Programming Tools

Microsoft® WINDOWS™

SOFTWARE DEVELOPMENT KIT

Microsoft® Windows™

Version 3.1

Programming Tools

For the Microsoft Windows Operating System

Microsoft Corporation

Information in this document is subject to change without notice and does not represent a commitment on the part of Microsoft Corporation. The software, which includes information contained in any databases, described in this document is furnished under a license agreement or nondisclosure agreement and may be used or copied only in accordance with the terms of that agreement. It is against the law to copy the software except as specifically allowed in the license or nondisclosure agreement. No part of this manual may be reproduced in any form or by any means, electronic or mechanical, including photocopying and recording, for any purpose without the express written permission of Microsoft Corporation.

© 1987–1992 Microsoft Corporation. All rights reserved.
Printed in the United States of America.

ITC Zapf Chancery and ITC Zapf Dingbats fonts. Copyright © 1991 International Typeface Corporation. All rights reserved.
Copyright © 1981 Linotype AG and/or its subsidiaries. All rights reserved. Helvetica, Palatino, Times and Times Roman typefont data is the property of Linotype or its licensors.
Arial and Times New Roman fonts. Copyright © 1991 Monotype Corporation PLC. All rights reserved.

Microsoft, MS, MS-DOS, QuickC, and CodeView are registered trademarks, and Windows and QuickBasic are trademarks of Microsoft Corporation.

U.S. Patent No. 4974159

Adobe and PostScript are registered trademarks of Adobe Systems, Inc.
The Symbol fonts provided with Windows version 3.1 are based on the CG Times font, a product of AGFA Compugraphic Division of Agfa Corporation.
Apple, Macintosh, and TrueType are registered trademarks of Apple Computer, Inc.
UNIX is a registered trademark of UNIX Systems Laboratories.
PANOSE is a trademark of ElseWare Corporation.
Epson and FX are registered trademarks of Epson America, Inc.
Hewlett-Packard, HP, and LaserJet are registered trademarks of Hewlett-Packard Company.
Lotus is a registered trademark of Lotus Development Company.
IBM and Personal System/2 are registered trademarks of International Business Machines Corporation.
ITC Zapf Chancery and ITC Zapf Dingbats are registered trademarks of the International Typeface Corporation.
Helvetica, Palatino, Times, and Times Roman are registered trademarks of Linotype AG and/or its subsidiaries.
Intel is a registered trademark of Intel Corporation.
Arial and Times New Roman are registered trademarks of the Monotype Corporation PLC.
Okidata is a registered trademark of Oki America, Inc.

Document No. PC28918-0492

Contents

Introduction .... **ix**

Organization of This Manual ix
Document Conventions x

Chapter 1 Creating and Editing Resources **1**

1.1 Designing Images: Image Editor 3
1.2 Designing Dialog Boxes: Dialog Editor 4
1.3 Designing Fonts: Font Editor 5

Chapter 2 Compiling Resources: Resource Compiler **7**

2.1 Including Resources in an Application 9
2.2 Creating a Resource-Definition File 9
2.2.1 Single-Line Statements 10
2.2.2 Multiline Statements 10
2.3 Using Resource Compiler 12
2.3.1 Command-Line Syntax 12
2.3.2 Compiling Resources Separately 15
2.3.3 Defining Names for the Preprocessor 16
2.3.4 Renaming the Compiled Resource File 16
2.3.5 Controlling Which Directories the Resource Compiler Searches 17
2.3.6 Displaying Progress Messages 18
2.4 Related Topics 18

Chapter 3 Creating Help Files **19**

3.1 About Windows Help Files 21
3.2 Creating Topic Files 21
3.2.1 Declaring Character Set, Fonts, and Colors 22
3.2.2 Defining Individual Topics 23
3.2.3 Setting Font Size and Name 24
3.2.4 Setting Space Before and After Paragraphs 24
3.2.5 Setting the Left and Right Indents 24
3.2.6 Setting Tab Stops 25
3.2.7 Breaking Lines 25
3.2.8 Creating Links and Pop-up Topics 26

3.2.9 Creating a Keyword List 27
3.2.10 Creating Browse Sequences 27
3.3 Using Graphics Files 28
3.3.1 Inserting a Bitmap in Text 29
3.3.2 Wrapping Text Around a Bitmap 29
3.3.3 Using a Bitmap as a Hot Spot 30
3.3.4 Using a Bitmap on Different Displays 30
3.4 Creating Help Project Files 32
3.4.1 Project File Sections 32
3.4.2 Using Macros in Project Files 33
3.4.3 Sample Project File 33
3.5 Using Help in a Windows Application 34
3.5.1 Choosing Help from the Help Menu 34
3.5.2 Choosing Help with the Keyboard 36
3.5.3 Choosing Help with the Mouse 39
3.5.4 Searching for Help with Keywords 42
3.5.5 Displaying Help in a Secondary Window 43
3.5.6 Canceling Help 44
3.6 Project File Sections and Options Reference 45

Chapter 4 Debugging: CodeView for Windows 67

4.1 Requirements for Using CodeView for Windows 69
4.1.1 Using CVW with a Single Monitor 70
4.1.2 Using CVW with a Secondary Monitor 70
4.2 Comparing CodeView for Windows with Other Microsoft Debuggers 71
4.2.1 Differences Between CVW and SYMDEB 71
4.2.2 Differences Between CVW and CodeView for MS-DOS 72
4.3 Preparing Windows Applications for Debugging 73
4.4 Setting Up the Debugging Version of Windows 73
4.5 Starting a Debugging Session 74
4.5.1 Display Options 75
4.5.2 Starting a Debugging Session for a Single Application 75
4.5.3 Starting a Debugging Session for Multiple Instances of an Application 76
4.5.4 Starting a Debugging Session for Multiple Applications 76
4.5.5 Starting a Debugging Session for Dynamic-Link Libraries 77
4.5.6 Command-Line Options 78
4.6 Saving Session Information 80

4.7 Working with the CodeView for Windows Screen 80
4.7.1 Using CVW Display Windows 80
4.7.2 Using the Menu Bar 83
4.8 Accessing Help 85
4.9 Displaying Application Data 85
4.9.1 Displaying Variables 86
4.9.2 Displaying Expressions 87
4.9.3 Displaying Arrays and Structures 87
4.9.4 Using the Quick Watch Command 91
4.9.5 Tracing Windows Messages 91
4.9.6 Displaying Memory 92
4.9.7 Displaying the Contents of Registers 97
4.9.8 Displaying Windows Modules 98
4.10 Modifying Application Data 98
4.11 Controlling Execution of Your Application 99
4.11.1 Continuous Execution 99
4.11.2 Single-Step Execution 103
4.11.3 Animated Execution 104
4.11.4 Jumping to a Particular Location 104
4.11.5 Interrupting Your Application 104
4.12 Handling Abnormal Termination of the Application 106
4.12.1 Handling a Fatal Exit 106
4.12.2 Handling a General Protection Fault 107
4.13 Ending a Session 108
4.14 Advanced Techniques 108
4.14.1 Using Multiple Source Windows 108
4.14.2 Checking for Undefined Pointers 108
4.14.3 Handling Register Variables 109
4.14.4 Redirecting CodeView for Windows Input and Output 109
4.15 Modifying the TOOLS.INI File 110
4.16 Related Topics 110

Chapter 5 Advanced Debugging: 80386 Debugger 111

5.1 Preparing Symbol Files for 80386 Debugger 113
5.2 Starting 80386 Debugger 114
5.3 Entering 80386 Debugger 116
5.4 Command Syntax 118
5.4.1 Command Keys 118
5.4.2 Command Parameters 118

5.4.3 Binary and Unary Operators 121
5.4.4 Regular Expressions 122
5.5 Common Commands 123
5.6 Reference of 80386 Debugger Commands 125
5.7 Related Topics 170

Chapter 6 Analyzing System Failures: Dr. Watson 171

6.1 Configuring Dr. Watson from the WIN.INI File 173
6.1.1 The SkipInfo Entry 173
6.1.2 The ShowInfo Entry 174
6.1.3 The DisLen Entry 174
6.1.4 The TrapZero Entry 175
6.1.5 The GPContinue Entry 175
6.1.6 The DisStack Entry 176
6.1.7 The LogFile Entry 176
6.2 Sample Dr. Watson Log File 177
6.3 Sample Dr. Watson Log File with Comments 179

Chapter 7 Monitoring Messages: Spy 183

7.1 Selecting Options: The Options! Menu 185
7.1.1 Selecting Message Types 185
7.1.2 Selecting the Output Device 186
7.1.3 Selecting Frequency of Output 186
7.2 Selecting a Window: The Window Menu 187
7.3 Starting and Stopping Spy: The Spy Menu 187
7.4 Related Topics 188

Chapter 8 Monitoring Dynamic Data Exchange Activity: DDESpy 189

8.1 The Output Menu 191
8.2 The Monitor Menu 191
8.2.1 Monitoring String-Handle Data 192
8.2.2 Monitoring Sent or Posted DDE Messages 192
8.2.3 Monitoring Callbacks 193
8.2.4 Monitoring Errors 193
8.3 Tracking Options 193
8.3.1 Tracking String Handles 194
8.3.2 Tracking Active Conversations 194
8.3.3 Tracking Active Links 194
8.3.4 Tracking Registered Servers 194

Chapter 9 Viewing the Heap: Heap Walker 195

9.1 The Heap Walker Window 197
9.2 Performing File Operations: The File Menu 198
9.3 Waiking the Heap: The Walk Menu 199
9.4 Sorting Memory Objects: The Sort Menu 199
9.5 Displaying Memory Objects: The Object Menu 200
9.5.1 The Show Command 200
9.5.2 The LocalWalk Commands 201
9.6 Allocating Memory: The Alloc Menu 203
9.7 Determining Memory Size: The Add! Menu 203
9.8 Suggestions for Using Heap Walker 204
9.9 Related Topics 204

Chapter 10 Analyzing Performance: Profiler 205

10.1 Overview of Profiler 207
10.2 Preparing to Run Profiler 208
10.3 Using Profiler Functions 208
10.4 Sampling Code 209
10.5 Displaying Samples 209

Chapter 11 Compressing and Decompressing Files 213

11.1 Compressing Files: Compress 215
11.2 Decompressing Compressed Files: Expand 216

Appendix A Resource Compiler Diagnostic Messages 217

Appendix B Help Compiler Error Messages 229

B.1 Interpreting Error Messages 231
B.2 Error Message Categories 231
B.3 File Errors 232
B.4 Project-File Errors 233
B.5 Macro Errors 237
B.6 Context-String Errors 237
B.7 Topic-File Errors 239
B.8 Miscellaneous Errors 240

Appendix C Windows Debugging Version 243

C.1 Debugging Programs 245
C.1.1 Logging Debugging Messages 246
C.1.2 Interpreting Debugging Messages 247
C.2 Debugging Functions and the WINDEBUGINFO Structure 249
C.2.1 WIN.INI Debugging Options 250
C.3 Debugging Messages 251
C.4 Common Programming Errors 254

Index 257

Introduction

The Microsoft® Windows™ operating system is a single-user system for personal computers. Microsoft provides a variety of tools you'll find useful as you develop Windows applications. This manual, *Microsoft Windows Programming Tools*, explains how to use these tools.

Organization of This Manual

Following are brief descriptions of the chapters and appendixes in this manual:

- Chapter 1, "Creating and Editing Resources," introduces three tools you can use to create and edit resources for Windows applications. These tools are Microsoft Image Editor (IMAGEDIT.EXE), Microsoft Dialog Editor (DLGEDIT.EXE), and Microsoft Windows Font Editor (FONTEDIT.EXE).
- Chapter 2, "Compiling Resources: Resource Compiler," describes how to use Microsoft Windows Resource Compiler (RC) to compile application resources and add them to an executable Windows application.
- Chapter 3, "Creating Help Files," describes how to use Microsoft Help Compiler, Microsoft Multiple Resolution Bitmap Compiler, and Microsoft Hotspot Editor to develop help files.
- Chapter 4, "Debugging: CodeView for Windows," explains how to use Microsoft CodeView® for Windows (CVW) to test the execution of your Windows applications and examine your data simultaneously.
- Chapter 5, "Advanced Debugging: 80386 Debugger," shows how to use Microsoft Windows 80386 Debugger (WDEB386.EXE) to test and debug Windows applications and dynamic-link libraries (DLLs).
- Chapter 6, "Analyzing System Failures: Dr. Watson," discusses how to use Microsoft Windows Dr. Watson (DRWATSON.EXE) to detect and analyze failures caused by Windows applications.
- Chapter 7, "Monitoring Messages: Spy," shows how to use Microsoft Windows Spy (SPY.EXE) to monitor messages sent to one or more windows in your Windows application.
- Chapter 8, "Monitoring Dynamic Data Exchange Activity: DDESpy," explains how to use Microsoft Windows DDESpy (DDESPY.EXE) to monitor dynamic data exchange (DDE) activity in the Windows operating system.

- Chapter 9, "Viewing the Heap: Heap Walker," shows how to use Microsoft Windows Heap Walker (HEAPWALK.EXE) to examine local and global heaps used by applications and DLLs in the Windows operating system.
- Chapter 10, "Analyzing Performance: Profiler," explains how to use Microsoft Windows Profiler to analyze and optimize the performance of applications running with the Windows operating system in 386 enhanced mode.
- Chapter 11, "Compressing and Decompressing Files," discusses how to use Microsoft File Compression Utility (COMPRESS.EXE) and Microsoft File Expansion Utility (EXPAND.EXE) to compress and decompress files.
- Appendix A, "Resource Compiler Diagnostic Messages," describes diagnostic messages produced by Microsoft Resource Compiler (RC).
- Appendix B, "Help Compiler Error Messages," lists error messages that Help Compiler can display if errors occur while you are building a help file.
- Appendix C, "Windows Debugging Version," presents information about Windows diagnostic messages to help you debug applications you develop for the Windows operating system.

Document Conventions

The following conventions are used throughout this manual to define syntax:

Convention	Meaning
Bold text	Denotes a term or character to be typed literally, such as a resource-definition statement or function name (**MENU** or **CreateWindow**), a command, or a command-line option (**/nod**). You must type these terms exactly as shown.
Italic text	Denotes a placeholder or variable: You must provide the actual value. For example, the statement **SetCursorPos**(*X*, *Y*) requires you to substitute values for the *X* and *Y* parameters.
[ ]	Enclose optional parameters.
|	Separates an either/or choice.
...	Specifies that the preceding item may be repeated.
BEGIN . . . END	Represents an omitted portion of a sample application.

In addition, certain text conventions are used to help you understand this material:

Convention	Meaning
SMALL CAPITALS	Indicate the names of keys, key sequences, and key combinations—for example, ALT+SPACEBAR.
FULL CAPITALS	Indicate filenames and paths, type names and most structure names (which are also bold), and constants.
`monospace`	Sets off code examples and shows syntax spacing.

Creating and Editing Resources

Chapter 1

1.1 Designing Images: Image Editor 3
1.2 Designing Dialog Boxes: Dialog Editor 4
1.3 Designing Fonts: Font Editor 5

This chapter introduces three tools you can use to create and edit resources for your Microsoft Windows applications. These tools are Microsoft Image Editor, Microsoft Dialog Editor, and Microsoft Windows Font Editor. You can find full documentation for these tools in Help.

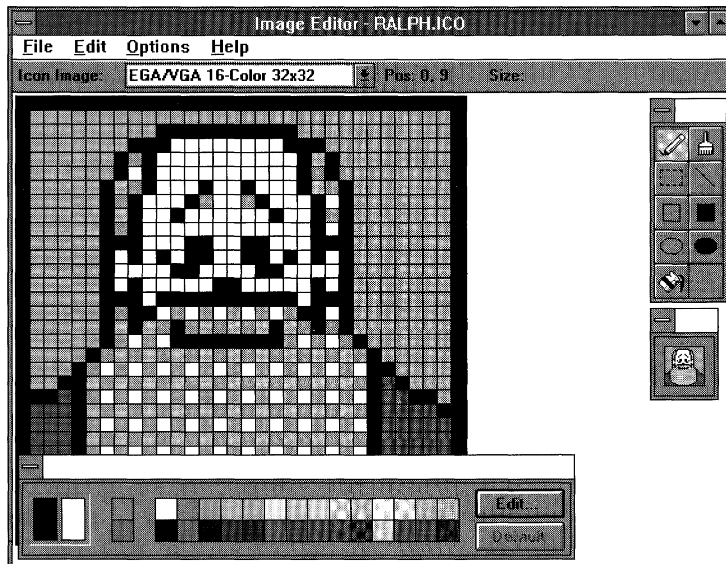
1.1 Designing Images: Image Editor

With Image Editor (IMAGEDIT.EXE), you can create graphical images to represent files, windows, cursors, and other objects in your Windows applications. Image Editor provides you with a set of drawing tools for creating commonly used shapes.

Image Editor contains context-sensitive Help that includes information about how to create the following kinds of images:

- Cursors, which represent the position the mouse is pointing to. They are also called pointers.
- Bitmaps, which represent static graphical images.
- Icons, which represent files or windows.

The following illustration shows the Image Editor window after a user has opened an icon file.



You must use a mouse or similar pointing device with Image Editor.

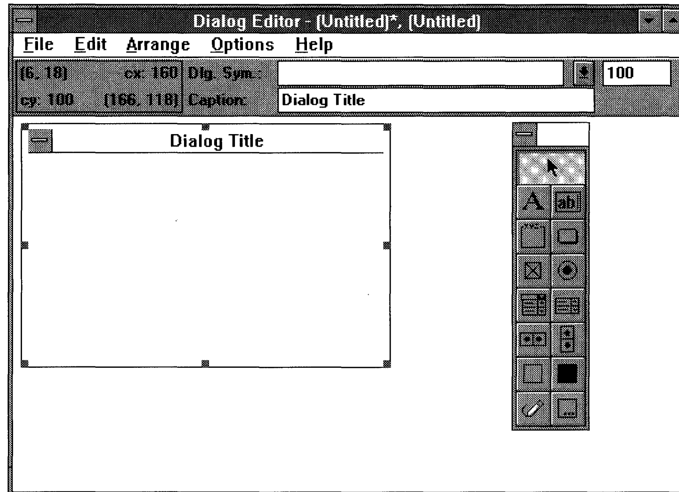
1.2 Designing Dialog Boxes: Dialog Editor

With Dialog Editor (DLGEDIT.EXE), you can design and test a dialog box on your screen instead of defining **DIALOG** statements in a resource-definition file. Using Dialog Editor, you can add, modify, and delete controls in a dialog box. Dialog Editor saves the changes you make as resource-definition statements. You then compile these statements into a binary resource file that is linked to your application's executable file.

Dialog Editor contains context-sensitive Help that includes information about the following topics:

- How Dialog Editor works with files
- Viewing the Dialog Editor window
- Opening resource files, header files, and dialog boxes
- Working with dialog boxes
- Editing individual controls
- Working with groups of controls
- Moving a dialog box between resources
- Working with header files
- Installing custom controls

The following illustration shows the Dialog Editor window after a user has chosen the New command from the File menu.



You must use a mouse or similar pointing device with Dialog Editor.

1.3 Designing Fonts: Font Editor

With Font Editor (FONTEDIT.EXE), you can modify existing fonts to create new fonts for your applications. The Font Editor Help describes how to use Font Editor to do the following:

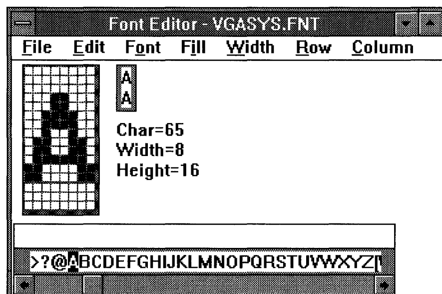
- Edit letters, numbers, and other characters in a font
- Modify the height, width, and character mapping of a font
- Change information in the font-file header

To view Help for Font Editor, start Microsoft Windows Help (WINHELP.EXE) and open FONTEDIT.HLP.

You can use Font Editor to create and edit raster fonts. Font Editor cannot create or modify vector or TrueType fonts.

You must use a preexisting font file to create a new font file with Font Editor. Two font files are supplied with Font Editor: ARTM1111.FNT and VGASYS.FNT. For a fixed-pitch (monospace) font, you can edit ATRM1111.FNT; for a variable-pitch font, you can edit VGASYS.FNT.

The following illustration shows the Font Editor window after a user has opened VGASYS.FNT from the Open File dialog box.



After creating a new font with Font Editor, you must add the new font to a font resource file. For information about adding a customized font to a font resource file and using it in a Windows application, see the *Microsoft Windows Guide to Programming*.

You must use a mouse or similar pointing device with Font Editor.

Compiling Resources: Resource Compiler

Chapter 2

2.1 Including Resources in an Application 9
2.2 Creating a Resource-Definition File 9
    2.2.1 Single-Line Statements 10
    2.2.2 Multiline Statements 10
        2.2.2.1 Directives 10
        2.2.2.2 Sample Resource-Definition File 11
2.3 Using Resource Compiler 12
    2.3.1 Command-Line Syntax 12
        2.3.1.1 Specifying Options 13
        2.3.1.2 Specifying the Resource-Definition File 15
        2.3.1.3 Specifying the Executable File 15
        2.3.1.4 Renaming the Executable File 15
    2.3.2 Compiling Resources Separately 15
    2.3.3 Defining Names for the Preprocessor 16
    2.3.4 Renaming the Compiled Resource File 16
    2.3.5 Controlling Which Directories the Resource Compiler Searches 17
        2.3.5.1 Adding a Directory to Search 17
        2.3.5.2 Suppressing the INCLUDE Environment Variable 18
    2.3.6 Displaying Progress Messages 18
2.4 Related Topics 18

Microsoft Windows Resource Compiler (RC) is a tool for the Microsoft Windows operating system. This chapter describes how to create a resource-definition file and how to compile your application's resources and add them to the application's executable file.

2.1 Including Resources in an Application

To include resources in your Windows application, do the following:

1. Create individual resource files for cursors, icons, bitmaps, dialog boxes, and fonts. To do this, you can use Microsoft Image Editor and Dialog Editor (IMAGEDIT.EXE and DLGEDIT.EXE) and Microsoft Windows Font Editor (FONTEDIT.EXE).
2. Create a resource-definition file that describes all the resources used by the application.
3. Use RC to compile the resource-definition file.
4. Add the compiled resource files to the application's compiled executable file.

2.2 Creating a Resource-Definition File

After creating individual resource files for your application's icon, cursor, font bitmap, and dialog box resources, you create a resource-definition file. A resource-definition file is an ASCII text file with the file extension .RC.

The .RC file lists every resource in your application and describes some types of resources in great detail. For a resource that exists in a separate file, such as an icon or cursor, the .RC file simply names the resource and the file that contains it. For some resources, such as a menu, the entire definition of the resource exists within the .RC file.

An .RC file can contain either or both of the following types of information:

- Statements, which name and describe resources.
- Directives, which instruct RC to perform actions on the resource-definition file before compiling it. Directives can also assign values to names.

The following sections describe the statements and directives you can use in a resource-definition file. For detailed descriptions and syntax, see the *Microsoft Windows Programmer's Reference, Volume 4.*

2.2.1 Single-Line Statements

A single-line resource-definition statement can begin with any of the following keywords:

Keyword	Description
BITMAP	Defines a bitmap by naming it and specifying the name of the file that contains it. (To use a particular bitmap, the application requests it by name.)
CURSOR	Defines a cursor by naming it and specifying the name of the file that contains it. (To use a particular cursor, the application requests it by name.)
FONT	Specifies the name of a file that contains a font.
ICON	Defines an icon by naming it and specifying the name of the file that contains it. (To use a particular icon, the application requests it by name.)

2.2.2 Multiline Statements

A multiline resource-definition statement can begin with any of the following keywords:

Keyword	Description
ACCELERATORS	Defines menu accelerator keys.
DIALOG	Defines a template that an application can use to create dialog boxes.
MENU	Defines the appearance and function of an application menu.
RCDATA	Defines data resources. Data resources let you include binary data directly into the executable file.
STRINGTABLE	Defines string resources. String resources are null-terminated ASCII strings that can be loaded from the executable file.

2.2.2.1 Directives

The following directives can be used as needed in the resource-definition file to instruct RC to perform actions or to assign values to names:

Keyword	Description
#define	Defines a specified name by assigning it a given value.
#elif	Marks an optional clause of a conditional compilation block.
#else	Marks the last optional clause of a conditional compilation block.
#endif	Marks the end of a conditional compilation block.

Keyword	Description
#if	Carries out conditional compilation if a specified expression is true.
#ifdef	Carries out conditional compilation if a specified name is defined.
#ifndef	Carries out conditional compilation if a specified name is not defined.
#include	Copies the contents of a file into the resource-definition file before RC processes the latter.
#undef	Removes the current definition of the specified name.

2.2.2.2 Sample Resource-Definition File

The following example shows an .RC file that defines the resources for an application named Shapes:

```
#include "SHAPES.H"

ShapesCursor  CURSOR  SHAPES.CUR
ShapesIcon    ICON    SHAPES.ICO

ShapesMenu    MENU
    BEGIN
        POPUP "&Shape"
            BEGIN
                MENUITEM "&Clear", ID_CLEAR
                MENUITEM "&Rectangle", ID_RECT
                MENUITEM "&Triangle", ID_TRIANGLE
                MENUITEM "&Star", ID_STAR
                MENUITEM "&Ellipse", ID_ELLIPSE
            END
    END
```

The **CURSOR** statement names the application's cursor resource ShapesCursor and specifies the cursor file SHAPES.CUR, which contains the image for that cursor.

The **ICON** statement names the application's icon resource ShapesIcon and specifies the icon file SHAPES.ICO, which contains the image for that icon.

The **MENU** statement defines an application menu named ShapesMenu, a pop-up menu with five menu items.

The menu definition, enclosed by the **BEGIN** and **END** keywords, specifies each menu item and the menu identifier that is returned when the user selects that item. For example, the first item on the menu, Clear, returns the menu identifier ID_CLEAR when the user selects it. The menu identifiers are defined in the application header file, SHAPES.H.

For more information about resource-definition files, the syntax of resource statements, and how to define your own resources, see the *Microsoft Windows Programmer's Reference, Volume 4*.

2.3 Using Resource Compiler

Resource Compiler (RC) serves the following functions:

- It compiles the resource-definition file and the resource files (such as icon and cursor files) into a binary resource (.RES) file.
- It combines the .RES file with the executable (.EXE) file created by the linker; the result is an executable Windows application.
- It marks the Windows application as a Windows version 3.0 or Windows 3.1 application.

Note Each Windows application and dynamic-link library (DLL) must be identified with a Windows version number. For this reason, use RC on each Windows application or DLL you build, even if it uses no resources. For more information about Windows versions, see the descriptions of the **/30** and **/31** options in Section 2.3.1.1, "Specifying Options."

2.3.1 Command-Line Syntax

To start RC, use the **rc** command. What you need to specify on the command line depends on whether you are compiling resources, adding compiled resources to an executable file, or both.

The following line shows **rc** command-line syntax:

rc [*options*] *definition-file* [*executable-file*]

Following are several ways you can use the **rc** command:

- To compile resources separately, use the **rc** command in the following form:

  rc /r [*options*] *definition-file*

  When you use this form, RC ignores any executable file you specify.
- To compile an .RC file and add the resulting .RES file to the executable file, use the **rc** command in the following form:

  rc [*options*] *definition-file* [*executable-file*]

- To compile an application or DLL that does not have a .RES file, use the **rc** command in the following form:

  rc [*options*] *dll-or-executable-file*

  When you use this form, the filename must explicitly have an .EXE, .DRV, or .DLL extension.

- To simply add a compiled resource (.RES) file to an executable file, use the **rc** command in the following form:

  rc [*options*] *res-file***.res** [*executable-file*]

2.3.1.1 Specifying Options

The **rc** command's *options* parameter can include one or more of the following options:

/30
Marks the executable file so it will run with Windows version 3.0 or Windows version 3.1. By default, RC marks the executable file to run only with Windows 3.1.

/31
Marks the executable file so it will run only with Windows 3.1. This is the default condition.

/?
Displays a list of **rc** command-line options.

/d
Defines a symbol for the preprocessor that you can test with the **#ifdef** directive.

/e
Changes the default location of global memory for a DLL from below the Expanded Memory Specification (EMS) bank line to above the EMS bank line. This option has no effect with Windows 3.1.

/fe *newname*
Uses *newname* for the name of the .EXE file.

/fo *newname*
Uses *newname* for the name of the .RES file.

/h
Displays a list of **rc** command-line options.

/i
Searches the specified directory before searching the directories specified by the INCLUDE environment variable.

/k

Disables the load-optimization feature of RC. If this option is not specified, the compiler arranges segments and resources in the executable file so that all preloaded information is contiguous.

This feature allows Windows to load the application much more quickly. If you do not specify the **/k** option, all data segments, nondiscardable code segments, and the entry-point code segment will be preloaded, unless any segment and its relocation information exceed 64K. If the **PRELOAD** attribute is not assigned to these segments in the module-definition (.DEF) file when you link your application, RC will add the **PRELOAD** attribute and display a warning. Resources and segments will have the same segment alignment. This alignment should be as small as possible to limit the size of the final executable file. You can set the alignment by using the **link** command with the **/a** option.

/l[**im32**]

Specifies to Windows that the application uses expanded memory directly, according to the Lotus/Intel/Microsoft Expanded Memory Specification (LIM EMS), version 3.2. This option has no effect with Windows 3.1.

/m[**ultinst**]

Assigns each instance of the application task to a distinct EMS bank when Windows is running with the EMS 4.0 memory configuration. (By default, all instances of a task share the same EMS bank.) This option has no effect with Windows 3.1.

/p

Creates a private DLL that is called by only one application. This allows Windows to use memory more efficiently, because only one application (or multiple instances of the same application) calls the DLL. For example, in the large-frame EMS memory model, the DLL is loaded above the EMS bank line, freeing memory below the bank line. This option has no effect with Windows 3.1.

/r

Creates an .RES file from an .RC file. Use this option when you do not want to add the compiled .RES file to the .EXE file.

/t

Creates an application that runs with Windows only in protected (standard or 386 enhanced) mode. If the user attempts to run the application in real mode, Windows will display a message that the application cannot run in real mode. This option has no effect with Windows 3.1.

/v

Displays messages that report on the progress of the compiler.

/x

Prevents RC from checking the INCLUDE environment variable when searching for header files or resource files.

/z

Prevents RC from checking for **RCINCLUDE** statements. When you have not used **RCINCLUDE** statements, using this option can greatly improve the speed of RC.

Options are not case-sensitive, and a hyphen (**-**) can be used in place of a slash mark (**/**). You can combine single-letter options if they do not require any additional parameters.

2.3.1.2 Specifying the Resource-Definition File

The **rc** command's *definition-file* parameter specifies the name of the resource-definition file that contains the names, types, filenames, and descriptions of the resources to be added to the .EXE file. It can also specify the name of a compiled .RES file, in which case RC adds the compiled resources to the executable file.

2.3.1.3 Specifying the Executable File

The **rc** command's *executable-file* parameter specifies the name of the executable file that the compiled resources should be added to. If you do not specify an executable file, RC uses the executable file with the same name as the resource-definition file (excluding the filename extension).

2.3.1.4 Renaming the Executable File

The **rc** command's **/fe** option makes it possible for you to specify the name of the final executable file. The following example combines MYEXE.EXE with MYRES.RES to produce the final executable file FINAL.EXE:

```
rc /fe final.exe myres.res myexe.exe
```

2.3.2 Compiling Resources Separately

By default, RC adds the compiled resources to the specified executable file. Sometimes you might want to first compile the resources and then add them to the executable file in separate steps. This can be useful because resource files typically change little after initial development. You can save time by compiling your application's resources separately and then adding the compiled .RES file to your executable file each time you recompile the .EXE file.

You can use the **/r** option to compile the resources separately without adding them to the executable file. When you use this option, RC compiles the .RC file and creates a compiled resource (.RES) file.

For example, the following command reads the resource-definition file SAMPLE.RC and creates the compiled resource file SAMPLE.RES:

```
rc -r sample.rc
```

In this case, RC does not add SAMPLE.RES to the executable file.

2.3.3 Defining Names for the Preprocessor

You can specify conditional branching in a resource-definition file, based on whether a term is defined on the **rc** command line with the **/d** option.

For example, suppose your application has a pop-up menu, the Debug menu, that should appear only during debugging. When you compile the application for normal use, the Debug menu is not included. The following example shows the statements that can be added to the resource-definition file to define the Debug menu:

```
MainMenu MENU
BEGIN
   .
   .
   .
#ifdef DEBUG
        POPUP "&Debug"
        BEGIN
                MENUITEM "&Memory usage", ID_MEMORY
                MENUITEM "&Walk data heap", ID_WALK_HEAP
        END
#endif
END
```

When compiling resources for a debugging version of the application, you could include the Debug menu by using the following **rc** command:

```
rc -r -d debug myapp.rc
```

To compile resources for a normal version of the application—one that does not include the Debug menu—you could use the following **rc** command:

```
rc -r myapp.rc
```

2.3.4 Renaming the Compiled Resource File

By default, when compiling resources, RC names the compiled resource (.RES) file with the same name as the .RC file (but not the same extension) and places it in the same directory as the .RC file. The following example compiles

MYAPP.RC and creates a compiled resource file named MYAPP.RES in the same directory as MYAPP.RC:

```
rc -r myapp.rc
```

The **/fo** option lets you give the resulting .RES file a name that differs from the name of the corresponding .RC file. For example, to name the resulting .RES file NEWFILE.RES, you would type the following command:

```
rc -r -fo newfile.res myapp.rc
```

The **/fo** option can also place the .RES file in a different directory. For example, the following command places the compiled resource file MYAPP.RES in the directory C:\SOURCE\RESOURCE:

```
rc -r -fo c:\source\resource\myapp.res myapp.rc
```

2.3.5 Controlling Which Directories the Resource Compiler Searches

By default, RC searches for header files and resource files (such as icon and cursor files) first in the current directory and then in the directories specified by the INCLUDE environment variable. (The PATH environment variable has no effect on which directories RC searches.)

2.3.5.1 Adding a Directory to Search

You can use the **/i** option to add a directory to the list of directories RC searches. The compiler then searches the directories in the following order:

1. The current directory
2. The directory or directories you specify by using the **/i** option, in the order in which they appear on the **rc** command line
3. The list of directories specified by the INCLUDE environment variable, in the order in which the variable lists them, unless you specify the **/x** option

The following example compiles the resource-definition file MYAPP.RC and adds the compiled resources to MYAPP.EXE:

```
rc /i c:\source\stuff /i d:\resources myapp.rc
```

When compiling the resource-definition file MYAPP.RC, RC searches for header files and resource files first in the current directory, then in C:\SOURCE\STUFF and D:\RESOURCES, and then in the directories specified by the INCLUDE environment variable.

2.3.5.2 Suppressing the INCLUDE Environment Variable

You can prevent RC from using the INCLUDE environment variable when determining the directories to search. To do so, use the **/x** option. The compiler then searches for files only in the current directory and in any directories you specify by using the **/i** option.

The following example compiles the resource-definition file MYAPP.RC and adds the compiled resources to MYAPP.EXE:

```
rc /x /i c:\source\stuff myapp.rc
```

When compiling the resource-definition file MYAPP.RC, RC searches for header files and resource files first in the current directory and then in C:\SOURCE\STUFF. It does not search the directories specified by the INCLUDE environment variable.

2.3.6 Displaying Progress Messages

By default, RC does not display messages that report on its progress as it compiles. You can, however, specify that RC is to display these messages. To do so, use the **/v** option.

The following example causes RC to report on its progress as it compiles the resource-definition file SAMPLE.RC, creates the compiled resource file SAMPLE.RES, and adds the .RES file to the executable file SAMPLE.EXE:

```
rc /v sample.rc
```

2.4 Related Topics

For information about creating icons, cursors, bitmaps, dialog boxes, and fonts, see Chapter 1, "Creating and Editing Resources."

For an introduction to menus, controls, and dialog boxes, see the *Microsoft Windows Guide to Programming*.

For the syntax and description of each resource statement and directive, see the *Microsoft Windows Programmer's Reference, Volume 4*.

Creating Help Files

Chapter 3

3.1	About Windows Help Files		21
3.2	Creating Topic Files		21
	3.2.1	Declaring Character Set, Fonts, and Colors	22
	3.2.2	Defining Individual Topics	23
	3.2.3	Setting Font Size and Name	24
	3.2.4	Setting Space Before and After Paragraphs	24
	3.2.5	Setting the Left and Right Indents	24
	3.2.6	Setting Tab Stops	25
	3.2.7	Breaking Lines	25
	3.2.8	Creating Links and Pop-up Topics	26
	3.2.9	Creating a Keyword List	27
	3.2.10	Creating Browse Sequences	27
3.3	Using Graphics Files		28
	3.3.1	Inserting a Bitmap in Text	29
	3.3.2	Wrapping Text Around a Bitmap	29
	3.3.3	Using a Bitmap as a Hot Spot	30
	3.3.4	Using a Bitmap on Different Displays	30
3.4	Creating Help Project Files		32
	3.4.1	Project File Sections	32
	3.4.2	Using Macros in Project Files	33
	3.4.3	Sample Project File	33
3.5	Using Help in a Windows Application		34
	3.5.1	Choosing Help from the Help Menu	34
	3.5.2	Choosing Help with the Keyboard	36
	3.5.3	Choosing Help with the Mouse	39

3.5.4 Searching for Help with Keywords 42
3.5.5 Displaying Help in a Secondary Window 43
3.5.6 Canceling Help 44
3.6 Project File Sections and Options Reference 45

Microsoft Windows Help provides online help for users working with a Windows application. Windows Help provides a practical way to present information about your application in a format users can access easily.

This chapter introduces the tools you can use to develop Windows Help files and to incorporate Help in Windows applications.

3.1 About Windows Help Files

Windows Help files can display information by using the following elements:

- Text in multiple fonts, sizes, and colors
- Bitmaps and metafiles with up to 16 colors
- Segmented-graphics bitmaps with embedded hot spots
- Cross-reference jumps for links to additional information
- Pop-up windows to present text and graphics
- Secondary windows to present information without the full menus and buttons of Windows Help
- Keywords to help users find the information they need

You create help files by creating topic and graphics files and a Help project file. A topic file contains the text for the help topic and contains the Help statements and macros that define the format of the text and the position of graphics in each topic. The graphics files contain the bitmaps and metafiles you want to display in the topics. The project file contains a description of how to build the help file.

You use the Microsoft Help Compiler to build the final help file. Combining the topic, graphics, and project files, the compiler creates a single help file (with the filename extension .HLP) that you can open and view by using Windows Help.

3.2 Creating Topic Files

A topic file contains the text for the help file, as well as the statements and macros that define the format of the text and the position of the graphics. Every topic file consists of one or more topics. A topic is any distinct unit of information, such as a contents screen, a conceptual description, a set of instructions, a keyboard table, a glossary definition, a list of jumps, a picture, and so on.

Windows Help displays only one topic at a time, but a user can view any topic in a help file by using a link to the topic or searching for keywords associated with the topic.

You create topic files directly by using a text editor and inserting Help statements. You can create them indirectly by using a word processor that generates rich-text format (RTF) files. The Help statements are an extended subset of the RTF statements, which provide a wide variety of formatting capabilities. For a complete list of Help statements, see the *Microsoft Windows Programmer's Reference, Volume 4.*

3.2.1 Declaring Character Set, Fonts, and Colors

When you create a topic file, you must ensure that the entire contents of the file are enclosed in braces ({ }). The first statement in the file must be the **\rtf** statement; it immediately follows the first opening brace. You should follow the **\rtf** statement with a **\ansi** statement (or a similar statement) that specifies the character set used in the file. The following example shows the general form for a topic file:

```
{\rtf1\ansi
     .
     .
     .
}
```

You must declare the names of the fonts you use in the file by using a **\fonttbl** statement. The **\fonttbl** statement, enclosed in braces, contains a list of font and family names and specifies a unique number for each font. You use these numbers with **\f** statements later in the file to set specific fonts. The following **\fonttbl** statement assigns font numbers 0, 1, and 2 to the TrueType fonts Times New Roman®, Courier New®, and Arial®, respectively:

```
{\fonttbl
\f0\froman Times New Roman;
\f1\fdecor Courier New;
\f2\fswiss Arial;}
```

You should also use the **\deff** statement to set the default font for the file. Windows Help uses this default font if no other font is specified. The following example sets the default font number to zero, corresponding to the Times New Roman font specified in the previous **\fonttbl** statement:

```
\deff0
```

If you use specific text colors or choose not to rely on the default text colors set by Windows, you must define your colors by using a **\colortbl** statement. The **\colortbl** statement, enclosed in braces, defines each color by specifying the amount of each primary color (red, green, and blue) used in it. The statement implicitly numbers the colors consecutively starting from zero. You use these color numbers with **\cf** statements later in the file to set the color. The following example creates four colors (black, red, green, and blue):

```
{\colortbl
\red0\green0\blue0;
\red255\green0\blue0;
\red0\green128\blue0;
\red0\green0\blue255;}
```

Although it is not shown here, you can put a semicolon immediately after the **\colortbl** statement to define the default color as color 0.

3.2.2 Defining Individual Topics

A topic starts with one or more **\footnote** statements and ends with a **\page** statement. All text and graphics specified between these statements belong to the topic.

