 12-1
stdlib.h A-14
stdout 12-1
strcat A-13
strcmp A-13
strcpy A-13
string formatting 12-2
string list editor 2-2, 4-1, 4-24
edit menu 4-26
file menu 4-25
string lists 2-2
creation of using string list editor 2-2
String Processing A-11
practices to avoid A-11
strings in Dialog Box Resource files A-45
strings in program source files A-44
untranslatable strings A-38
Wide Characters A-13
Strings in Dialog Box Resource Files A-45
Strings in Program Source Files A-44
common mistakes A-44
strlen A-11
structure handling 5-14
strupr A-28
stub commands 3-6
style sheets *see on-line help*
suppressREQCmds pragma *see pragmas, suppressREQCmds*
Switched Function Calls 21-13
swprintf A-20
swscanf A-20
synchronization 15-9, 15-12
example of 18-16
purpose of 15-12
synonym resources 15-3, 15-12, 15-17
example of 15-12, 18-1, 18-9

T

Tab Page 16-113
item hook function messages 16-115
item list specification 16-113
item resource specification 16-114
Tab Page item 16-110
tab page list resource 16-110
tab page resource 16-110
uses of 16-110
Tab Page list
item hook function messages 16-113
item list specification 16-110
item resource specification 16-111
table identifier *see command tables*
table resources 6-8
Target Languages A-2
task 8-1
task ID 1-8, 8-1
task ID dispatcher 1-14
TCB (Terminal Control Block) 15-9
templates *see on-line help*
Testing
see Developing and Testing
text item *see dialog box items, text item*
text pull-down menu *see dialog box items, text pull-down menu*
TextParam Structure A-37
TextParamWide Structure A-37
There 17-1
title bar *see windows*
toggle button item *see dialog box items*
toggle button item *see dialog button items, toggle button item*
Tool Boxes 16-66
adding into icon command frame 16-70
comparison to palettes 16-66
icon palette conversion 16-67
summary 16-73
tool decriptions 16-72
tool tips 16-72
tool palette *see dialog box items, tool palette*
tool palette item *see dialog box items*
towlower A-16
towupper A-16, A-28
translate pragma *see pragmas, translate*
TrueType Fonts A-7, A-35, A-49
TSR (Terminate and Stay Resident) 1-3

U

undeclaredFunction *see pragmas, undeclaredFunction*
undefined length structures 6-3
ungetwc A-20
Unicode A-6
creating Unicode fonts A-7
Untranslatable Strings A-38
cell names A-38
database column names A-38
MDL command names A-38
user interface 3-7

userHook_allMdlUnload 21-14
userHook_dlmUnload 20-33
userHook_mdlUnload 20-33, 21-14
userHook_
sharedLibNoMoreClients 21-14
userpref.h A-33
userSystem_reloadProgram 8-3
userSystem_unloadProgram
8-4, 12-1
ustation.rsc 4-1
ustnhelp.mde 2-3
utility commands 3-6, 3-7

V

Vector Fonts A-34
character mapping of A-34
common problems A-35
creating libraries for foreign languages A-35
Version pragma *see pragmas, Version*
vfwprintf A-20
view commands 1-10
interaction with primitive commands 1-10
View Window Icons 16-28
mdlView_getViewRectangle 16-28
mdlWindow_contentRectGet Global 16-28
mdlWindow_
contentRectGetLocal 16-28
vswprintf A-20
vwprintf A-20

W

Warnings
see MDL Compiler Warnings
wchar.h A-13
wchar_t
see Wide Characters
wcscat A-16, A-28
wcschr A-16
wcscmp A-16
wcscoll A-17
wcscpn A-16
wcscpy A-16, A-28
wcsftime A-19
wcslen A-16, A-28
wcsncat A-16
wcsncmp A-16
wcspbrk A-16
wcsrchr A-16
wcsspn A-16
wcsstr A-16, A-28
wcstod A-20
wcstok A-16
wcstol A-20
wcstombs A-14, A-16, A-28
wcstoul A-20
wcsxfrm A-17, A-19
wctomb A-14
wctype_t
see Wide Characters
Wide Character Functions
A-3, A-15
case conversion functions A-16
character classification functions A-16
character collation functions A-17
date functions A-19
I/O functions A-20
locale initialization A-15
number conversion functions A-20
printing functions A-20
scanning functions A-20
string functions A-16
time functions A-19
Worldwide Portability Interfaces A-15
Wide Character Processing A-12
limitations A-20
locale A-12
overview A-12
performance considerations A-21
Wide Character String Functions
A-16
wcscat A-16
wcschr A-16
wcscpn A-16
wcscpy A-16
wcslen A-16
wcsncat A-16
wcsncmp A-16
wcspbrk A-16
wcsrchr A-16
wcsspn A-16
wcsstr A-16
wcstok A-16
Wide Characters A-13
data type A-13
declaration of A-13
String Format A-13
wchar_t A-13
wctype_t A-13
wint_t A-13
widechar.h A-13, A-16
windows 15-6
components of 15-6
defined 15-6
resizing 15-6
wint_t
see Wide Characters
Worldwide Portability Interfaces
see Wide Character Functions
wprintf A-20
wscanf A-20
WTF (Words To Follow) 11-1

X

X Windows A-30
XC *see dialog box coordinate system*

Y

YC *see dialog box coordinate system*