Every topic must have a context string. Windows Help uses the context string to locate the topic when the user requests to view it. You assign a context string to a topic by using the **\footnote** statement and the number sign (#) footnote character. Context strings can consist of letters, digits, and the underscore character (_). To prevent conflicts, each context string in a help file must be unique.

You can also assign a title to the topic by using the **\footnote** statement and the dollar sign ($) footnote character. Windows Help uses the title to identify the topic in the History and Search dialog boxes. You must provide a title if you assign keywords to the topic.

The following example defines a small topic having the context string "topic1" and the title My Topic:

```
#{\footnote topic1}
${\footnote My Topic}
This is my first topic.
\par
\page
```

In general, you use the **\par** statement to mark the end of each paragraph. In this example, the **\par** statement marks the end of the only paragraph in the topic.

You can add a macro to a topic by using the **\footnote** statement and the exclamation point (!) as the footnote character. For example, the following **\footnote** statement adds the **CopyTopic** macro to the topic:

```
!{\footnote CopyTopic()}
```

Windows Help executes the macro each time it displays the topic.

The total size of text and graphics data stored in a topic must not exceed 64K. (Bitmaps included by using the **bmc**, **bml**, and **bmr** statements do not contribute to this total.)

3.2.3 Setting Font Size and Name

You can set the font name and size by using the **\f** and **\fs** statements. The name is set by using a font number specified in the **\fonttbl** statement. The size of the font is specified in half-points. The following example sets the text to 10-point Times New Roman (if the **\fonttbl** statement matches the example given earlier):

```
\f0\fs20
```

Once you set the font name and size, the settings apply to all subsequent text up to the next **\plain** statement or until you change the name or size by using the **\f** or **\fs** statement again. The **\plain** statement resets the name and font to the defaults. The default font name is as set by the **\deff** statement; the default font size is 12 points.

3.2.4 Setting Space Before and After Paragraphs

You can set the amount of space before and after each paragraph by using the **\sb** and **\sa** statements. These statements let you control the amount of space that appears between paragraphs. You specify the space in twips. (A twip is 1/1440 inch, or 1/20 of a printer's point.) The following example sets the space before a paragraph to 360 twips:

```
\sa360
This paragraph has 360 twips space immediately before it.
\par
This paragraph also has 360 twips before it.
\par
```

Once you set the space before or after a paragraph, the spacing applies to all subsequent paragraphs up to the next **\pard** statement or until you change the spacing by using the **\sa** and **\sb** statements again. The **\pard** statement restores the default spacing.

3.2.5 Setting the Left and Right Indents

When Windows Help displays its window, it automatically creates left and right margins and wraps text to fit within these margins. The margins are positioned slightly within the left and right edges of the window to prevent text in the topic from being clipped by the window.

You can override these margins by setting the left and right indents for a paragraph. The **\li** and **\ri** statements set an indent to a position relative to the corresponding left and right margins. For example, the following paragraph is indented 1 inch (1440 twips) from the left margin:

```
\\li1440
This paragraph is indented 1 inch.
\par
\pard
This paragraph is not indented.
```

Once indents are set, they apply to all subsequent paragraphs up to the next **\pard** statement. Note that the **\pard** statement must follow the **\par** statement that ends the paragraph to be indented.

You can set an indent for the first line in a paragraph by using the **\fi** statement. This allows you to create paragraphs with hanging indents. It is also useful for creating two-column lists.

3.2.6 Setting Tab Stops

You can set tab stops by using the **\tx** statement. You can use one or more **\tx** statements, each setting a specific position in twips relative to the left margin. Once you have set tab stops, you can use the **\tab** statement to align subsequent text with the next tab. The tab settings remain active until you use the **\pard** statement. The following example creates a two-column list by using a tab stop and paragraph indenting:

```
\fi-1440\li1440\tx1440
left
\tab
right
\par
left
\tab
right
\par
\pard
```

3.2.7 Breaking Lines

Ordinarily, Windows Help wraps all lines in a paragraph, fitting as many words on a line as will fit between the current left and right indents. You can force Windows Help to break a line at a given place by using the **\line** statement. You can also direct Windows Help to forego wrapping by using the **\keep** statement. You can control wrapping by using the **\keep** and **\pard** statements.

The following example uses the **\keep** statement to turn off word wrapping for three short lines and uses the **\pard** statement to restore the default properties:

```
\keep
3 pairs black socks\line
5 pairs blue socks\line
2 pairs brown socks\line
\par
\pard
```

The following example uses the **\keep** and **\pard** statements to create three non-wrapping paragraphs:

```
\keep
3 pairs black socks
\par
5 pairs blue socks
\par
2 pairs brown socks
\par
\pard
```

3.2.8 Creating Links and Pop-up Topics

Windows Help displays only one topic at a time. To enable users to view other topics, you must create hot spots that link your topics to other topics. You create a hot spot by using the **\strike**, **\ul**, or **\uldb** statement and a corresponding **\v** statement. When you create a link, you provide the text for the hot spot and the context string for the topic that is to be jumped to or displayed. The following example creates a hot spot named Glossary and establishes a link from the hot spot to the topic having the context string "glo1":

```
You can find a list of terms used in this
help file in the {\uldb Glossary}{\v glo1}.
```

When Windows Help displays the topic with this hot spot, it places a line under the word *Glossary* and colors the word green. The context string is not shown, but if the user clicks on the hot spot, Windows Help jumps to and displays the corresponding topic.

The **\strike** and **\uldb** statements are used to create jumps to other topics. The **\ul** statement creates a link to a pop-up topic. Windows Help displays pop-up topics in a pop-up window and leaves the current topic in the main window.

You can also associate a Help macro with a hot spot in a topic. For example, the following **\uldb** and **\v** statements create a hot spot for the **ExecProgram** macro:

```
{\uldb Clock}{\v !ExecProgram("clock.exe", 1)}
```

Windows Help executes the macro whenever the user chooses the hot spot. Windows Help continues displaying the topic while it executes the macro, unless the macro causes a jump to another topic.

3.2.9 Creating a Keyword List

You can also enable users to find and view topics by assigning keywords to the topics. You assign a keyword by using the **\footnote** statement and the letter *K* as the footnote character. Windows Help collects all keywords in a help file and displays them in its Search dialog box. Using this dialog box, a user can select a keyword and view the help topics associated with it. The following example assigns the keyword "Sample Topics" to the current topic:

```
#{\footnote topic1}
${\footnote My Topic}
K{\footnote Sample Topics}
This is my first topic.
\par
\page
```

If a keyword begins with the letter *K*, you must place an extra space before the word. Multiple keywords for a topic are separated by semicolons.

A keyword can be assigned to any number of topics. When the user selects the keyword in the Search dialog box, Windows Help displays all topics associated with the keyword. The user then picks the one to view.

You can also create alternative keywords for a help file for use with the **WinHelp** function.

3.2.10 Creating Browse Sequences

You can enable users to browse through a sequence of help topics by creating a browse sequence and adding browse buttons to your help file. A browse sequence typically consists of two or more related topics that are intended to be read sequentially. You create a browse sequence by using the **\footnote** statement and the plus-sign (+) footnote character to assign a sequence identifier. The following example assigns a sequence identifier to the topic titled A Topic:

```
#{\footnote topic5}
${\footnote A Topic}
+{\footnote shorttopics}
This is one topic in a browse sequence.
\par
\page
```

Windows Help adds topics with sequence identifiers to the browse sequence and determines the order of topics in the sequence by sorting the identifiers alphabetically. If two topics have the same identifier, Windows Help assumes that the topic that was compiled first is to be displayed first.

Windows Help uses the sequence only if the browse buttons have been enabled. You can enable the buttons by placing the following in the Help project file:

```
[CONFIG]
BrowseButtons()
```

For more information about the project file, see Section 3.4, "Creating Help Project Files."

You can create more than one browse sequence in a help file by using sequence numbers with sequence identifiers. The sequence number consists of a colon (:) followed by an integer. Windows Help combines all topics having the same sequence identifier (but different sequence numbers) into a single browse sequence and determines the order of the topics by sorting them alphabetically. To ensure that numerals are sorted correctly, they should have the same number of digits. For example, the numerals 1 through 10 should be 01 through 10.

```
#{\footnote topic10}
${\footnote Alpha Topic #3}
+{\footnote alpha:3}
This topic is part of the alpha browse sequence.
\par
\page
```

3.3 Using Graphics Files

You can add bitmaps and metafiles to your help files by using the **bml**, **bmc**, and **bmr** statements. These statements take the name of a graphics file and insert the corresponding bitmap or metafile into the help file at the specified position.

Windows Help requires graphics files to be in one of the following formats:

- Windows bitmap (.BMP)
- Placeable Windows metafile (.WMF)
- Multiple-resolution bitmap (.MRB)
- Segmented-graphics bitmap (.SHG)

Multiple-resolution bitmaps can be created by using the Microsoft Multiple-Resolution Bitmap Compiler (MRBC). Segmented-graphics bitmaps can be created by using Microsoft Windows Hotspot Editor. Only 16-color and monochrome bitmaps may be used. Windows Help does not support 256-color bitmaps.

Although the **\pict** statement can also be used to add bitmaps and metafiles to a help file, the bitmap or metafile data must be inserted into the topic file rather than specified as a separate file.

3.3.1 Inserting a Bitmap in Text

You can insert a bitmap into a paragraph as if it were a character by using the **bmc** statement. The statement aligns the bottom of the bitmap with the base line of the current line of text and places the left edge of the bitmap at the next character position. The following example inserts a bitmap into a line of text:

```
Press the \{bmc enter.bmp\} key to complete the task and return to
the main window.
```

Since the bitmap is treated as text, any paragraph properties assigned to the paragraph also apply to the bitmap. Windows Help places text following the bitmap on the same base line at the next available character position.

In general, bitmaps inserted as characters should be clipped to the smallest possible size. Any extra white space at the top or bottom of the bitmap image affects the alignment of the bitmap with the text and may affect the spacing between lines.

You must not specify negative line spacing for paragraphs that contain **bmc** statements. Doing so might cause the bitmap to appear on top of the paragraph.

3.3.2 Wrapping Text Around a Bitmap

You can place a bitmap at the left or right margin of the Help window and have subsequent text wrap around the bitmap by using the **bml** or **bmr** statement. The **bml** statement inserts a bitmap at the left margin; **bmr** inserts it at the right.

If you want text to wrap around a bitmap, you must place the **bml** or **bmr** statement at the beginning of a paragraph. Windows Help aligns the start of the paragraph with the top of the bitmap and wraps around the left or right edge of the bitmap. The following example places the bitmap at the left margin and subsequent text wraps around its right edge:

```
\{bml mybitmap.bmp\}
The text in this paragraph wraps around the right edge of the bitmap.
\par
```

If you place a **bml** or **bmr** statement at the end of a paragraph, Windows Help places the bitmap under the paragraph instead of wrapping the text around the bitmap. If you do not want text to wrap around a bitmap, place **\par** statements immediately before and after the **bml** or **bmr** statement.

3.3.3 Using a Bitmap as a Hot Spot

You can use bitmaps as hot spots. This enables you to create graphics, such as icons or buttons, and use them as "jumps" to particular topics or as hot spots for macros. The following example uses the bitmap in the MYBUTTON.BMP file to create a link. When the user clicks the bitmap, Windows Help jumps to the topic identfied by the context string "topic15":

```
{\strike \{bml mybutton.bmp\}}{\v topic15}
```

You can also divide a single bitmap into several hot spots and assign a different link or macro to each hot spot. Such bitmaps, called segmented-graphics bitmaps, are created by using Hotspot Editor. For example, if you have a bitmap of a dialog box, you can assign links to each of the control windows in the dialog box, enabling the user to click a control window and view information about it. Segmented-graphics bitmaps already contain the context strings needed for the links; only a **bml** or **bmr** statement is needed to insert the bitmap. The **\strike** and **\v** statements must not be used.

```
\{bml mydialog.shg\}
```

3.3.4 Using a Bitmap on Different Displays

A multiple-resolution bitmap is a single bitmap file that contains one or more bitmaps that have been marked for use with specific displays, such as the CGA, EGA, VGA, or 8514 displays. You use multiple-resolution bitmaps to avoid problems associated with displaying bitmaps designed for a single type of display. Single-resolution bitmaps can have the following problems:

- Appear too big or too small on displays having different resolutions
- Appear stretched or compressed on displays with different aspect ratios
- Lack colors or use unintended colors on displays with different color capabilities.

You create multiple-resolution bitmaps by using MRBC. The compiler, an MS-DOS program, has the following command-line syntax:

mrbc [/s] *filename* ...

The *filename* parameter specifies the name of a Windows bitmap file. Typically, you specify several filenames, one for each type of display. Wildcards can be used. The compiler uses the filename of the first bitmap file as the name of the output file but gives the output file the filename extension .MRB. The following example combines the bitmap files MYBUTTON.EGA, MYBUTTON.VGA, and MYBUTTON.854 into the multiple-resolution bitmap file MYBUTTON.MRB:

```
mrbc mybutton.ega mybutton.vga mybutton.854
```

In this example, the compiler checks the **biXPelsPerMeter** and **biYPelsPerMeter** members of the **BITMAPINFOHEADER** structure in each bitmap file to determine the display type for the bitmap. (For a description of the **BITMAPINFO-HEADER** structure, see the *Microsoft Windows Programmer's Reference, Volume 3*.) If these members are set to zero, the compiler prompts for the display type with a message such as the following:

```
Please enter the monitor type for the bitmap mybutton.ega:
```

You must enter at least the first character of one of the following display-type names: CGA, EGA, VGA, or 8514. The compiler sets the display type you specify, but it does not check that the type is valid. For example, if you specify VGA for an EGA bitmap, the compiler marks it as a VGA bitmap. The result may be undesirable.

The **/s** option, specifying silent mode, speeds up compilation if the names of the bitmap files conform to the MRBC filename convention. If you use the **/s** option, the compiler uses the first character of the filename extension to determine the display type for the bitmap, as described in the following list:

Letter	Meaning
C	CGA bitmap
E	EGA bitmap
V	VGA bitmap
8	8514 bitmap

If the filename extension starts with any other character, MRBC assumes a VGA bitmap. The following example creates the multiple-resolution bitmap file MYBUTTON.MRB, containing bitmaps for EGA, VGA, and 8514 displays:

```
mrbc /s mybutton.ega mybutton.vga mybutton.854
```

The compiler never writes over existing multiple-resolution bitmap files. If the output file already exists, the compiler displays an error message.

You insert multiple-resolution bitmaps into your help file by using the same statements as for Windows bitmaps. For example, the following **bmc** statement inserts the bitmaps from the MYBUTTON.MRB file:

```
Click the \{bmc mybutton.mrb\} button to complete the task and
return to the main window.
```

Before displaying a multiple-resolution bitmap, Windows Help checks the display type for the computer and then selects the bitmap that has the closest matching resolution, aspect ratio, and color capabilities. Windows Help never displays more than one bitmap from a multiple-resolution bitmap file.

3.4 Creating Help Project Files

This section describes the format and contents of the Help project file (.HPJ) used to build the help file. The project file contains all the information the Microsoft Help Compiler needs to combine topic files and other elements into a help file.

3.4.1 Project File Sections

Every project file consists of one or more sections. Each section has a section name, enclosed in brackets ([]), that defines the purpose and format of statements and options in the section. Following are the sections used in project files:

Section	Description
[OPTIONS]	Specifies options that control the build process. This section is optional. If this section is used, it should be the first section listed in the project file, so that the options will apply during the entire build process.
[FILES]	Specifies topic files to be included in the build. This section is required.
[BUILDTAGS]	Specifies valid build tags. This section is optional.
[CONFIG]	Specifies Help macros that define nonstandard menus, buttons, and macros used in the help file. This section is required if the help file uses any of these features. This section is new for Windows 3.1.
[BITMAPS]	Specifies bitmap files to be included in the build. This section is not required if the project file lists a path for bitmap files by using the **BMROOT** or **ROOT** option.
[MAP]	Associates context strings with context numbers. This section is optional.
[ALIAS]	Assigns one or more context strings to the same topic. This section is optional.
[WINDOWS]	Defines the characteristics of the primary Help window and the secondary-window types used in the help file. This section is required if the help file uses secondary windows. This section is new for Windows 3.1.
[BAGGAGE]	Lists files that are to be placed within the help file (which contains its own file system). This section is optional.

Every project file requires a **[FILES]** section. This section names the topic files. Most project files also have an **[OPTIONS]** section that specifies how to build the help file. A very useful option in the **[OPTIONS]** section is the **COMPRESS** option, which specifies whether the help file should be compressed or uncompressed. Compressing a help file reduces its size considerably and saves valuable disk space.

The following example creates a compressed help file from two topic files, MAIN.RTF and MENUS.RTF:

```
[OPTIONS]
COMPRESS=TRUE

[FILES]
MAIN.RTF
MENUS.RTF
```

3.4.2 Using Macros in Project Files

You can add macros to the **[CONFIG]** section of a project file. Since Windows Help executes the macros when it first opens the help file, macros that create menus, menu items, and buttons are typically placed in this section. If there is more than one macro listed in the **[CONFIG]** section, Windows Help executes them in the order in which they are listed.

You can create new menu items and buttons for Windows Help by using such macros as **CreateButton** and **InsertMenu**. These macros define other Help macros and associate them with the menu items and buttons. Windows Help executes these macros when the user chooses a corresponding menu item or button. Macros that create Help buttons, menus, or menu items remain in effect until the user quits Windows Help or opens a new help file.

You can extend the capabilities of Windows Help by developing your own dynamic-link libraries (DLLs) and defining Help macros that call functions in the libraries. To define Help macros that call DLL functions, you must register each function and its corresponding library by using the **RegisterRoutine** macro in the [CONFIG] section of the project file.

For a complete list of Help macros, see the *Microsoft Windows Programmer's Reference, Volume 4.*

3.4.3 Sample Project File

The following example is a sample project file for the Cardfile application. Comments, marked by a beginning semicolon (;), indicate the purpose of each section in the file:

```
; Options used to define the Help title bar and icon
[OPTIONS]
ROOT=C:\HELP
BMROOT=C:\HELP\ART
CONTENTS=cont_idx_card
TITLE=Cardfile Help
ICON=CARDHLP.ICO
COMPRESS=OFF
WARNING=3
REPORT=ON
ERRORLOG=CARD.BUG

; Files used to build Cardfile Help
[FILES]
RTFTXT\COMMANDS.RTF
RTFTXT\HOWTO.RTF
RTFTXT\KEYS.RTF
RTFTXT\GLOSSARY.RTF

; Button macros and Using Help file
[CONFIG]
CreateButton("btn_up", "&Up", "JumpContents(`HOME.HLP')")
BrowseButtons()
SetHelpOnFile("APPHELP.HLP")

; Secondary-window characteristics
[WINDOWS]
picture = "Samples", (123, 123, 256, 256), 0, (0, 255, 255), (255, 0, 0)
```

3.5 Using Help in a Windows Application

Windows applications can offer help to their users by using the **WinHelp** function to start Windows Help and display topics in the application's help file. The **WinHelp** function gives a Windows application complete access to the help file, as well as to the menus and commands of Windows Help. Many applications use **WinHelp** to implement context-sensitive Help. Context-sensitive Help enables users to view topics about specific windows, menus, menu items, and control windows by selecting the item with the keyboard or the mouse. For example, a user can learn about the Open command on the File menu by selecting the command (using the direction keys) and pressing the F1 key.

3.5.1 Choosing Help from the Help Menu

Every application should provide a Help menu to allow the user to open the help file with either the keyboard or the mouse. The Help menu should contain at least one Contents menu item that, when chosen, displays the contents or the main topic in the help file. To support the Help menu, the application's main window proce-

dure should check for the Contents menu item and call the **WinHelp** function, as in the following example:

```
case WM_COMMAND:
    switch (wParam) {
    case IDM_HELP_CONTENTS:
        WinHelp(hwnd, "myhelp.hlp", HELP_CONTENTS, 0L);
        return 0L;
        .
        .
        .

    }
    break;
```

You can add other menu items to the Help menu for topics containing general information about the application. For example, if your help file contains a topic that describes how to use the keyboard, you can place a Keyboard menu item on the Help menu. To support additional menu items, your application must specify either the context string or the context identifier for the corresponding topic when it calls the **WinHelp** function. The following example uses a Help macro to specify the context string IDM_HELP_KEYBOARD for the Keyboard topic:

```
case IDM_HELP_KEYBOARD:
    WinHelp(hwnd, "myhelp.hlp", HELP_COMMAND,
            (LPSTR)"JumpID(\"myhelp.hlp\",\"IDM_HELP_KEYBOARD\")");
    return 0L;
```

A better way to display a topic is to use a context identifier. To do this, the help file must assign a unique number to the corresponding context string, in the **[MAP]** section of the project file. For example, the following section assigns the number 101 to the context string IDM_HELP_KEYBOARD:

```
[MAP]
IDM_HELP_KEYBOARD   101
```

An application can display the Keyboard topic by specifying the context identifier in the call to the **WinHelp** function, as in the following example:

```
#define IDM_HELP_KEYBOARD 101

WinHelp(hwnd, "myhelp.hlp", HELP_CONTEXT, (DWORD)IDM_HELP_KEYBOARD);
```

To make maintenance of an application easier, most programmers place their defined constants (such as IDM_HELP_KEYBOARD in the previous example) in a single header file. As long as the names of the defined contants in the header file are identical to the context strings in the help file, you can include the header file in the **[MAP]** section to assign context identifiers, as shown in the following example:

```
[MAP]
#include <myhelp.h>
```

If a help file contains two or more Contents topics, the application can assign one as the default by using the context identifier and the HELP_SETCONTENTS value in a call to the **WinHelp** function.

3.5.2 Choosing Help with the Keyboard

An application can enable the user to choose a help topic with the keyboard by intercepting the F1 key. Intercepting this key lets the user select a menu, menu item, dialog box, message box, or control window and view Help for it with a single keystroke.

To intercept the F1 key, the application must install a message-filter procedure by using the **SetWindowsHook** function. This allows the application to examine all keystrokes for the application, regardless of which window has the input focus. If the filter procedure detects the F1 key, it posts a WM_F1DOWN message (application-defined) to the application's main window procedure. The procedure then determines which help topic to display.

The filter procedure should have the following form:

```
int FAR PASCAL FilterFunc(nCode, wParam, lParam)
int nCode;
WORD wParam;
DWORD lParam;
{
    LPMSG lpmsg = (LPMSG)lParam;

    if ((nCode == MSGF_DIALOGBOX || nCode == MSGF_MENU) &&
            lpmsg->message == WM_KEYDOWN && lpmsg->wParam == VK_F1) {
        PostMessage(hWnd, WM_F1DOWN, nCode, 0L);
    }

    DefHookProc(nCode, wParam, lParam, &lpFilterFunc);

    return 0;
}
```

The application should install the filter procedure after creating the main window, as shown in the following example:

```
lpProcInstance = MakeProcInstance(FilterFunc, hInstance);
if (lpProcInstance == NULL)
    return FALSE;

lpFilterFunc = SetWindowsHook(WH_MSGFILTER, lpProcInstance);
```

Like all callback functions, the filter procedure must be exported by the application.

The filter procedure sends a WM_F1DOWN message only when the F1 key is pressed in a dialog box, message box, or menu. Many applications also display the Contents topic if no menu, dialog box, or message box is selected when the user presses the F1 key. In this case, the application should define the F1 key as an accelerator key that starts Help.

To create an accelerator key, the application's resource-definition file must define an accelerator table, as follows:

```
1 ACCELERATORS
BEGIN
    VK_F1, IDM_HELP_CONTENTS, VIRTKEY
END
```

To support the accelerator key, the application must load the accelerator table by using the **LoadAccelerators** function and translate the accelerator keys in the main message loop by using the **TranslateAccelerator** function.

In addition to installing the filter procedure, the application must keep track of which menu, menu item, dialog box, or message box is currently selected. In other words, when the user selects an item, the application must set a global variable indicating the current context. For dialog and message boxes, the application should set the global variable immediately before calling the **DialogBox** or **MessageBox** function. For menus and menu items, the application should set the variable whenever it receives a WM_MENUSELECT message. As long as identifiers for all menu items and controls in an application are unique, an application can use code similar to the following example to monitor menu selections:

```
case WM_MENUSELECT:
    /*
     * Set dwCurrentHelpId to the Help ID of the menu item that is
     * currently selected.
     */

    if (HIWORD(lParam) == 0)            /* no menu selected    */
        dwCurrentHelpId = ID_NONE;

    else if (lParam & MF_POPUP) {       /* pop-up selected     */
        if ((HMENU)wParam == hMenuFile)
            dwCurrentHelpId = ID_FILE;
        else if ((HMENU)wParam == hMenuEdit)
            dwCurrentHelpId = ID_EDIT;
        else if ((HMENU)wParam == hMenuHelp)
            dwCurrentHelpId = ID_HELP;
        else
            dwCurrentHelpId = ID_SYSTEM;
    }

    else                                /* menu item selected */
        dwCurrentHelpId = wParam;

    break;
```

In this example, the *hMenuFile*, *hMenuEdit*, and *hMenuHelp* parameters must previously have been set to specify the corresponding menu handles. An application can use the **GetMenu** and **GetSubMenu** functions to retrieve these handles.

When the main window procedure finally receives a WM_F1DOWN message, it should use the current value of the global variable to display a help topic. The application can also provide Help for individual controls in a dialog box by determining which control has the focus at this point, as shown in the following example:

```
case WM_F1DOWN:
    /*
     * If there is a current Help context, display it.
     */

    if (dwCurrentHelpId != ID_NONE) {
        DWORD dwHelp = dwCurrentHelpId;

        /*
         * Check for context-sensitive Help for individual dialog
         * box controls.
         */

        if (wParam == MSGF_DIALOGBOX) {
            WORD wID = GetWindowWord(GetFocus(), GWW_ID);
            if (wID != IDOK && wID != IDCANCEL)
                dwHelp = (DWORD) wID;
        }
```

```
        WinHelp(hWnd, szHelpFileName, HELP_CONTEXT, dwHelp);

        /*
         * This call is used to remove the highlighting from the
         * System menu, if necessary.
         */

        DrawMenuBar(hWnd);
    }

    break;
```

When the application ends, it must remove the filter procedure by using the **UnhookWindowsHook** function and free the procedure instance for the function by using the **FreeProcInstance** function.

3.5.3 Choosing Help with the Mouse

An application can enable the user to choose a help topic with the mouse by intercepting mouse input messages and calling the **WinHelp** function. To distinguish requests to view Help from regular mouse input, the user must press the SHIFT+F1 key combination. In such cases, the application sets a global variable when the user presses the key combination and changes the cursor shape to a question-mark pointer to indicate that the mouse can be used to choose a help topic.

To detect the SHIFT+F1 key combination, an application checks for the VK_F1 virtual-key value in each WM_KEYDOWN message sent to its main window procedure. It also checks for the VK_ESCAPE virtual-key code. The user presses the ESC key to quit Help and restore the mouse to its regular function. The following example checks for these keys:

```
case WM_KEYDOWN:
    if (wParam == VK_F1) {

        /* If Shift-F1, turn Help mode on and set Help cursor. */

        if (GetKeyState(VK_SHIFT)) {
            bHelp = TRUE;
            SetCursor(hHelpCursor);
            return (DefWindowProc(hwnd, message, wParam, lParam));
        }

        /* If F1 without shift, call Help main index topic. */

        else {
            WinHelp(hwnd,"myhelp.hlp",HELP_CONTENTS,0L);
        }
    }

    else if (wParam == VK_ESCAPE && bHelp) {

        /* Escape during Help mode: turn Help mode off. */

        bHelp = FALSE;
        SetCursor((HCURSOR) GetClassWord(hWnd, GCW_HCURSOR));
    }

    break;
```

Until the user clicks the mouse or presses the ESC key, the application responds to WM_SETCURSOR messages by resetting the cursor to the arrow and question-mark combination.

```
case WM_SETCURSOR:
    /*
     * In Help mode, it is necessary to reset the cursor in response
     * to every WM_SETCURSOR message. Otherwise, by default, Windows
     * will reset the cursor to that of the window class.
     */

    if (bHelp) {
        SetCursor(hHelpCursor);
        break;
    }

    return (DefWindowProc(hwnd, message, wParam, lParam));

case WM_INITMENU:
    if (bHelp) {
        SetCursor(hHelpCursor);
    }

    return (TRUE);
```

If the user clicks the mouse button in a nonclient area of the application window while in Help mode, the application receives a WM_NCLBUTTONDOWN message. By examining the *wParam* value of this message, the application can determine which context identifier to pass to **WinHelp**.

```
case WM_NCLBUTTONDOWN:
    /*
     * If in Help mode (Shift+F1), display context-sensitive
     * Help for nonclient area.
     */

    if (bHelp) {
        dwHelpContextId =
            (wParam == HTCAPTION) ?(DWORD) HELPID_TITLE_BAR:
            (wParam == HTSIZE) ? (DWORD) HELPID_SIZE_BOX:
            (wParam == HTREDUCE) ? (DWORD) HELPID_MINIMIZE_ICON:
            (wParam == HTZOOM) ? (DWORD) HELPID_MAXIMIZE_ICON:
            (wParam == HTSYSMENU) ?(DWORD) HELPID_SYSTEM_MENU:
            (wParam == HTBOTTOM) ? (DWORD) HELPID_SIZING_BORDER:
            (wParam == HTBOTTOMLEFT) ? (DWORD) HELPID_SIZING_BORDER:
            (wParam == HTBOTTOMRIGHT) ?(DWORD) HELPID_SIZING_BORDER:
            (wParam == HTTOP) ?(DWORD) HELPID_SIZING_BORDER:
            (wParam == HTLEFT) ?(DWORD) HELPID_SIZING_BORDER:
            (wParam == HTRIGHT) ?(DWORD) HELPID_SIZING_BORDER:
            (wParam == HTTOPLEFT) ?(DWORD) HELPID_SIZING_BORDER:
            (wParam == HTTOPRIGHT) ? (DWORD) HELPID_SIZING_BORDER:
            (DWORD) 0L;

        if (!((BOOL) dwHelpContextId))
            return (DefWindowProc(hwnd, message, wParam, lParam));
        bHelp = FALSE;
        WinHelp(hWnd, szHelpFileName, HELP_CONTEXT, dwHelpContextId);
        break;
    }

    return (DefWindowProc(hWnd, message, wParam, lParam));
```

If the user clicks a menu item while in Help mode, the application intercepts the WM_COMMAND message and sends the Help request:

```
case WM_COMMAND:

    /* In Help mode (Shift-F1)? */

    if (bHelp) {
        bHelp = FALSE;
        WinHelp(hWnd,szHelpFileName,HELP_CONTEXT, (DWORD)wParam);
        return NULL;
    }
```

3.5.4 Searching for Help with Keywords

An application can enable the user to search for help topics based on full or partial keywords. This method is similar to employing the Search dialog box in Windows Help to find useful topics. The following example searches for the keyword "Keyboard" and displays the corresponding topic, if found:

```
WinHelp(hwnd, "myhelp.hlp", HELP_KEY, "Keyboard");
```

If the topic is not found, Windows Help displays an error message. If more than one topic has the same keyword, Windows Help displays only the first topic.

An application can give the user more options in a search by specifying partial keywords. When a partial keyword is given, Windows Help usually displays the Search dialog box to allow the user to continue the search or return to the application. However, if there is an exact match and no other topic exists with the given keyword, Windows Help displays the topic. The following example opens the Search dialog box and selects the first keyword in the list starting with the letters *Ke*:

```
WinHelp(hwnd, "myhelp.hlp", HELP_PARTIALKEY, "Ke");
```

When the HELP_KEY and HELP_PARTIALKEY values are specified in the **WinHelp** function, Windows Help searches the K keyword table. This table contains keywords generated by using the letter *K* with **\footnote** statements in the topic file. An application can search alternative keyword tables by specifying the HELP_MULTIKEY value in the **WinHelp** function. In this case, the application must specify the footnote character and the full keyword in a **MULTIKEYHELP** structure, as follows:

```
HANDLE hmkh;
MULTIKEYHELP far *mkh;
char *szKeyword = "Frame";
WORD wSize;

wSize = sizeof(MULTIKEYHELP) + lstrlen(szKeyword);

hmkh = GlobalAlloc(GHND, (DWORD)wSize);
if (hmkh == NULL)
    break;
mkh = (MULTIKEYHELP far *) GlobalLock(hmkh);

mkh->mkSize    = wSize;
mkh->mkKeylist = 'L';
lstrcpy(mkh->szKeyphrase, szKeyword);

WinHelp(hwnd, "myhelp.hlp", HELP_MULTIKEY, (DWORD)mkh);

GlobalUnlock(hmkh);
GlobalFree(hmkh);
```

If the keyword is not found, Windows Help displays an error message. If more than one topic has the keyword, Windows Help displays only the first topic. (For a full description of the **MULTIKEYHELP** structure, see the *Microsoft Programmer's Reference, Volume 3.*)

Applications cannot use alternative keyword tables unless the **MULTIKEY** option is specified in the **[OPTIONS]** section of the project file.

3.5.5 Displaying Help in a Secondary Window

An application can display help topics in secondary windows instead of in Windows Help's main window. Secondary windows are useful whenever the user does not need the full capabilities of Windows Help. The Windows Help menus and buttons are not available in secondary windows.

To display Help in a secondary window, the application specifies the name of the secondary window along with the name of the help file. The following example displays the help topic having the context identifier IDM_FILE_SAVE in the secondary window named wnd_menu:

```
WinHelp(hwnd, "myhelp.hlp>wnd_menu", HELP_CONTEXT, IDM_FILE_SAVE);
```

The name and characteristics of the secondary window must be defined in the **[WINDOWS]** section of the project file, as in the following example:

```
[WINDOWS]
wnd_menu = "Menus", (128, 128, 256, 256), 0
```

Windows Help displays the secondary window with the initial size and position specified in the **[WINDOWS]** section. However, an application can set a new size and position by specifying the HELP_SETWINPOS value in the **WinHelp** function. In this case, the application sets the members in a **HELPWININFO** structure to specify the window size and position. The following examples sets the secondary window wnd_menu to a new size and position:

```
HANDLE hhwi;
LPHELPWININFO lphwi;
WORD wSize;
char *szWndName = "wnd_menu";
```

```
wSize = sizeof(HELPWININFO) + lstrlen(szWndName);
hhwi = GlobalAlloc(GHND, wSize);
lphwi = (LPHELPWININFO)GlobalLock(hhwi);

lphwi->wStructSize = wSize;
lphwi->x           = 256;
lphwi->y           = 256;
lphwi->dx          = 767;
lphwi->dy          = 512;
lphwi->wMax        = 0;
lstrcpy(lphwi->rgchMember, szWndName);

WinHelp(hwnd, "myhelp.hlp", HELP_SETWINPOS, lphwi);

GlobalUnlock(hhwi);
GlobalFree(hhwi);
```

3.5.6 Canceling Help

Windows Help requires an application to explicitly cancel Help so that Windows Help can free any resources it used to keep track of the application and its help files. The application can do this at any time.

An application cancels Windows Help by calling the **WinHelp** function and specifying the HELP_QUIT value, as shown in the following example:

```
WinHelp(hwnd, "myhelp.hlp", HELP_QUIT, NULL);
```

If the application has made any calls to the **WinHelp** function, it must cancel Help before it closes its main window (for example, in response to the WM_DESTROY message in the main window procedure). An application needs to call **WinHelp** only once to cancel Help, no matter how many help files it has opened. Windows Help remains running until all applications or dynamic-link libraries that have called the **WinHelp** function have canceled Help.

3.6 Project File Sections and Options Reference

This section describes the different sections and options in a project file and gives examples of their use. The entries are in alphabetic order.

[ALIAS]

[ALIAS]
context_string = alias
.
.
.

The **[ALIAS]** section assigns one or more context strings to the same topic alias. This section is optional.

Parameters

context_string
Specifies the context string that identifies a particular topic.

alias
Specifies the alternative string or alias name. An alias string has the same form and follows the same conventions as the topic context string. That is, it is not case-sensitive and may contain the alphabetic characters A through Z, the numeric characters 0 through 9, and the period and underscore characters.

Comments

Because context strings must be unique for each topic and cannot be used for any other topic in the Help project, the **[ALIAS]** section provides a way to delete or combine help topics without recoding your files. For example, if you create a topic that replaces information in three other topics, you could manually search through your files for invalid cross-references to the deleted topics. The easier approach, however, would be to use the **[ALIAS]** section to assign the name of the new topic to the deleted topics.

The **[ALIAS]** section can also be used when your application has multiple context identifiers for one help topic. This situation occurs in context-sensitive Help.

Alias names can be used in a **[MAP]** section, but only if the **[ALIAS]** section precedes the **[MAP]** section.

Example

The following example creates several aliases:

```
[ALIAS]
sm_key=key_shrtcuts
cc_key=key_shrtcuts
st_key=key_shrtcuts     ; combined into Keyboard Shortcuts topic
clskey=us_dlog_bxs
maakey=us_dlog_bxs      ; covered in Using Dialog Boxes topic.
chk_key=dlogprts
drp_key=dlogprts
lst_key=dlogprts
opt_key=dlogprts
tbx_key=dlogprts        ; combined into Parts of Dialog Box topic.
frmtxt=edittxt
wrptxt=edittxt
seltxt=edittxt          ; covered in Editing Text topic.
```

See Also

[MAP]

[BAGGAGE]

[BAGGAGE]
filename
.
.
.

The **[BAGGAGE]** section lists files (typically multimedia elements) that the Microsoft Help Compiler stores within the help file's internal file system. Windows Help can access data files stored in the help file more efficiently than it can access files in the normal MS-DOS file system, since it doesn't have to read the file allocation table from CD-ROM.

Parameters

filename
Specifies the full path of a file. If a file cannot be found, the compiler reports an error.

Comments

A maximum of 1,000 bitmap files can be stored as baggage files.

If a file is listed in the **[BAGGAGE]** section, you refer to that file in the topic file by prefixing the filename with an exclamation point (!). The filename must appear exactly the same in the topic file as it appears in the **[BAGGAGE]** section. To

avoid having to specify a full path, use the **ROOT** option in the **[OPTIONS]** section to specify the path. All filenames that you give in the topic file are relative to the **ROOT** path.

See Also

ROOT

[BITMAPS]

[BITMAPS]
filename
.
.
.

The **[BITMAPS]** section specifies the names and locations of the bitmap files specified in the **bmc**, **bml**, and **bmr** statements.

Parameters

filename
Specifies the full path of a bitmap file. If a file cannot be found, the compiler reports an error.

Comments

The **[BITMAPS]** section is not required if the bitmaps are located in the Help project directory or if the path containing the bitmaps is listed in the **BMROOT** or **ROOT** option. If the project file does not include either of these options, each bitmap filename must be listed in the **[BITMAPS]** section of the project file.

Example

The following example specifies three bitmap files:

```
[BITMAPS]
BMP01.BMP
BMP02.BMP
BMP03.BMP
```

See Also

BMROOT, **ROOT**

BMROOT

BMROOT = *path*[**,** *path*]...

The **BMROOT** option specifies the directory containing the bitmap files specified in the **bmc**, **bml**, and **bmr** statements.

Parameters

path
Specifies a drive and full path.

Comments

If the project file has a **BMROOT** option, you do not need to list the bitmap files in the **[BITMAPS]** section.

If the project file does not have a **BMROOT** option, the Help compiler looks for bitmaps in the directories specified by the **ROOT** option. If the project file does not have a **ROOT** option or if the **ROOT** option does not specify the directory containing the bitmap files, the filename for each bitmap must be specified in the **[BITMAPS]** section.

Example

The following example specifies that bitmaps are in the \HELP\BMP directory on drive C: and the \GRAPHICS\ART directory on drive D:

```
[OPTIONS]
BMROOT=C:\HELP\BMP, D:\GRAPHICS\ART
```

See Also

[BITMAPS], **[OPTIONS]**, **ROOT**

BUILD

BUILD = *expression*

The **BUILD** option specifies which topics containing build tags are included in a build. The **BUILD** option does not apply to topics that do not contain build tags.

A topic contains a build tag if it contains a build-tag **\footnote** statement. Topics without build tags are always compiled, regardless of the current build expression.

Parameters

expression
Specifies the build expression. This parameter consists of a combination of build tags (specified in the **[BUILDTAGS]** section) and the following operators:

Operator	Description
~	Applies the NOT operator to a single tag. The Help compiler compiles a topic only if the tag is *not* present. This operator has the highest precedence; the compiler applies it before any other operator.
&	Combines two tags by using the AND operator. The Help compiler compiles a topic only if it contains both tags. The compiler applies this operator only after the ~ operator has been applied.
|	Combines two tags by using the OR operator. The Help compiler compiles a topic if it has at least one tag. This operator has the lowest precedence; the compiler applies it only after all other operators have been applied.

Parentheses may be used to override operator precedence. Expressions enclosed in parentheses are always evaluated first.

Comments

Only one **BUILD** option can be given per project file.

The Help compiler evaluates all build expressions from left to right, using the specified precedence rules.

Example

The following examples assume that the **[BUILDTAGS]** section in the project file defines the build tags DEMO, MASTER, and TEST_BUILD. Although the following examples show several **BUILD** options on consecutive lines, only one **BUILD** option per project file is allowed.

```
BUILD = DEMO             ; compile topics that have the DEMO tag
BUILD = DEMO & MASTER    ; compile topics with both DEMO and MASTER
BUILD = DEMO | MASTER    ; compile topics with either DEMO or MASTER
BUILD = ~DEMO            ; compile topics that do not have DEMO
BUILD = (DEMO | MASTER) & TEST_BUILD
                         ; compile topics that have TEST_BUILD and
                         ; either DEMO or MASTER
```

See Also

[BUILDTAGS], **[OPTIONS]**

[BUILDTAGS]

[BUILDTAGS]
tag
.
.
.

The **[BUILDTAGS]** section defines the build tags for the help file. The Help compiler uses these tags to determine which topics to include when building the help file.

This section is used in conjunction with the build-tag **\footnote** statements. These **\footnote** statements associate a build tag with a given topic. If the build tag is also defined in the **[BUILDTAGS]** section, the Help compiler compiles the topic; otherwise, it ignores the topic.

Parameters

tag
Specifies a build tag consisting of any combination of characters except spaces. The Help compiler strips any space characters from the tag. Also, the compiler treats uppercase and lowercase characters as the same characters (that is, it is case-insensitive).

Comments

The **[BUILDTAGS]** section is optional. If given, it can contain up to 30 build tags.

Example

The following example shows the form of the **[BUILDTAGS]** section in a sample project file:

```
[BUILDTAGS]
DEMO            ; topics to include in demo build
MASTER          ; topics to include in master help file
DEBUGBUILD      ; topics to include in debugging build
TESTBUILD       ; topics to include in a mini-build for testing
```

See Also

BUILD

COMPRESS

COMPRESS = *compression-level*

The **COMPRESS** option specifies the level of compression to be used when building the help file. Compression levels indicate either no compression, medium compression (approximately 40%), or high compression (approximately 50%).

Parameters

compression-level
Specifies the level of compression. This parameter can be one of the following values:

Value	Meaning
0	No compression
1	High compression
FALSE	No compression
HIGH	High compression
MEDIUM	Medium compression
NO	No compression
TRUE	High compression
YES	High compression

Comments

Depending on the degree of compression requested, the build uses either block compression or a combination of block and key-phrase compression. Block compression compresses the topic data into predefined units known as blocks. Key-phrase compression combines repeated phrases found within the source file(s). The compiler creates a phrase-table file with the .PH extension if one does not already exist. If the compiler finds a file with the .PH extension, it uses that file for the current compilation. Because the .PH file speeds up the compression process when little text has changed since the last compilation, you might want to keep the phrase file if you compile the same Help file several times with compression. However, you will get maximum compression if you delete the .PH file before starting each build.

See Also

[OPTIONS]

[CONFIG]

[CONFIG]
macro
.
.
.

The **[CONFIG]** section contains one or more macros that carry out actions, such as enabling browse buttons and registering dynamic-link library (DLL) functions. Windows Help executes the macros when it opens the help file.

Parameters

macro
Specifies a Windows Help macro. For more information about these macros, see the *Microsoft Windows Programmer's Reference, Volume 4.*

Comments

The **[CONFIG]** section may include any number of lines. Each line of the **[CONFIG]** section may be up to 254 characters long.

Example

The following example registers a DLL, creates a button, enables the browse buttons, and sets the name of the help file containing information about how to use Help:

```
[CONFIG]
RegisterRoutine("bmp","HDisplayBmp","USSS")
RegisterRoutine("bmp","CopyBmp", "v=USS")
CreateButton("btn_up", "&Up", "JumpContents(`HOME.HLP')")
BrowseButtons()
SetHelpOnFile("APPHELP.HLP")
```

CONTENTS

CONTENTS = *context-string*

The **CONTENTS** option identifies the context string of the highest-level or Contents topic. This topic is usually a table of contents or index within the help file. Windows Help displays the Contents topic whenever the user clicks the Contents button.

Parameters

context-string
Specifies the context string of a topic in the help file. The string can be any combination of characters, except spaces, and must also be specified in a context-string **\footnote** statement in some topic in the help file.

Comments

If the **[OPTIONS]** section does not include a **CONTENTS** option, the compiler assumes that the Contents topic is the first topic encountered in the first listed topic file in the **[FILES]** section of the project file.

Example

The following example sets the topic containing the context string "main_contents" as the Contents topic:

```
CONTENTS=main_contents
```

See Also

[FILES], **[OPTIONS]**

COPYRIGHT

COPYRIGHT = *copyright-notice*

The **COPYRIGHT** option places a custom copyright notice in the About dialog box of Windows Help. Windows Help displays the notice immediately below the Microsoft copyright notice.

Parameters

copyright-notice
Specifies the copyright notice. The notice can be any combination of characters; its length must be in the range 35 through 75 characters.

See Also

[OPTIONS]

ERRORLOG

ERRORLOG = *error-filename*

The **ERRORLOG** option directs the Help compiler to write all error messages to the specified file. The compiler also displays the error messages on the screen.

Parameters

error-filename
Specifies the name of the file to receive the error messages. This parameter can be a full or partial path if the error file should be written to a directory other than the project root directory.

Example

The following example writes all errors during the build to the HLPBUGS.TXT file in the Help project root directory.

```
ERRORLOG=HLPBUGS.TXT
```

See Also

[OPTIONS]

[FILES]

[FILES]
filename
.
.
.

The **[FILES]** section lists all topic files used to build the help file. Every project file requires a **[FILES]** section.

Parameters

filename
Specifies the full or partial path of a topic file. If a partial path is given, the Help compiler uses the directories specified by the **ROOT** option to construct a full path. If a file cannot be found, the compiler reports an error.

Comments

The **#include** directive can also be used in the **[FILES]** section to specify the topic files indirectly by designating a file that contains a list of the topic files.

Example

The following example specifies four topic files:

```
[FILES]
rtftxt\COMMANDS.RTF
rtftxt\HOWTO.RTF
rtftxt\KEYS.RTF
rtftxt\GLOSSARY.RTF
```

The following example uses the **#include** directive to specify the topic files indirectly. In this case, the file RTFFILES.H must be in the project file (the Help compiler does not use the INCLUDE environment variable to search for files).

```
[FILES]
#include <rtffiles.h>
```

See Also

ROOT

FORCEFONT

FORCEFONT = *fontname*

The **FORCEFONT** option forces the specified font to be substituted for all requested fonts. The option is used to create help files that can be viewed on systems that do not have all fonts available.

Parameters

fontname
Specifies the name of an available font. Font names must be spelled the same as they are in the Fonts dialog box in Control Panel. Font names cannot exceed 20 characters. If an invalid font name is given, the Help compiler uses the MS Sans Serif font as the default.

See Also

[OPTIONS]

ICON

ICON = *icon-file*

The **ICON** option identifies the icon file to display when the user minimizes Windows Help.

Parameters

icon-file
Specifies the name of the icon file. This file must have the standard Windows icon-file format.

See Also

[OPTIONS]

LANGUAGE

LANGUAGE = *language-name*

The **LANGUAGE** option sets the sorting order for keywords in the Search dialog box.

Parameters

language-name
Specifies the language on which to base sorting. This parameter can be the following:

Value	Meaning
scandanavian	Sets the sorting order to the Scandavanian-language order.

Comments

The default sorting order is the English-language order.

Microsoft Windows Help version 3.1 supports only English and Scandanavian sorting.

See Also

[OPTIONS]

[MAP]

[MAP]
context-string context-number

.
.
.

The **[MAP]** section associates context strings (or aliases) with context numbers for context-sensitive Help. The context number corresponds to a value the parent application passes to Windows Help in order to display a particular topic. This section is optional.

Parameters

context-string
Specifies the context string of a topic in the help file. The string can be any combination of characters, except spaces, and must also be specified in a context-string **\footnote** statement in some topic in the help file.

context-number
Specifies the context number to associate with the context string. The number can be in either decimal or standard C hexadecimal format. Only one context number may be assigned to a context string or alias. Assigning the same number to more than one context string generates a compiler error. At least one space must separate the context number from the context string.

Comments

If you do not explicitly assign context numbers to topics, the Help compiler generates default values by converting topic context strings into context numbers.

You can define the context strings listed in the **[MAP]** section either in a help topic or in the **[ALIAS]** section. The compiler generates a warning message if a context string appearing in the **[MAP]** section is not defined in any of the topic files or in the **[ALIAS]** section.

If you use an alias name, the **[ALIAS]** section must precede the **[MAP]** section in the Help project file.

The **[MAP]** section supports two additional statements for specifying context strings and their associated context numbers. The first statement has the following form:

#define *context-string context-number*

The *context-string* and *context-number* parameters are as described in the Parameters section.

The second statement has the following form:

#include "*filename*"

The *filename* parameter, which can be enclosed in either double quotation marks or angle brackets(<>), specifies the name of a file containing one or more **#define** statements. The file may contain additional **#include** statements as well, but files may not be nested in this way more than five deep.

Example

The following example assigns hexadecimal context numbers to the context strings:

```
[MAP]
Edit_Window     0x0001
Control_Menu    0x0002
Maximize_Icon   0x0003
Minimize_Icon   0x0004
Split_Bar       0x0005
Scroll_Bar      0x0006
Title_Bar       0x0007
Window_Border   0x0008
```

See Also

[ALIAS]

MAPFONTSIZE

MAPFONTSIZE = *m:p*

The **MAPFONTSIZE** option maps font sizes specified in topic files to different sizes when they are displayed in the Help window. This option is especially useful if there is a significant size difference between the authoring display and the intended user display.

Parameters

m
Specifies the size of the source font. This parameter is either a single point size or a range of point sizes. A range of point sizes consists of the low and high point sizes separated by a hyphen (-). If a range is specified, all fonts in the range are changed to the size specified by the *p* parameter.

p
Specifies the size of the desired font for the help file.

Comments

Although the **[OPTIONS]** section can contain up to five font ranges, only one font size or range is allowed with each **MAPFONTSIZE** statement. If more than

one **MAPFONTSIZE** statement is included, the source font size or range specified in subsequent statements cannot overlap previous mappings.

Example

The following examples illustrate the use of the **MAPFONTSIZE** option:

```
MAPFONTSIZE=8:12      ; display all 8-pt. fonts as 12-pt.
MAPFONTSIZE=12-24:16  ; display fonts from 12 to 24 pts. as 16 pts.
```

See Also

[OPTIONS]

MULTIKEY

MULTIKEY = *footnote-character*

The **MULTIKEY** option specifies the footnote character to use for an alternative keyword table. This option is intended to be used in conjunction with topic files that contain **\footnote** statements for alternative keywords.

Parameters

footnote-character
Specifies the case-sensitive letter to be used for the keyword footnote.

Comments

Since keyword footnotes are case-sensitive, you should limit your keyword-table footnotes to one case, usually uppercase. If an uppercase letter is specified, the compiler will not include footnotes with the lowercase form of the same letter in the keyword table.

You may use any alphanumeric character for a keyword table except *K* and *k*, which are reserved for Help's standard keyword table. There is an absolute limit of five keyword tables, including the standard table. However, depending upon system configuration and the structure of your Help system, a practical limit of only two or three tables may be more realistic. If the compiler cannot create an additional keyword table, the additional table is ignored in the build.

Example

The following example illustrates how to enable the letter *L* for a keyword-table footnote:

```
MULTIKEY=L
```

See Also

[OPTIONS]

OLDKEYPHRASE

OLDKEYPHRASE = *onoff*

The **OLDKEYPHRASE** option specifies whether an existing key-phrase file should be used to build the help file.

Parameters

onoff

Specifies whether the existing file should be used. This parameter can be one of the following values:

Value	Meaning
0	Recreate the file
1	Use the existing file
FALSE	Recreate the file
NO	Recreate the file
OFF	Recreate the file
ON	Use the existing file
TRUE	Use the existing file
YES	Use the existing file

See Also

[OPTIONS]

OPTCDROM

OPTCDROM = *yesvalue*

The **OPTCDROM** option optimizes a help file for display on CD-ROM by aligning topic files on predefined block boundaries.

Parameters

yesvalue

Specifies that the file should be optimized for CD-ROM. This parameter can be any of the following values:

YES
TRUE
1
ON

See Also

[OPTIONS]

[OPTIONS]

[OPTIONS]
option
.
.
.

The **[OPTIONS]** section includes options that control how a help file is built and what feedback the build process displays. If this section is included in the project file, it should be the first section listed, so that the options will apply during the entire build process.

Parameters

option
Specifies one of the following project-file options:

Option	Description
BMROOT	Specifies the directory containing the bitmap files named in the **bmc**, **bml**, and **bmr** statements in topic files.
BUILD	Specifies which topics to include in the build.
COMPRESS	Specifies the type of compression to use during the build.
CONTENTS	Specifies the context string of the Contents topic for a help file.
COPYRIGHT	Adds a unique copyright message for the help file to the About dialog box.
ERRORLOG	Puts compilation errors in a file during the build.
FORCEFONT	Forces all authored fonts in the topic files to appear in a different font when displayed in the Help window.
ICON	Specifies the icon file to be displayed when the help file is minimized.
LANGUAGE	Specifies a different sorting order for help files authored in a Scandanavian language.
MAPFONTSIZE	Maps a font size in the topic file to a different font size in the compiled help file.
MULTIKEY	Specifies an alternative keyword table to use for mapping topics.
OLDKEYPHRASE	Specifies whether the compiler should use the existing key-phrase table or create a new one during the build.
OPTCDROM	Optimizes the help file for CD-ROM use.
REPORT	Controls the display of messages during the build process.
ROOT	Specifies the directories containing the topic and data files listed in the project file.

Option	Description
TITLE	Specifies the text displayed in the title bar of the Help window when the file is open.
WARNING	Specifies the level of error-message reporting the compiler is to display during the build.

Comments

These options can appear in any order within the **[OPTIONS]** section. The **[OPTIONS]** section is not required.

REPORT

REPORT = ON

The **REPORT** option displays messages on the screen during the build. These messages indicate when the Help compiler is performing the different phases of the build, including compiling the file, resolving jumps, and verifying browse sequences.

See Also

[OPTIONS], **[WARNING]**

ROOT

ROOT = *pathname*[**,** *pathname*]...

The **ROOT** option specifies the directories where the Help compiler looks for files listed in the project file.

Parameters

pathname
Specifies either a drive and full path or a relative path from the project directory. If the project file has a **ROOT** option, all relative paths in the project file refer to one of these paths. If the project file does not have a **ROOT** option, all paths are relative to the directory containing the project file.

Comments

If the project file does not have a **BMROOT** option, the compiler looks in the directories specified in the **ROOT** option to find bitmaps positioned by using the **bmc**, **bml**, and **bmr** statements. If none of these directories contains these bitmaps, the bitmap filenames must be listed in the **[BITMAPS]** section of the project file.

Example

The following example specifies that the project root directory is C:\WINHELP\HELPDIR and is found on drive C:

```
[OPTIONS]
ROOT=C:\WINHELP\HELPDIR
```

Given this root directory, if the **[FILES]** section contains the entry TOPICS\FILE.RTF, the full path for the topic file is C:\WINHELP\HELPDIR\TOPICS\FILE.RTF.

See Also

[BITMAPS], **BMROOT**, **[OPTIONS]**

TITLE

TITLE = *titlename*

The **TITLE** option sets the title for the help file. Windows Help displays the title in its title bar whenever it displays the help file.

Parameters

titlename
Specifies the title displayed in the Windows Help title bar. The title must not exceed 50 characters.

Comments

If no title is specified by using the **TITLE** option, Windows Help displays the title Windows Help in the title bar.

Example

The following example sets the help-file title to ABC Help.

```
[OPTIONS]
TITLE=ABC Help
```

See Also

[OPTIONS]

WARNING

WARNING = *level*

The **WARNING** option specifies the amount of debugging information the Help compiler is to report.

Parameters

level
Specifies the warning level. This parameter may be one of the following values:

Value	Meaning
1	Report only the most severe errors.
2	Report an intermediate number of errors.
3	Report all errors and warnings.

Example

The following example specifies an intermediate level of error reporting:

```
[OPTIONS]
WARNING=2
```

See Also

[OPTIONS], **REPORT**

[WINDOWS]

[WINDOWS]
type = "*caption*", (*x*, *y*, *width*, *height*), *sizing*,
(*clientRGB*), (*nonscrollRGB*)
.
.
.

The **[WINDOWS]** section defines the size, location, and colors for the primary Help window and any secondary-window types used in a help file.

The secondary windows defined in this section are intended to be used with Windows applications that specify secondary windows when calling the **WinHelp** function.

Parameters

type
Specifies the type of window that uses the defined attributes. For the primary Help window, this parameter is **main**. For a secondary window, this parameter

may be any unique name of up to 8 characters. Any jumps that display a topic in a secondary window give this type name as part of the jump.

caption
Specifies the title for a secondary window. Windows Help places the title in the title bar of the window. To set the title for the primary Help window, use the **TITLE** option in the **[OPTIONS]** section.

x
Specifies the x-coordinate, in help units, of the window's upper-left corner. Windows Help always assumes the screen is 1024 help units wide, regardless of resolution. For example, if the x-coordinate is 512, the left edge of the Help window is in the middle of the screen.

y
Specifies the y-coordinate, in help units, of the window's upper-left corner. Windows Help always assumes the screen is 1024 help units high, regardless of resolution. For example, if the x-coordinate is 512, the top edge of the Help window is in the middle of the screen.

width
Specifies the default width, in help units, for a secondary window.

height
Specifies the default height, in help units, for a secondary window.

sizing
Specifies the relative size of a secondary window when Windows Help first opens the window. This parameter can be one of the following values:

Value	Meaning
0	Set the window to the size specified by the *x*, *y*, *width*, and *height* parameters.
1	Maximize the window; ignore the *x*, *y*, *width*, and *height* parameters.

clientRGB
Specifies the background color of the window. This parameter is an RGB color value consisting of three 8-bit hexadecimal numbers enclosed in parentheses and separated by commas. If this parameter is not given, Windows Help uses the default window color specified by Control Panel.

nonscrollRGB
Specifies the background color of the non-scrolling region (if any) in the Help window. This parameter is an RGB color value consisting of three 8-bit hexadecimal numbers enclosed in parentheses and separated by commas. If this parameter is not given, Windows Help uses the default window color specified by Control Panel.

Example

The following example defines two windows, the main window and a secondary window named "picture". The main-window definition sets the background color of non-scrolling regions in the main Help window to (128, 0, 128) but leaves several other values empty (for which Windows Help will supply its own default values). The secondary-window definition sets the caption to "Samples" and sets the width and height of the window to about one-quarter of the width and height of the screen. The background colors for the window and non-scrolling region are (0, 255, 255) and (255, 0, 0), respectively. The *sizing* parameter for both the main and secondary windows is zero.

```
[WINDOWS]
main=, (, , , ), 0, (, , ), (128, 0, 128)
picture = "Samples", (123, 123, 256, 256), 0, (0, 255, 255), (255, 0, 0)
```

See Also

[Options], **TITLE**

Debugging: CodeView for Windows

Chapter 4

4.1	Requirements for Using CodeView for Windows		69
	4.1.1	Using CVW with a Single Monitor	70
	4.1.2	Using CVW with a Secondary Monitor	70
4.2	Comparing CodeView for Windows with Other Microsoft Debuggers		71
	4.2.1	Differences Between CVW and SYMDEB	71
	4.2.2	Differences Between CVW and CodeView for MS-DOS	72
4.3	Preparing Windows Applications for Debugging		73
4.4	Setting Up the Debugging Version of Windows		73
4.5	Starting a Debugging Session		74
	4.5.1	Display Options	75
	4.5.2	Starting a Debugging Session for a Single Application	75
	4.5.3	Starting a Debugging Session for Multiple Instances of an Application	76
	4.5.4	Starting a Debugging Session for Multiple Applications	76
	4.5.5	Starting a Debugging Session for Dynamic-Link Libraries	77
	4.5.6	Command-Line Options	78
4.6	Saving Session Information		80
4.7	Working with the CodeView for Windows Screen		80
	4.7.1	Using CVW Display Windows	80
		4.7.1.1 Opening Display Windows	81
		4.7.1.2 Selecting Display Windows	81
		4.7.1.3 Adjusting Display Windows	82
	4.7.2	Using the Menu Bar	83
4.8	Accessing Help		85
4.9	Displaying Application Data		85
	4.9.1	Displaying Variables	86
	4.9.2	Displaying Expressions	87

4.9.3 Displaying Arrays and Structures 87
4.9.3.1 Displaying Character Arrays 88
4.9.3.2 Displaying Multidimensional Arrays 89
4.9.3.3 Displaying Dynamic Array Elements 90
4.9.4 Using the Quick Watch Command 91
4.9.5 Tracing Windows Messages 91
4.9.6 Displaying Memory 92
4.9.6.1 Displaying Local and Global Memory Objects 93
4.9.6.2 Displaying Variables with a Live Expression 95
4.9.6.3 Dereferencing Memory Handles 96
4.9.7 Displaying the Contents of Registers 97
4.9.8 Displaying Windows Modules 98
4.10 Modifying Application Data 98
4.11 Controlling Execution of Your Application 99
4.11.1 Continuous Execution 99
4.11.1.1 Selecting Breakpoint Lines 99
4.11.1.2 Setting Breakpoint Values 101
4.11.1.3 Setting Breakpoints on Windows Messages 101
4.11.1.4 Using Breakpoints 102
4.11.2 Single-Step Execution 103
4.11.3 Animated Execution 104
4.11.4 Jumping to a Particular Location 104
4.11.5 Interrupting Your Application 104
4.12 Handling Abnormal Termination of the Application 106
4.12.1 Handling a Fatal Exit 106
4.12.2 Handling a General Protection Fault 107
4.13 Ending a Session 108
4.14 Advanced Techniques 108
4.14.1 Using Multiple Source Windows 108
4.14.2 Checking for Undefined Pointers 109
4.14.3 Handling Register Variables 109
4.14.4 Redirecting CodeView for Windows Input and Output 109
4.15 Modifying the TOOLS.INI File 110
4.16 Related Topics 110

The Microsoft CodeView for Windows (CVW) debugger is a powerful, easy-to-use tool for the Microsoft Windows operating system. With CVW, you have the power to test the execution of your application and examine your data simultaneously. You can isolate problems quickly because you can display any combination of variables—global or local—while you interrupt or trace an application's execution.

CVW provides a variety of ways to analyze an application. You can use the debugger to examine source code, disassemble machine code, or examine a mixed display that shows you precisely which machine instructions correspond to each of your C-language statements. You can also monitor the occurrence of specific Windows messages.

CVW is similar to Microsoft CodeView (CV) version 3.0 for Microsoft® MS-DOS®. If you are familiar with CV for MS-DOS, see Section 4.2.2, "Differences Between CVW and CodeView for MS-DOS" for a concise description of the unique features of CVW.

This chapter serves as a complement to the CVW Help system. A significant portion of the CVW documentation is online. For information about using the CVW Help system, see Section 4.8, "Accessing Help."

Note CVW supports the Microsoft Mouse or any fully compatible pointing device. This chapter describes both mouse and keyboard procedures.

4.1 Requirements for Using CodeView for Windows

Following are the system requirements for using CVW:

- Your system must have at least 384K of extended memory. For applications compiled with many symbols, 1 megabyte or more of extended memory is required.
- For 80386-based systems, the following required entry is automatically added to the [386enh] section of your SYSTEM.INI file when you install CVW:

```
device=windebug.386
```

- Your PATH environment variable must include the directory (or directories) containing CVW.EXE, CVWIN.DLL, WINDEBUG.386, and CV.HLP.

4.1.1 Using CVW with a Single Monitor

It is possible to use CVW version 3.07 with a single monitor. For single-monitor debugging, you must have one of the following:

- A VGA display. CVW directly supports single-monitor debugging with a VGA display in both 386 enhanced and standard modes. No additional driver is needed.
- An EGA or other display with an 80386-based or 80486-based system running in 386 enhanced mode (you must use a VGA display in standard mode). With a non-VGA (or nonstandard VGA) display, you must install the VCV.386 driver. Place the driver in your Windows \SYSTEM directory and add the following entry to the [386enh] section of your Windows SYSTEM.INI file:

```
device=vcv.386
```

4.1.2 Using CVW with a Secondary Monitor

You may find it more convenient to use a dual-monitor configuration. With the secondary monitor connected to your system, you can view CVW output and Windows output simultaneously. (CVW version 3.07 does not support a serial terminal.)

If you are using a secondary monochrome monitor for your CVW display, you need a monochrome adapter card and monochrome display monitor.

To set up a secondary monitor for debugging, do the following:

1. Install a secondary monochrome adapter card in a free slot in your computer, and connect the monochrome monitor to the port in the back.
2. Set the switches for the secondary display adapter to the appropriate settings, according to the display adapter and computer manufacturers' recommendations.

To use the secondary monochrome monitor, you must specify the **/2** option on the command line when you start CVW.

If your system is an IBM Personal System/2, it must be configured with an IBM 8514/a display as the primary monitor and a VGA display as the secondary monitor. To use this configuration, specify the **/8** (8514/a) option on the **cvw** command line when you choose the Run command from the File menu in Program Manager. If your VGA display is monochrome, you must also use the **/b** (black-and-white) option. The 8514/a display serves as the Windows screen and the VGA display as the debugging screen.

Do not attempt to run non-Windows applications or MS-DOS Shell while running CVW with the **/8** option.

By default, the debugging screen operates in 50-line mode in this configuration. If you specify the **/8** option, you can optionally specify the **/25** or **/43** option for 25- or 43-line mode, respectively, on the VGA debugging screen.

For more information about the command-line display options for CVW, see Section 4.5.1, "Display Options."

4.2 Comparing CodeView for Windows with Other Microsoft Debuggers

If you have programmed in the Windows environment, you may have used the Microsoft Symbolic Debugger (SYMDEB) to debug Windows applications. You may also be familiar with CodeView (CV) for MS-DOS. This section describes the features and functions of CVW that are different from the features and functions of these other Microsoft debugging tools.

4.2.1 Differences Between CVW and SYMDEB

CVW has all the capabilities of SYMDEB and a number of features that SYMDEB does not provide. Following is a summary of the differences between SYMDEB and CVW:

SYMDEB feature	CVW feature
Debugs applications in real mode.	Debugs applications in protected mode.
Examines only global (static) variables.	Examines both global and local variables.
Examines memory only when you specify simple memory addresses or symbol names.	Examines memory directly, but also uses the C-language expression evaluators to combine any variables with higher-level-language syntax.
Provides only breakpoints to interrupt execution.	Provides breakpoints, tracepoints, and watchpoints to set Boolean conditions and then break execution whenever these conditions become true.
Does not set breakpoints or tracepoints on Windows messages.	Sets breakpoints and tracepoints on Windows messages.
Works through command line.	Works through command line or menus.

4.2.2 Differences Between CVW and CodeView for MS-DOS

With CVW, as with CV for MS-DOS, you can display and modify *any* variable, section of addressable memory, or processor register; monitor the path of execution; and precisely control where execution pauses. However, CV for MS-DOS and CVW differ in the following ways:

CV feature	CVW feature
Starts from the MS-DOS prompt.	Starts from within Windows.
Repeats a search when you press ALT+/.	Repeats a search when you press CTRL+R.
Returns to MS-DOS upon termination.	Returns to Windows under normal termination conditions. An abnormal termination of CVW may cause the Windows session to be terminated.

In addition to these differences, CVW includes the following unique features:

- The ability to track your application's segments and data as Windows moves their locations in memory. As items are moved, the debugger readjusts its symbol table accordingly.
- The **(lh)** and **(gh)** type casts, which you can use to dereference local and global handles of a memory object into near and far pointer addresses. For a more detailed description, see Section 4.9.6.3, "Dereferencing Memory Handles."
- Windows-specific commands. CVW has the following six new commands:

Command	Action
wdl (Windows Display Local Heap)	Displays a list of the memory objects in the local heap. For more information, see Section 4.9.6.1, "Displaying Local and Global Memory Objects."
wdg (Windows Display Global Heap)	Displays a list of the memory objects in the global heap. For more information, see Section 4.9.6.1, "Displaying Local and Global Memory Objects."
wdm (Windows Display Modules)	Displays a list of the application and library modules available to Windows. For more information, see Section 4.9.8, "Displaying Windows Modules."
wwm (Windows Watch Message)	Displays a Windows message or class of messages in the CVW Command window. For more information, see Section 4.9.5, "Tracing Windows Messages."

Command	Action
wbm (Windows Breakpoint Message)	Sets a breakpoint on a Windows message or class of messages. For more information, see Section 4.11.1.3, "Setting Breakpoints on Windows Messages."
wka (Windows Kill Application)	Terminates the task that is running. You should use this command with caution. For more information, see Section 4.11.5, "Interrupting Your Application."

4.3 Preparing Windows Applications for Debugging

If you want to use symbolic information and access source files with CVW, preparation depends on your compiler and linker.

Suppose, for example, that you were using Microsoft C Optimizing Compiler (CL), version 5.1 or later, and Microsoft Segmented Executable Linker (LINK). You would compile with the **/Zi** option to produce object files containing symbolic information and the **/Od** (disable optimization) option to ensure that code generated by the compiler would match the statements in the C-language source code. You would link with the **/co** option to produce an executable file containing symbolic information.

For further information about the settings you need to use, see the documentation that accompanied your compiler and linker.

4.4 Setting Up the Debugging Version of Windows

You can run CVW with either the debugging or retail version of Windows. The debugging version performs error checking that is not available with the retail version.

For example, the debugging version of Windows checks whether a window handle passed to a Windows function is valid. When the debugging version of Windows detects such an error, it reports a fatal exit. If this happens while you are running CVW, the fatal exit is reported in the CVW Command window. For details about this error handling, see Section 4.12, "Handling Abnormal Termination of the Application."

When you use the debugging version of Windows with CVW, the Windows core dynamic-link libraries (DLLs) provide debugging support. These DLLs (KRNL286.EXE, KRNL386.EXE, GDI.EXE, and USER.EXE) contain symbol information that makes it easier to determine the cause of an error. For example, if your application were to cause a general protection (GP) fault while running with

the debugging version, Windows would display symbol information for the Windows code that was running when the GP fault was detected. If, instead, your application were running with the retail version of Windows, Windows would be able to display only CS:IP address values of the code that was being executed when the fault occurred.

CVW does not automatically use these Windows core DLL symbols. To provide CVW access to these symbols, you must specify one or more of the core DLLs either by using the **/l** command-line option or in response to the DLL prompt within CVW. If you are running CVW with Windows in standard mode, specify KRNL286.EXE. In 386 enhanced mode, specify KRNL386.EXE. For an explanation of how to load symbols from a DLL, see Section 4.5.5, "Starting a Debugging Session for Dynamic-Link Libraries."

To install the debugging version of Windows, run the batch program N2D.BAT from your Windows system directory. This batch program replaces the nondebugging Windows core files with the debugging versions. (It copies both symbol files and executable files.) When the batch program has finished running, you start the debugging version of Windows by typing the **win** command. No special command-line options are required. To restore the nondebugging version of Windows, follow the same procedure using the batch program D2N.BAT.

4.5 Starting a Debugging Session

As with Windows applications, you can start CVW in any of several ways. For a complete description of how to start Windows applications, see the *Microsoft Windows User's Guide*. To specify CVW options and parameters, you must choose the Run command from the File menu in Program Manager. For more information about CVW options, see Section 4.5.6, "Command-Line Options."

You can run CVW to debug any of the following:

- A single application
- Multiple instances of an application
- Multiple applications
- DLLs

This section describes the methods you use to perform these tasks and summarizes the display options you can specify when you start CVW from the Run dialog box. This dialog box appears when you choose the Run command from the File menu in Program Manager.

4.5.1 Display Options

You must specify your display selection on the command line when you start CVW. The following list describes the display options:

Option	Display configuration
None	VGA; debugging on single monitor
/v (VCV.386 must be installed)	Non-VGA; debugging on single monitor
/2	Any; debugging on secondary monochrome monitor
/8	8514/a; debugging on secondary VGA monitor

4.5.2 Starting a Debugging Session for a Single Application

After you start CVW from Windows, CVW displays the Command Line dialog box. To start debugging a single application, do the following:

1. In the Command Line dialog box, type the name of the application. If you do not include an extension, CVW assumes the .EXE extension by default. You can also include any arguments that the application recognizes. Following is the syntax of the command to start debugging a single application:

   app_name[**.exe**] [*app_arguments*]

2. Press ENTER, or choose the OK button.

   CVW displays a dialog box with the following message:

   ```
   Name any other DLL or executable with debug info.
   ```

3. Because you are debugging only one application and no DLLs, press ENTER or choose the OK button. CVW loads the application and displays on the debugging screen the source code for the application's **WinMain** function.
4. Set any breakpoints you want in the code.
5. To continue running the application, choose the <F5=Go> button on the status line or press the F5 key.

You can avoid startup dialog boxes and start CVW more quickly by specifying the application name as an argument on the command line, as follows:

1. From the Program Manager File menu, choose Run.
2. Type the application name and any application arguments on the command line. Following is the command syntax to start debugging a single application:

   cvw [*cvw_options*] *app_name*[**.exe**] [*app_arguments*]

3. Press ENTER, or choose the OK button.

4.5.3 Starting a Debugging Session for Multiple Instances of an Application

Windows can run multiple instances of an application simultaneously, which can cause a problem for your application. For example, two instances of an application might interfere with each other, or one application might corrupt the data of the other.

To help you solve problems associated with running multiple instances of an application, CVW allows you to debug multiple instances of an application at the same time. You can determine which instance of an application you are looking at by examining the DS register at any breakpoint.

To debug multiple instances of an application, perform the following steps:

1. Start CVW as usual for your application.
2. Run one or more additional instances of your application by choosing Run from the Program Manager File menu.

Specifying your application name more than once when starting CVW does not have the effect of loading multiple instances of the application.

The breakpoints you set in your application apply to all instances of the application. To determine which instance of the application has the current focus in CVW, examine the DS register.

4.5.4 Starting a Debugging Session for Multiple Applications

You can debug two or more applications at the same time, such as a dynamic data exchange (DDE) client and server. However, when global symbols are shared by applications (such as the symbol name WINMAIN), CVW resolves symbol references to the first application named when you started CVW.

Perform the following steps to debug two applications at the same time:

1. Start CVW as usual for a single application.
2. Type the name of the second application when CVW displays a dialog box with the following message:

   ```
   Name any other DLL or executable with debug info.
   ```

   You *must* include the .EXE extension after the filename of the second application.
3. Set breakpoints in either or both applications, choosing Open Module from the CVW File menu to display the source code for the different modules.
4. Press F5 to continue running the first application.

5. From the Program Manager File menu, choose Run, type the application name and any application arguments, and press ENTER or choose the OK button to start execution of the second application.

An alternative way to load the symbols for a second application is to use the **/l** option on the command line when you start CVW, as follows:

```
cvw /l second.exe first.exe
```

The **/l** option and the name of the second application must precede the name of the first application on the command line in the Run dialog box. You can repeat the **/l** option for each application to be included in the debugging session. Once CVW starts, choose the Run command from the Program Manager File menu to start the second application.

4.5.5 Starting a Debugging Session for Dynamic-Link Libraries

You can debug one or more DLLs while you are debugging an application. However, no distinction is made between global symbols shared by the applications and any DLLs.

Perform the following steps to debug a DLL at the same time as an application:

1. Start CVW as usual for the application.
2. Type the name of the DLL when CVW displays a dialog box with the following message:

   ```
   Name any other DLL or executable with debug info.
   ```

   CVW assumes the .DLL extension if you do not supply an extension with the filename. If your DLL has another extension (such as .DRV), you must specify it explicitly.
3. From the File menu, choose Open Module to display the source code for the different modules. Set breakpoints in either the application or the DLL.
4. Press F5 to continue running the application.

Alternatively, you can use the **/l** option to specify the DLL on the command line in the Run dialog box, as follows:

```
cvw /l appdll appname.exe
```

The **/l** option and the name of the DLL must precede the name of the first application on the command line. You can repeat the **/l** option for each DLL to be included in the debugging session. The .DLL extension is the default extension for the **/l** option.

CVW allows you to debug the LibEntry initialization routine of a DLL. If your application implicitly loads the library, a special technique is required to debug the LibEntry routine. An application implicitly loads a DLL if the library routines are imported in the application's module-definition (.DEF) file or if your application imports library routines through an import library when you link the application. An application explicitly loads a DLL by calling the **LoadLibrary** function.

If you type in the Command Line dialog box the name of an application that implicitly loads a DLL, CVW automatically loads the DLL and executes the DLL's LibEntry routine when CVW loads the application. In this case, you have no opportunity to debug the LibEntry routine. To avoid this problem, perform the following steps:

1. Instead of typing the name of your application in the Command Line dialog box, type the name of a dummy application that does not implicitly load the library.
2. Type the name of your DLL, being sure to include the extension if it is not .DLL, when the following message is displayed:

   ```
   Name any other DLL or executable with debug info.
   ```

3. From the File menu, choose Open Module to display the source code for the library module containing the LibEntry routine. Set breakpoints in the LibEntry routine.
4. From the File menu, choose Open Module to display the source code for other library or application modules. Set breakpoints.
5. Press F5 to start running the dummy application.
6. Run the application that implicitly loads the DLL by choosing Run from the Program Manager File menu. CVW will resume control when the breakpoint in the LibEntry routine is encountered.

Alternatively, you can use a command line of the following form to specify the dummy application, your application, and the DLL:

```
cvw /l appdll dummyapp
```

After this command starts CVW, you need to perform steps 5 and 6 of the preceding procedure.

4.5.6 Command-Line Options

Following is the command-line syntax to start CVW from the Run dialog box, which is displayed when you choose the Run command from the Program Manager File menu:

cvw [*cvw_options*] *app_name*[**.exe**] [*app_arguments*]

Parameters are not case-sensitive. Following are the command-line parameters:

cvw_options
Specifies one or more options that modify how CVW runs. Options are not case-sensitive. Valid options are as follows:

Option	Purpose
/b	Specifies a monochrome VGA display used as the secondary display with an 8514/a display. This option is valid only in conjunction with the **/8** option.
/c *command*	Specifies one or more commands that CVW is to carry out when it loads the application specified by the *app_name* parameter. The group of commands must be enclosed in double quotation marks ("). Commands must be separated with semicolons (;).
/l *dll_or_exe*	Specifies the name of an application or DLL that has been compiled and linked with CVW symbols. CVW assumes the default filename extension .DLL if no extension is supplied. You can use the **/l** option more than once to specify multiple DLLs or executable files.
/m	Disables the use of the mouse on the debugging screen. You should use this option when you set breakpoints in code that is responsive to mouse movements on the Windows application screen.
/tsf	Inverts save-state-file status for the current session. For more information, see Section 4.6, "Saving Session Information."
/v	Allows single-monitor debugging on a non-VGA display.
/2	Allows CVW to use a secondary monochrome monitor for debugger output while displaying Windows output on your primary monitor.
/8	Allows CVW to use an 8514/a display as the Windows display and a VGA display for debugger output.
/25	Specifies 25-line mode for the secondary VGA display. This option is valid only in conjunction with the **/8** option.
/43	Specifies 43-line mode for the secondary VGA display. This option is valid only in conjunction with the **/8** option.
/50	Specifies 50-line mode for the secondary VGA display. This option is valid only in conjunction with the **/8** option. The **/50** option is not required, because 50-line mode is the default for the dual-monitor configuration.

app_name[**.exe**]
Specifies the location and name of the application for which CVW is to load symbols and issue an initial breakpoint. The .EXE extension is optional.

app_arguments
Specifies one or more arguments recognized by the application that CVW loads.

4.6 Saving Session Information

After your session, CVW stores session information in a file called CURRENT.STS, which is located in the directory pointed to by the INIT environment variable or in the current directory. If this file does not already exist, CVW automatically creates it. Session information includes the following:

- CVW display windows that were opened
- Breakpoint locations

CVW saves this information, which becomes the default session information the next time you run a CVW session for that application.

By default, this feature is enabled. You can disable this feature by placing the following entry in your TOOLS.INI file:

```
[cvw]
StateFileRead: n
```

The **/tsf** option temporarily inverts this setting when you run CVW. That is, if TOOLS.INI disables this feature, running CVW with the **/tsf** option saves session information for that session only.

If your Windows session abnormally terminates while CVW is running, the CURRENT.STS file may be corrupted. This may cause CVW to fail when it first tries to execute the application you are debugging. If this happens, delete the CURRENT.STS file before attempting to run CVW again.

Note Microsoft Programmer's WorkBench (PWB) version 2.0 modifies the CURRENT.STS file. Once PWB has modified this file, CVW cannot read the command settings.

4.7 Working with the CodeView for Windows Screen

When you start CVW, the CVW menu bar and three display windows—the Local window, the Source window, and the Command window—appear.

4.7.1 Using CVW Display Windows

CVW divides the screen into logically separate sections called display windows, so that a large amount of information can be displayed in an organized and easy-to-read presentation. Each CVW display window is a distinct area on your monitor that operates independently of the other display windows. The name of each display window appears in the window's title bar. The following list describes the eight types of CVW display windows:

CVW display window	Purpose
Source window	Displays the source code. You can open a second source window to view a header file, another source file, or the same source file at a different location.
Command window	Accepts debugging commands.
Watch window	Displays the current values of selected variables.
Local window	Lists the values of all variables local to the current function or block.
Memory window	Shows the contents of memory. You can open a second Memory window to view a different section of memory.
Reg window	Displays the contents of the microprocessor's registers, including flags.
8087 window	Displays the registers of the coprocessor or its software emulator.
Help window	Displays the Help options or any Help information that you request.

4.7.1.1 Opening Display Windows

Following are the two ways to open CVW display windows:

- Choose a window from the View menu. (Note that you can open two Source windows and two Memory windows.)
- Perform an operation that automatically opens a window if it is not already open. For example, selecting a Watch variable automatically opens the Watch window.

CVW continually and automatically updates the contents of all its display windows.

4.7.1.2 Selecting Display Windows

To select a window, click anywhere in it. You can also press F6 or SHIFT+F6 to move the focus from one window to the next.

The selected window is called the active window and is marked in three ways:

- The window's name is displayed in reverse video.
- The cursor appears in the window.
- Vertical and horizontal scroll bars appear in the window.

Typing commands in the Source window causes CVW to temporarily shift its focus to the Command window. Whatever you type is appended to the last line in

the Command window. If the Command window is closed, CVW beeps in response to your input and ignores the input.

4.7.1.3 Adjusting Display Windows

CVW display windows often contain more information than they can display on the screen. Although you cannot change the relative positions of the display windows, you can manipulate a selected window by using the mouse, as follows:

- To scroll through the information in the window, use the vertical or horizontal scroll bar.
- To maximize a window so that it fills the screen, click the Maximize arrow at the right end of the window's top border. To restore the window to its previous size and position, click the Maximize arrow again.
- To change the size of a window:
  1. Position the cursor anywhere on the border between two windows.
  2. Press and hold down the left mouse button.

     Two double-headed arrows appear on the line.
  3. Drag the mouse to enlarge or reduce the window.
- To close a window, click the Close box at the left end of the top border.

  The adjacent windows automatically expand to recover the empty space.

You can also use the following keyboard commands:

Keyboard command	Description
PAGE UP or PAGE DOWN	Scrolls through the text vertically.
CTRL+F10	Maximizes a selected display window.
CTRL+F8	Enables the arrow keys to resize the active window.
CTRL+F4	Removes a selected display window.

You can also choose the Maximize, Size, and Close commands from the View menu to manipulate a selected display window.

The different CVW display windows can help you to conduct a variety of debugging activities simultaneously. These activities are initiated and controlled with CVW debugging commands, which you can type on the command line when you start CVW or choose from CVW menus.

4.7.2 Using the Menu Bar

In addition to display windows, the CVW screen includes a menu bar, which contains the following menus. For a more detailed description of CVW menus and commands, see CVW Help.

Menu	Contents
File	This menu contains the following commands:

Command	Description
Open Source	Opens any text file, and reads it into the active Source window.
Open Module	Opens the source file of any module for which CVW information has been loaded, and reads it into the active Source window.
Exit	Ends your CVW session, and returns you to Windows.

Menu	Contents
Edit	This menu contains the following commands:

Command	Description
Undo	Retracts the most recent edit, and restores the current line to its previous condition.
Copy	Copies selected text to the paste buffer.
Paste	Inserts text from the paste buffer into the active window at the present cursor location, if that location is valid (for example, text cannot be pasted into the Source window).

Menu	Contents
View	This menu contains the following commands:

Command	Description
Source	Opens a new Source window.
Memory	Opens a new Memory window.
Register	Acts as a switch to open and close the Reg window.
8087	Acts as a switch to open and close the 8087 window.
Local	Acts as a switch to open and close the Local window.
Watch	Acts as a switch to open and close the Watch window.
Command	Acts as a switch to open and close the Command window.
Help	Acts as a switch to open and close the Help window.
Maximize	Enlarges the active window so that it fills the screen.
Size	Enables the arrow keys to resize the active window.
Close	Closes the active window.

Menu	Contents
Search	This menu contains the following commands:

Command	Description
Find	Searches for the next occurrence of a text string or a regular expression that you supply in the Find dialog box.
Selected Text	Searches for the next occurrence of a string of selected text.
Repeat Last Find	Searches for the next occurrence of the string or regular expression specified in the previous Find dialog box.
Label/Function	Searches for a label definition or function in the active Source window; if one is found, moves the input focus to the found label definition or function in the active Source window.

Menu	Contents
Run	This menu contains the following command:

Command	Description
Animate	Continues running an application while displaying the execution path in the Source window. This type of display is called an animated trace display.

Menu	Contents
Watch	This menu contains the following commands:

Command	Description
Add Watch	Adds an expression to the Watch window.
Delete Watch	Deletes an expression from the Watch window.
Set Breakpoint	Specifies where to interrupt execution of an application. You can set breakpoints on lines of source code, variables, expressions, and Windows messages.
Edit Breakpoints	Performs editing functions on breakpoints; they can be added, removed, modified, enabled, or disabled.
Quick Watch	Selects one expression for the Quick Watch dialog box. For a description of the Quick Watch window, see Section 4.9.4, "Using the Quick Watch Command."

Menu	Contents
Options	This menu contains the following commands:

Command	Description
Source Window	Sets the display characteristics of the active Source window.
Memory Window	Sets the display characteristics of the active Memory window.
Trace Speed	Sets the speed of tracing and execution of an application.
Case Sensitivity	Turns case sensitivity on or off.
386 Instructions	Reads all 80386 instructions as 32-bit values when this command is checked; otherwise, reads all instructions as 16-bit values.

Menu	Contents
Calls	The contents and size of this menu change as your application runs. The Calls menu shows the currently executing routine and the trail of routines from which it was called. Your application must execute at least the beginning of the **WinMain** function before CVW will display the current routine. When you select one of the lines in the Calls menu, CVW displays the source code corresponding to the calling location in the active source window.
Help	This menu can be used to access Help.

4.8 Accessing Help

CVW Help contains detailed information and examples not found in this chapter. You can access Help by choosing a command from the Help menu described in the preceding section or by selecting an item on your screen and pressing F1. Help is available on such items as commands, menus, dialog boxes, and error messages.

4.9 Displaying Application Data

CVW offers a variety of ways to display variables, processor registers, and memory. You can also modify the values of any of these items as the application runs. This section describes how to display the following:

- Variables in the Watch window
- Expressions in the Watch window
- Arrays and structures in the Watch window
- A single expression in the Quick Watch dialog box

- Windows messages in the Command window
- Memory in the Memory window
- Contents of registers in the Reg window

4.9.1 Displaying Variables

You can use the Watch window to monitor the value of a given variable throughout the execution of your application. For example, **do**, **for**, and **while** loops can cause problems when they don't terminate correctly. By displaying loop variables in the Watch window, you can determine whether a loop variable achieves its proper value.

To add a variable to the Watch window, perform the following steps:

1. In the Source window, use the mouse or the arrow keys to position the cursor on the name of the variable you want to watch.
2. From the Watch menu, choose Add Watch, or press CTRL+W.

   An Add Watch dialog box appears with the selected variable's name displayed in the Expression field.
3. Choose the OK button or press ENTER to add the variable to the Watch window.

   If you want to add a variable other than the one shown in the dialog box, type its name over the one displayed and press ENTER.

Adding a Watch variable opens the Watch window automatically if it is not already open. The Watch window appears at the top of the screen.

When you add a local variable, the following message may be displayed:

```
Watch Expression Not in Context
```

This message appears when execution has not yet reached the C-language function that defines the local variable. Global variables (those declared outside C-language functions) never cause CVW to display this message; you can watch them from anywhere in the application.

If any two or more applications or DLLs you are debugging contain global variables with the same name, CVW displays the variable of only the first application or DLL containing that variable name.

For example, if you are debugging App1 and App2, which both contain a global variable named hInst, CVW always displays the value of hInst in App1—even if CVW stopped at a breakpoint in App2.

The Watch window can display as many variables as you like; the quantity is limited only by available memory. You can scroll through information in the

Watch window to view other variables. CVW automatically updates all watched variables as the application runs, including those not currently visible.

To remove a variable from the Watch window, do the following:

1. From the Watch menu, choose Delete Watch.
2. Scroll through information in the Delete Watch dialog box, and select the variable you want to remove.

Alternatively, you can position the cursor on any line in the Watch window and press CTRL+Y to delete the line.

4.9.2 Displaying Expressions

You may have noticed that the Add Watch dialog box prompts for an expression, not simply a variable name. You can add any valid combination of variables, constants, or operators as an expression for CVW to evaluate and display in the Watch window.

The advantage of evaluating expressions is that you can reduce several variables to a single value, which may be easier to interpret than the components that make it up. For example, imagine a **for** loop in which the ratio between two variables, var1 and var2, should remain constant. You suspect that one of these variables sometimes has the wrong value. To see when the quotient changes, without having to mentally divide two numbers, you can specify the following expression for display in the Watch window:

```
(var1 / var2)
```

You can also display Boolean expressions. For example, if the variable var is never supposed to be greater than 100 or less than 25, the following expression evaluates to 1 (TRUE) when var exceeds its limits:

```
(var < 25 || var > 100)
```

4.9.3 Displaying Arrays and Structures

An application variable is usually a scalar quantity (a single character, integer, or floating-point value). The variable appears in the Watch window with the variable name to the left, followed by an equal sign (=) and the current value.

The Watch window provides a different way to display aggregate data items, such as arrays and structures. Arrays and structures contain multiple values that can be arranged in one or more layers. You can control how these variables appear in the Watch window—whether all, part, or none of their internal structure is displayed.

For example, the array WordHolder initially appears in the Watch window in the following form:

```
+WordHolder[] = [...]
```

The brackets indicate that this variable contains more than one element. The plus sign (+) indicates that the variable has more elements than are displayed on the screen. You can expand the variable to display any or all of its components; this technique is called dereferencing.

To dereference (expand) the array, you can double-click anywhere on the displayed line or you can position the cursor on the line and press ENTER. For example, if WordHolder is a six-character array containing the word Basic, the Watch window display changes to the following:

```
-WordHolder[]
   [0]  =  66 'B'
   [1]  =  97 'a'
   [2]  =  115 's'
   [3]  =  105 'i'
   [4]  =  99 'c'
   [5]  =  0 ''
```

Note that both the individual character values and their ASCII decimal equivalents are listed. The minus sign (-) indicates that no further expansion is possible. To contract the array, you can double-click its line again or you can position the cursor on the line and press ENTER.

4.9.3.1 Displaying Character Arrays

If viewing a character array in this form is inconvenient, use either of the following methods to specify the watchpoint:

- Type the variable name, a comma (**,**), and the letter **s**, as shown in the following example:

  ```
  WordHolder,s
  ```

  CVW displays the contents of the array, as follows:

  ```
  WordHolder,s[] = "Basic"
  ```

- Cast the variable's name to a character pointer, as shown in the following example:

  ```
  (char *)WordHolder
  ```

  CVW displays the address of the array and its contents, as follows:

  ```
  (char *)WordHolder = 0x8C7:0x0010 "Basic"
  ```

4.9.3.2 Displaying Multidimensional Arrays

You can display an array with more than one dimension. For example, imagine an integer array (5 by 5) named Matrix, whose diagonal elements are the numbers 1 through 5 and whose other elements are zero. Unexpanded, the array is displayed like this:

```
+Matrix[] = [...]
```

Double-click on the word Matrix (or position the cursor on that line and press ENTER) to change the display to the following:

```
-Matrix[]
  +[0][]  =  [...]
  +[1][]  =  [...]
  +[2][]  =  [...]
  +[3][]  =  [...]
  +[4][]  =  [...]
```

The actual values of the elements are not shown yet. You have to descend one more level to see them. For example, to view the elements of the third row of the array, position the cursor anywhere on its subscript line (the +[2] line) and press ENTER. The following example shows the third row of the array dereferenced:

```
-Matrix[]
  +[0][]  =  [...]
  +[1][]  =  [...]
  -[2][]
     [0]  = 0
     [1]  = 0
     [2]  = 3
     [3]  = 0
     [4]  = 0
  +[3][]  =  [...]
  +[4][]  =  [...]
```

Dereferencing the fifth row (+[4]) of the array produces this display:

```
-Matrix[]
  +[0][]  =  [...]
  +[1][]  =  [...]
  -[2][]
     [0]  = 0
     [1]  = 0
     [2]  = 3
     [3]  = 0
     [4]  = 0
  +[3][]  =  [...]
  -[4][]
     [0]  = 0
     [1]  = 0
     [2]  = 0
     [3]  = 0
     [4]  = 5
```

Any element of an array or structure can be independently expanded or contracted; you need not display every element of the variable. If you want to view only one or two elements of a large array, specify the particular array or structure elements in the Expression field of the Add Watch dialog box.

You can dereference a pointer in the same way as an array or structure. The Watch window displays the pointer address, followed by all the elements of the variable to which the pointer currently refers. You can display multiple levels of indirection (that is, pointers referencing other pointers) simultaneously.

4.9.3.3 Displaying Dynamic Array Elements

An array may have dynamic elements that change as some other variable changes. Just as you can display a particular element of an array by selecting its subscript, you can also display a dynamic array element by specifying its variable subscript. For example, suppose that the loop variable p is a subscript for the array variable Catalogprice. The Watch window expression Catalogprice[p] displays only the array element currently specified by the variable p, not the entire array.

You can mix constant and variable subscripts. For example, the expression BigArray[3][i] displays only the element in the third row of the array to which the index variable i points.

4.9.4 Using the Quick Watch Command

Using the Quick Watch command is a convenient way to take a quick look at a variable or expression. Because the Quick Watch dialog box can display only one variable at a time, it's best to use the Watch window to view most variables.

Selecting the Quick Watch command from the Watch menu (or pressing SHIFT+F9) displays the Quick Watch dialog box. If the cursor is in the Source, Local, or Watch window, the variable at the current cursor position appears in the Quick Watch dialog box.

The Quick Watch display automatically expands arrays and structures to their first level. For example, an array with three dimensions expands to the first dimension. You can expand or contract an element just as you would in the Watch window; position the cursor on the appropriate line and press ENTER. If the array has more lines than the Quick Watch dialog box can display, you can view the rest of the array either by using the scroll bar or by pressing the DOWN ARROW or PAGE DOWN key.

To add a Quick Watch item to the Watch window, choose the Add Watch button. Arrays and structures appear in the Watch window expanded as they were displayed in the Quick Watch dialog box.

You can also display a Quick Watch dialog box for a variable by typing two question marks and the variable name in the Command window. For example, the following command shows the contents of the Index variable:

```
?? Index
```

4.9.5 Tracing Windows Messages

You can trace occurrences of a Windows message or an entire class of Windows messages by using the **wwm** (Windows Watch Message) command. CVW displays the messages in the CVW Command window.

To trace a Windows message or message class, type the **wwm** command in the Command window. The syntax for the command is as follows:

wwm *winproc msgname* | *msgclasses*

The *winproc* parameter is the symbol name or address of an application's window procedure. The *msgname* parameter is the name of a Windows message, such as WM_PAINT. The *msgclasses* parameter is a string of characters that identify one or more classes of messages to be traced. If *msgclasses* is not specified, CVW traces all message classes. The class, if specified, is consistent with those defined in Microsoft Windows Spy (SPY.EXE); they are as follows:

Message class	Type of Windows message
c	Clipboard
d	DDE
i	Initialization
m	Mouse
n	Input
s	System
w	Window management
z	Nonclient

For example, the following command traces all mouse and input messages sent to the MainWndProc procedure:

```
wwm MainWndProc mn
```

The following example illustrates how the CVW Command window displays a Windows message:

```
HWND:1c00 wParm:0000 lParm:000000 msg:000F WM_PAINT
```

4.9.6 Displaying Memory

Selecting the Memory command from the View menu opens a Memory window. You can have two CVW Memory windows open at a time.

By default, memory is displayed as byte values in hexadecimal format, with 16 bytes per line. At the end of each line is a second display of the same memory in ASCII form. Values that correspond to printable ASCII characters (decimal values 32 through 127) are displayed in decimal format. Values outside that range are represented by periods (.).

Byte values are not always the most convenient way to view memory. If the area of memory you are examining contains character strings or floating-point values, you might prefer to view them in a directly readable form. The Memory Window command on the Options menu displays a dialog box with the display options in the following categories:

- ASCII characters
- Byte, word, or doubleword binary values
- Signed or unsigned integer decimal values
- Short (32-bit), long (64-bit), or 10-byte (80-bit) floating-point values

You can also cycle through these display formats directly by pressing SHIFT+F3.

If a section of memory cannot be displayed as a valid floating-point number, the value shown includes the characters NAN (not a number).

4.9.6.1 Displaying Local and Global Memory Objects

CVW is also useful for displaying global and local memory objects in their respective Windows heaps. You can use the **wdg** (Windows Display Global Heap) command to display the entire heap of global memory objects in the Command window, or you can use the **wdl** (Windows Display Local Heap) command to display the entire heap of local memory objects in the Command window.

For the **wdg** command, you can specify a global handle to display a partial list of the global heap. The Command window displays the first five memory objects in the global heap, starting at the handle rather than at the beginning of the heap. The following example illustrates the **wdg** output format:

```
❶      ❷       ❸          ❹          ❺    ❻
047E  (0A7D)  00000020b  MYAPP     PRIV MOVEABLE DISCARDABLE

                                                ❼
0A6D          00000134b  MYAPP     DATA FIXED PGLOCKED=0001

                              ❽
0806  (0805)  00000600b  PDB (0465)

 ❾
FREE          000000A0b
```

The following table describes the indicated fields:

Field	Description
❶	The value of the handle of a global memory object. Global memory objects are displayed in the order in which Windows manages them, which is typically not in ascending handle order.
❷	A memory selector. This value is not displayed if the selector value is the same as the global handle, as is the case for **DATA** objects.
❸	The length, in bytes, of the global memory object.
❹	The name of the application or library module that allocated the object.
❺	The type of global memory object, which can be the following:

Field	Description
	Type / **Meaning**: **PRIV** — Application or DLL global data, or system object; **CODE** — Code segment; **DATA** — Data segment of application or DLL; **FREE** — Free memory object in the global heap
❻	One of the following memory allocation attributes: **MOVEABLE**, **MOVEABLE DISCARDABLE**, **FIXED**
❼	One of the following dispositions if the object is movable: **LOCKED=***number* — Number of times the object has been locked with any of the Windows functions that lock data; **PGLOCKED=***number* — Number of times Windows has locked the object in its linear address space
❽	The handle of the application or library module that allocated the process descriptor block (PDB).
❾	A free memory object, followed by the size of the free object, in bytes.

The following example shows sample output of the **wdl** (Windows Display Local Heap) command:

```
❶      ❷      ❸      ❹
190A:  000A   BUSY   (16DA)
```

The following table describes the indicated fields:

Field	Description
❶	The offset of the local memory object in the local data segment
❷	The length of the object, in bytes
❸	One of the following dispositions:

Disposition	Meaning
BUSY	A currently allocated object
FREE	A free object in the local heap

Field	Description
❹	A local memory handle

4.9.6.2 Displaying Variables with a Live Expression

Section 4.9.4, "Using the Quick Watch Command," explains how to display a specific array element by adding the appropriate expression to the Watch window. It is also possible to view a particular array element or structure element in the Memory window. This CVW display feature is called a live expression, because the displayed area of memory changes to reflect the value of a pointer or subscript. For example, if Buffer is an array and pBuf is a pointer to that array, then *pBuf points to the array element currently referenced. A live expression displays the section of memory beginning with this element.

CVW displays live expressions in a Memory window. To create a live expression:

1. From the Options menu, choose Memory Window.
2. Select the Live Expression check box, and type the name of the element you want to view.

   For example, if pszMsg is a pointer to a null-terminated array of characters and you want to see what it currently points to, type the following:

   ```
   *pszMsg
   ```

3. Choose the OK button, or press ENTER.

A new Memory window opens. The first memory location in the window is the first memory location of the live expression. The section of memory displayed changes to the section the pointer currently references.

You can use the Memory Window command on the Options menu to display the value of the live expression in a readable form. This is especially convenient when the live expression represents strings or floating-point values, which are difficult to interpret in hexadecimal form.

It is usually more convenient to view an item in the Watch window than as a live expression. However, you might find some items easier to view as live expressions. For example, you can examine what is currently at the top of the stack by specifying SS:SP as the live expression.

4.9.6.3 Dereferencing Memory Handles

In a Windows application, the **LocalLock** and **GlobalLock** functions are used to dereference memory handles into near or far pointers. In a debugging session, you may know the handle of the memory object, but might not know which near or far address it dereferences to, unless you are debugging in an area where the application has just completed a **LocalLock** or **GlobalLock** function call. To get the near and far pointer addresses for your local and global handles, use the **(lh)** and **(gh)** type casts. For example, you could use **(lh)** to dereference the array in the following code:

```
HANDLE hLocalMem;
PBYTE pbArray;

hLocalMem = LocalAlloc(LMEM_MOVEABLE, 100);
pbArray = (PBYTE)LocalLock(hLocalMem);

/* Use the array.     */

LocalUnlock(hLocalMem);
```

To properly display this array in CVW, you can use the following command:

```
dw (lh)hLocalMem
```

If you set a breakpoint immediately after the **LocalLock** function, you could find out where the local object was allocated in the application's data segment by looking at the value of the pbArray variable. To display the value of pbArray, use the following CVW command:

```
dw pbArray
```

Note that you cannot rely on the value of pbArray anywhere else in the application, because it may change or the memory object may move.

In the following example, the memory object lpszTest is a string:

```
HANDLE hGlobalMem;
LPSTR  lpszTest;

hGlobalMem = GlobalAlloc(GMEM_MOVEABLE, 10L)
lpszTest = GlobalLock(hGlobalMem);

lstrcpy(lpszTest, "ABCDEF");

GlobalUnlock(hGlobalMem);
```

To display the contents of the string, you could use double type casting, as follows:

```
? *(char far*) (gh)lpszTest,s
```

The **(gh)** type cast returns a pointer to the far address of the global memory object.

4.9.7 Displaying the Contents of Registers

Selecting the Register command from the View menu (or pressing F2) opens a Reg window on the right side of the screen. The current values of the microprocessor's registers appear in this window.

At the bottom of the window are a group of mnemonics representing the processor flags. When your application first starts running, all values are shown in normal-intensity video. Any subsequent changes are marked in high-intensity video. For example, suppose the overflow flag is not set when the application starts. The corresponding mnemonic is NV, and it appears in normal-intensity video. If the overflow flag is subsequently set, the mnemonic changes to OV and appears in high-intensity video.

Selecting the 386 Instructions command from the Options menu displays the contents of the registers as 32-bit values. This command is valid only if your computer uses an 80386 processor. Selecting this command a second time changes the registers back to 16-bit values.

You can also display the registers of an 8087/80287/80387 coprocessor in a separate window by choosing the 8087 command from the View menu. If your application uses a coprocessor emulator, the emulated registers are displayed instead.

4.9.8 Displaying Windows Modules

The **wdm** (Windows Display Modules) command displays a list of all the DLL and task modules that Windows has loaded. For each module, the list shows the module handle, the type of module (DLL or task), the name of the module, and the path of the module.

4.10 Modifying Application Data

You can easily change the values of variables, memory locations, or registers displayed in the Watch, Memory, Reg, or 8087 window. Simply position the cursor at the value you want to change, and type the appropriate value. If you change your mind, press ALT+BACKSPACE to undo the last change you made.

The Memory window displays the starting address of each line in segment:offset form. Altering the address automatically shifts the display to the corresponding section of memory. If that section is not used by your application, memory locations are displayed as double question marks (??). You cannot change memory that is displayed as question marks.

You can also change the values of memory locations by modifying the right side of the memory display, which shows memory values in ASCII form. For example, you can change a byte from decimal value 75 (ASCII value for uppercase K) to decimal value 85 (ASCII value for uppercase U). To do so, place the cursor over the letter K, which corresponds to the position where the memory value is 75, and type **U**.

To change a processor flag, you can click its mnemonic or you can position the cursor on a mnemonic and press any key (except TAB or SPACEBAR). Repeat these operations to restore the flag to its previous setting.

Although you can alter most items from the Watch window, sometimes it is useful to modify a register or memory directly. For example, if a function returns a value in the AX register, you can modify the AX register to change a returned value without executing the function.

Caution You should be especially cautious when altering machine-level values. The effect of changing a register, flag, or memory location may vary from having no effect at all to causing the operating system to crash.

4.11 Controlling Execution of Your Application

This section describes how you can use CVW to control the execution of your application.

Following are the three possible forms of execution in CVW:

Application execution	Description
Continuous	The application runs until either a previously specified breakpoint has been reached or the application terminates normally.
Single-step	The application pauses after each line of code has been executed.
Animated	The application pauses after each line of code has been executed, but execution continues after a short pause. The application continues to run until you press a key.

4.11.1 Continuous Execution

With continuous execution, you can quickly execute bug-free sections of code. To initiate continuous execution, either you can click the right mouse button on the line of code you want to debug or examine in more detail or you can position the cursor on this line and then press F7. Execution proceeds at full speed and pauses when it reaches the selected line.

You can also use a breakpoint to cause execution to pause at a specific line of code. CVW provides you with several types of breakpoints to control your application's execution. The sections that follow describe how to use breakpoints.

4.11.1.1 Selecting Breakpoint Lines

By specifying one or more lines as breakpoints, you can skip over the parts of the application that you don't want to examine. Execution of the application proceeds at full speed up to the first breakpoint, at which execution is interrupted; pressing F5 causes execution to continue up to the next breakpoint; and so on. You can set as many breakpoints as you want, provided that you have available memory.

Following are several ways to set breakpoints:

- Double-click anywhere on the desired breakpoint line. The selected line is highlighted to show that it is a breakpoint. To remove the breakpoint, double-click on the line a second time.

- Position the cursor anywhere on the line at which you want execution to pause. Press F9 to select the line as a breakpoint and to highlight it. Press F9 a second time to remove the breakpoint and highlighting.
- Display the Set Breakpoint dialog box by choosing the Set Breakpoint command from the Watch menu. Select one of the breakpoint options that permits you to specify a line (location). The line on which the cursor rests is the default breakpoint line in the Location field. If this line is not the location you want, replace it by typing another line number in the Location field. When you type a new line number, make sure that you precede it with a period.
- Your application can call the Windows **DebugBreak** function to interrupt execution and return control to CVW. When your application calls the **DebugBreak** function, execution may stop within the **DebugBreak** code rather than in your application. You may have to single-step out of the **DebugBreak** code and back into your application.

A breakpoint line must contain executable code. You cannot select a blank line, a comment line, or a declaration line (such as a variable declaration or a preprocessor statement) as a breakpoint.

To set a breakpoint on a multiline statement, you must position the cursor on the last line of the statement. If you try to set a breakpoint on any other line of the statement, CVW does not accept it.

If your compiler optimizes your code, some lines of code may be repositioned or reorganized for more efficient execution. These changes can prevent CVW from recognizing the corresponding lines of source code as breakpoints. Therefore, it is a good idea to disable optimization during development. You can restore optimization once debugging is completed.

A breakpoint can also be set at a function or an explicit address. To set a breakpoint at a function, simply enter the name of the function in the Set Breakpoint dialog box. To set a breakpoint at an address, enter the address in CS:IP form.

If any of the applications or DLLs you are debugging share names for certain window procedures (such as MainWndProc), you can refer by name only to the procedure that is defined in the first application or DLL.

You can remove a breakpoint by choosing the Edit Breakpoints command from the Watch menu or by selecting the breakpoint in the Source window and pressing F9. When your application pauses at a breakpoint, you can continue execution by pressing F5. You cannot remove a breakpoint set by an application calling the **DebugBreak** function.

4.11.1.2 Setting Breakpoint Values

Breakpoints are not limited to specific lines of code. CVW can also break execution when an expression changes value or reaches a particular value. Use one of the following methods to set a breakpoint value:

- To interrupt execution when an expression *changes* value, type the name of the expression in the Expression field of the Set Breakpoint dialog box.
- To interrupt execution when an expression *reaches* a particular value, use that value in the expression you type in the Expression field of the Set Breakpoint dialog box.

  For example, if you want the application to pause when a variable named looptest equals 17, type the following in the Expression field:

  ```
  looptest==17
  ```

  The application pauses when this statement becomes true.

You can also use the Set Breakpoint dialog box to combine value breakpoints with line breakpoints so that execution stops at a specified line only if an expression has simultaneously changed value or reached a specified value.

For large variables (such as arrays and character strings), you can specify the number of bytes you want checked (up to 32K) in the Length field.

Note When a breakpoint is tied to a variable, CVW must check the variable's value after each machine instruction is executed. This computational overhead slows execution greatly. For maximum speed when debugging, either tie value breakpoints to specific lines or set value breakpoints only after you have reached the section of code that needs to be debugged.

4.11.1.3 Setting Breakpoints on Windows Messages

You can also set a breakpoint on a Windows message or an entire class of Windows messages. By using this feature, you can track your application's response to user input and window-management messages.

To set a breakpoint on a Windows message or message class, type the **wbm** (Windows Breakpoint Message) command in the Watch window. The syntax for the command is:

wbm *winproc msgname* | *msgclasses*

The *winproc* parameter is the symbol name or address of an application's window procedure. The *msgname* parameter is the name of a Windows message, such as WM_PAINT. The *msgclasses* parameter is a string of characters that identify one or more classes of messages. If *msgclasses* is not specified, CVW traces all

message classes. If it is specified, the classes are consistent with those defined in Microsoft Windows Spy (SPY.EXE); they are as follows:

Message class	Type of Windows message
c	Clipboard
d	DDE
i	Initialization
m	Mouse
n	Input
s	System
w	Window management
z	Nonclient

For example, if your application is failing to refresh the client area of a window, you might set a breakpoint on the WM_PAINT message so that you can watch your application's behavior as it processes the message. The following command interrupts execution whenever the application's MainWndProc procedure receives a WM_PAINT message:

```
wbm MainWndProc WM_PAINT
```

4.11.1.4 Using Breakpoints

This section shows how breakpoints can help you find the cause of a problem.

One of the most common bugs is a **for** loop that executes too many or too few times. If you set a breakpoint that encloses the loop statements, the application pauses after each iteration. You can then monitor the loop variable or critical program variables in the Watch or Local window to find the error in loop processing.

You can specify that a breakpoint is to be ignored. To set the number of times a breakpoint is to be ignored before execution is interrupted, perform the following steps:

1. From the Watch menu, choose Set Breakpoint.
2. In the Pass Count field of the Set Breakpoint dialog box, type the decimal number.

For example, suppose your application repeatedly calls a function to create a binary tree. You suspect that something goes wrong approximately halfway through the process. You could mark the line that calls the function as the breakpoint, then specify how many times this line is to be executed before execution pauses. Running the application creates a representative (but unfinished) tree structure that can be examined from the Watch window. You can then continue your analysis by using single-step execution, which is described in the next section.

Another programming error is assignment of the wrong value to a variable. If you enter a variable in the Expression field of the Set Breakpoint dialog box, execution is interrupted every time the variable changes value.

Breakpoints make it possible for you to interrupt execution of an application so that you can assign new values to variables. For example, if a limit value is set by a variable, you can change the value to see if it affects the application's execution. Similarly, you can pass a variety of values to a **switch** statement to see if they are correctly processed. This ability to alter variables provides an especially convenient way to test new functions without having to write a stand-alone test application.

When your application reaches a breakpoint and you change a variable, you might want to watch each step be executed while you check the value of that variable. This technique is called single-stepping.

4.11.2 Single-Step Execution

When single-stepping, CVW pauses after each line of code is executed. If a line contains more than one executable statement, CVW executes all the statements on the line before pausing. The next line to be executed is displayed in reverse video. You can use either the Trace command or the Step command to single-step through an application.

To use Trace, press F8. Trace displays each step of every function for which CVW has symbolic information. Each line of the function is a separate step. If CVW does not have symbolic information for a function, the function runs in a single step.

To use Step, press F10. Step displays each step of the current function but does not step into function calls. Instead, the called function runs as a single step.

You can alternate between Trace and Step as you like. Which method you should use depends on whether you want to see what happens within a particular function.

Attempting to step or trace through Windows startup code while viewing assembly-language instructions causes unpredictable results. To step through your application while viewing assembly-language instructions, set a breakpoint at the **WinMain** function and begin stepping through the application only after the breakpoint has been reached.

Using the Trace command to step out of a window procedure causes CVW to step into Windows system code.

4.11.3 Animated Execution

To trace through the application continuously without having to press F8, choose the Animate command from the Run menu. The speed of execution is controlled by the Trace Speed command on the Options menu. You can interrupt animated execution at any time by pressing any key.

4.11.4 Jumping to a Particular Location

At times, you may wish to force the system to jump to a particular location in your application during execution. For example, you may want to avoid executing code that you know has bugs, or you may want to repeatedly execute a particularly troublesome portion of your application.

To jump to a specific location in your application, do the following:

1. From the Options menu, choose Source. Select the Mix Source and Assembly radio button and the Show Machine Code check box.
2. In the Source window, view the line of source code to which you want to jump.
3. Examine the code offset of the first machine instruction for the assembled statement.
4. To change the IP register to this code offset, type the **rip** (Register IP) command in the command window, supplying the value in hexadecimal format.

CVW highlights the line to which you have jumped.

Caution Do not jump from one procedure to another. Jumping from one procedure to another disrupts the stack.

Assembled source code for a given statement may rely on memory values and registers set in previous instructions. If you cause execution to jump to a specific point in your application, values and registers may not be correctly set, particularly if optimization was not disabled during compiling.

4.11.5 Interrupting Your Application

There may be times when you want to interrupt your application immediately. You can force an immediate interruption of a CVW session by pressing CTRL+ALT+SYS RQ. You then have the opportunity to change debugging options; for example, you can add breakpoints and modify variables. To resume continuous execution, just press F5; to single-step, press F10.

You should take care when you interrupt the CVW session. For example, if you interrupt the session while Windows code or other system code is being executed,

attempting to use the Step command or the Trace command could produce unpredictable results. When you interrupt the CVW session, it is usually safer to set breakpoints in your code and resume continuous execution than to use Step or Trace.

An infinite loop in your code presents a special problem. Again, because you should avoid using Step or Trace after interrupting your application, you should try to locate the loop by setting breakpoints in places you suspect are in the loop.

Whether or not you locate the infinite loop, you will have to terminate your application. The **wka** (Windows Kill Application) command terminates the task that is currently running. You should use the **wka** command only when your application is the one being executed.

If your application is currently executing a module that contains symbol information, the CVW Source window highlights the current instruction. However, if your application contains modules without symbolic information, it is more difficult to determine whether the assembly-language code displayed in the Source window belongs to your application or to another task.

In this case, use the **wdg** (Windows Display Global Heap) command, supplying the value in the CS register as the parameter. CVW displays a listing that indicates whether the code segment belongs to your application. If the code segment does belong to your application, you can use the **wka** command without affecting other tasks. The **wka** command does not perform all the cleanup tasks associated with the normal termination of a Windows application. For example, graphics device interface (GDI) objects created during the execution of the application but not destroyed before you terminated the application remain allocated in the systemwide global heap. This reduces the amount of memory available during your Windows session. Because of this, you should use the **wka** command to terminate the application only if you cannot terminate it normally.

The **wka** command simulates a fatal error in your application. Because of this, when you use the **wka** command, Windows displays an error message. After you close the message box, Windows may not release subsequent mouse input messages from the system queue until you press a key. If this happens, the cursor moves on the Windows screen, but Windows does not appear to respond to the mouse. After you press any key, Windows responds to all mouse events that occurred before you pressed the key.

4.12 Handling Abnormal Termination of the Application

Your application can terminate abnormally in one of two ways while you are debugging it with CVW. It can cause a fatal exit, or it can cause a GP fault. In both cases, CVW regains control, giving you the opportunity to examine the state of the system when your application terminated. In particular, you can often determine the location in your application's code where the error occurred or which call caused the error. CVW makes it possible for you to view registers, display the global heap, display memory, and examine the source code.

Once you have determined where the error occurred, type the **q** (Quit) command in the Command window to terminate CVW. In most cases, control returns to Windows.

4.12.1 Handling a Fatal Exit

If the abnormal termination was a fatal exit and the application was running with the retail version of Windows, CVW displays a fatal exit code and the CS:IP register contains an address in the Windows code itself. This small amount of information provides little to help you locate the last call that your application made before the error was detected.

If, however, your application was running with the debugging version of Windows, the CVW Command window displays a stack trace that is much more useful for finding the error in your source code.

After the stack trace appears in the CVW Command window, Windows prompts you with the following message:

```
Abort, Break, or Ignore?
```

To locate the cause of the error, press the B key. This allows CVW to regain control from Windows.

In most cases, the stack trace will have been scrolled past the top of the CVW Command window; but once CVW regains control, you can scroll the information in the window to examine the entire stack trace. The following information appears at the top of the stack trace:

- A fatal exit number. For more information about Windows debugging messages, see Appendix C, "Windows Debugging Version."
- The CS:IP address, the name of the Windows function where the error was detected, or the name of the last Windows function called before the error was detected.

Following this information, additional Windows functions may be listed in the stack trace. Somewhere near the top of the stack trace, a CS:IP address is listed without a Windows function name. In most cases, this is the location in the source code of your application at which the call to a Windows function occurred, triggering the fatal exit.

To examine this location in your source code, open or switch to a Source window and use the **v** (View) command followed by the CS:IP address; be sure to precede both the segment and the offset with the hexadecimal prefix **0x**. For example, if CVW indicates that the error occurred at 07DA:0543 in your application, type the following command:

```
v 0x07DA:0x0543
```

If the module at which the error occurred was compiled to produce object files containing symbolic information, the CVW Source window displays the location in your code at which the errant call to a Windows function occurred.

The first CS:IP address without a name in the stack trace may point to a location in your code without symbols. For example, the code may be in a DLL you didn't specify with the **/l** command-line option or when CVW prompted you for a DLL, or the address might be in a module that was not compiled to produce symbolic information. In such cases, CVW reports that no source code is available. If this happens, continue down the stack trace, using the **v** command to examine each unnamed CS:IP address. You are likely to find a location in a module that was compiled to produce symbolic information and to find this location made a call into one of your modules that was not compiled to produce symbolic information.

4.12.2 Handling a General Protection Fault

When a general protection (GP) fault occurs, CVW displays a message in the Command window to notify you of the event. If the GP fault occurred at an instruction in one of your modules, CVW displays the corresponding source code if the module was compiled to produce symbolic information. You can obtain information about the chain of calls leading up to the GP fault by using the CVW Call menu. This menu displays a backtrace of calls in the form of a series of segments and offsets, starting at the most recent call.

If your application was running with the debugging version of Windows, the backtrace shows function names next to some of the segment:offset pairs. By examining the function names, you may be able to determine where in your code the error occurred.

4.13 Ending a Session

To terminate a CVW session, you can choose the Exit command from the File menu or type the **q** (Quit) command in the Command window.

You can also terminate your application without terminating CVW. While Windows is terminating the application, it notifies CVW. CVW then displays the following message:

```
Program terminated normally (0)
```

The value in parentheses is the return value of the **WinMain** function. This value is usually the *wParam* parameter of the WM_QUIT message, which in turn is the value of the *nExitCode* parameter passed to the **PostQuitMessage** function.

If you were debugging more than one application or DLL, you can press F5 to continue the debugging session.

4.14 Advanced Techniques

Once you are comfortable displaying variables, changing variables, and controlling the execution of your application, you may want to experiment with the following advanced techniques:

- Using multiple Source windows
- Checking for undefined pointers
- Handling register variables
- Redirecting CVW input and output

4.14.1 Using Multiple Source Windows

You can have two Source windows open at the same time. The windows can display two different sections of source code for the same application. They can both track CS:IP addresses, or one can display a high-level listing and one can display an assembly-language listing. You can move freely between the Source windows, executing a single line of source code or a single assembly-language instruction at a time.

4.14.2 Checking for Undefined Pointers

Until a pointer has been explicitly assigned a value, its value is undefined. Its value can be completely random, or it can be some consistent value (such as 1) that does not point to a useful data address.

Accessing a value through an uninitialized pointer address can cause inexplicable or erratic application behavior, because the data is not being read from or written to the intended location. For example, suppose that var1 is mistakenly written to the address specified by an uninitialized pointer and that then var2 is written there. When var1 is read back, it does not have its original value, having been replaced by var2.

4.14.3 Handling Register Variables

A register variable is stored in one of the microprocessor's registers, rather than in random-access memory (RAM). This speeds up access to the variable.

A conventional variable can become a register variable in either of the following ways:

- The variable is declared as a register variable. If a register is free, the compiler stores the variable there.
- The compiler stores a frequently used variable (such as a loop variable) in a register during optimization to speed up execution.

Register variables can cause problems during debugging. As with local variables, they are visible only within the function where they are defined. In addition, a register variable may not always be displayed with its current value.

Usually, it is a good idea to turn off all optimization and to avoid declaring register variables until the application has been fully debugged. Any side effects produced by optimization or register variables can then be easily isolated.

4.14.4 Redirecting CodeView for Windows Input and Output

You can cause CVW to receive input from an input file and generate output to an output file. To redirect CVW input and output, you can use the **/c** option on a command line of the following form to start CVW:

cvw /c "<*infile*; **t** >*outfile*"

When you redirect input in this way, CVW carries out any commands in *infile* during startup. When CVW exhausts all commands in the input file, focus automatically shifts to the Command window.

When you redirect output, it is sent to both *outfile* and the Command window. You can use the **t** parameter before the right angle bracket (>) on the command line to send output to the Command window. You can also redirect output from the command line after CVW has started.

Redirection is a useful way to automate CVW startup. Although redirection makes it possible for you to keep a viewable record of command-line input and output, you cannot record mouse operations. Some applications—particularly interactive ones—may need modification to allow for redirection of input to the application itself.

4.15 Modifying the TOOLS.INI File

To customize the behavior and user interface of CVW, modify the [cvw] section of your TOOLS.INI file. The TOOLS.INI file is an ASCII text file. You should place it in a directory pointed to by the INIT environment variable. (If you do not use the INIT environment variable, CVW looks for TOOLS.INI only in the CVW source directory.)

Most TOOLS.INI customizations control screen colors, but you can also specify startup commands or the name of the file that receives CVW output. The Help system contains complete information about all the TOOLS.INI entries for CVW.

4.16 Related Topics

For more information about CVW commands, consult CVW Help.

For an introduction to programming Windows applications, see the *Microsoft Windows Guide to Programming*.

For more information about Windows debugging messages, see Appendix C, "Windows Debugging Version."

Advanced Debugging: 80386 Debugger

Chapter 5

5.1 Preparing Symbol Files for 80386 Debugger 113
5.2 Starting 80386 Debugger 114
5.3 Entering 80386 Debugger 116
5.4 Command Syntax 118
5.4.1 Command Keys 118
5.4.2 Command Parameters 118
5.4.3 Binary and Unary Operators 121
5.4.4 Regular Expressions 122
5.5 Common Commands 123
5.6 Reference of 80386 Debugger Commands 125
5.7 Related Topics 170

Microsoft Windows 80386 Debugger (WDEB386.EXE) is used to test and debug Windows applications and dynamic-link libraries (DLLs) running with the Microsoft Windows operating system in standard or 386 enhanced mode. With 80386 Debugger commands, you can inspect and manipulate test code and environment status, install breakpoints, and perform other debugging operations.

Although 80386 Debugger offers debugging features not available in CodeView for Windows (CVW), 80386 Debugger lacks the convenient window interface of CVW and does not provide source-level debugging.

To use 80386 Debugger, you must have a serial terminal connected to the computer on which you are running the debugger and test application.

The terminal connection requirements are described in Section 5.2, "Starting 80386 Debugger."

This chapter describes the following:

- Preparing symbol files for 80386 Debugger
- Starting 80386 Debugger
- Entering 80386 Debugger
- Command format for 80386 Debugger
- List of common commands

It also contains a reference of 80386 Debugger commands.

5.1 Preparing Symbol Files for 80386 Debugger

To prepare application symbol files, perform the following steps:

1. Compile your C-language source files, using the appropriate command-line option to generate object files with line-number information for use by 80386 Debugger. For more information about compiler options, see the documentation that accompanied your compiler.
2. Link the compiled code with the standard Windows libraries, using the appropriate command-line option to prepare a symbol map (.MAP) file that includes **PUBLIC** symbols. The map file is required by Microsoft Symbol File Generator (MAPSYM).

   You may also want to use the linker option for display of line-number information. For more information about linker options, see the documentation that accompanied your linker.
3. Run MAPSYM to create a symbol file for symbolic debugging. MAPSYM converts the contents of your application's symbol map (.MAP) file into a form

suitable for loading with 80386 Debugger; then MAPSYM copies the result to a symbol (.SYM) file.

Following is the command-line syntax for MAPSYM:

mapsym [**/l**][**/n**] *mapfilename*

/l
Directs MAPSYM to display information on the screen about the conversion. The information includes the names of groups defined in the application, the application start address, the number of segments, and the number of symbols per segment.

/n
Directs MAPSYM to ignore line-number information in the map file. The resulting symbol file contains no line-number information.

mapfilename
Specifies the filename for a symbol map file that was created during linking. If you do not give a filename extension, .MAP is assumed. If you do not give a full path, the current directory and drive are assumed. MAPSYM creates a new symbol file having the same name as the map file but with the .SYM extension.

In the following example, MAPSYM uses the symbol information in FILE.MAP to create FILE.SYM in the current directory on the current drive:

```
mapsym /l file.map
```

Information about the conversion is sent to the screen.

Note MAPSYM always places the new symbol file in the current directory on the current drive.

MAPSYM can process up to 10,000 symbols for each segment in the application and up to 1024 segments.

5.2 Starting 80386 Debugger

A three-wire null modem cable is the minimum cable requirement for the serial terminal. In a three-wire null modem cable, the TxD (transmit data) and RxD (receive data) lines are in opposite positions at the two ends of the cable, but the signal ground is connected straight through.

The command-line syntax is as follows:

wdeb386 [**/C:***comport*] [**/D:"***commands***"**] [**/F:***filename*] [**/N**] [**/T:***hhhh*] [**/S:***symfile*] [**/V**[**P**]] [**X**] *winfile* [*parameters*]

Following are the command-line options and parameters:

/C:*comport*
Specifies a COM port for debugger output. If this option is not specified, 80386 Debugger checks first for COM2. If COM2 is not found, the debugger then checks for COM1. If neither COM1 nor COM2 exists, the debugger checks for any other COM port in the read-only memory (ROM) data area (40:0). A three-wire null modem cable is all that is needed for terminal connection; no DTR (data-terminal-ready) and CTS (clear-to-send) handshaking is used.

/D:"*commands*"
Carries out the 80386 Debugger command line specified by the string enclosed in quotation marks. Spaces, semicolons (;), and other punctuation can be included in the command string. To use a single quote (') on the command line, use double quotation marks (") before and after the single quotation mark.

The commands specified in this option are carried out after symbols are loaded. This means you can set breakpoints in code even before the code has been loaded. Before a segment or module has been loaded or defined, breakpoints can be set on the logical address (a combination of map number and group number) until the segment or module is defined, at which point the breakpoint turns into a real breakpoint.

/F: *filename*
Specifies a file containing command-line options for 80386 Debugger. Maximum file size is 4K, and the input file cannot contain the **/F** option.

/N
Sets the following options:

dislwr
codebytes
symaddrs
int3line
newvec
newreg
newprompt

For information about these options, see the **y** command in Section 5.6, "Reference of 80386 Debugger Commands."

/S: *symfile*
Specifies a symbol file to be loaded. This option can be repeated to load more than one symbol file. If the symbol files are not in your current directory, you must supply a full path, because 80386 Debugger does not use the PATH environment variable to locate any of the files supplied on the command line.

When memory is low, you can use more symbol files by running 80386 Debugger in the Windows directory and specifying the full path of WIN386.EXE (such as \WINDOWS\SYSTEM\WIN386.EXE) instead of WIN.COM.

/T:*hhhh*
Sets the port number for the timing card. (The default number is 250h.)

/V
Enables verbose mode, which displays messages indicating which segments are being being loaded. This option displays the messages for both Windows in 386 enhanced mode and Windows applications.

/VP
Enables verbose mode, which displays messages indicating which segments are being loaded. This option displays the messages for applications only.

/X
Causes symbols to be loaded into Extended Memory Specification (XMS) memory. This option has no effect with Windows version 3.1.

winfile
Specifes the Windows application to run under 80386 Debugger control. You will usually specify WIN.COM.

parameters
Specifies any parameters to be passed to the application.

Note The length of the command line cannot exceed 128 characters.

Following are two examples of valid commands:

```
wdeb386 /V /S:\windows\system\krnl286.sym /S:myapp.sym \windows\win.com /s myapp

wdeb386 /C:1 /S:krnl386.sym /s:user.sym /S:\myapp\myapp.sym \windows\win.com /3
myapp
```

You can start 80386 Debugger as a device driver by placing the following line in your CONFIG.SYS file:

```
device=c:\windev\wdeb386.exe
```

You must specify the full path to the WDEB386.EXE file. You can specify any command-line options on the line with device= (for example, you can load symbol files).

5.3 Entering 80386 Debugger

To enter 80386 Debugger at any time interrupts are not disabled, press the CTRL+C key combination on the debugging terminal. A nonmaskable interrupt (NMI) can be used to enter the debugger even when interrupts are disabled.

An **int 3** instruction or a call to the Windows **DebugBreak** function passes control to the 80386 Debugger.

When a Windows application running in standard or 386 enhanced mode attempts to read or write memory with a bad selector, beyond a selector limit, or with a selector set to 0, a general protection (GP) fault occurs.

In such cases, Windows displays a dialog box notifying the user of a problem. When 80386 Debugger is loaded, the dialog box has a Cancel button. If the user chooses the Cancel button, Windows passes control to the debugger at the instruction that caused the fault with a display of the following form:

```
GENERAL PROTECTION VIOLATION
AX=00000000 BX=00002136 CX=06040079 DX=00001EF5 SI=000000C3 DI=00002283
IP=00000028 SP=80012126 BP=0000212C CR2=80501FFC CR3=0293 IOPL=0 F=-- --
CS=0915 SS=091D DS=091D ES=0000 FS=0000 GS=0000 -- NV UP EI PLZR NA PE
NC
00AD:00000FA0  MOV BX, WORD PTR ES:[BX]
ES:65DF=INV:0003#
```

For more information about commands shown in the remaining examples in this section, see Section 5.4.2, "Command Parameters," and Section 5.6, "Reference of 80386 Debugger Commands."

You can determine the cause of the GP fault by looking at the value and the limit of the selector. To dump the local descriptor table (LDT) entry, you can use a command of the following form:

dl *selector*

The ability to continue execution depends on the cause of the fault. If the fault was caused by reading or writing beyond the selector limit, it may be possible to skip the instruction by incrementing the IP register.

To determine how many bytes the instruction contains, you may need to display the actual code bytes when disassembling the instruction. To do this, use the following commands:

```
y codebytes
r
```

If the fault is caused by a critical logic error, such as trying to use a selector for a temporary variable, there probably is no way to continue execution of the application. You may need to restart the computer.

5.4 Command Syntax

To enter 80386 Debugger commands, you use a debugging terminal rather than your computer's keyboard.

Commands and parameters are not case-sensitive.

If a syntax error occurs in a debugger command, 80386 Debugger redisplays the command line and indicates the error with a caret (^) and the word Error, as in the following example:

```
A100
 ^ Error
```

5.4.1 Command Keys

Following are the command keys:

Key	Action
CTRL+A	Repeats the previous command.
CTRL+C	Halts 80386 Debugger output, and returns to the debugger prompt.
CTRL+S	Freezes an 80386 Debugger display.
CTRL+Q	Restarts the display.

If the target system is executing code, CTRL+S and CTRL+Q are ignored.

5.4.2 Command Parameters

You can separate 80386 Debugger command parameters with delimiters (spaces or commas), but a delimiter is required only between two consecutive hexadecimal values. The following commands are equivalent:

```
dCS:100 110
d CS:100 110
d,CS:100,110
```

Following are the parameters you can use with 80386 Debugger commands:

addr

Represents an address parameter in one of four forms. For more information about the operators shown in the following address forms, see Section 5.4.3, "Binary and Unary Operators."

Address	Mode
#1f:02C0	Protected-mode address (selector:offset)
%31020	Linear address
%%31020	Physical address
&0100:02FF	Real-mode address (segment:offset)

Any of these specified address forms overrides the current address type.

byte
Specifies a two-digit hexadecimal value.

cmds
Specifies an optional set of debugger commands to be executed with the **bp** (Set Breakpoint) or **j** (Conditionally Execute) command.

count
Specifies a count. Valid values depend on the command with which this parameter is being used.

dword
Represents an eight-digit (4-byte) hexadecimal value. The **DWORD** data type is most commonly used as a physical address.

expr
Represents a combination of parameters and operators that evaluates to an 8-bit, 16-bit, or 32-bit value. An *expr* parameter can be used as a value in any command. An *expr* parameter can combine any symbol, number, or address with any of the binary and unary operators.

flags
Specifies one or more conditions. Valid conditions depend on the command with which this parameter is being used.

group-name
Specifies the name of a group that contains the map symbols you want to display.

list
Specifies a series of byte values or a string. The *list* parameter must be the last parameter on the command line. Following is an example of the **f** (Fill) command with a *list* parameter:

```
fCS:100 42 45 52 54 41
```

map-name
Specifies the name of a symbol map file.

name-chars
Specifies one or more characters.

number
Specifies a numeric value. Valid values depend on the command with which this parameter is being used.

object
Specifies a handle, a selector, or (in 386 enhanced mode) a heap address.

option
Specifies an option. Valid options depend on the command with which this parameter is being used.

range
Specifies the block of memory on which the command should operate. The *range* parameter can be two addresses (*addr addr*); or it can be one address and a length (*addr* **L** *word*, where *word* is the number of items on which the command should operate; 80h is the default value). Following are three valid examples:

```
CS:100 110
CS:100 L 10
CS:100
```

The limit for *range* is 10000h. To specify a word of 10000h using only four digits, use 0000h or 0h.

reg
Specifies the name of a microprocessor register.

string
Represents any number of characters enclosed in single quotation marks (') or double quotation marks ("). For quotation marks that must appear within *string*, you must use two sets of quotation marks. For example, the following strings are valid:

```
'This ''string'' is OK.'
\"This \"\"string\"\" is OK.\"
```

However, the following strings are not valid:

```
\"This \"string\" is not OK.\"
\"This 'string' is not OK.\"
```

The ASCII values of the characters in the string are used as a list of byte values.

word
Specifies a four-digit (2-byte) hexadecimal value.

5.4.3 Binary and Unary Operators

Following, in descending order of precedence, are the binary operators that can be used in 80386 Debugger commands:

Operator	Meaning
( )	Parentheses
:	Address binder
*	Multiplication
/	Integer division
MOD	Modulus (remainder)
+	Addition
–	Subtraction
>	Greater-than relational operator
<	Less-than relational operator
>=	Greater-than/equal-to relational operator
<=	Less-than/equal-to relational operator
==	Equal-to relational operator
!=	Not-equal-to relational operator
AND	Bitwise Boolean AND
XOR	Bitwise Boolean exclusive OR
OR	Bitwise Boolean OR
&&	Logical AND
||	Logical OR

Following, in descending order of precedence, are the unary operators that can be used in 80386 Debugger commands:

Operator	Meaning
&(*seg*)	Address of segment value
#(*sel*)	Address of selector value
%%(*phy*)	Address as a physical value
%(*lin*)	Address as a linear value
–	Two's complement
!	Logical NOT operator
NOT	One's complement
SEG	Segment address of operand
OFF	Address offset of operand
BY	Low-order byte from given address
WO	Low-order word from given address

Operator	Meaning
DW	Doubleword from given address
POI	Pointer (4 bytes) from given address—this operator works only with 16:16 addresses
PORT	1 byte from given port
WPORT	Word from given port

5.4.4 Regular Expressions

The set of regular expressions that 80386 Debugger supports for matching symbols is similar to the set supported by UNIX grep. The 80386 Debugger set includes a few enhancements.

Following are the 80386 Debugger wildcards:

Wildcard	Description
.	Matches any single character.
[]	Defines a character class; matches a set or range of characters.
	Negates a character class.

Following are the 80386 Debugger postfix operators:

Operator	Description
*	Causes the previous wildcard or single character to match zero or more characters.
#	Matches zero or one.
+	Plus sign, matches one or more.

Anywhere a symbol is accepted, a regular expression can be used. If there is more than one match, a list of matching symbols is displayed and you must select the proper symbol. The symbol match is not case-sensitive.

The asterisk (*), number sign (#), and plus sign (+) are already math expression operators. To be recognized as a regular expression operator, each of these characters must be immediately preceded by an escape character—the backslash (\). The period (.), opening bracket ([), and closing bracket (]) do not require escape characters. Anything inside the brackets of a character class does not have to be escaped. Following are valid character classes:

```
[a-z]
[;*+#]
```

Characters are escaped at two levels: in the expression evaluator and in the regular expression parser. A character special to the expression evaluator (*, #, +, or \)

must be escaped to make it to the regular expression parser. If a character special to the regular expression parser must be escaped (for example, to match symbols with * or # in them), it must be escaped twice. If a backslash is needed in an expression, it must be double escaped.

Following are sample regular expressions:

Regular expression	Description
`sym.*`	Matches any symbols beginning with the string sym.
`sym*`	Matches sym alone and sym followed by any characters.
`.*sym.*`	Matches any symbols containing the string sym.
`sym[0-9]`	Matches sym0, sym1, sym2, and so on.
`sym*`	Matches sym*.
`sym\\\\`	Matches sym\.
`sym\\\\.*`	Matches any symbols beginning with the string sym\.

5.5 Common Commands

This section documents the commands available in all environments in which you can use 80386 Debugger. A command that begins with a period (.) is called a dot command.

Command	Description
?	Display expression, or display help menu.
.?	Display external commands.
.b	Set baud rate for COM port.
.df	Display global free list.
.dg	Display global heap.
.dh	Display local heap.
.dm	Display global module list.
.dq	Dump task queues.
.du	Display list of least recently used (LRU) global memory objects.
.reboot	Restart target system.
bc	Clear breakpoint.
bd	Disable breakpoint.
be	Enable breakpoint.
bl	List breakpoints.
bp	Set breakpoint.
br	Set breakpoint on debug register.
c	Compare memory locations.

Command	Description
d	Display memory.
db	Display bytes.
dd	Display doublewords.
dg	Display global descriptor table (GDT).
di	Display interrupt descriptor table (IDT).
dl	Display local descriptor table (LDT).
dp	Display page directory and page tables.
dt	Display task state segment (TSS).
dw	Display words.
dx	Dump loadall buffer.
e	Enter byte.
f	Fill memory.
g	Go.
h	Perform hexadecimal arithmetic.
i	Display 1 byte of input.
j	Conditionally execute command.
k	Display current stack frame.
ka	Set backtrace argument.
kt	Display stack frame of task.
la	List absolute symbols.
lg	List groups.
lm	List maps.
ln	List nearest symbol.
ls	List symbols.
m	Move memory.
o	Write output to a port.
p	Execute instruction, returning from any call or interrupt.
r	Display register.
s	Search for a byte.
t	Execute instruction.
u	Disassemble bytes.
v	Display debugger version.
vc	Clear interrupt vector.
vl	List debugger interrupt vectors.
vo	List debugger interrupt vectors in specified format.
vs	Add debugger interrupt vector (not at ring 0).
vt	Add debugger interrupt vector.
w	Change active map list.

Command	Description
wa	Add map to active list.
wr	Remove map from active list.
y	Change debugger configuration.
z	Zap embedded **int 1** or **int 3** instruction.
zd	Execute default command string.
zl	Display default command string.
zs	Change default command string.

5.6 Reference of 80386 Debugger Commands

This section describes the common commands that can be used with 80386 Debugger.

?

? [[*option*.]*expr*]
? ["*string*", *expr*, *expr*, [...]]

The **?** command evaluates an expression and displays the result.

The **?** command with no arguments displays a list of commands and syntax recognized by the debugger.

Parameters

option

Specifies the format in which to display the expression specified by *expr*. The *option* parameter can be one of the following characters:

Character	Format
h	Hexadecimal
d	Decimal
t	Decimal
o	Octal
q	Octal
y	Binary

If *option* is given, a period (.) must be used to separate *option* and *expr*. If *option* is not given, the command displays all formats, an ASCII character representation, and whether the expression is TRUE or FALSE.

expr

Specifies an expression consisting of one or more addresses, numbers, and operators. The operators in the expression can be any of the 80386 Debugger operators listed in Section 5.4.3, "Binary and Unary Operators." The addresses in the expression can be 32-bit physical addresses or protected-mode addresses (selector:offset). The number sign (#) operator overrides the current address type.

string

Specifies a **printf** formatting string. Supported **printf** format characters are as follows:

Format character	Meaning
%%	%
%c	Character
%[-][+][ ][0][*width*][.*precision*][p][n]d	Decimal
%[-][0][*width*][.*precision*][p][n]u	Unsigned decimal
%[-][#][0][*width*][.*precision*][p][n]x	Hexadecimal
%[-][#][0][*width*][.*precision*][p][n]X	Hexadecimal
%[-][0][*width*][.*precision*][p][n]o	Octal
%[-][0][*width*][.*precision*][p][n]b	Binary
%[-][*width*][.*precision*][a]s	String
%[-][*width*][.*precision*][a][p][n][L][H][N]S	Symbol
%[-][*width*][.*precision*][a][p][n][L][H][N]G	Group:symbol
%[-][*width*][.*precision*][a][p][n][L][H][N]M	Map:group:symbol
%[-][*width*][.*precision*][a][p][n][L][H][N]A	Address

Specifying an asterisk (*) for the *width* or *precision* parameter causes the field width or precision, respectively, to be picked up from the next parameter. Decimal values can also be specified for the *width* and *precision* parameters.

The following escape sequences are supported:

Escape sequence	Description
\a	Alert (bell) character
\b	Backspace
\n	New line
\r	Carriage return
\t	Horizontal tab

The following table describes the optional prefixes:

Prefix	Format character(s)	Meaning
a	s,S,G,M,A	Address argument size
H	S,G,M,A	16-bit offset
L	S,G,M,A	32-bit offset
N	S,G,M,A	Offset only
p	S,G,M	Get the previous symbol
n	S,G,M	Get the next symbol
p	A	Get the previous symbol address
n	A	Get the next symbol address
p	d,u,x,X,o,b	Get the previous symbol offset
n	d,u,x,X,o,b	Get the next symbol offset

Example

The following example looks up the physical address of selector 1Fh in the current local descriptor table (LDT) and adds 220h to it:

```
?%(#001F:0220)
```

The following example displays the value of the expression DS:SI + BX:

```
? ds:si+bx
```

The debugger returns a display similar to the following:

```
987A:000001B3 %00098953 %%00098953
```

The following example displays the value of the arithmetic expression 3*4:

```
? 3*4
```

The debugger returns the following display:

```
0Ch 12T 14Q 00001100Y '.' TRUE
```

.?

.?

The **.?** command displays a list of external commands. These commands are part of 80386 Debugger, but they are specific to the environment in which the debugger is running.

.b

.b *number* [*addr*]

The **.b** command sets the baud rate for the debugging port (COM2).

Parameters

number
Specifies the baud rate. It can be one of the following values: 150, 300, 600, 1200, 2400, 4800, 9600, or 19200. Because the default radix for the debugger is 16, you must type **t** after the number to indicate a decimal value.

addr
Specifies 1 for COM1 or 2 for COM2; anything else is taken as a base port address. If there is no COM2, 80386 Debugger checks for COM1 and then for any other COM port address in the read-only memory (ROM) data area to use as the console.

Example

The following example sets the baud rate to 1200:

```
#.b 1200t
```

.df

.df

The **.df** command displays a list of the free global memory objects in the global heap.

The list has the following form:

address: *size owner* [*chain*]

address
Specifies the selector of the memory in standard mode. In 386 enhanced mode, the *address* field specifies physical and heap addresses.

size
Specifies the size, in paragraphs (multiples of 16 bytes), of the object in standard mode. In 386 enhanced mode, the *size* field specifies the size of the object, in bytes.

owner
Always specifies that the module is free.

chain
Specifies the previous and next addresses in the list of least recently used (LRU) objects. 80386 Debugger displays the addresses only if the segment is movable and discardable.

.dg

.dg [*object*]

The **.dg** command displays a list of the global memory objects in the global heap.

Parameters

object
Specifies the first object to be listed. The *object* parameter can be a handle, a selector, or (in 386 enhanced mode) a heap address.

The list has the following form:

address: *size segment-type owner* [*handle flags chain*]

address
Specifies the selector of the memory in standard mode. In 386 enhanced mode, the *address* field specifies physical and heap addresses.

size
Specifies the size, in paragraphs (multiples of 16 bytes), of the object in standard mode. In 386 enhanced mode, the *size* field specifies the size of the object, in bytes.

segment-type

Specifies the type of object. The type can be any one of the following:

Segment type	Meaning
CODE	Segment contains application code.
DATA	Segment contains application data and possible stack and local heap data.
FREE	Segment belongs to pool of free memory objects ready for allocation by an application.
PRIV	Segment contains private data.
SENTINAL	Segment marks the beginning or end of the global heap.

owner

Specifies the module name of the application or library that allocated the memory object. The acronym PDB is used for memory objects that represent process descriptor blocks. These blocks contain execution information about applications.

handle

Specifies the handle of the global memory object. If 80386 Debugger displays no handle, the segment is fixed.

flags

Specifies either of the following:

Flag	Meaning
D	The segment is movable and discardable.
L	The segment is locked. If the segment is locked, the lock count appears to the right of the flag.

If 80386 Debugger displays a handle but no flag, the segment is movable but not discardable.

chain

Specifies the previous and next addresses in the list of least recently used (LRU) objects. Addresses are displayed only if the segment is movable and discardable (specified by the D flag).

.dh

.dh

The **.dh** command displays a list of the local memory objects in the local heap (if any) belonging to the current data segment. The command uses the current value of the DS register to locate the data segment and check for a local heap.

The list of memory objects has the following form:

offset: *size* { **BUSY** | **FREE** }

offset
: Specifies the address offset from the beginning of the data segment to the local memory object.

size
: Specifies the size of the object, in bytes.

If **BUSY** is displayed, the object has been allocated and is currently in use. If **FREE** is displayed, the object is in the pool of free objects ready to be allocated by the application. A special memory object, **SENTINAL**, may also be displayed.

.dm

.dm

The **.dm** command displays a list of the global modules in the global heap.

The list has the following form:

module-handle module-type module-name filename

module-handle
: Specifies the handle of the module.

module-type
: Specifies either a dynamic-link library (DLL) or the name of the application you are debugging.

module-name
: Specifies the name of the module.

filename
: Specifies the name of the file from which you loaded the application.

.dq

.dq

The **.dq** command displays a list containing information about the various task queues supported by the system.

The list has the following form:

*task-descriptor-block stack-segment***:***stack-pointer number-of-events priority internal-messaging-information module*

task-descriptor-block
Specifies the selector or segment address.

The task descriptor block is identical to the process descriptor block (PDB).

*stack-segment***:***stack-pointer*
Specifies the stack segment and pointer.

number-of-events
Specifies the number of events waiting for the segment.

priority
Specifies the priority of the segment.

internal-messaging-information
Specifies information about internal messages.

module
Specifies the module name.

.du

.du

The **.du** command displays a list of the least recently used (LRU) global memory objects in the global heap.

The list has the following form:

*address***:** *size segment-type owner* [*handle flags chain*]

address
Specifies the selector of the memory in standard mode. In 386 enhanced mode, the *address* field specifies physical and heap addresses.

size
Specifies the size, in paragraphs (multiples of 16 bytes), of the object in standard mode. In 386 enhanced mode, the *size* field specifies the size of the object, in bytes.

segment-type
Specifies the type of object. The type can be any one of the following:

Segment type	Meaning
CODE	Segment contains application code.
DATA	Segment contains application data and possible stack and local heap data.
FREE	Segment belongs to pool of free memory objects ready for allocation by an application.
PRIV	Segment contains private data.
SENTINAL	Segment marks the beginning or end of the global heap.

owner
Specifies the module name of the application or library that allocated the memory object. The acronym PDB is used for memory objects that represent process descriptor blocks. These blocks contain execution information about applications.

handle
Specifies the handle of the global memory object.

flags
Specifies D, which means the segment is movable and discardable.

chain
Specifies the previous and next addresses in the LRU list.

.reboot

.reboot

The **.reboot** command causes the target system to restart.

bc

bc *list* | *

The **bc** command removes one or more defined breakpoints.

Parameters

list
Specifies any combination of integer values in the range 0 through 9. If you specify *list*, the debugger removes the specified breakpoints.

*
Clears all breakpoints.

Example

The following example removes breakpoints 0, 4, and 8:

```
bc 0 4 8
```

The following example removes all breakpoints:

```
bc *
```

---

bd

bd *list* | *

The **bd** command temporarily disables one or more breakpoints. To restore breakpoints disabled by the **bd** command, use the **be** (Enable Breakpoints) command.

Parameters

list
Specifies any combination of integer values in the range 0 through 9. If you specify *list*, the debugger disables the specified breakpoints.

*
Disables all breakpoints.

Example

The following example disables breakpoints 0, 4, and 8:

```
bd 0 4 8
```

The following example disables all breakpoints:

```
bd *
```

be

be *list* | *

The **be** command restores (enables) one or more breakpoints that have been temporarily disabled by a **bd** (Disable Breakpoints) command.

Parameters

list
Specifies any combination of integer values in the range 0 through 9. If you specify *list*, the debugger enables the specified breakpoints.

*
Enables all breakpoints.

Example

The following example enables breakpoints 0, 4, and 8:

```
be 0 4 8
```

The following example enables all breakpoints:

```
be *
```

---

bl

bl

The **bl** command lists current information about all breakpoints created by the **bp** (Set Breakpoints) command.

Example

If no breakpoints are currently defined, the debugger displays nothing. Otherwise, the breakpoint number, enabled status, breakpoint address, number of passes remaining, initial number of passes (in parentheses), and any optional debugger commands to be executed when the breakpoint is reached are displayed on the screen, as in the following example:

```
0 e 04BA:0100
4 d 04BA:0503 4 (10)
8 e 0D2D:0001 3 (3) "R;DB DS:SI"
9 e xxxx:0012
```

In this example, breakpoints 0 and 8 are enabled (e) and 4 is disabled (d). Breakpoint 4 had an initial pass count of 10h and has four remaining passes to be taken before the breakpoint. Breakpoint 8 had an initial pass count of 3 and must make

all three passes before it halts execution and forces the debugger to execute the optional debugger commands enclosed in quotation marks. Breakpoint 0 shows no initial pass count, which means it was set to 1. Breakpoint 9 shows a virtual breakpoint (a breakpoint set in a segment that has not been loaded into memory).

bp

bp[*number*]*addr* [*count*] ["*cmds*"]

The **bp** command creates a software breakpoint at an address. When the application is running, software breakpoints stop execution and force the debugger to execute the default or optional command string. Unlike breakpoints created by the **g** (Go) command, software breakpoints remain in memory until you remove them with the **bc** (Clear Breakpoints) command or temporarily disable them with the **bd** (Disable Breakpoints) command.

The debugger allows up to 10 software breakpoints (0 through 9). If you specify more than 10 breakpoints, the debugger returns the following message:

```
Too Many Breakpoints
```

The *addr* parameter is required for all new breakpoints.

Parameters

number
Specifies which breakpoint is being created. No space is allowed between the **bp** and *number*. If *number* is omitted, the first available breakpoint number is used.

addr
Specifies any valid instruction address—the first byte of an operation code (opcode).

count
Specifies the number of times the breakpoint is to be ignored before being executed. It can be any 16-bit value.

cmds
Specifies an optional list of debugger commands to be executed in place of the default command when the breakpoint is reached. You must enclose optional commands in quotation marks and separate optional commands with semicolons (;).

Example

The following example creates a breakpoint at address CS:123:

```
bp 123
```

The following example creates breakpoint 8 at address 400:23 and executes a **db** (Display Bytes) command:

```
bp8 400:23 "db DS:SI"
```

The following example creates a breakpoint at address 100 in the current CS selector and displays the registers before comparing a block of memory. The breakpoint is ignored 16 (10h) times before being executed.

```
bp 100 10 "r;c100 L 100 300"
```

---

br

br[*number*] *flags* [*count*] ["*cmds*"]

The **br** command sets an 80386 debug register breakpoint. Debug registers can be used to break on data reads and writes and instruction execution. Up to four debug registers can be set and enabled at one time.

Parameters

number
Specifies which breakpoint is being created. No space is allowed between the **br** command and the *number* parameter. If *number* is omitted, the first available breakpoint number is used.

flags
Specifies the length and break conditions for the breakpoint. This parameter can be some combination of the following values:

Value	Meaning
1	Set 1-byte length (default value).
2	Set word length on word boundary.
4	Set doubleword length on doubleword boundary.
E	Break on instruction execution only (1-byte length only).
W	Break on writes only.
R	Break on reads and writes.

count
Specifies the number of times the breakpoint is to be ignored before being executed. It can be any 16-bit value.

cmds
Specifies an optional list of debugger commands to be executed in place of the default command when the breakpoint is reached. You must enclose the group of optional commands in quotation marks and separate optional commands with semicolons (;).

C

c *range addr*

The **c** command compares one memory location with another memory location.

If the two memory areas are identical, the debugger displays nothing and returns the debugger prompt. Differences, when they exist, are displayed in the following form:

addr1 byte1 byte2 addr2

Parameters

range
Specifies the block of memory that is to be compared with a block of memory starting at *addr*.

addr
Specifies the starting address of the second block of memory.

Example

This section shows two forms of the **c** command that have the same effect. Each compares the block of memory from 100h to 1FFh with the block of memory from 300h to 3FFh.

The first example specifies a *range* with a starting address of 100h and an ending address of 1FFh. This block of memory is compared with a block of memory of the same size starting at 300h.

```
c100 1FF 300
```

The second example compares the same block of memory but specifies the *range* by using the **L** (length) option.

```
c100 L 100 300
```

d

d [*range*]

The **d** command displays the contents of memory at a given address or in a range of addresses. The **d** command displays one or more lines, depending on the *range* given. Each line displays the address of the first item displayed. The command always displays at least one value. The memory display is in the format defined by a previously executed **db** (Display Bytes), **dd** (Display Doublewords), or **dw** (Display Words) command. Each subsequent **d** (typed without parameters) displays the bytes immediately following those last displayed.

Parameters

range
Specifies the block of memory to display. If you omit *range*, the **d** command displays the next byte of memory after the last one displayed. The **d** command must be separated by at least one space from any *range* value.

Example

The following example displays 20h bytes at CS:100:

```
d CS:100 L 20
```

The following example displays all the bytes in the range 100h to 115h in the CS selector:

```
d CS:100 115
```

db

db [*range*]

The **db** command displays the values of the bytes at a given address or in a given range.

The display is in two portions: a hexadecimal display (each byte is shown in hexadecimal format) and an ASCII display (the bytes are shown as ASCII characters). A nonprinting character is denoted by a period (.) in the ASCII portion of the display. Each display line shows 16 bytes, with a hyphen between the eighth and ninth bytes. Each displayed line begins on a 16-byte boundary.

Parameters

range
Specifies the block of memory to display. If you omit *range*, 128 bytes are displayed beginning at the first address after the address displayed by the previous **db** command.

Example

The following example displays 0Ah bytes of memory, beginning at the specified address:

```
db CS:100 0A
```

This example displays lines in a format similar to the following:

```
04BA:0100 54 4F 4D 20 53  . . . 45 52 TOM SAWYER
```

Each line of the display begins with an address, incremented by 10h from the address on the previous line.

dd

dd [*range*]

The **dd** command displays the hexadecimal values of the doublewords at the address specified or in the specified range of addresses.

The **dd** command displays one or more lines, depending on the *range* given. Each line displays the address of the first doubleword in the line, followed by up to four hexadecimal doubleword values. The hexadecimal values are separated by spaces. The **dd** command displays values up to the end of the *range* or until the first 32 doublewords have been displayed.

Typing **dd** displays 32 doublewords at the current dump address. For example, if the last byte in the previous **dd** command was 04BA:0110, the display starts at 04BA:0111.

Parameters

range
Specifies the block of memory to display. If you omit *range*, 32 doubleword values are displayed beginning at the first address after the address displayed by the previous **dd** command.

Example

The following example displays the doubleword values from CS:100 to CS:110:

```
dd CS:100 110
```

The resulting display is similar to the following:

```
04BA:0100 7473:2041 676E:6972 5405:0104 0A0D:7865
04BA:0110 0000:002E
```

No more than four values per line are displayed.

dg

dg[a] [*range*]

The **dg** command displays the specified range of entries in the global descriptor table (GDT).

Parameters

range
Specifies the range of entries in the GDT. If you omit *range*, the debugger displays the entire contents of the GDT.

a
Causes all entries in the table to be displayed, not just the valid entries. By default, only the valid GDT entries are displayed. If the command is passed a local descriptor table (LDT) selector, it displays the appropriate LDT entry.

Example

The following example displays only the valid entries from 0h to 40h in the GDT:

```
dg 0 40
```

The resulting display is similar to the following:

```
0008  Data Seg  Base=01D700 Limit=3677 DPL=0 Present ReadWriteAccessed
0010  TSS Desc  Base=007688 Limit=002B DPL=0 Present Busy
0018  Data Seg  Base=020D7A Limit=03FF DPL=0 Present ReadWrite
0020  Data Seg  Base=000000 Limit=03FF DPL=0 Present ReadWrite
0028  LDT Desc  Base=000000 Limit=0000 DPL=0 Present
0030  Data Seg  Base=000000 Limit=0000 DPL=0 Present ReadWrite
0040  Data Seg  Base=000400 Limit=03BF DPL=3 Present ReadWrite
```

di

di[**a**] [*range*]

The **di** command displays the specified range of entries in the interrupt descriptor table (IDT).

Parameters

a
Causes all entries in the table to be displayed, not just the valid ones. The default is to display just the valid IDT entries.

range
Specifies the range of entries to be displayed. If you omit *range*, the debugger displays all IDT entries.

Example

The following example displays the valid IDT entries in the range 0h through 10h:

```
di 0 10
```

The resulting display is similar to the following:

```
0000  Int Gate  Sel=1418    Offst=03D8 DPL=3 Present
0001  Int Gate  Sel=2D38    Offst=0049 DPL=3 Present
0002  Int Gate  Sel=1418    Offst=03E4 DPL=3 Present
0003  Int Gate  Sel=2D38    Offst=006F DPL=3 Present
0004  Int Gate  Sel=1418    Offst=0417 DPL=3 Present
0005  Int Gate  Sel=1418    Offst=041D DPL=3 Present
0006  Int Gate  Sel=1418    Offst=0423 DPL=3 Present
0007  Int Gate  Sel=2D38    Offst=00A3 DPL=3 Present
0008  Int Gate  Sel=1418    Offst=042F DPL=3 Present
0009  Int Gate  Sel=2D38    Offst=00CA DPL=3 Present
000A  Int Gate  Sel=2D38    Offst=00D3 DPL=3 Present
000B  Int Gate  Sel=2D38    Offst=0156 DPL=3 Present
000C  Int Gate  Sel=2D38    Offst=01A4 DPL=3 Present
000D  Int Gate  Sel=2D38    Offst=01C6 DPL=3 Present
```

dl

dl[a | p | s | h] [*range*]

The **dl** command displays the specified range of entries in the local descriptor table (LDT).

Parameters

a
Causes all entries in the table to be displayed, not just the valid ones. By default, only the valid LDT entries are displayed. If the command is passed a global descriptor table (GDT) selector, it displays the appropriate GDT entry.

p
Causes private segment selectors to be displayed.

s
Causes shared segment selectors to be displayed.

h
Causes huge segment selectors to be displayed. To display the huge segment selectors, give the shadow selector followed by the maximum number of selectors reserved for that segment plus 1.

range
Specifies the range of entries to be displayed. If you omit *range*, the entire table is displayed.

Example

The following example displays all the LDT entries:

```
dla 4 57
```

The command produces a display similar to the following:

```
0014  Call Gate Sel=1418    Offst=0417 DPL=0 NotPres WordCount=1D
001C  Code Seg  Base=051418 Limit=0423 DPL=0 NotPres ExecOnly
0027  Reserved  Base=87F000 Limit=FEA5 DPL=3 Present
0034  Code Seg  Base=05F000 Limit=1805 DPL=0 NotPres ExecOnly
003C  Code Seg  Base=05F000 Limit=EF57 DPL=0 NotPres ExecOnly
0047  Code Seg  Base=4DC000 Limit=0050 DPL=3 Present ExecOnly
004D  Reserved  Base=71F000 Limit=F841 DPL=1 NotPres
0057  Code Seg  Base=59F000 Limit=E739 DPL=3 Present ExecOnly
```

dp

dp[a|d] [*range*]

The **dp** command displays the page directory and page tables. Page tables are always skipped if the corresponding page directory entry is not present. Page directory entries appear with an asterisk next to the page frame.

Parameters

a
Displays all present page directory and page table entries; by default, page directory and page table entries that are zero are skipped.

d
Displays only page directory entries. If a count is given as part of the optional range, it will be interpreted as a page directory entry count.

range
Specifies the range of linear addresses for page tables.

Example

The following example displays the page directory and page table in the range 0 through 12h:

```
dp 0 12
```

The resulting display is similar to the following:

```
%00000000 *frame=00FCE  state=3  res=0  c A  pb1=0  pb0=0  U W P
%00000000  frame=00000  state=3  res=0  c u  pb1=0  pb0=0  U W P
%00001000  frame=00001  state=3  res=0  c u  pb1=0  pb0=0  U W P
```

The display produced by the **dp** command can contain flags that have the following meanings:

Bit set	Bit clear	Meaning
D	c	Dirty/clean
A	u	Accessed/unaccessed
U	s	User/supervisor
W	r	Writable/read-only
P	n	Present/not-present

dt

dt [*addr*]

The **dt** command displays the current task state segment (TSS) or the selected TSS if you specify the optional address.

Parameters

addr
Specifies the address of the TSS to display. If no *addr* is given, **dt** displays the current TSS pointed to by the TR register.

Example

The following example displays the current TSS:

```
dt
```

The resulting display is similar to the following:

```
AX=0000 BX=0000 CX=0000 DX=0000 SP=0000 BP=0000 SI=0000 DI=0000
IP=0000 CS=0000 DS=0000 ES=0000 SS=0000 NV UP DI PL NZ NAPO NC
SS0=0038  SP0=08DE  SS1=0000  SP1=0000  SS2=0000  SP2=0000
IOPL=0    LDTR=0028 LINK=0000
```

dw

dw [*range*]

The **dw** command displays the hexadecimal values of the words at a given address or in a given range of addresses.

The command displays one or more lines, depending on the *range* given. Each line displays the address of the first word in the line, followed by up to eight hexadecimal word values. The hexadecimal values are separated by spaces. The command displays values until the end of the *range* or until the first 64 words have been displayed.

Typing **dw** displays 64 words at the current dump address. For example, if the last word in the previous **dw** command was displayed at address 04BA:0110, the next display will start at 04BA:0112.

Parameters

range
Specifies the range of addresses to display. If you omit *range*, 64 words are displayed beginning at the first address after the address displayed by the previous **dw** command.

Example

The following example displays the word values from CS:100 to CS:110:

```
dw CS:100 110
```

The resulting display is similar to the following:

```
04BA:0100 2041 7473 6972 676E 0104 5404 7865 0A0D
04BA:0110 002E
```

e

e *addr* [*list*]

The **e** command enters byte values into memory at a specified address. You can specify the new values on the command line or let the debugger prompt you for values. If the debugger prompts you, it displays the address and its contents and then waits for you to perform one of the following actions:

- Replace a byte value with a value you type. Type the value after the current value. If the byte you type is an invalid hexadecimal value or contains more than two digits, the system does not echo the illegal or extra character.
- Press the SPACEBAR to advance to the next byte. To change the value, type the new value after the current value. If, when you press the SPACEBAR, you move beyond an 8-byte boundary, 80386 Debugger starts a new display line with the address displayed at the beginning.
- Type a hyphen (-) to return to the preceding byte. If you decide to change a byte before the current position, typing the hyphen returns the current position to the previous byte. When you type the hyphen, a new line is started with its address and byte value displayed.
- Press ENTER to terminate the **e** command. You can press ENTER at any byte position.

Parameters

addr
Specifies the address of the first byte to be entered.

list
Specifies the byte values used for replacement. These values are inserted automatically. If an error occurs when you are using the list form of the command, no byte values are changed.

Example

The following example prompts you to change the value EB at CS:100:

```
eCS:100
04BA:0100 EB.
```

To step through the subsequent bytes without changing values, press the SPACEBAR. In the following example, the SPACEBAR is pressed three times:

```
04BA:0100 EB.41  10. 00. BC.
```

To return to a value at a previous address, type a hyphen, as shown in the following example:

```
04BA:0100  EB.41    10. 00. BC.-
04BA:0102  00.-
04BA:0101  10.
```

This example returns to the address CS:101.

---

f

f *range list*

The **f** command fills the addresses in a specified range with the values in the specified list.

Parameters

range
Specifies the block of memory to be filled. If *range* contains more bytes than the number of values in *list*, the debugger uses *list* repeatedly until all bytes in *range* are filled. If any of the memory in *range* is not valid (bad or nonexistent), an error occurs in all succeeding locations.

list
Specifies the list of values to fill the given *range*. If *list* contains more values than the number of bytes in *range*, the debugger ignores the extra values in *list*.

Example

The following example fills memory locations 04BA:100 through 04BA:1FF with the bytes specified, repeating the five values until it has filled all 100h bytes:

```
f04BA:100 L 100 42 45 52 54 41
```

g

g[**s**|**h**|**t**|**z**] [=*addr* [*addr*[**...**]]]

The **g** command executes the application currently in memory. If you type the **g** command by itself, the current application runs as if it had been run outside the debugger. If you specify =*addr*, execution begins at the specified address.

Specifying an optional breakpoint address causes execution to halt at the first address encountered, regardless of the position of the address in the list of addresses that halts execution or application branching. When execution of the application reaches a breakpoint, the default command string is executed.

The stack (SS:SP) must be valid and have 6 bytes available for this command. The **g** command uses an **iret** instruction to cause a jump to the application being tested. The stack is set, and the user flags, CS register, and IP register are pushed on the user stack. (If the user stack is not valid or is too small, the operating system may crash.) An interrupt code (0CCh) is placed at the specified breakpoint addresses.

When the debugger encounters an instruction with the breakpoint code, it restores all breakpoint addresses listed with the **g** command to their original instructions. If you do not halt execution at one of the breakpoints, the interrupt codes are not replaced with the original instructions.

Parameters

s
Shows the time, in microseconds, from when the system is started with **gs** until the next entry to the debugger. No attempt is made to calculate and remove debugger overhead from the measurement. Requires a timing card.

h
Displays the approximate debugger overhead in the **s** option. Requires a timing card.

t or **z**
Allows trapped exceptions to resume at the original trap handler address without having to unhook the exception. Use these options instead of the **vcp d; t; vsp d** command.

=*addr*
Specifies the address at which execution is to begin. The equal sign (=) is needed to distinguish the starting address from the breakpoint address.

addr
Specifies one or more breakpoint addresses where execution is to halt. You can specify up to 10 breakpoints, but only at addresses containing the first byte of an operation code (opcode). If you attempt to set more than 10 breakpoints, an error message is displayed.

Example

The following example executes the application currently in memory until address 7550 in the CS selector is executed. The debugger then executes the default command string, removes the **int 3** trap from this address, and restores the original instruction. When you resume execution, the original instruction is executed.

```
gCS:7550
```

h

h *word word*

The **h** command performs hexadecimal arithmetic on the two specified parameters.

The debugger adds, subtracts, and multiplies the two parameters; divides the second parameter by the first; and then displays the results on one line. The debugger does 32-bit multiplication and displays the result as doublewords. The debugger displays the result of division as a 16-bit quotient and a 16-bit remainder.

Parameters

word
Specifies a 16-bit word parameter.

Example

The following example performs the calculations on 300h and 100h:

```
h 300 100
```

The resulting display is the following:

```
+0400  -0200 *0000 0003 /0003 0000
```

i

i *word*

The **i** command accepts and displays 1 byte from a specified port.

Parameters

word
Specifies the 16-bit port address.

Example

The following example displays the byte at port address 2F8h:

```
i2F8
```

---

j

j *expr* ["*cmds*"]

The **j** command executes the specified commands when the specified expression is TRUE. If *expr* is FALSE, the debugger continues to the next command line (excluding the commands in *cmds*).

The **j** command is useful in breakpoint commands to conditionally break execution when an expression becomes TRUE.

Parameters

expr
Evaluates to a Boolean TRUE or FALSE.

cmds
Specifies a list of debugger commands to be executed when *expr* is TRUE. The list must be enclosed in single or double quotation marks. You must separate optional commands with semicolons (;). Single commands do not require quotation marks.

Example

The following example causes execution to break if AX does not equal zero when the breakpoint is reached:

```
bp 167:1454 "J AX == 0;G"
```

The following example displays the registers and continues execution when the byte pointed to by DS:SI +3 is equal to 40h; otherwise, it displays the descriptor table:

```
bp 167:1462 "J BY (DS:SI+3) == 40 'R;G';DG DS"
```

k

k[b|s|v] [*addr*] [*addr*]

This command displays the current stack frame. Each line shows the name of a procedure, its arguments, and the address of the statement that called it. The command displays four 2-byte arguments by default. The **ka** command changes the number of arguments displayed by this command.

Using the **k** command at the beginning of a function (before the function prolog has been executed) gives incorrect results. The command uses the BP register to compute the current backtrace, and this register is not correctly set for a function until its prolog has been executed.

Parameters

b
Indicates the stack frame is 32 bits wide.

s
Indicates the stack frame is 16 bits wide.

v
Displays the verbose version of stack information—that is, information about stack location and frame pointer values for each frame.

addr
Specifies an optional stack-frame address (SS:BP) or an optional code address (CS:IP).

---

ka

ka *count*

The **ka** command sets the number of arguments displayed for all subsequent stack trace commands. The initial default value is 4.

Parameters

count
Specifies the number of arguments to be displayed. The *count* parameter must be in the range 0 through 1Fh.

kt

k[b|s|v]t [*addr*]

This command displays the stack frame of the current task or the task specified by the *addr* parameter. Each line shows the name of a procedure, its arguments, and the address of the statement that called it. The command displays four 2-byte arguments by default. The **ka** command changes the number of arguments displayed by this command.

Parameters

b
Indicates the stack frame is 32 bits wide.

s
Indicates the stack frame is 16 bits wide.

v
Displays the verbose version of stack information—that is, information about stack location and frame pointer values for each frame.

addr
Specifies the segment address of the process descriptor block (PDB) for the task to be traced. To obtain the *addr* value, use the **.dq** (Dump Task Queue) command. If *addr* is not supplied, the **kt** command displays the stack frame of the current task.

---

la

la

The **la** command lists the absolute symbols in the active map.

lg

lg

The **lg** command lists the selector (or segment) and the name of each group in the active map.

Example

The **lg** command produces a display similar to the following:

```
#0090:0000 DOSCODE
#0828:0000 DOSGROUP
#1290:0000 DBGCODE
#16C0:0000 DBGDATA
#1A38:0000 TASKCODE
#1AD8:0000 DOSRING3CODE
#1AE0:0000 DOSINITCODE
#2018:0000 DOSINITRMCODE
#20A8:0000 DOSINITDATA
#23F8:0000 DOSMTE
#2420:0000 DOSHIGHDATA
#28D0:0000 DOSHIGHCODE
#3628:0000 DOSHIGH2CODE
#0090:0000 DOSCODE
```

lm

lm

The **lm** command lists the symbol files currently loaded and indicates which one is active.

The last symbol file loaded is made active by default. Use the **w** (Change Map) command to change the active file.

Example

The **lm** command returns a display similar to the following:

```
COMSAM2D is active.
DISK01D.
```

ln

ln [*addr*]

The **ln** command lists the symbol nearest the specified address. The command lists the nearest symbol before and after the specified *addr* parameter. This command also shows line-number information if it is available in the symbol file.

Parameters

addr
Specifies any valid instruction address. The default value is the current disassembly address.

Example

The **ln** command without the *addr* parameter displays the nearest symbols before and after the current disassembly address. The output is similar to the following:

```
6787 VerifyRamSemAddr + 10
67AA PutRamSemID - 13
```

ls

ls *group-name* | *name-chars* | *

The **ls** command lists the symbols in the specified group or lists names that match the search specification in all groups. The only valid wildcard is a single asterisk (*) as the last character on the command line; all other characters are ignored.

Parameters

group-name
Names the group that contains the symbols you want to list.

name-chars
Specifies the beginning characters of the symbols you want to list.

Example

The following example displays all the symbols in the DOSRING3CODE group:

```
ls DOSRING3CODE
```

Symbols are displayed in a format similar to the following:

```
0000 Sigdispatch
001A LibInitDisp
```

The following example displays all the symbols that begin with the string vkd:

```
ls vkd*
```

Group names are displayed as they are searched, in a form similar to the following:

```
GROUP: [0028] CODE
       60003A74 VKD_Control_Debug
GROUP: [0030] DATA
       6001DFFC VKD_CB_Offset
GROUP: [0030} IDATA
```

The following example displays the address and group for the symbol VMM_base:

```
ls vmm_base
```

---

m

m *range addr*

The **m** command moves a block of memory from one memory location to another.

Overlapping moves—those in which part of the block overlaps some of the current addresses—are always performed without loss of data. Addresses that could be overwritten are moved first. For moves from higher to lower addresses, the sequence of events is first to move the data at the block's lowest address and then to work toward the highest. For moves from lower to higher addresses, the sequence is first to move the data at the block's highest address and then to work toward the lowest.

Note that if the addresses in the block being moved will not have new data written to them, the data that was in the block before the move will remain. The **m** command copies the data from one area into another, in the sequence described, and writes over the new addresses—hence, the importance of the moving sequence.

To review the results of a memory move, use the **d** (Display Memory) command, specifying the same address you used with the **m** command.

Parameters

range
Specifies the block of memory to be moved.

addr
Specifies the starting address at which the memory is to be relocated.

Example

The following example first moves the data at address CS:110 to CS:510 and then moves the data at CS:10F to CS:50F, and so on, until the data at CS:100 is moved to CS:500:

```
mCS:100 110 CS:500
```

o

o *word byte*

The **o** command writes a byte to a 16-bit port address.

Parameters

word
Specifies the 16-bit port address to be written to.

byte
Specifies the 8-bit value to be written to the port.

Example

The following example writes the byte value 4Fh to output port 2F8h:

```
o 2F8 4F
```

---

p

p[n] [=*addr*][*count*]

The **p** command executes the instruction at a specified address and displays the current values of all the registers and flags (whatever the **zd** command has been set to). It then executes the default command string, if any.

The **p** command is identical to the **t** (Trace Instructions) command, except that it automatically executes and returns from any calls or software interrupts it encounters. The **t** command always stops after executing into the call or interrupt, leaving execution control inside the called routine.

Parameters

n
Suppresses the register display so just the assembly line is displayed. The suppression results only if the default command, **zd**, is set to a normal setting, **r**.

addr
Specifies the starting address at which to begin execution. If you omit the optional *addr* parameter, execution begins at the instruction pointed to by the CS and IP registers. Use the equal sign (=) only if you specify *addr*.

count
Specifies the number of instructions to execute before stopping and executing the default command string. The command executes the default command string for each instruction before executing the next.

Example

The following example executes the instruction pointed to by the current CS and IP register values before it executes the default command string:

```
p
```

The following example executes the instruction at address CS:120 before it executes the default command string:

```
p=120
```

---

r

r *reg*=*word*

The **r** command displays the contents of one or more central processing unit (CPU) registers and allows the contents to be changed to new values. If you specify the *reg* parameter with the **r** command, the 16-bit value of that register is displayed in hexadecimal format followed by a colon (:) prompt on the next line. You can then enter a new *word* value for the specified register or press ENTER if you do not want to change the register value.

If you specify **f** for *reg*, the debugger displays the flags in a row at the beginning of a new line and displays a hyphen (-) after the last flag.

You can type new flag values in any order as alphabetic pairs. You do not have to leave spaces between these values. To terminate the **r** command, press ENTER. Any flags for which you did not specify new values remain unchanged.

If you type more than one value for a flag or enter an invalid flag name, the flags up to the error in the list are changed and those flags at and after the error are not changed. In addition, 80386 Debugger returns the following error message:

```
Bad Flag
```

Parameters

reg
Specifies the register to be displayed. If you omit *reg*, the debugger displays the contents of all registers and flags along with the next executable instruction.

word
Specifies the new value for the register. For the Flags register, set or clear a flag by using one of the following names:

Flag code	Meaning
OV	Overflow set

Flag code	Meaning
NV	Overflow clear
DN	Direction decrement
UP	Direction increment
EI	Interrupt enabled
DI	Interrupt disabled
NG	Sign negative
PL	Sign positive
ZR	Zero set
NZ	Zero clear
AC	Auxiliary carry set
NA	Auxiliary carry clear
PE	Parity even
PO	Parity odd
CY	Carry set
NC	Carry clear
NT	Nested task switch (on and off)

For the machine status word (MSW) register, use the following names to set a flag:

Flag name	Action
TS	Sets the task switch bit.
EM	Sets the emulation processor extension bit.
MP	Sets the monitor processor extension bit.
PM	Sets the protected-mode bit.

Comments

Setting the protected-mode bit from within the debugger does *not* set the target system to run in protected mode. The debugger simulates the setting. To configure the target system to run in protected mode, you would have to set the PM bit in the MSW register and reset the target system to restart in protected mode.

Example

The **r** command without parameters produces a display similar to the following:

```
AX=0698  BX=2008  CX=2C18  DX=18AB  SP=1B7A  BP=00FF  SI=0020  DI=10CD
IP=0450  CS=18B0  DS=1BE8  ES=0DA8  SS=0048  NV UP DI PL NZ NA PONC
GDTR=01BE80 3687  IDTR=01F508 03FF  TR=0010  LDTR=0028 IOPL=3 MSW=PM
18B0:0450 C3              RET
```

The following example displays each flag with a two-letter code. To change any flag, type the two-letter code that inverts the setting. The flags are either set or cleared.

```
rf
```

The example produces a display similar to the following:

```
NV UP DI NG NZ AC PE NC - _
```

To change the value of a flag's setting, type the two-letter code that inverts the setting for that flag. The following example changes the sign flag to positive, enables interrupts, and sets the carry flag:

```
NV UP DI NG NZ AC PE NC - PLEICY
```

The following command modifies the MSW bits:

```
rmsw
```

Then 80386 Debugger displays the status of the MSW register and prints a colon on the next line.

s

s *range list* | "*string*"

The **s** command searches an address range for a specified list of bytes or an ASCII character string.

You can include one or more bytes in *list*, but multiple bytes must be separated by a space or comma. When you search for more than one byte, the command returns the address of only the first byte in the string. When *list* contains only one byte, the debugger displays the addresses of all occurrences of the byte in *range*.

Parameters

range
Specifies the block of memory to be searched.

list
Specifies one or more byte values to search for.

string
Specifies an ASCII character string to be searched for. The string must be enclosed in quotation marks.

Example

The following example searches for byte 41h in the address range CS:100 to CS:110:

```
sCS:100 110 41
```

If it finds the value, this command produces a display similar to the following:

```
04BA:0104
04BA:010D
```

t

t[a|c|n|s|x|z][=*start_addr***][***count***][***addr***]**

The **t** command executes one or more instructions along with the default command string and then displays the decoded instruction. If you include the *start_addr* parameter, tracing starts at the specified address. Otherwise, the command steps through the next machine instruction and then executes the default command string.

The **t** command uses the hardware trace mode of the Intel microprocessor. Consequently, you can also trace instructions stored in read-only memory (ROM).

Parameters

a
Indicates that an ending address is specified for the trace. Instructions are traced until the address in *addr* is reached.

c
Suppresses all output and counts instructions traced. An ending address is required for this command. Instructions are traced until the address in *addr* is reached.

n
Suppresses the register display so just the assembly line is displayed. This works only if the default command, **zd**, is set to **r** (the normal setting).

s
Suppresses output; the instruction and count are displayed for each **call** and the return from that call.

x
Forces the debugger to trace regions of code known to be untraceable (_PGSwitchContext, for example).

z
Allows original trap handler address to be traced into without having to unhook the exception. Use this option instead of **vcp d; t; vsp d**.

start_addr
Specifies the instruction address at which to start tracing. The equal sign (=) is required.

count
Specifies the number of instructions to execute and trace.

addr
Specifies the instruction address at which to stop tracing.

Example

The following example traces the current position (04BA:011A) and uses the default command string (**r** command) to display registers:

```
t
```

The resulting output is similar to the following:

```
AX=0E00  BX=00FF  CX=0007  DX=01FF  SP=039D  BP=0000  SI=005C  DI=0000
IP=011A  CS=04BA  DS=04BA  ES=04BA  SS=04BA  NV UP DI NG NZ AC PENC
GDTR=01D700 3677  IDTR=020D7A 03FF  TR=0010  LDTR=0028 IOPL=3 MSW=PM
04BA:011A  CD21          PUSH   21
```

The following command causes the debugger to execute 16 (10h) instructions beginning at 011A in the current selector:

```
t=011A 10
```

The debugger executes and displays the results of the default command string for each instruction. The display is scrolled until the last instruction is executed. Press the CTRL+S key combination to stop the scrolling and CTRL+Q to resume.

u

u [*range*]

The **u** command disassembles bytes and displays the source statements, with addresses and byte values, that correspond to them.

The display of disassembled code looks similar to a code listing for an assembled file. If you type the **u** command by itself, 20h bytes are disassembled at the first address after the one displayed by the previous **u** command.

Parameters

range
Specifies the block of memory in which instructions are to be disassembled. If no *range* is given, the command disassembles the next 20h bytes.

Example

The following example disassembles and displays 20h bytes from the specified address:

```
uCS:046C
```

The resulting display is similar to the following:

```
1A60:046C C3                RET
1A60:046D 9A6B3E100D        CALL  0D10:3E6B
1A60:0472 33C0              XOR  AX,AX
1A60:0474 50                PUSH  AX
1A60:0475 9D                POPF
1A60:0476 9C                PUSHF
1A60:0477 58                POP  AX
1A60:0478 2500F0            AND  AX,F000
1A60:047B 3D00F0            CMP  AX,F000
1A60:047E 7508              JNZ  0488
1A60:0480 689C26            PUSH  269C
1A60:0483 9AF105100D        CALL  0D10:05F1
```

If the bytes at some addresses are altered, the disassembler alters the instruction statements. You can also use the **u** command for the changed locations, for the new instructions viewed, and for the disassembled code used to edit the source file.

V

v

The **v** command displays the current 80386 Debugger version number and date.

vc

vc[n | p | r | v] *number*[,*number* [,...]]

The **vc** command clears the specified interrupt vector and reinstalls the previous interrupt vector.

Parameters

n
Removes the beep from traps that beep when encountered; does not clear the traps.

p
Clears protected-mode vectors only.

r
Clears real-mode vectors only.

v
Clears virtual 8086 (V86) mode vectors only.

number
Specifies the interrupt vector to clear.

---

vl

vl[n | p | r | v]

Lists the interrupt vectors that the debugger intercepts. Vectors that have been set with the **vt** command (as opposed to **vs**) are listed with an asterisk (*) following the vector number.

Parameters

n
Lists the traps that beep when encountered.

p
Lists the protected-mode vectors only.

r
Lists the real-mode vectors only.

v
Lists the virtual 8086 (V86) mode vectors only.

VO

vo[n | p | r | v]

The **vo** command lists interrupt vectors in the display format based on the **newvec** option. For details, see the **y** command.

Parameters

n
Lists the traps that beep when encountered.

p
Lists the protected-mode vectors only.

r
Lists the real-mode vectors only.

v
Lists the virtual 8086 (V86) mode vectors only.

VS

vs[n | p | r | v] *number*[,*number*[,...]]

The **vs** command adds a new interrupt vector to the list of intercepted vectors. Vectors set by this command do not intercept interrupts that occur at ring 0.

Parameters

n
Lists the traps that beep when encountered.

p
Lists the protected-mode vectors only.

r
Lists the real-mode vectors only.

v
Lists the virtual 8086 (V86) mode vectors only.

number
Specifies the interrupt vector to intercept.

vt

vt[n | p | r | v] *number*[,*number*[,...]]

The **vt** command adds a new interrupt vector to the list of intercepted vectors.

Parameters

n
Lists the traps that beep when encountered.

p
Lists the protected-mode vectors only.

r
Lists the real-mode vectors only.

v
Lists the virtual 8086 (V86) mode vectors only.

number
Specifies the interrupt vector to intercept.

---

w

w [*map-name*]

The **w** command changes the active map file.

Parameters

map-name
Specifies the name of the map file you want to make active. Use the **lm** (List Map) command to display a list of available map files.

If *map-name* is not specified, the loaded maps are displayed and the user is prompted to select a map by pressing its corresponding number.

Example

The **lm** command can be used to display the loaded map files in a form similar to the following:

```
COMSAM2D is active.
DISK01D.
```

Then the following command can be used to change the active map file to DISK01D:

```
w DISK01D
```

The following command displays the list of loaded maps:

```
w
```

The resulting display is similar to the following, prompting the user to type the number corresponding to the map to activate:

```
1. KERNEL
2. Win386 is active
activate which map?
```

In this case, pressing 1 activates the KERNEL map; pressing 2 leaves the Win386 map activated; and pressing the SPACEBAR leaves the current map activated. Any other key is ignored, and the debugger will continue to wait for input.

---

wa

wa *map-name*

The **wa** command adds the specified map to the list of active maps.

Parameters

map-name
Specifies the map to add to the list of active maps.

---

wr

wr *map-name*

The **wr** command removes the specified map from the list of active maps.

Parameters

map-name
Specifies the map to remove from the list of active maps.

y

y[**?** | *option*]

The **y** command changes the debugger configuration. The following list describes the available configuration options. All settings are toggles.

Parameters

?
Displays a list of supported options.

option
Following are the available configuration options:

/a
Controls automatic symbol loading. If this option is set, Windows will not load symbols automatically.

/n
Sets the following options:

codebytes
dislwr
int3line
newprompt
newreg
newvec
symaddrs

/v
Controls segment load notification messages. If this option is set, all segment load notifications will be displayed.

386env
Controls the size of addresses, registers, and so on when displayed. When this option is on, addresses, registers, and so on are shown in 32-bit format; otherwise, they are shown in 16-bit format.

codebytes
Causes the disassembler to display the code bytes along with the disassembled instructions.

disaddr
Causes the disassembler to display the disassembly address.

disline
Causes the disassembler to display the filename and line number of each operation code (opcode).

dislwr
Controls the disassembler's lowercase option. When the flag is on, disassembly is in lowercase.

int3line
Causes the disassembler to display the filename and line number on **int 3** instructions.

newprompt
Causes 80386 Debugger to produce a double prompt when paging is enabled and a nesting level if the debugger is reentered.

newreg
Controls the format of the register display.

newvec
Controls the display format for the intercepted interrupt vectors.

regterse
Controls the number of registers displayed by the **r** (Register Dump) command. In the 80386 environment, when **regterse** is on, only the first three lines are displayed (instead of the normal six lines plus disassembly line). In the 80286 environment (**386env** off), only the first two lines are displayed (instead of the normal three lines plus disassembly line).

scrncols
Sets the number of screen columns in the debug display. The default is 79 columns.

scrnlines
Sets the number of screen lines in the debug display. The default is 24 lines.

skipint3s
Causes the debugger to ignore inline **int 3** instructions.

symaddrs
Causes the disassembler to display symbol values along with the symbols.

teftibase
Sets the base port address for the timing card.

Z

z

Replaces the instruction bytes of the current **int 3** instruction or the previous **int 1** instruction with **nop** instructions. This allows the user to avoid **int 1** or **int 3** instructions that were assembled into the executable file by breaking into the debugger more than once.

zd

zd

The **zd** command executes the default command string.

The default command string is initially set to the **r** (Display Registers) command by the debugger. The default command string is executed every time a breakpoint is encountered during execution of the application or whenever a **p** (Program Trace) or **t** (Trace Instructions) command is executed.

Use the **zl** command to display the default command string and the **zs** command to change the default command string.

---

zl

zl

The **zl** command displays the default command string.

Example

The following example displays the default command string:

```
zl
```

The resulting output is similar to the following:

```
"R"
```

ZS

zs "*string*"

The **zs** command makes it possible for you to change the default command string.

Parameters

string

Specifies the new default command string. The string must be enclosed in single or double quotation marks. You must separate the debugger commands within the string with semicolons.

Example

The following example changes the current default command string to an **r** (Display Register) command followed by a **c** (Compare Memory) command:

```
zs "r;c100 L 100 300"
```

The following example begins execution whenever an **int 3** instruction is executed in your test application. This example executes a **g** (Go) command every time an **int 3** instruction is executed.

```
zs "j (by cs:ip) == cc 'g'"
```

You can use **zs** as follows to set up a watchpoint:

```
zs "j (wo 40:1234) == 0eeed;t"
```

This command traces until the word at 40:1234 is *not* equal to 0EEED. This does not work if you are tracing through the mode switching code in MS-DOS or other sections of code that cannot be traced.

5.7 Related Topics

For information about programming Windows applications, see the *Microsoft Windows Guide to Programming*.

Analyzing System Failures: Dr. Watson

Chapter 6

6.1 Configuring Dr. Watson from the WIN.INI File 173
- 6.1.1 The SkipInfo Entry 173
- 6.1.2 The ShowInfo Entry 174
- 6.1.3 The DisLen Entry 174
- 6.1.4 The TrapZero Entry 175
- 6.1.5 The GPContinue Entry 175
- 6.1.6 The DisStack Entry 176
- 6.1.7 The LogFile Entry 176

6.2 Sample Dr. Watson Log File 177

6.3 Sample Dr. Watson Log File with Comments 179

Microsoft Windows Dr. Watson is a diagnostic tool for the Microsoft Windows operating system. It detects system and application failures caused by Windows applications and can store information in a disk file. This file can help you find and fix problems in your applications.

Only a single instance of Dr. Watson can be run at a time. Dr. Watson uses the dynamic-link library TOOLHELP.DLL, so it runs only in standard or 386 enhanced mode. Dr. Watson cannot trap faults in a Windows MS-DOS session.

6.1 Configuring Dr. Watson from the WIN.INI File

You can configure Dr. Watson to meet your needs by including settings for any of the following entries in the [Dr. Watson] section of your WIN.INI file (note the space between Dr. and Watson):

SkipInfo
ShowInfo
DisLen
TrapZero
GPContinue
DisStack
LogFile

The following sections describe the Dr. Watson WIN.INI entries.

6.1.1 The SkipInfo Entry

The SkipInfo entry controls which parts of the failure report are actually written to disk. Following are the values you can set to disable parts of the failure report:

Value	Meaning
32bitregs	Disable values of 32-bit registers and of the FS and GS registers on 80386/80486 processors.
clues	Disable the dialog box titled "Dr. Watson's Clues."
information	Disable system information, such as the Windows version number, processor type, and memory available.
registers	Disable 16-bit registers.
segments	Disable segment contents, base addresses, length, and flags.
stack	Disable stack backtrace.
summary	Disable four-line summary at beginning of error report.
tasks	Disable list of all active tasks (running applications).
time	Disable Dr. Watson start and stop times.

Each of the SkipInfo values can be abbreviated to its first three letters. The following example disables the Dr. Watson's Clues dialog box and the stack backtrace:

```
[Dr. Watson]

SkipInfo=clu sta
```

6.1.2 The ShowInfo Entry

Some parts of the Dr. Watson failure report are disabled by default. They can be enabled with the ShowInfo entry. Following are the values you can set to enable parts of the failure report:

Value	Meaning
disassembly	Enable separate disassembly of the fault address. This does not affect disassembly of stack frames. (See Section 6.1.3, "The DisLen Entry.")
errorlog	Enable error logging.
locals	Enable stack dump of local variable and parameter values.
modules	Enable list of all loaded modules, including dynamic-link libraries (DLLs) and font files.
paramlog	Enable parameter-validation error logging.
sound	Enable audible warnings.

Each of the ShowInfo values can be abbreviated to its first three letters. The following example sets all six values for the ShowInfo entry, enabling those six parts of the failure report:

```
[Dr. Watson]

ShowInfo=dis err loc mod par sou
```

6.1.3 The DisLen Entry

The DisLen entry controls how many instructions are disassembled in stack traces and the disassembly portion of the failure report. The default value is 8. The following example sets the value to 4:

```
[Dr. Watson]

DisLen=4
```

6.1.4 The TrapZero Entry

By default, Dr. Watson does not trap divide overflow exceptions, because many applications provide their own handling. The TrapZero entry can be used to enable trapping of divide overflow exceptions, as shown in the following example:

```
[Dr. Watson]

TrapZero=1
```

6.1.5 The GPContinue Entry

One of the most advanced features of Dr. Watson enables an application to continue even after a general protection (GP) fault occurs. Because a GP fault means that a bug has been encountered, continuing is dangerous. However, some application developers requested the ability to continue running an application even after a GP fault. If the GPContinue entry is used, Dr. Watson performs the following tests when a GP fault occurs. If each of the following four conditions is true, Dr. Watson allows the application to continue:

1. Bit 0 of GPContinue is set.
2. The faulting instruction is one that can be allowed to continue.

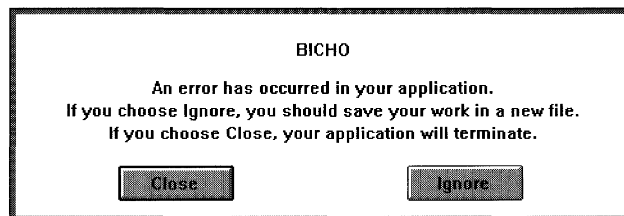
   The following example, which happens to be beyond the end of a segment, would be allowed to continue:

   ```
   mov     ax,[ffff]
   ```

   The following instruction, which involves an invalid address, would not be allowed to proceed:

   ```
   jmp     seg:offs
   ```

3. The fault is *not* in KERNEL or USER. (Or the fault is in KERNEL or USER, and you have set the appropriate bit in the GPContinue value to continue in spite of the risk.)
4. The user wants to continue. Dr. Watson displays the following dialog box so that the user can decide.



If the user chooses the Close button, an error message box appears.

Although it is very risky, you can also allow continuation in KERNEL or USER by setting GPContinue as required. Following are the bits and values for the GPContinue entry:

Bit	Value	Meaning
0	1	Allow continuation. (This is the default setting.)
1	2	Write only three-line reports.
2	4	Continue even if the fault is in KERNEL.
3	8	Continue even if the fault is in USER.

You must combine these values. The following example permits continuation after a GP fault in USER:

```
set GPContinue=9
```

6.1.6 The DisStack Entry

The DisStack entry controls how many levels back on the stack are to be disassembled. The default value is 2. The following example sets the value to 100:

```
[Dr. Watson]

DisStack=100
```

6.1.7 The LogFile Entry

By default, the Dr. Watson log file is named DRWATSON.LOG and placed in the Windows directory. The filename can be changed to any valid filename, even the name of a printer or debugging terminal. For example, to write to a terminal on COM1, use the following setting:

```
[Dr. Watson]

LogFile=com1
```

6.2 Sample Dr. Watson Log File

To save disk space, Dr. Watson generates a complete report for only the first three errors. The next 17 errors generate a report summary. After 20 errors, Dr. Watson stops writing to the log file. If you close Dr. Watson and rerun it, writing to the log file resumes. You can determine the number of error reports generated in the current session by double-clicking the Dr. Watson icon.

When the log file reaches 100K, Dr. Watson displays a warning message. After you have analyzed the error reports in the log file, you should delete the log file.

The following example shows a typical Dr. Watson log file.

```
Start Dr. Watson 0.80 - Thu Sep 26 10:51:28 1991
************************************************************************
Dr. Watson 0.80 Failure Report - Thu Sep 26 10:51:36 1991
BICHO had a 'Exceed Segment Bounds (Read)' fault at BICHO
_DoCommand+006b
$tag$BICHO$Exceed Segment Bounds (Read)$BICHO _DoCommand+006b
    $push word ptr [fffe]$Thu Sep 26 10:51:36 1991

CPU Registers (regs)
ax=1e54  bx=0014  cx=0d7f  dx=0111  si=1e54  di=0111
ip=02fd  sp=230c  bp=237a  O- D- I+ S- Z- A+ P+ C-
cs = 0e57  8059fbc0:083f Code Ex/R
ss = 0d7f  8059d5e0:25df Data R/W
ds = 0d7f  8059d5e0:25df Data R/W
es = 0d7f  8059d5e0:25df Data R/W

CPU 32 bit Registers (32bit)
eax = 00001e54  ebx = 00000014  ecx = ffff0d7f  edx = 00000111
esi = 00001e54  edi = 00000111  ebp = 0000237a  esp = 800422fc
fs = 0000        0:0000 Null Ptr
gs = 0000        0:0000 Null Ptr
eflag = 00000002

System Info (info)
Windows version 3.10
Debug build
Windows Build 3.1.048
Username Unknown User
Organization Unknown Organization
System Free Space 7131008
Stack base 1122, top 9164, lowest 7504, size 8042
System resources:  USER: 87% free, seg 0777  GDI: 85% free, seg 05d7
LargestFree 6594560, MaxPagesAvail 1610, MaxPagesLockable 267
TotalLinear 1948, TotalUnlockedPages 274, FreePages 52
TotalPages 614, FreeLinearSpace 1611, SwapFilePages 7158
Page Size 4096
4 tasks executing.
WinFlags -
```

```
  Math coprocessor
  80386 or 80386 SX
  Enhanced mode
  Protect mode

Stack Dump (stack)
Stack Frame 0 is BICHO _DoCommand+006b       ss:bp 0d7f:237a
0e57:02f0  e9 02b9                  jmp    near 05ac
0e57:02f3  6a 00                    push   00
0e57:02f5  9a 8db0 0477             callf  0477:8db0
0e57:02fa  e9 02af                  jmp    near 05ac
(BICHO:_DoCommand+006b)
0e57:02fd  ff 36 fffe               push   word ptr [fffe]
0e57:0301  68 0110                  push   0110
0e57:0304  e8 fe5d                  call   near 0164
0e57:0307  83 c4 04                 add    sp, 04

Stack Frame 1 is BICHO MAINWNDPROC+0027      ss:bp 0d7f:2388
0e57:0670  eb 16                    jmp    short 0688
0e57:0672  ff 76 0a                 push   word ptr [bp+0a]
0e57:0675  56                       push   si
0e57:0676  e8 fc19                  call   near 0292
(BICHO:MAINWNDPROC+0027)
0e57:0679  83 c4 04                 add    sp, 04
0e57:067c  99                       cwd
0e57:067d  eb 1f                    jmp    short 069e
0e57:067f  6a 00                    push   00

Stack Frame 2 is USER IDISPATCHMESSAGE+007e  ss:bp 0d7f:239e
Stack Frame 3 is BICHO WINMAIN+0050          ss:bp 0d7f:23bc
Stack Frame 4 is BICHO 1:00a3                ss:bp 0d7f:23ca

System Tasks (tasks)
Task  WINEXIT, Handle 0daf, Flags 0001, Info    9248 08-09-90 16:52
  FileName C:\MS\WIN\DON\WINEXIT.EXE
Task DRWATSON, Handle 0ea7, Flags 0001, Info   26256 09-23-91 12:00
  FileName C:\WIN31\DRWATSON.EXE
Task  PROGMAN, Handle 060f, Flags 0001, Info  110224 09-23-91 12:02
  FileName C:\WIN31\PROGMAN.EXE
Task    BICHO, Handle 0da7, Flags 0001, Info   16537 09-11-91  8:45
  FileName D:\BICHO.EXE

1> I ran a test app that accessed a value
2> beyond the limits of the segment bounds.
Stop Dr. Watson 0.80 - Thu Sep 26 10:52:10 1991
```

6.3 Sample Dr. Watson Log File with Comments

The following version of the Dr. Watson log file includes comments. These comments do not appear in the normal Dr. Watson log. They have been added here to explain the sections of the log.

```
Start Dr. Watson 0.80 - Thu Sep 26 10:51:28 1991
        # This line is inserted each time you start Dr. Watson.
        # You can disable it with SkipInfo=time.

*************************************************************************
        # This line marks the beginning of a Dr. Watson report.

Dr. Watson 0.80 Failure Report - Thu Sep 26 10:51:36 1991
        # Version 0.80 of Dr. Watson - date report was generated

BICHO had a 'Exceed Segment Bounds (Read)' fault at BICHO
_DoCommand+006b
        # Application 'BICHO' had an 'Exceed Segment Bounds' fault
        # while reading memory.  The actual code that failed
        # was also in BICHO, 0x6b bytes past the start of the
        # DoCommand function.

$tag$BICHO$Exceed Segment Bounds (Read)$BICHO _DoCommand+006b
    $push word ptr [fffe]$Thu Sep 26 10:51:36 1991
        # This line repeats the previous information in a format
        # easier for automatic code to parse.  It also includes
        # the actual faulting instruction (a push instruction here).

CPU Registers (regs)
ax=1e54  bx=0014  cx=0d7f  dx=0111  si=1e54  di=0111
        # The 16-bit CPU registers.  This can be useful for decoding
        # what address an instruction was modifying.

ip=02fd  sp=230c  bp=237a  O- D- I+ S- Z- A+ P+ C-
        # The IP is the instruction pointer (Program Counter).
        # SP and BP are the stack pointer and base pointer.
        # The last 8 items show the states of the flag bits.
        # In this case, Overflow, Direction, Sign, Zero, and Carry
        # bits are clear (0); the Interrupt, AuxCarry, and Parity
        # bits are set (1).

cs = 0e57  8059fbc0:083f Code Ex/R
        # Code segment selector is 0E57, linear address is 8059FBC0
        # (enhanced-mode linear addresses often start with 8xxx),
        # and the limit is 083F.  Accessing code and data segments
        # beyond their limits is a common cause of GP faults.

ss = 0d7f  8059d5e0:25df Data R/W
        # Stack selector
```

```
ds = 0d7f  8059d5e0:25df Data R/W
        # Data selector--note that the limit is 25DF, and we
        # tried to read the value at FFFE, beyond the limit.

es = 0d7f  8059d5e0:25df Data R/W

CPU 32 bit Registers (32bit)
eax = 00001e54  ebx = 00000014  ecx = ffff0d7f  edx = 00000111
esi = 00001e54  edi = 00000111  ebp = 0000237a  esp = 800422fc
fs = 0000         0:0000 Null Ptr
        # If the selector is 0, it indicates a NULL pointer.  Trying
        # to use a NULL pointer is another common cause of GP faults.

gs = 0000         0:0000 Null Ptr
eflag = 00000002

System Info (info)
Windows version 3.10
Debug build
        # The debugging version of Windows was running.

Windows Build 3.1.048
        # This is an internal Microsoft build of Windows, #48.

Username Unknown User
        # Your name here

Organization Unknown Organization
        # Your organization here

System Free Space 7131008
Stack base 1122, top 9164, lowest 7504, size 8042
        # Stack size of current task

System resources:  USER: 87% free, seg 0777  GDI: 85% free, seg 05d7
LargestFree 6594560, MaxPagesAvail 1610, MaxPagesLockable 267
        # These statistics are almost useless.

TotalLinear 1948, TotalUnlockedPages 274, FreePages 52
TotalPages 614, FreeLinearSpace 1611, SwapFilePages 7158
Page Size 4096
4 tasks executing.
WinFlags -
  Math coprocessor
  80386 or 80386 SX
  Enhanced mode
  Protect mode

Stack Dump (stack)
        # We dump the stack to see what called the routine that failed.
```

```
Stack Frame 0 is BICHO _DoCommand+006b       ss:bp 0d7f:237a
        # The failure occurred in BICHO, 0x6B bytes past the
        # start of DoCommand.

0e57:02f0  e9 02b9              jmp    near 05ac
0e57:02f3  6a 00                push   00
0e57:02f5  9a 8db0 0477         callf  0477:8db0
0e57:02fa  e9 02af              jmp    near 05ac
(BICHO:_DoCommand+006b)
        # The failure happened on the following instruction:

0e57:02fd  ff 36 fffe           push   word ptr [fffe]
        # We tried to read a value from memory at DS:FFFE and
        # push it on the stack.  However, the limit of the DS
        # segment is 25DF.

0e57:0301  68 0110              push   0110
0e57:0304  e8 fe5d              call   near 0164
0e57:0307  83 c4 04             add    sp, 04

Stack Frame 1 is BICHO MAINWNDPROC+0027      ss:bp 0d7f:2388
        # The Bicho MainWndProc probably called DoCommand.

0e57:0670  eb 16                jmp    short 0688
0e57:0672  ff 76 0a             push   word ptr [bp+0a]
0e57:0675  56                   push   si
0e57:0676  e8 fc19              call   near 0292
(BICHO:MAINWNDPROC+0027)
0e57:0679  83 c4 04             add    sp, 04
0e57:067c  99                   cwd
0e57:067d  eb 1f                jmp    short 069e
0e57:067f  6a 00                push   00

Stack Frame 2 is USER IDISPATCHMESSAGE+007e   ss:bp 0d7f:239e
        # USER is the Windows USER.EXE.  It is what calls your
        # window and dialog box procedures.  In this case,
        # it called the BICHO MainWndProc.

Stack Frame 3 is BICHO WINMAIN+0050          ss:bp 0d7f:23bc
        # Here is the BICHO WinMain, which called
        # DispatchMessage, which called MainWndProc.

Stack Frame 4 is BICHO 1:00a3                ss:bp 0d7f:23ca
        # Here is where the startup code calls WinMain.

System Tasks (tasks)
Task  WINEXIT, Handle 0daf, Flags 0001, Info    9248 08-09-90 16:52
  FileName C:\MS\WIN\DON\WINEXIT.EXE
Task DRWATSON, Handle 0ea7, Flags 0001, Info   26256 09-23-91 12:00
  FileName C:\WIN31\DRWATSON.EXE
        # This task will always be listed.
```

```
Task  PROGMAN, Handle 060f, Flags 0001, Info  110224 09-23-91 12:02
  FileName C:\WIN31\PROGMAN.EXE
        # This task (or whatever shell you use) will always be listed.

Task    BICHO, Handle 0da7, Flags 0001, Info   16537 09-11-91  8:45
  FileName D:\BICHO.EXE
        # This is the name of the program that caused the failure.

1> I ran a test app that accessed a value
2> beyond the limits of the segment bounds.
        # Anything you type in the Dr. Watson's Clues dialog box
        # is added to the log file, so you can type what you
        # want to remember.

Stop Dr. Watson 0.80 - Thu Sep 26 10:52:10 1991
        # This line is written each time Dr. Watson terminates.
```

Monitoring Messages: Spy

Chapter 7

7.1 Selecting Options: The Options! Menu 185
  7.1.1 Selecting Message Types 185
  7.1.2 Selecting the Output Device 186
  7.1.3 Selecting Frequency of Output 186
7.2 Selecting a Window: The Window Menu 187
7.3 Starting and Stopping Spy: The Spy Menu 187
7.4 Related Topics 188

Microsoft Windows Spy (SPY.EXE) is a tool for the Microsoft Windows operating system. Spy makes it possible for you to monitor messages sent to one or more windows and to examine the values of message parameters.

Note If you are using the Microsoft CodeView for Windows (CVW) debugger to debug your application, you can use CVW instead of Spy to trace messages.

This chapter describes how to use the Options!, Window, and Spy menus to specify how Spy is to operate.

7.1 Selecting Options: The Options! Menu

The Options! menu displays a dialog box in which you make selections about the following:

- Which message types Spy is to monitor
- Which output device Spy is to send messages to
- Whether Spy is to send output synchronously or asynchronously

You make your selections from items displayed under Messages, Output, and Display.

7.1.1 Selecting Message Types

Under Messages, you can select any of the following message types you want Spy to monitor:

Message	Description
Mouse	Mouse messages, such as WM_MOUSEMOVE and WM_SETCURSOR
Input	Input messages, such as WM_CHAR and WM_COMMAND
System	Systemwide messages, such as WM_ENDSESSION and WM_TIMECHANGE
Window	Window manager messages, such as WM_SIZE and WM_SHOWWINDOW
Init	Initialization messages, such as WM_INITMENU and WM_INITDIALOG
Clipboard	Clipboard messages, such as WM_RENDERFORMAT
Other	Messages other than the types explicitly listed

Message	Description
DDE	Dynamic data exchange (DDE) messages, such as WM_DDE_REQUEST
Non Client	Windows nonclient messages, such as WM_NCDESTROY and WM_NCHITTEST

By default, Spy monitors all messages.

7.1.2 Selecting the Output Device

Under Output, you can select which of the following output devices you want Spy to send messages to:

Device	Description
Window	Spy displays messages in the Spy window. You can specify how many messages Spy stores in its buffer. By default, Spy stores up to 100 lines of messages, which you can view by scrolling through the Spy window. You can also change the maximum number of lines that can be stored in the buffer.
Com1	Spy sends messages to the COM1 port.
File	Spy sends messages to the specified file. The default output file is SPY.OUT.

7.1.3 Selecting Frequency of Output

Under Display, you can select which of the following frequency options you want Spy to use:

Option	Description
Synchronous	Spy displays messages as it receives them.
Asynchronous	Spy queues messages for display.

By default, Spy sends messages synchronously.

7.2 Selecting a Window: The Window Menu

Use the Window menu to select the window you want Spy to monitor. The Window menu contains the following commands:

Command	Description
Window	Specifies the window that Spy is to monitor. When you choose the Window command, Spy displays the Spy Window dialog box. This dialog box displays information about the window in which the cursor is located. As you move the cursor from window to window, the following information is updated:

Item	Description
Window	Handle of the window.
Class	Window class.
Module	Program that created the window.
Parent	Handle of the parent window and the name of the program that created the parent window.
Rect	Upper-right and lower-left coordinates of the window and the window size in screen coordinates.
Style	Style bits of the window in which the cursor is located, the principal style of the window, and an identifier if the window is a child window. The principal style can be WS_POPUP, WS_ICONIC, WS_OVERLAPPED, or WS_CHILD.

Command	Description
All Windows	Specifies that Spy is to display messages received by all windows.
Clear Window	Clears the Spy window.

7.3 Starting and Stopping Spy: The Spy Menu

After using the Options! and Window menus to make your selections, start Spy by clicking the window you selected and choosing the OK button in the dialog box.

To stop monitoring messages, resume monitoring messages, or close Spy, use the Spy menu. The Spy menu contains the following commands:

Command	Description
Spy On/Off	Starts and stops message monitoring.
Exit	Closes Spy.
About Spy	Provides information about the version of Spy you are using.

7.4 Related Topics

For information about monitoring DDE messages, see Chapter 8, "Monitoring Dynamic Data Exchange Activity: DDESpy."

For an introduction to input messages, see the *Microsoft Windows Guide to Programming*.

For information about message syntax and content, see the *Microsoft Windows Programmer's Reference, Volume 3*.

Monitoring Dynamic Data Exchange Activity: DDESpy

Chapter 8

8.1 The Output Menu 191
8.2 The Monitor Menu 191
8.2.1 Monitoring String-Handle Data 192
8.2.2 Monitoring Sent or Posted DDE Messages 192
8.2.3 Monitoring Callbacks 193
8.2.4 Monitoring Errors 193
8.3 Tracking Options 193
8.3.1 Tracking String Handles 194
8.3.2 Tracking Active Conversations 194
8.3.3 Tracking Active Links 194
8.3.4 Tracking Registered Servers 194

Microsoft Windows DDESpy (DDESPY.EXE) is a monitoring application for dynamic data exchange (DDE) activity in your Microsoft Windows operating system.

This chapter describes how to use DDESpy. For more information about dynamic data exchange, see the *Microsoft Windows Programmer's Reference, Volume 1.*

DDESpy is a typical DDE monitoring application. Because DDE is a cooperative activity, DDE monitoring applications must follow certain guidelines for your Windows system to operate properly while they are in use. In particular, DDE monitoring applications should not perform DDE server or client communications—problems may arise when the monitoring application intercepts its own communications.

8.1 The Output Menu

DDESpy can display DDE information in a window or on your debugging terminal or can save the displayed information in a file for later use.

You use the Output menu to select where DDESpy is to send output. If you choose the File command, you must specify the name of an output file. After you have chosen the File command once, DDESpy prompts you for an output filename every time you restart the application.

From the Output menu, you can choose the Clear Screen command to clear the display window. You can choose the Mark command to add text to the display as a marker—for example, before a DDE event to make it easier to find the event in the output file.

8.2 The Monitor Menu

You use the Monitor menu to specify one or more types of DDE information that DDESpy is to display. The following information can be displayed:

- String handle data
- Sent DDE messages
- Posted DDE messages
- Callbacks
- Errors

The Dynamic Data Exchange Management Library (DDEML) passes information by using shared memory. The contents of the shared memory depend on the type of DDE transaction. Several structures have been defined to allow applications

using DDE to access the information in the shared memory. DDESpy displays the contents of the appropriate structure for the DDE activity being monitored.

8.2.1 Monitoring String-Handle Data

The DDEML uses the **MONHSZSTRUCT** structure to pass string-handle data. DDESpy displays the following information from this structure:

- Task (application instance)
- Time, in milliseconds, since you started Windows
- Activity type (create, destroy, or increment)
- String handle
- String contents

The following example shows a typical DDESpy display of string-handle data:

```
Task:0x94f, Time:519700, String Handle Created: c4a4(this is a test)
Task:0x94f, Time:526126, String Handle Created: c4aa(another test)
```

8.2.2 Monitoring Sent or Posted DDE Messages

The DDEML uses the **MONMSGSTRUCT** structure to send and post DDE messages. DDESPY displays the following information from this structure:

- Task
- Time
- Handle of receiving window
- Transaction type (sent or posted)
- Message type
- Handle of sending application
- Other message-specific information

The following example shows a typical DDESpy display of DDE message activity:

```
Task:0x8df Time:642402 hwndTo=0x38dc Message(Sent)=Initiate:
       hwndFrom=9224, App=0xc35d("Server")
       Topic=*
Task:0x94f Time:642457 hwndTo=0x2408 Message(Sent)=Ack:
       hwndFrom=9396, App=0xc35d("Server")status=c35d(fAck fBusy )
       Topic=Item=0xc361("System")
```

8.2.3 Monitoring Callbacks

The DDEML uses the **MONCBSTRUCT** structure to pass information to application callback functions. DDESpy displays the following information from this structure:

- Task
- Time
- Transaction type
- Exchanged-data format, if any
- Conversation handle
- String handles and their referenced strings
- Transaction-specific data

The following example shows a typical DDESPY display of callback activity:

```
Task:0x8df Time:2882628 Callback:
        Type=Advstart, fmt=0x1("CF_TEXT"), hConv=0xc24b4,
        hsz1=0xc361("System") hsz2=0xc4df("xxcall"), hData=0x0,
        lData1=0x83f0000, lData2=0x0
        return=0x0
```

8.2.4 Monitoring Errors

When an error occurs during a DDE transaction, the DDEML places the error value and associated information in a **MONERRSTRUCT** structure. DDESpy uses this structure to display the following information about the error:

- Task (the handle of the application that caused the error)
- Time
- Error value and name

8.3 Tracking Options

DDESPY can also display information about aspects of DDE communication in your Windows system:

- String handles
- Active conversations
- Active links
- Registered servers

You use the Track menu to specify which DDE activity DDESpy is to track. When you choose a command from the Track menu, DDESpy creates a separate window for the display of information in conjunction with the DDE functions. For each window created, DDESpy updates the displayed information as DDE activity occurs. Events that occurred prior to creation of the tracking window are not displayed in the tracking window.

DDESpy can sort the displayed information in the tracking window. If you select the heading for a particular column in the tracking window, DDESpy will sort the displayed information based on the column you selected. This can be useful if you are searching for a particular event or handle.

8.3.1 Tracking String Handles

Windows maintains a systemwide string table containing the string handles applications use in DDE transactions. To display the system string table so that the string, the string handle, and the string usage count are shown, choose the String Handles command from the Track menu.

8.3.2 Tracking Active Conversations

To see a display of all active DDE conversations in your Windows system, choose the Active Conversations command from the Track menu. The Active Conversations window shows the server name, the current topic, and the server and client handles for each active conversation.

8.3.3 Tracking Active Links

To see a display of all active DDE advise loops, choose the Active Links command from the Track menu. The Active Links window shows the server name, topic, item format, transaction type, client handle, and server handle for every active advise loop in your Windows system.

8.3.4 Tracking Registered Servers

Server applications use the **DdeNameService** function to register with the DDEML. When the DDEML receives the **DdeNameService** function call, it adds the server name and an instance-specific name to a list of registered servers. To see a list of registered servers, choose the Registered Servers command from the Track menu.

Viewing the Heap: Heap Walker

Chapter 9

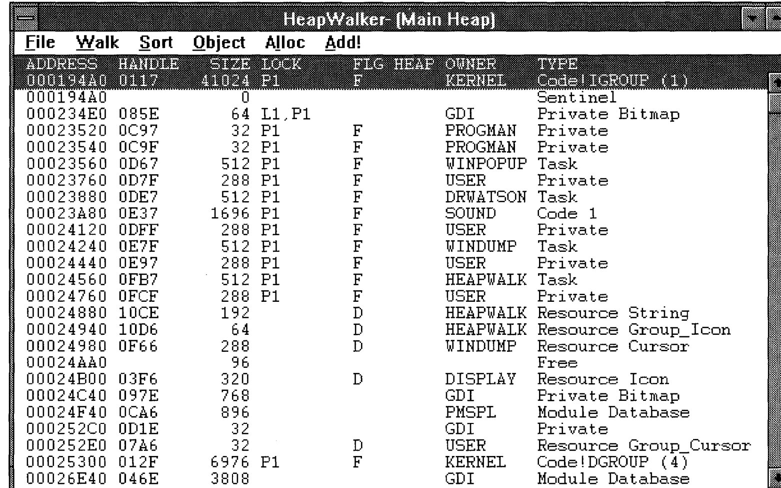
9.1 The Heap Walker Window 197
9.2 Performing File Operations: The File Menu 198
9.3 Walking the Heap: The Walk Menu 199
9.4 Sorting Memory Objects: The Sort Menu 199
9.5 Displaying Memory Objects: The Object Menu 200
    9.5.1 The Show Command 200
    9.5.2 The LocalWalk Commands 201
        9.5.2.1 Local Walk: The Heap Menu 202
        9.5.2.2 Local Walk: The Sort Menu 202
        9.5.2.3 Local Walk: The Add! Menu 203
9.6 Allocating Memory: The Alloc Menu 203
9.7 Determining Memory Size: The Add! Menu 203
9.8 Suggestions for Using Heap Walker 204
9.9 Related Topics 204

Microsoft Windows Heap Walker (HEAPWALK.EXE) lets you examine the global heap (the system memory that the Microsoft Windows operating system uses) and local heaps used by active applications and dynamic-link libraries (DLLs) in your Windows system. Heap Walker is useful for analyzing the effects your application has when it allocates memory from the global heap or when it creates user interface objects or graphics objects.

9.1 The Heap Walker Window

When you start Heap Walker, it scans the global heap and displays information about the allocated and free memory objects.

The following illustration shows a HeapWalker window.



Heap Walker displays the following information about each object:

Column heading	Information displayed
ADDRESS	Address of the memory object (displayed in hexadecimal format).
HANDLE	Handle of the memory object (displayed in hexadecimal format).
SIZE	Size of the memory object, in bytes (displayed in decimal format).

Column heading	Information displayed
LOCK	Lock count of the object. There are two types of lock counts: page-locked (P) and object-locked (L). Page-locked means that virtual memory will not be used to page the object (pieces of the object will not be written to the swap file); object-locked means the entire object will not be discarded.
FLG	D if the object is discardable; F if the object is fixed (not movable or discardable).
HEAP	Y if the object has a local heap.
OWNER	Owner of the object (name of the module that allocated the object).
TYPE	Type of object (code segment, data segment, resource, and so on). Heap Walker searches for symbol files and lists names for segments whenever corresponding symbol files are found.

9.2 Performing File Operations: The File Menu

The following commands are on the File menu:

Command	Action
Save	Saves in a file the current listing of objects in the heap. Heap Walker writes the first listing you save to the file HWG00.TXT and numbers subsequent files consecutively (HWG01.TXT, HWG02.TXT, and so on).
Exit	Closes Heap Walker.
About	Displays information about the current version of Heap Walker.

When you save a current heap listing to a file, Heap Walker includes all the information shown in the HeapWalker-[Main Heap] window, the number of free blocks in the heap, the size of the largest free block, the total free global heap space, and the following information about each module that has allocated memory from the global heap:

- Module name
- Number of discardable segments loaded in memory
- Number of bytes in discardable segments
- Number of bytes in nondiscardable segments
- Total number of bytes used by the module

9.3 Walking the Heap: The Walk Menu

The following commands are on the Walk menu:

Command	Action
Walk Heap	Displays all objects in the global heap.
Walk LRU List	Displays only discardable objects. Heap Walker lists objects from the least recently used to the most recently used. The object at the top of the list has been least recently used and, therefore, is most eligible for discarding.
Walk Free List	Displays only free blocks of memory.
GC(0) and Walk	Compacts the global heap, asking for 0 bytes, and then displays the heap.
GC(−1) and Walk	Attempts to discard all discardable objects and then displays the heap.
GC(−1) and Hit A:	Attempts to discard all discardable objects and then accesses drive A. This command is used to test critical error handling.
Set Swap Area	Resets the code fence. The code fence defines an area of memory reserved for discardable code.
Segmentation Test	Dumps the heap to a file called HWG*xx*.TXT and then compacts the heap.

9.4 Sorting Memory Objects: The Sort Menu

The Sort menu is useful for sorting memory objects in a variety of ways. The following commands are on the Sort menu:

Command	Action
Address	Sorts numerically by address.
Module	Sorts alphabetically by the owning module's name and sorts alphabetically by object type within each owner name.
Size	Sorts numerically by object size.
Type	Sorts alphabetically by object type and sorts alphabetically by owner name within each object type.
Refresh Seg Names	Searches for symbol files and lists segment names. This command can be used to list segment names for applications loaded after you start Heap Walker.

9.5 Displaying Memory Objects: The Object Menu

The Object menu is useful for viewing objects selectively. The following commands are on the Object menu:

Command	Action
Show	Displays the contents of a selected object in hexadecimal format and ASCII format. When possible for resources, this command displays the resource (such as an icon, menu, or dialog box).
Discard	Discards a selected object.
Oldest	Marks a selected object as the next candidate for discarding.
Newest	Marks a selected object as the last candidate for discarding.
LocalWalk	Displays the local heap of the currently selected object, if it has one.
LC(−1) and LocalWalk	Compacts the selected local heap and then displays the heap.
GDI LocalWalk	Displays the GDI local heap and provides information about the objects in the heap.
User LocalWalk	Displays the USER local heap and provides information about the objects in the heap.

9.5.1 The Show Command

To display a hexadecimal dump of an object, select the object in the HeapWalker-[Main Heap] window and either double-click the left mouse button or choose the Show command from the Object menu. In addition to the hexadecimal dump, the Show command can display the following kinds of resources:

- Bitmaps
- Cursors
- Dialog boxes
- Icons
- Menus

For example, the following illustration shows how the Show command displays the memory and icon associated with the selected memory object.

```
HeapWalker- (Main Heap)
File  Walk  Sort  Object  Alloc  Add!
ADDRESS  HANDLE  SIZE LOCK  FLG HEAP OWNER  TYPE
00022FC0 02A6   64        D        DISPLAY  Resource Group_Icon
0002F400 033E   32        D        DISPLAY  Resource Group_Icon
00074FE0 0296   64        D        DISPLAY  Resource Group_Icon
  Global Object - 0005E080 1946   672   D   DDEPRINT Resource Ic
  0000  10 00 10 00 20 00 20 00 04 00 04 01 00 00 00 00
  0010  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0030  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0040  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0050  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0060  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0005E080 1946  672        D        DDEPRINT Resource Icon
806963C0 03EE  672        D        DISPLAY  Resource Icon
0006CA60 03D6  672        D        DISPLAY  Resource Icon
806DB020 0406  672        D        DISPLAY  Resource Icon
00067340 03CE  320        D        DISPLAY  Resource Icon
806DAAE0 040E  768        D        DISPLAY  Resource Icon
000659C0 03C6  768        D        DISPLAY  Resource Icon
806DA5A0 0416  320        D        DISPLAY  Resource Icon
00052280 03E6  320        D        DISPLAY  Resource Icon
00023380 041E  67   Resource Icon   PLAY  Resource Icon
8068DEC0 03DE  76                   PLAY  Resource Icon
0007B8C0 03F6  76                   PLAY  Resource Icon
0007B780 03FE  32                   PLAY  Resource Icon
000523C0 14F6  67                   ATSON Resource Icon
000755C0 0B16  672        D        PROGMAN  Resource Icon
```

9.5.2 The LocalWalk Commands

You can choose the LocalWalk command from the Object menu to view the local heap for a selected object. You can also choose the GDI LocalWalk or User LocalWalk command to view the GDI or USER local heap, respectively, at any time. Local Walk windows show the following information:

Window heading	Information displayed
OFFSET	Offset of the object from the beginning of the heap. You can use this information to locate the contents of the object within the hexadecimal display of the heap.
HANDLE	Handle of the object.
SIZE	Size of the object, in bytes.
FLAGS	Whether the object is movable, fixed, or free.
LCK	Lock count for the object.
TYPE	Object type (shown only for GDI and USER heaps).

The following illustration shows a Local Walk window.

```
DDEPRINT Heap (Local Walk)
Heap  Sort  Add!
OFFSET HANDLE   SIZE FLAGS      LCK
 1B06              6 Fixed
 1B12             74 Fixed
 1B62            206 Fixed
 1C36             18 Fixed
 1C4E             98 Fixed
 1CB6            130 Fixed
 1D3E              6 Fixed
 1D4A            922 Free
 20EA   1CBA      34 Moveable
 2112   1CB6     514 Moveable
```

Windows allocates the first object in the local heap, so there are always at least two objects in a local heap.

9.5.2.1 Local Walk: The Heap Menu

Following are the commands on the Heap menu:

Command	Action
Info	Displays a message box showing the number of free, movable, and fixed objects; the number of bytes they use; the total number of allocated objects; and the number of bytes they use.
Save	Saves the Local Walk display to a file. The first file saved is named HWL00.TXT; subsequent files are numbered sequentially (HWL01.TXT, HWL02.TXT, and so on). The file contains all the information shown in the Local Walk window and a summary of local objects by type (free, movable, fixed, and total allocated).

9.5.2.2 Local Walk: The Sort Menu

Following are the commands on the Sort menu:

Command	Action
Address	Sorts the Local Walk display numerically by address.
Flags	Sorts the Local Walk display alphabetically by flags (fixed, free, or movable).
Size	Sorts the Local Walk display numerically by object size.

9.5.2.3 Local Walk: The Add! Menu

The Add! menu displays a message box showing the total number of bytes used by selected objects.

9.6 Allocating Memory: The Alloc Menu

The Alloc menu is useful for allocating memory for test purposes. You can allocate all free memory and then run your program to see how it behaves when no memory is available. You can free all or a specified part of the allocated memory.

The following commands are available from the Alloc menu:

Command	Action
Allocate All of Memory	Allocates all free memory. This command is useful for testing out-of-memory conditions in applications.
Free All	Frees memory allocated by the Allocate All of Memory command.
Free 1K	Frees 1K of the memory allocated by the Allocate All of Memory command.
Free 2K	Frees 2K of the memory allocated by the Allocate All of Memory command.
Free 5K	Frees 5K of the memory allocated by the Allocate All of Memory command.
Free 10K	Frees 10K of the memory allocated by the Allocate All of Memory command.
Free 25K	Frees 25K of the memory allocated by the Allocate All of Memory command.
Free 50K	Frees 50K of the memory allocated by the Allocate All of Memory command.
Free XK	Frees a specified number of kilobytes of the memory allocated by the Allocate All of Memory command. A dialog box is displayed, in which you can specify the number.

The last eight commands apply only to memory allocated when you chose the first command—it is not possible to free memory allocated by another program.

9.7 Determining Memory Size: The Add! Menu

The Add! menu on the Heap Walker menu bar adds the total number of bytes of selected memory objects. Opening this menu displays a dialog box that shows the number of selected segments and total segment sizes.

9.8 Suggestions for Using Heap Walker

One error that frequently occurs in applications is the failure to free memory objects when they are no longer needed. This can cause Windows to fail when one of its data segments grows beyond the 64K limit.

You can use Heap Walker to help determine if your application is not freeing memory objects. With Heap Walker, you can view changes in the sizes of all Windows data segments to observe the effect your application has on these segments.

To check how your application changes the sizes of the Windows data segments, follow these steps:

1. Make sure that your application does not generate fatal exits.
2. Start the debugging version of Windows.
3. Start Heap Walker, and note the sizes of the GDI and USER data segments. This establishes the reference for comparing the size of the data segments later.
4. From the Object menu, choose the GDI LocalWalk command to display the GDI Heap (Local Walk) window, which lists the different objects in the GDI data segment. Then choose the Save command from the Heap menu to copy this list to a file; the file will also contain a summary of GDI objects.
5. Run your application, and exercise it fully over a long period of time, noting the changes in the size of the GDI and USER data segments that Heap Walker displays as your application runs. While your application is running, repeat step 4 a number of times to take "snapshots" of the effect your application has on the GDI data segment.
6. Close your application, take a final snapshot of the GDI data segment, and note the total sizes of the GDI and USER data segments.

As you analyze the data that you've recorded, you should look for GDI objects that your application creates but does not delete when they are no longer needed.

9.9 Related Topics

For more information about heaps and memory management, see the *Microsoft Windows Guide to Programming.*

Analyzing Performance: Profiler

Chapter 10

10.1 Overview of Profiler 207
10.2 Preparing to Run Profiler 208
10.3 Using Profiler Functions 208
10.4 Sampling Code 209
10.5 Displaying Samples 209

Microsoft Windows Profiler is an analytical tool that helps you optimize the performance of Windows applications. Profiler lets you determine the amount of time the Microsoft Windows operating system spends executing sections of code.

Profiler analyzes applications running with Windows in 386 enhanced mode; however, it cannot analyze applications running with Windows in standard mode.

10.1 Overview of Profiler

Profiler contains the following:

- A sampling utility
- A reporting utility
- A set of functions your application can call

The sampling utility gathers information about the time spent between adjacent labels and records memory addresses of code. The utility is a special device driver, VPROD.386. To run Profiler, install VPROD.386 and then run Windows directly.

Profiler stores the information it gathers in a buffer. It writes the buffer to disk when Windows terminates, producing a CSIPS.DAT file and a SEGENTRY.DAT file in the directory that was your current directory when you started Windows. The CSIPS.DAT file contains statistical samplings of the code segment (CS) and instruction pointer (IP) registers. The SEGENTRY.DAT file contains information about the movement of code segments. Because code segments can be located at different physical addresses during the execution of the program, information from both the CSIPS.DAT and SEGENTRY.DAT files is required to prepare the profiling report.

After the sampling utility has finished gathering information, the SHOWHITS.EXE reporting utility organizes and displays the results.

With Profiler's functions, you start and stop examining code, manage the output of code samples, and get information about Profiler. All applications that Profiler examines must include the two functions that start and stop the sampling of code. Other Profiler functions are optional.

10.2 Preparing to Run Profiler

To profile an application running with Windows in 386 enhanced mode, you can use any system that is capable of running Windows in 386 enhanced mode.

In addition to ensuring that your system is compatible with Profiler, you must do the following:

1. Ensure that the Windows directory is defined in your PATH environment variable.
2. Include in your application at least the two mandatory Profiler functions **ProfStart** and **ProfStop**.

   ProfStart indicates when you want Profiler to start sampling code; **ProfStop** indicates when you want Profiler to stop sampling. Other Profiler functions are optional.
3. Compile your application. Then link the compiled code with the standard Windows libraries, using the appropriate command-line option to prepare a symbol map (.MAP) file that includes **PUBLIC** symbols. The .MAP file is required by Microsoft Symbol File Generator (MAPSYM). For information about how to create the .MAP file during linking, see the documentation that accompanied your linker. For more information about MAPSYM, see Chapter 5, "Advanced Debugging: 80386 Debugger."
4. Use MAPSYM to convert the .MAP file to a symbol (.SYM) file.

10.3 Using Profiler Functions

In addition to the mandatory **ProfStart** and **ProfStop** functions, Profiler includes functions that determine whether Profiler is installed, specify a rate for sampling, and control the output buffer. Following are the available Profiler functions:

Function	Description
ProfClear	Discards all buffered Profiler samples.
ProfFinish	Stops profile sampling and flushes profile buffer.
ProfFlush	Flushes the Profiler sampling buffer to disk.
ProfInsChk	Determines whether Profiler is installed.
ProfSampRate	Sets the Profiler sampling rate.
ProfSetup	Sets Profiler buffer size and sample quantity.
ProfStart	Starts Profiler sampling.
ProfStop	Stops Profiler sampling.

10.4 Sampling Code

To use the Profiler functions, you must first install VPROD.386, a virtual device driver. Your application can call the **ProfSetup** function to set the size of the output buffer (up to 1064K).

Profiler sampling uses memory that is otherwise available to Windows. Therefore, using Profiler may decrease the performance of the application you are analyzing. By specifying a small output buffer for Profiler, you can reduce the amount of memory used. However, a small output buffer may cause sample loss.

Profiler can write samples to disk only when Windows indicates it is safe to do so. When the sampling buffer is full, Profiler ignores additional samples until the buffer is flushed to disk. To minimize sample loss, either increase the buffer size or periodically call the **ProfFlush** function.

To profile applications, do the following:

1. Install the VPROD.386 driver by adding the following setting to the [386enh] section of your SYSTEM.INI file:

   ```
   device=vprod.386
   ```

2. Run Windows in 386 enhanced mode.
3. Run the application you want to profile.
4. When you have finished profiling your application, remove the SYSTEM.INI file setting you added in step 1.

10.5 Displaying Samples

To display the data Profiler gathers, run the SHOWHITS.EXE application from the MS-DOS command line. This reporting utility reads CSIPS.DAT, SEGENTRY.DAT, and .SYM files and then organizes and displays the data. The CSIPS.DAT and SEGENTRY.DAT files are located where the sampling utility placed them—that is, in the directory that was your current directory when you started Windows. To ensure that SHOWHITS.EXE can locate these files, either run SHOWHITS.EXE from the same directory or specify full paths for the CSIPS.DAT and SEGENTRY.DAT files. If the .SYM files are not in the current directory, use the **/i***path* option on the **showhits** command line to specify the directory or directories containing them.

SHOWHITS.EXE reads .SYM files to match instruction pointer samples with global symbols in the application. When you run SHOWHITS.EXE, the utility searches for .SYM files that contain symbolic names identical to the names of modules that Profiler sampled. Each match is called a hit. If the sampled program

is written in the C language, the symbolic names are typically function names. If the sampled program is written in assembly language, the symbolic names can be either procedure names or **PUBLIC** symbols within procedures.

SHOWHITS.EXE reports the number of times sampling occurred between adjacent symbols.

The syntax for the **showhits** command line is as follows:

showhits [**/i***path* [**/i***path* [...]]] [*cs_file*] [*seg_file*]

Following are the command-line options and parameters:

/i*path*
: Specifies one or more directories to search for .SYM files. SHOWHITS.EXE loads all .SYM files from the specified directories, regardless of their relevance to the application you are profiling. The default value is the current directory.

cs_file
: Specifies the full path of the CSIPS.DAT file. If no path is specified, SHOWHITS.EXE looks for the file in the current directory.

seg_file
: Specifies the full path of the SEGENTRY.DAT file. If no path is specified, SHOWHITS.EXE looks for the file in the current directory.

SHOWHITS.EXE displays information about hits, which are instruction pointer samples, in the following four categories:

Category	Description
Unrecognized segments	A list of instruction pointer values that occur within segments for which there are no symbols of module names. Unrecognized segments are typically code for device drivers, terminate-and-stay-resident (TSR) programs, and other code that Windows does not use.
Known segments	The number of hits that occur within known modules. Hits on known segments typically include counts for the application and counts for such Windows modules as KERNEL, GDI, and DISPLAY. Profiler also counts hits in MS-DOS and the read-only memory (ROM) basic input-and-output system (BIOS). In addition to displaying hits, SHOWHITS.EXE lists the total number of hits and the segment's percentage of total hits.
Breakdown	A detailed breakdown of the hits between labels of the modules for which SHOWHITS.EXE finds .SYM files. SHOWHITS.EXE also displays the total number of hits and the percentage of the total number.
Summary	A list of the top hits.

The following example illustrates a profiling-report display:

```
Here are the Hits for Unrecognized Segments

Here are the Hits for Known Segments

 0.3%        3  Hits on SYSTEM!
 0.5%        5  Hits on HELLO!
76.5%      786  Hits on DISPLAY!
11.3%      116  Hits on GDI!
11.5%      118  Hits on KERNEL!

   1028  TOTAL HITS

HELLO!_TEXT

 0.4%        4  Hits between labels _HelloPaint and _HelloInit
 0.1%        1  Hits between labels __cintDIV and __fptrap

Profiler Summary (Top 10 Hits):

 0.4%        4  HELLO! _TEXT! _HelloPaint - _HelloInit
 0.1%        1  HELLO! _TEXT! __cintDIV - __fptrap
```

Compressing and Decompressing Files

Chapter 11

11.1 Compressing Files: Compress 215
11.2 Decompressing Compressed Files: Expand 216

Microsoft File Compression Utility (Compress) and Microsoft File Expansion Utility (Expand) are two tools for the Microsoft Windows operating system. These utilities are useful when you prepare your application files for distribution and as part of the installation process for your application. Compress compresses files to a smaller size, so you can fit more files on a disk. Expand decompresses files after they have been compressed, expanding them to their original sizes. You run these utilities from the MS-DOS command line.

The following sections describe these utilities.

11.1 Compressing Files: Compress

Compress (COMPRESS.EXE) creates compressed versions of one or more files. The resulting files are typically 25 to 45 percent smaller than the original files.

Command-line syntax for Compress is as follows:

compress [**/?**][**/r**] *source destination*

Following are command-line options and parameters for Compress:

/?
: Displays information about how to use Compress.

/r
: Specifies that compressed files should be renamed.

source
: Specifies the source filename. The name can include a drive letter, a directory path, or both; and it can contain wildcards.

destination
: Specifies the destination. This parameter can consist of a directory (with optional drive letter), a filename, or any combination of the two.

If the *source* parameter contains wildcards and the *destination* parameter does not specify only a directory, the **/r** option must be used.

If the *destination* parameter does not contain a filename, Compress uses the filename or filenames specified by the *source* parameter when Compress copies the file or files to the location specified by the *destination* parameter.

11.2 Decompressing Compressed Files: Expand

Expand (EXPAND.EXE) decompresses files previously compressed by Compress. Expand restores these files to their original sizes.

Command-line syntax for Expand is as follows:

expand [**/?**][**/r**] *source destination*

Following are command-line options and parameters for Expand:

/?
Displays information about how to use Expand.

/r
Specifies that compressed files should be renamed.

source
Specifies the source filename. The name can include a drive letter, a directory path, or both; and it can contain wildcards.

destination
Specifies the destination. This parameter can consist of a directory (with optional drive letter), a filename, or any combination of the two.

If the *source* parameter contains wildcards and the *destination* parameter does not specify only a directory, the **/r** option must be used.

If the *destination* parameter does not contain a filename, Expand uses the filename or filenames specified by the *source* parameter when Expand copies the file or files to the location specified by the *destination* parameter.

The following example shows how to create decompressed versions of all the files on drive A, writing them to a directory on drive C:

```
expand a:*.* c:\mydir
```

Resource Compiler Diagnostic Messages

Appendix A

Alphabetic Reference 219

This appendix contains descriptions of diagnostic messages produced by Microsoft Windows Resource Compiler (RC). Many of these messages appear when RC is not able to compile resources properly. The descriptions in this appendix clarify the causes. The messages are listed in alphabetic order.

A capital V in parentheses (V) at the beginning of a message description indicates that the message is displayed only if RC is run with the **–V** (verbose) option. These messages are generally informational and do not necessarily indicate errors.

For information on the keywords and fields specified in this appendix, see the *Microsoft Windows Programmer's Reference, Volume 4.*

A

Accelerator Type required (ASCII or VIRTKEY)

The *type* parameter in the **ACCELERATORS** statement must contain either the **ASCII** or **VIRTKEY** value.

B

BEGIN expected in Accelerator Table

An **ACCELERATORS** statement was not followed by the **BEGIN** keyword.

BEGIN expected in Dialog

A **DIALOG** statement was not followed by the **BEGIN** keyword.

BEGIN expected in menu

A **MENU** statement was not followed by the **BEGIN** keyword.

BEGIN expected in RCData

An **RCDATA** statement was not followed by the **BEGIN** keyword.

BEGIN expected in String Table

A **STRINGTABLE** statement was not followed by the **BEGIN** keyword.

BEGIN expected in VERSIONINFO resource

A **VERSIONINFO** statement was not followed by the **BEGIN** keyword.

Bitmap file *resource-file* **is not in** *version-number* **format.**

Use Microsoft Image Editor (IMAGEDIT.EXE) to convert version 2.*x* resource files to the version 3.1 format.

C

Cannot Re-use String Constants

You are using the same value twice in a **STRINGTABLE** statement. Make sure that you have not mixed overlapping decimal and hexadecimal values.

Control Character out of range [A – Z]

A control character in the **ACCELERATORS** statement is invalid. The character following the caret (^) must be in the range A through Z.

Copying segment *id* (*size* **bytes)**

(V) Microsoft Windows Resource Compiler (RC) is copying the specified segment to the executable (.EXE) file.

Could not find RCPP.EXE

The preprocessor (RCPP.EXE) must be in the current directory or in a directory specified in the PATH environment variable.

Could not open *in-file-name*

Microsoft Windows Resource Compiler (RC) could not open the specified file. Make sure that the file exists and that you typed the filename correctly.

Couldn't open *resource-name*

Microsoft Windows Resource Compiler (RC) could not open the specified file. Make sure that the file exists and that you typed the filename correctly.

Creating *resource-name*

(V) Microsoft Windows Resource Compiler (RC) is creating a new binary resource (.RES) file.

E

Empty menus not allowed

An **END** keyword appears before any menu items are defined in the **MENU** statement. Empty menus are not permitted by Microsoft Windows Resource Compiler (RC). Make sure that you do not have any opening quotation marks within the **MENU** statement.

END expected in Dialog

The **END** keyword must appear at the end of a **DIALOG** statement. Make sure that there are no opening quotation marks left from the preceding statement.

END expected in menu

The **END** keyword must appear at the end of a **MENU** statement. Make sure that there are no mismatched **BEGIN** and **END** statements.

Error Creating *resource-name*

Microsoft Windows Resource Compiler (RC) could not create the binary resource specified (.RES) file. Make sure that it is not being created on a read-only drive. Use the **/V** option to find out whether the file is being created.

Errors occurred when linking file.

The linker failed. For more information, see the documentation for your linker.

EXE file too large; relink with higher /ALIGN value

The executable (.EXE) file is too large. Relink the .EXE file with a larger value. For more information, see the documentation for your linker.

Expected Comma in Accelerator Table

Microsoft Windows Resource Compiler (RC) requires a comma between the *event* and *idvalue* parameters in the **ACCELERATORS** statement.

Expected control class name

The *class* parameter of a **CONTROL** statement in the **DIALOG** statement must be one of the following control types: **BUTTON**, **COMBOBOX**, **EDIT**, **LISTBOX**, **SCROLLBAR**, **STATIC**, or user-defined. Make sure that the class is spelled correctly.

Expected font face name

The *typeface* parameter of the **FONT** statement in the **DIALOG** statement must be an ASCII character string enclosed in double quotation marks. This parameter specifies the name of a font.

Expected ID value for Menuitem

The **MENU** statement must contain a **MENUITEM** statement, which contains either an integer or a symbolic constant in the *MenuID* parameter.

Expected Menu String

Each **MENUITEM** and **POPUP** statement must contain a *text* parameter. This parameter is a string enclosed in double quotation marks that specifies the name of the menu item or pop-up menu. A **MENUITEM SEPARATOR** statement requires no quoted string.

Expected numeric command value

Microsoft Windows Resource Compiler (RC) was expecting a numeric *idvalue* parameter in the **ACCELERATORS** statement. Make sure that you have used a **#define** constant to specify the value and that the constant used is spelled correctly.

Expected numeric constant in string table

A numeric constant, defined in a **#define** statement, must immediately follow the **BEGIN** keyword in a **STRINGTABLE** statement.

Expected numeric point size

The *pointsize* parameter of the **FONT** statement in the **DIALOG** statement must be an integer point-size value.

Expected Numerical Dialog constant

A **DIALOG** statement requires integer values for the *x, y, width*, and *height* parameters. Make sure that these values, which are included after the **DIALOG** keyword are not negative.

Expected String in STRINGTABLE

A string is expected after each numeric *stringid* parameter in a **STRINGTABLE** statement.

Expected String or Constant Accelerator command

Microsoft Windows Resource Compiler (RC) was not able to determine which key was being set up for the accelerator. The *event* parameter in the **ACCELERATORS** statement might be invalid.

Expected VALUE, BLOCK, or END keyword.

The **VERSIONINFO** structure requires a **VALUE**, **BLOCK**, or **END** keyword.

Expecting number for ID

A number is expected for the *id* parameter of a control statement in the **DIALOG** statement. Make sure that you have a number or a **#define** statement for the control identifier.

Expecting quoted string for key

The key string following the **BLOCK** or **VALUE** keyword should be enclosed in double quotation marks.

Expecting quoted string in dialog class

The *class* parameter of the **CLASS** statement in the **DIALOG** statement must be an integer or a string enclosed in double quotation marks.

Expecting quoted string in dialog title

The *captiontext* parameter of the **CAPTION** statement in the **DIALOG** statement must be an ASCII character string, enclosed in double quotation marks.

F

Fast-load area is [*size*] bytes at offset 0x[*address*]

(V) This is the size, in bytes, of all the following segments:

- Segments with the **PRELOAD** attribute
- Segments with the **DISCARDABLE** attribute
- Code segments that contain the entry point, **WinMain**
- Data segments (which should not be discardable)

To disable fast loading, use the **–k** option. Fast loading is the placement of segments in a contiguous area in the executable (.EXE) file for quicker loading. The offset is from the the beginning of the file.

File not created by LINK

You must create the executable (.EXE) file with an appropriate version of the linker.

File not found: *filename*

The file specified in the **rc** command was not found. Make sure that the file has not been moved to another directory and that the filename or path is typed correctly.

Font names must be ordinals

The *pointsize* parameter in the **FONT** statement must be an integer, not a string.

I

Insufficient memory to spawn RCPP.EXE

There wasn't enough memory to run the preprocessor (RCPP.EXE). Try disabling any memory-resident software that might be taking up too much memory. To verify the amount of memory you have, use the **chkdsk** command.

Invalid Accelerator

An *event* parameter in the **ACCELERATORS** statement was not recognized or was more than two characters long.

Invalid Accelerator Type (ASCII or VIRTKEY)

The *type* parameter in the **ACCELERATORS** statement must contain either the **ASCII** or **VIRTKEY** value.

Invalid control character

A control character in the **ACCELERATORS** statement is invalid. A valid control character consists of a caret (^) followed by a single letter.

Invalid Control type

The control statement in a **DIALOG** statement must be one of the following: **CHECKBOX**, **COMBOBOX**, **CONTROL**, **CTEXT**, **DEFPUSHBUTTON**, **EDITTEXT**, **GROUPBOX**, **ICON**, **LISTBOX**, **LTEXT**, **PUSHBUTTON**, **RADIOBUTTON**, **RTEXT**, or **SCROLLBAR**.

Invalid directive in preprocessed RC file

The specified filename has an embedded newline character.

Invalid .EXE file

The executable (.EXE) file is invalid. Make sure that the linker created it correctly and that the file exists.

Invalid switch, *option*

An option used was invalid. For a list of the command-line options, use the **rc –?** command.

Invalid type

The resource type was not among the types defined in the include file.

Invalid usage. Use rc –? for Help

Make sure that you have at least one filename to work with. For a list of the command-line options, use the **RC –?** command.

I/O error reading file.

Read failed. Since this is a generic routine, no specific filename is supplied.

I/O error seeking in file

Seeking in file failed. Since this is a generic routine, no specific filename is supplied.

I/O error writing file.

Write failed. Since this is a generic routine, no specific filename is supplied.

N

No executable filename specified.

The **–FE** option was used, but no executable (.EXE) file was specified.

No resource binary filename specified.

The **–FO** option was used, but no binary resource (.RES) file was specified.

Not a Microsoft Windows format .EXE file

Make sure that the linker created the executable (.EXE) file correctly and that the file exists.

O

Old DIB in ***resource-name*****. Pass it through IMAGEDIT.**

The resource file specified is not compatible with Windows version 3.1. Make sure you have read and saved this file using the latest version of Microsoft Image Editor (IMAGEDIT.EXE). (Image Editor has replaced SDK Paint.)

Out of far heap memory

There wasn't enough memory. Try disabling any memory-resident software that might be taking up too much space. To find out how much memory you have, use the **chkdsk** command.

Out of memory, needed ***n*** **bytes**

Microsoft Windows Resource Compiler (RC) was not able to allocate the specified amount of memory.

R

RC: Invalid swap area size: –S ***string***

Invalid swap area size. Check your syntax for the **–S** option on the command line for the Microsoft Windows Resource Compiler (RC). The following examples show acceptable command lines:

```
RC  S123
RC  S123K  ; where K is kilobytes
RC  S123p  ; where p is paragraphs
```

RC: Invalid switch: ***option***

An option used was invalid. For a list of the command-line options, use the **rc –?** command.

RC: RCPP.ERR not found

The RCPP.ERR file must be in the current directory or in a directory specified in the PATH environment variable.

RC terminated by user

A CTRL+C key combination was pressed, exiting Microsoft Windows Resource Compiler (RC).

RC terminating after preprocessor errors

For information about preprocessor errors, see the documentation for the preprocessor.

RCPP.EXE command line greater than 128 bytes

The command line for the preprocessor (RCPP.EXE) was too long.

RCPP.EXE is not a valid executable

The preprocessor (RCPP.EXE) may have been altered. Try copying the file again from the Microsoft Windows Software Development Kit (SDK) disks.

Resource file *resource-name* is not in version-number format.

Make sure your icons and cursors have been read and saved using the latest version of Microsoft Image Editor (IMAGEDIT.EXE).

Resources will be aligned on *number* byte boundaries

(V) The alignment is determined by an option on the command line for the linker.

S

Sorting preload segments and resources into fast-load section

(V) Microsoft Windows Resource Compiler (RC) is sorting the preloaded segments into a contiguous area in the executable (.EXE) file (the fast-load section) so that they can be loaded quickly.

T

Text string or ordinal expected in Control

The *text* parameter of a **CONTROL** statement in the **DIALOG** statement must be either a text string or an ordinal reference to the type of control that is expected. If using an ordinal, make sure that you have a **#define** statement for the control.

The EXETYPE of this program is not Windows

The **EXETYPE WINDOWS** statement did not appear in the module-definition (.DEF) file. Since the linker might make optimizations that are not appropriate for Windows, the **EXETYPE WINDOWS** statement must be specified.

U

Unable to create *destination*

Microsoft Windows Resource Compiler (RC) was not able to create the destination file. Make sure that there is enough disk space.

Unable to open *exe-file*

Microsoft Windows Resource Compiler (RC) could not open the executable (.EXE) file. Make sure that the linker created it correctly and that the file exists.

Unbalanced Parentheses

Make sure that you have closed every opening parenthesis in the **DIALOG** statement.

Unexpected value in RCData

The values for the *raw-data* parameter in the **RCDATA** statement must be integers or strings, separated by commas. Make sure that you did not leave out a comma or a quotation mark around a string.

Unexpected value in value data

A statement contained information whose format or size was different from the expected value for that parameter.

Unknown DIB header format

The device-independent bitmap (DIB) header is not a **BITMAPCOREHEADER** or **BITMAPINFOHEADER** structure.

Unknown error spawning RCPP.EXE

For an unknown reason, the preprocessor (RCPP.EXE) has not started. Try copying the file again from the SDK disks and use the chkdsk command to verify the amount of available memory.

Unknown Menu SubType

The *item-definitions* parameter of the **MENU** statement can contain only **MENUITEM** and **POPUP** statements.

Unrecognized VERSIONINFO field; BEGIN or comma expected

The format of the information following a **VERSIONINFO** statement is incorrect.

V

Version WORDs separated by commas expected

Values in an information block for a **VERSIONINFO** statement should be separated by commas.

W

Warning: ASCII character not equivalent to virtual key code

An invalid virtual-key code exists in the **ACCELERATORS** statement. The ASCII values for some characters (such as *, ^, or &) are not equivalent to the virtual-key codes for the corresponding keys. (In the case of the asterisk ([*]), the virtual-key code is equivalent to the ASCII value for 8, the numeric character on the same key. Therefore, the statement **VIRTKEY** `'*'` is invalid. For information about these values, see the *Microsoft Windows Programmer's Reference, Volume 3*.

Warning: SHIFT or CONTROL used without VIRTKEY

The **ALT**, **SHIFT**, and **CONTROL** options apply only to virtual keys in the **ACCELERATORS** statement. Make sure that the **VIRTKEY** option is used with one of these other options.

Warning: *string* **segment** *number* **set to PRELOAD**

Microsoft Windows Resource Compiler (RC) displays this warning when it copies a segment that must be preloaded but is not marked **PRELOAD** in the module-definition (.DEF) file for the linker. All nondiscardable segments should be preloaded, including automatic data segments, fixed segments, and the entry point of the code (**WinMain**). (The attributes of code segments are set by the .DEF file.)

Writing resource *resource-name or ordinal-id. resource type* **(***resource size***)**

(V) Microsoft Windows Resource Compiler (RC) is writing the resource name or ordinal identifier, followed by a period and the resource type and size, in bytes.

Help Compiler Error Messages

Appendix B

B.1 Interpreting Error Messages 231
B.2 Error Message Categories 231
B.3 File Errors 232
B.4 Project-File Errors 233
B.5 Macro Errors 237
B.6 Context-String Errors 237
B.7 Topic-File Errors 239
B.8 Miscellaneous Errors 240

This appendix lists the error messages displayed by the Microsoft Help Compiler when it encounters errors in building a help file. Whenever possible, the compiler displays the name of the file that contains the error, as well as the number used to identify the specific line of the project file or the topic that produced the error. Since topics are not actually numbered, the topic number given with an error message refers to that topic's sequential position in the topic file.

B.1 Interpreting Error Messages

The Microsoft Help Compiler displays either warning or fatal-error messages. A warning message indicates a problem during compilation that was not severe enough to prevent the help file from being created. Microsoft Windows Help should be able to open the file but may encounter problems when displaying some topics. A fatal error indicates a problem that prevents the compiler from creating a help file. The compiler always reports fatal errors, regardless of the current warning level or reporting option.

While the Microsoft Help Compiler processes the project file, it ignores lines that contain errors and attempts to continue. This means that errors encountered early in the file may result in many more errors being reported as the compiler continues.

When the Microsoft Help Compiler processes topic files, it reports any errors it encounters and, if the errors are not fatal, compilation continues. A single error in a topic file may result in more than one error message being displayed by the compiler. For example, a typographic mistake in a topic's context string will cause an error to be reported every time the compiler encounters a reference to the correct topic identifier.

B.2 Error Message Categories

Error-message numbers have four digits; the first one or two of those digits identify the message category. The message-number prefixes and the categories they identify are defined as follows:

Prefix	Error
1	Problems with files used to build the help file
2	Problems with the project file
30–31	Problems with build tags or build-tag expressions
35–36	Problems with Help macros
40–41	Problems with context strings

Prefix	Error
42–45	Problems with footnotes
46–47	Problems with the topic file
5	Other problems

B.3 File Errors

The following messages result from problems with files used to build a help file. A description is given for messages that are not self-explanatory.

Number	File error message
1019	**Project file extension cannot be .HLP or .PH.**
1030	**File name exceeds limit of 259 characters.**
	The combined length of the path and filename must not be more than the MS-DOS limit of 259 characters.
1079	**Out of file handles.**
	The compiler does not have enough available file handles to continue. If possible, increase the **FILES** setting in the CONFIG.SYS file.
1100	**Cannot open file** *filename*: **permission denied.**
	Requested files must have at least read privilege to be opened.
1150	**Cannot overwrite file** *filename*.
	Files with the read-only attribute cannot be overwritten.
1170	**File** *filename* **is a directory.**
	A directory in the project directory has the same name as the requested help file.
1190	**Cannot use reserved DOS file name** *filename*.
	Do not use reserved MS-DOS filenames, such as COM1, LPT2, or PRN, when specifying topic or other data files.
1230	**File** *filename* **not found.**
	The specified file could not be found or is unreadable.
1292	**File** *filename* **is not a valid bitmap.**
	The specified bitmap file could not be found or is not in a recognizable bitmap format.
1319	**Disk full.**
1513	**Bitmap name** *filename* **duplicated.**
	The [BITMAPS] section contains duplicate bitmap names. The compiler uses the first occurrence of the name.
1536	**Not enough memory to compress bitmap** *filename*.
	The specified bitmaps cannot be compressed due to insufficient memory.

B.4 Project-File Errors

The following messages result from errors in the Help project file (with the .HPJ filename extension) used to build a help file. A description is given for messages that are not self-explanatory.

Number	Project-file error message
2010	**Include statements nested more than 5 deep.** The **#include** statement on the specified line has exceeded the maximum of five include levels.
2030	**Comment starting at line** *linenumber* **of file** *filename* **unclosed at end of file.** The compiler has unexpectedly come to the end of the project file. There may be an open comment in the project file or in an include file.
2050	**Invalid #include syntax.** The **#include** statement requires a filename.
2091	**Bracket missing from section heading** [*sectionname*]**.**
2111	**Section heading missing.** The section heading on the specified line is not complete. This error is also reported if the first entry in the project file is not a section heading.
2131	**Invalid OPTIONS syntax: 'option=value' expected.**
2141	**Invalid ALIAS syntax: 'context=context' expected.**
2151	**Incomplete line in** [*sectionname*] **section**
2171	**Unrecognized text.**
2191	**Section heading** [*sectionname*] **unrecognized.**
2214	**Line in .HPJ file exceeds length limit of 2047 characters.**
2273	**[OPTIONS] should precede [FILES] and [BITMAPS] for all options to take effect.** The [OPTIONS] section should be the first section in the project file. Also, if the **ERRORLOG** option is used, that option should be the first line in the [OPTIONS] section.
2291	**Section** *sectionname* **previously defined.** The compiler ignores the lines under the duplicated section and continues from the next valid section heading.
2305	**No valid files in [FILES] section.** The file section is empty or contains only invalid files.
2322	**Context string** *context_name* **cannot be used as alias string.** A context string that has been assigned an alias cannot be used later as an alias for another context string. That is, you cannot map **a=b** and then **c=a** in the [ALIAS] section. The compiler ignores the attempted reassignment.

Number	Project-file error message
2331	**Context number already used in [MAP] section.** The context number on the specified line in the project file was previously mapped to a different context string.
2341	**Invalid or missing context string.** The specified line is missing a context string before an equal sign.
2351	**Invalid context identification number.** The context number on the specified line is empty or contains invalid characters.
2362	**Context string** *context_name* **already assigned an alias.** A context string can have only one alias. That is, you cannot map **a=b** and then **a=c** in the [ALIAS] section. The specified context string has already been assigned an alias in the [ALIAS] section. The compiler ignores the attempted reassignment.
2372	**Alias string** *aliasname* **already assigned.** You cannot alias an alias. That is, an alias string cannot, in turn, be assigned another alias. You cannot map **a=b** and then **b=c** in the [ALIAS] section. The compiler ignores the attempted reassignment.
2391	**Limit of 6 window definitions exceeded.** The maximum number of window definitions is one main-window definition and five secondary-window definitions.
2401	**Window maximization state must be 0 or 1.** The *sizing* parameter in a window definition must be either zero or 1.
2411	**Invalid syntax in window color.** A window color in a window definition consists of three decimal numbers enclosed in parentheses and separated by commas.
2421	**Invalid window position.** The window position in a window definition consists of four decimal numbers enclosed in parentheses and separated by commas.
2431	**Missing quote in window caption.** The window caption in a window definition must be enclosed in quotation marks.
2441	**Window name** *windowname* **is too long.** The window name exceeds the maximum length of 8 characters.
2451	**Window position value out of range 0...1023.** One or more of the window-position coordinates exceed the maximum limit of 1023.
2461	**Window name missing.** A window definition in the project file is missing the window name.
2471	**Invalid syntax in [WINDOWS] section.**

Number	Project-file error message
2481	**Secondary-window position required.** A window definition for a secondary window must specify the four window-position parameters.
2491	**Duplicate window name** *windowname.* Window names must be unique.
2501	**Window caption** *windowcaption* **exceeds limit of 50 characters.**
2511	**Unrecognized option** *optionname* **in [OPTIONS] section.**
2532	**Option** *optionname* **previously defined.** The compiler ignores the attempted redefinition.
2550	**Invalid path** *pathname* **in** *optionname* **option.** The compiler cannot find the path specified by the **ROOT** or **BMROOT** option. The compiler uses the current working directory.
2570	**Path in** *optionname* **option exceeds** *number of* **characters.** The specified root path exceeds the maximum limit for MS-DOS. The compiler ignores the path and uses the current working directory.
2591	**Invalid MAPFONTSIZE option.** The font range syntax used is invalid. A font range consists of a low and high point size, separated by a hyphen (-).
2612	**Maximum of 5 font ranges exceeded.** The compiler ignores additional ranges.
2632	**Current font range overlaps previously defined range.** The compiler ignores the second mapping.
2651	**Font name exceeds limit of 20 characters.**
2672	**Unrecognized font name** *fontname* **in FORCEFONT option.** The compiler ignores the font name and uses the default Helvetica font.
2691	**Invalid MULTIKEY syntax.** The **MULTIKEY** option must specify a single capital letter other than the letter *K*.
2711	**Maximum of 5 keyword tables exceeded.** The compiler ignores the additional tables.
2732	**Character already used.** A character used for indicating the keyword table was previously used. The compiler ignores the line.
2752	**Characters 'K' and 'k' cannot be used.** These characters are reserved for Help's standard keyword table.
2771	**REPORT option must be 'ON' or 'OFF'.**
2811	**OLDKEYPHRASE option must be 'ON' or 'OFF'.**
2832	**COMPRESS option must be 'OFF', 'MEDIUM' or 'HIGH'.**
2842	**OPTCDROM option must be 'TRUE' or 'FALSE'.**

Number	Project-file error message
2852	**Invalid TITLE option.** The **TITLE** option defines a string that is empty or contains more than 32 characters.
2872	**Invalid LANGUAGE option.** You have specified an ordering that is not supported by the compiler. The compiler uses English sorting order.
2893	**Warning option must be 1, 2, or 3.** The compiler uses full reporting (level 3).
2911	**Invalid icon file** *filename*. The compiler cannot find the icon file specified in the **ICON** option, or the file is not a valid icon file.
2932	**Copyright string exceeds limit of 50 characters.** The maximum length of the copyright string in the About box is limited to 50 characters.
3011	**Maximum of 32 build tags exceeded.** The compiler ignores the additional tags.
3031	**Build tag length exceeds 32 characters.** The compiler ignores the build tag.
3051	**Build tag** *tagname* **contains invalid characters.** Build tags can contain only alphanumeric characters or the underscore (_) character.
3076	**[BUILDTAGS] section missing.** The **BUILD** option declared a conditional build, but there is no [BUILDTAGS] section in the project file. The compiler includes all topics in the build.
3096	**Build expression too complex.** The build expression has too many expressions ("~", "|", or "&") or is too deeply nested.
3116	**Invalid build expression.** The syntax used in the build expression on the specified line contains one or more logical or syntax errors.
3133	**Duplicate build tag in [BUILDTAGS] section.**
3152	**Build tag** *tagname* **not defined in [BUILDTAGS] section.** The specified build tag has been assigned to a topic but not declared in the project file. The compiler ignores the tag for the topic.
3178	**Build expression missing from project file.** The topics have build tags, but there is no **build** expression in the project file. The compiler includes all topics in the build.

B.5 Macro Errors

The following messages result from errors in the use of Help macros in footnotes, hot spots, and the [CONFIG] section of the Help project file. A description is given for messages that are not self-explanatory.

Number	Macro error message
3511	**Macro** *macrostring* **exceeds limit of 254 characters.**
3532	**Undefined function in macro** *macroname.* The specified macro is not on the list of macros supported by the compiler, nor is it specified in the **RegisterRoutine** macro. The compiler passes the macro to the help file, however.
3552	**Undefined variable in macro** *macroname.*
3571	**Wrong number of parameters to function in macro** *macroname.*
3591	**Syntax error in macro** *macroname.*
3611	**Function parameter type mismatch in macro** *macroname.* There is a type mismatch (string or numeric) in the function call.
3631	**Bad macro prototype.** The prototype string passed to the **RegisterRoutine** macro is invalid.
3652	**Empty macro string.** The ! footnote or a hidden text starting with "!" does not contain a macro.
3672	**Macro** *macroname* **nested too deeply.** Macro strings that contain macro strings as parameters may not nest more than three deep.

B.6 Context-String Errors

The following messages are caused by problems with context-string footnotes or context strings specified in jumps or in Help project-file options. A description is given for messages that are not self-explanatory.

Number	Context-string error messages
4011	**Context string** *contextname* **already used.** The specified context string was previously assigned to another topic. The compiler ignores the latter string and the topic has no identifier.
4031	**Invalid context string** *contextname.* The context string footnote contains non-alphanumeric characters or is empty. The compiler does not assign the topic an identifier.

Number	Context-string error messages
4056	**Unresolved context string specified in CONTENTS option.** The Contents topic defined in the project file could not be found. The compiler uses the first topic in the build as the Contents topic.
4072	**Context string exceeds limit of 255 characters.** The compiler ignores the context string.
4098	**Context string(s) in [MAP] section not defined in any topic.** The compiler cannot find a context string listed in the [MAP] section in any of the topics in the build.
4113	**Unresolved jump or popup** *contextname*. The specified topic contains a context string that identifies a nonexistent topic.
4131	**Hash conflict between** *contextname* **and** *contextname*. The hash algorithm has generated the same hash value for both of the listed context strings. Change one of the context strings and recompile.
4151	**Invalid secondary window name** *windowname*. The window name for the secondary window is "main" or another disallowed member name.
4171	**Cannot use secondary window with popup.** The hidden text defining the pop-up identifier contains a secondary-window name.
4196	**Jumps and lookups not verified.** Due to low memory conditions, the build continues without verifying the validity of jumps and popups. (The reference to "lookups" in the error message is incorrect.)
4211	**Footnote text exceeds limit of 1023 characters.** Footnote text cannot exceed the limit of 1,023 characters. The compiler ignores the footnote.
4231	**Footnote text missing.** The specified topic contains a footnote that has no characters.
4251	**Browse sequence not in first paragraph.** The browse-sequence footnote is not in the first paragraph of the topic. The compiler ignores the browse sequence.
4272	**Empty browse sequence string.** The browse-sequence footnote for the specified topic contains no sequence characters.
4292	**Missing sequence number.** A browse-sequence number ends in a colon (:) for the specified topic. Remove the colon or enter a "minor" sequence number and then recompile.

Number	Context-string error messages
4312	**Browse sequence already defined.** A browse-sequence footnote already exists for the specified topic. The compiler ignores the latter sequence.
4331	**Title not in first paragraph.** The title footnote ($) is not in the first paragraph of the topic. The topic will not have a topic title string.
4352	**Empty title string.** The title footnote for the specified topic contains no characters. The compiler does not assign the topic a title.
4372	**Title defined more than once.** There is more than one title footnote in the specified topic. The compiler uses the first title string.
4393	**Title exceeds limit of 128 characters.** The compiler ignores the additional characters.
4412	**Keyword string exceeds limit of 255 characters.**
4433	**Empty keyword string.** There are no characters in the keyword footnote.
4452	**Keyword(s) defined without title.** The topic has a keyword assigned to it, but no title.
4471	**Build tag footnote not at beginning of topic.** The build-tag footnote marker, if used, must be the first character in the topic.
4492	**Build tag exceeds limit of 32 characters.** The compiler ignores the tag for the topic.
4551	**Entry macro not in first paragraph.** The ! footnote (for executing a macro) is not in the first paragraph of the topic. The compiler ignores the macro.

B.7 Topic-File Errors

The following messages result from problems in rich-text format (RTF) formatting in one or more topic files. A description is given for messages that are not self-explanatory.

Number	Topic-file error message
4616	**File *filename* is not a valid RTF topic file.**
4639	**Error in file *filename* at byte offset 0x*offset*.** The specified file contains unrecognized RTF at that byte offset.
4649	**File *filename* contains more than 32767 topics.**

Number	Topic-file error message
4652	**Table formatting too complex.** The compiler encountered a table with borders, shading, or right justification.
4662	**Side by side paragraph formatting not supported.** The side-by-side paragraph formatting is not supported in Microsoft Windows Help 3.1.
4671	**Table contains more than 32 columns.**
4680	**Font** *fontname* **in file** *filename* **not in RTF font table.** The compiler uses the default system font.
4692	**Unrecognized graphic format.** The compiler supports only Windows bitmaps, Windows metafiles, segmented graphics, and multi-resolution graphics. The compiler ignores the graphic.
4733	**Hidden page break.** A page break is a part of the hidden text. A page break formatted as hidden text will not separate two topics.
4753	**Hidden paragraph.** A paragraph marker is part of the hidden text. The compiler ignores the paragraph marker.
4763	**Hidden carriage return.** A carriage return is part of the hidden text. The compiler ignores the carriage return.
4774	**Paragraph exceeds limit of 64K.** A single paragraph has more than 64K of text or 64K of graphics. (This limit does not include graphics stored separately from the data, using the **bmc**, **bml**, or **bmr** statements.)
4792	**Non-scrolling region defined after scrolling region.** A **\keepn** statement precedes a paragraph that is not the first paragraph in the topic. The compiler ignores the statement; the paragraph is treated as regular text and is part of the regular topic text.
4813	**Non-scrolling region crosses page boundary.** A **\pard** statement must appear before the **\page** statement in a topic containing a **\keepn** statement.

B.8 Miscellaneous Errors

The following messages are caused by conditions such as MS-DOS file errors or out-of-memory conditions. A description is given for messages that are not self-explanatory.

Number	Error message
5035	**File** *filename* **not created.** There are no topics to compile or the build expression is false for all topics. The compiler does not create a help file.
5059	**Not enough memory to build help file.** To free memory, unload any unneeded applications, device drivers, and memory-resident programs.
5075	**Help Compiler corrupted. Please reinstall HC.EXE.** Virus-checking code has detected a corruption in the compiler. Reinstall the compiler.
5098	**Using old key-phrase table.** Maximum compression can result only by deleting the .PH file before each recompilation of the Help topics or by setting the **OLDKEY-PHRASE** option to "0".
5115	**Write failed.** Write-to-disk operation failed. Contact Microsoft Product Support Services.
5139	**Aborted by user.** Compilation was terminated when the user pressed CTRL+C.

Windows Debugging Version

Appendix C

C.1 Debugging Programs 245
C.1.1 Logging Debugging Messages 246
C.1.1.1 Settings Command 246
C.1.1.2 Alloc Break Command 247
C.1.2 Interpreting Debugging Messages 247
C.2 Debugging Functions and the WINDEBUGINFO Structure 249
C.2.1 WIN.INI Debugging Options 250
C.3 Debugging Messages 251
C.4 Common Programming Errors 254

The debugging version of the Microsoft Windows operating system generates diagnostic messages whenever it encounters an error that would otherwise cause the system to fail. You use the debugging version by itself or in conjunction with a debugger to debug Windows applications and dynamic-link libraries (DLLs). The debugging functions described in this appendix are not available in the retail version of the system: The API elements exist, but they have no effect. However, the retail version of Windows version 3.1 contains parameter-validation capabilities that an application can use with the Tool Helper library (TOOLHELP.DLL) to retrieve system errors and information about invalid parameters. For more information about the Tool Helper library, see the *Microsoft Windows Programmer's Reference, Volume 1.*

The debugging version of Windows consists of the executable and symbol files for the GDI, KERNEL, and USER modules. These modules are identical to those provided with standard Windows except that they contain extra code that checks for errors and then reports them.

The best way to use the debugging version of Windows is to install it on a computer you use for testing and debugging and use a second computer for development. Output from the debugging system and debugger can be directed to a debugging terminal.

Developers who write and debug applications on a single computer often place copies of the standard and debugging versions of Windows in separate directories. When they need to switch from one system to the other, they use batch files to copy the appropriate files to the Windows system directory. (Switching between systems is a good idea because the standard Windows system is faster that the debugging version—it is a better environment for compilers and editors.) You can use the installation program supplied for the Microsoft Windows Software Development Kit (SDK) to set up this two-directory system and then use the batch files D2N.BAT and N2D.BAT to switch between the debugging and standard versions of Windows.

C.1 Debugging Programs

The Microsoft Windows System Debugging Log Application (DBWIN.EXE) allows you to see messages produced by the debugging version of Windows even if you are not using a debugging terminal or debugging application. DBWIN.EXE allows you to control the kinds of messages that are displayed and to save your preferences in the WIN.INI file. DBWIN.EXE also provides a feature that allows you to test the performance of your application during out-of-memory failures.

The Microsoft Windows Dr. Watson application detects system and application failures and can store information in a disk file. This program can help you find and fix problems in your applications. For more information about it, see Chapter 6, "Analyzing System Failures: Dr. Watson."

C.1.1 Logging Debugging Messages

You can log messages to the DBWIN window, to a debugging monitor, or to the device attached to the COM1 port. The Options menu allows you to change the destination of debugging messages.

C.1.1.1 Settings Command

Choosing the Settings command from the Options menu produces a dialog box that allows you to control the display of debugging messages produced by the debugging system. This dialog box contains the following check boxes:

Check box	Description
Break	Controls whether and how a message causes a break to the debugger with a stack trace.
Trace	Controls whether certain kinds of informational messages are produced.
Debugging	Controls the kind of debugging features enabled in the system.

Following is a selected list of debugging options:

Option	Meaning
Validate Heap	Check the consistency of global and local heaps before every call to a memory-management function. This option affects the global heap only when it is one of the default start-up settings (that is, when it is saved by choosing the Save Settings command from the File menu). This option affects local heaps only if it is set before the application is started.
Check Free Blocks	Ensure that freed local blocks are not written into. The value 0xFB is written into free blocks and when the heap is validated, a check is performed to ensure that the blocks are still filled with this value. This option works only with local heaps. It must be used with the Validate Heap option.
Buffer Fill	Fill buffers that are passed to Windows functions with the value 0xF9. This option ensures that all of the supplied buffer is writable and helps detect overwrite problems that can occur when the buffer is too small.
Break with INT 3	Break to the debugger with an **int 3** instruction, instead of a fatal exit. This option does not display a stack back-trace.

Note Some applications will not run when the Buffer Fill option is turned on. If the supplied buffer is smaller than the size specified in the *count* parameter of the calling function, the application data is overwritten.

C.1.1.2 Alloc Break Command

The Alloc Break command on the Options menu ensures that an application deals properly with out-of-memory conditions. This command displays a dialog box into which you can enter the module name of your application and the number of memory allocations you want to succeed before subsequent allocations fail.

The system counts each global or local memory allocation performed by your application. When the number of allocations reaches the allocation break count, that allocation and all subsequent allocations fail. Because memory allocations made by the system fail once the break count is reached, calls to certain functions (such as **CreateWindow**, **CreateBrush**, and **SelectObject**) will fail as well. Only allocations made within the context of the application you specify are affected by the allocation break count.

The module name is limited to 8 characters. In some cases the module name may be different from the filename. (The module name is specified in the module-definition file for the application.) You cannot specify the module name of a DLL.

If you set the break count to zero, no allocation break is set, but the system counts allocations made by the specified application. You can choose the Show Count button to display the current allocation count.

You can set an allocation break before the named application is run. The allocation count is then set to zero and allocations are counted as soon as the application starts. If you run more than one instance of an application, the allocation break applies only to the most recent instance.

The allocation count is also reset to zero when you choose the Set command or the Inc & Set command. You can set an allocation break before performing an operation, to ensure that your application handles the problem effectively, and then choose Inc & Set and repeat the operation, to ensure that the next allocation failure is also handled properly.

C.1.2 Interpreting Debugging Messages

Windows debugging messages are the primary feature of the debugging version of Windows. These messages identify errors caused by applications and report the type of each error and the information you need to locate the error in your application.

Windows debugging messages have the following form:

FatalExit Code = *fatalexit-code*

Stack trace:
*module-name***!***segment-name***:**[*function-name***+**]*address*

.
.
.

Abort, Break or Ignore?

The *fatalexit-code* parameter identifies the type of error. For a complete set of possible error codes, see Section C.3, "Debugging Messages."

The stack trace consists of one or more addresses representing a chain of return addresses from the function that detected the error to the application that made the original function call.

Windows displays the "Abort, Break or Ignore?" prompt at the end of each debugging message.

The following variables are found in Windows debugging messages:

Variable	Description
fatalexit-code	Identifies the type of error (a hexadecimal value).
module-name	Specifies the name of the application or of a Windows module (such as USER, GDI, or KERNEL).
segment-name	Specifies the name of a segment in the application or module.
function-name	Specifies the name of a function in the segment.

Note The segment and function names are available only if a symbol file (.SYM extension) exists for the given module. Otherwise, Windows displays addresses instead of names.

The following example shows a typical debugging message:

```
FatalExit Code = 0x6040

Stack trace:
USER!_FFFE:SHOWCURSOR+0389
USER!_MSGBOX:08D7
USER!_FFFE:922D
MYAPP!_TEXT:WINMAIN+001B
MYAPP!_TEXT:__astart+0060

Abort, Break or Ignore?
```

In this example, the stack trace shows that the **ShowCursor** function in the USER module (USER.EXE) detected the error. The error type is 0x6040. This value is associated with the ERR_BAD_HWND constant; it means that the window handle passed to the function is not valid. The MYAPP application initially called the USER module at the address WINMAIN+001B in its _TEXT segment. A check of the application code at that location will probably reveal the error.

The "Abort, Break or Ignore?" prompt gives you the opportunity to terminate Windows, pass control to the debugger, or ignore the error. When you receive this prompt, you must type one of the following responses:

Response	Action
A	Terminates Windows, returning control to the MS-DOS prompt or to the debugger (if one was running).
B	Generates a breakpoint interrupt. If no debugger is running, this response terminates Windows as if you had typed **A**. If a debugger is running, control passes to the debugger as if you had set a breakpoint in the application. In this case, the CS:IP registers point to an **int 3** instruction. To continue execution or to enable single-stepping, you must change the IP register to the address of the next instruction.
I	Ignores the error and continues running the application that caused the error.
SPACE or NEWLINE	Directs Windows to redisplay the debugging message. This is helpful if the stack trace for the message is exceptionally long.

Note Not all debuggers support the same type of stack trace that Windows displays. If you use the **B** response to enter a debugger that does not support stack tracing, there is no way to regenerate the trace.

C.2 Debugging Functions and the WINDEBUGINFO Structure

Applications can use the **DebugOutput** function to display information on either the debugging terminal or the current debugging computer. The function is especially useful for displaying the full details of calls to functions that generate debugging messages.

DebugOutput includes formatting and message-filtering features that are not available with the **OutputDebugString** function.

Debugging-system options and filters are provided in the **WINDEBUGINFO** structure. The **WINDEBUGINFO** structure has the following form:

```
typedef struct tagWINDEBUGINFO {
    UINT    flags;              /* valid WINDEBUGINFO members      */
    DWORD   dwOptions;          /* debugging options               */
    DWORD   dwFilter;           /* filter  for trace messages      */
    char    achAllocModule[8];  /* module for alloc break          */
    DWORD   dwAllocBreak;       /* allocs to succeed before break  */
    DWORD   dwAllocCount;       /* number of successful allocs     */
} WINDEBUGINFO;
```

The values in **WINDEBUGINFO** can be set and retrieved by using the **SetWinDebugInfo** and **GetWinDebugInfo** functions.

You can generate your own debugging messages by using the **FatalExit** function. This function displays a message that has the same form as a debugging message generated by Windows, using the error value supplied as its only parameter. This function is especially useful for debugging DLLs.

In general, you should remove calls to debugging functions when compiling the final version of your application or library.

C.2.1 WIN.INI Debugging Options

Applications use the **GetWinDebugInfo** and **SetWinDebugInfo** functions to retrieve or set debugging options or filter values at run time. To control the same options and filter values in a system-wide, persistent manner, you can use two entries in the **[WINDOWS]** section of the WIN.INI file. These entries are **DebugOptions** and **DebugFilter**. They have the following form:

[WINDOWS]
DebugOptions = *hexadecimal value*
DebugFilter = *hexadecimal value*

The setting for the **DebugOptions** entry corresponds to the values for the **dwOptions** member of the **WINDEBUGINFO** structure. The setting for the **DebugFilter** entry corresponds to the values for the **dwFilter** member of **WINDEBUGINFO**. To determine the proper hexadecimal value for a setting, add the values of the options to be set. For example, to specify DBO_CHECKHEAP and DBO_FREEFILL, the setting for the **DebugOptions** entry would be 0x0021 (0x0001 + 0x0020). For information about the possible values for these options and a full description of the **WINDEBUGINFO** structure, see the *Microsoft Windows Programmer's Reference, Volume 3.*

C.3 Debugging Messages

The following table gives the possible error values in a Windows debugging message. For a list that includes the strings that are associated with these error values, see the "Debugging Messages" topic in the online reference.

Value	Constant	Meaning
0x0001	ERR_GALLOC	**GlobalAlloc** failed. This error value is sent by KERNEL.
0x0002	ERR_GREALLOC	**GlobalReAlloc** failed. This error value is sent by KERNEL.
0x0003	ERR_GLOCK	**GlobalLock** failed. This error value is sent by KERNEL.
0x0004	ERR_LALLOC	**LocalAlloc** failed. This error value is sent by KERNEL.
0x0005	ERR_LREALLOC	**LocalReAlloc** failed. This error value is sent by KERNEL.
0x0006	ERR_LLOCK	**LocalLock** failed. This error value is sent by KERNEL.
0x0007	ERR_ALLOCRES	**AllocResource** failed. This error value is sent by KERNEL.
0x0008	ERR_LOCKRES	**LockResource** failed. This error value is sent by KERNEL.
0x0009	ERR_LOADMODULE	**LoadModule** failed. This error value is sent by KERNEL.
0x0040	ERR_CREATEDLG	Dialog box could not be created because **LoadMenu** failed. This error value is sent by USER.
0x0041	ERR_CREATEDLG2	Dialog box could not be created because **CreateWindow** failed. This error value is sent by USER.
0x0042	ERR_REGISTERCLASS	**RegisterClass** failed because the class is already registered. This error value is sent by USER.
0x0043	ERR_DCBUSY	Device-context cache is full. This error value is sent by USER.
0x0044	ERR_CREATEWND	Window could not be created because the class was not found. This error value is sent by USER.
0x0045	ERR_STRUCEXTRA	Program is using unallocated space. This error value is sent by USER.
0x0046	ERR_LOADSTR	**LoadString** failed. This error value is sent by USER.

Value	Constant	Meaning
0x0047	ERR_LOADMENU	**LoadMenu** failed. This error value is sent by USER.
0x0048	ERR_NESTEDBEGINPAINT	Program contains nested **BeginPaint** functions. This error value is sent by USER.
0x0049	ERR_BADINDEX	Index to **GetClassLong**, **GetClassWord**, **GetWindowLong**, **GetWindowWord**, **SetClassLong**, **SetClassWord**, **SetWindowLong**, or **SetWindowWord** is invalid. This error value is sent by USER.
0x004A	ERR_CREATEMENU	Menu could not be created. This error value is sent by USER.
0x0080	ERR_CREATEDC	**CreateCompatibleDC**, **CreateDC**, or **CreateIC** failed. This error value is sent by GDI.
0x0081	ERR_CREATEMETA	**CreateMetaFile** failed. This error value is sent by GDI.
0x0082	ERR_DELOBJSELECTED	Program is trying to delete a bitmap that is selected into the device context. This error value is sent by GDI.
0x0083	ERR_SELBITMAP	Program is trying to select a bitmap that is already selected. This error value is sent by GDI.
0x6001	ERR_BAD_VALUE	A 16-bit signed or unsigned value is invalid.
0x6002	ERR_BAD_FLAGS	One or more bit flags are invalid.
0x6003	ERR_BAD_INDEX	Index is invalid or out of range.
0x6009	ERR_BAD_SELECTOR	Selector is invalid.
0x600B	ERR_BAD_HANDLE	Generic handle is invalid.
0x6020	ERR_BAD_HINSTANCE	Instance handle is invalid. This error value is sent by KERNEL.
0x6021	ERR_BAD_HMODULE	Module handle is invalid. This error value is sent by KERNEL.
0x6022	ERR_BAD_GLOBAL_HANDLE	Global handle is invalid. This error value is sent by KERNEL.
0x6023	ERR_BAD_LOCAL_HANDLE	Local handle is invalid. This error value is sent by KERNEL.
0x6024	ERR_BAD_ATOM	Atom is invalid. This error value is sent by KERNEL.
0x6025	ERR_BAD_HFILE	File handle is invalid. This error value is sent by KERNEL.

Value	Constant	Meaning
0x6040	ERR_BAD_HWND	Window handle is invalid. This error value is sent by USER.
0x6041	ERR_BAD_HMENU	Menu handle is invalid. This error value is sent by USER.
0x6042	ERR_BAD_HCURSOR	Cursor handle is invalid. This error value is sent by USER.
0x6043	ERR_BAD_HICON	Icon handle is invalid. This error value is sent by USER.
0x6044	ERR_BAD_HDWP	Handle to a window-position structure is invalid. This error value is sent by USER.
0x6045	ERR_BAD_CID	Communications identifier (CID) is invalid. This error value is sent by USER.
0x6046	ERR_BAD_HDRVR	Installable-driver handle is invalid. This error value is sent by USER.
0x6061	ERR_BAD_GDI_OBJECT	GDI object is invalid. This error value is sent by GDI.
0x6062	ERR_BAD_HDC	Device-context handle is invalid. This error value is sent by GDI.
0x6063	ERR_BAD_HPEN	Pen handle is invalid. This error value is sent by GDI.
0x6064	ERR_BAD_HFONT	Font handle is invalid. This error value is sent by GDI.
0x6065	ERR_BAD_HBRUSH	Brush handle is invalid. This error value is sent by GDI.
0x6066	ERR_BAD_HBITMAP	Bitmap handle is invalid. This error value is sent by GDI.
0x6067	ERR_BAD_HRGN	Region handle is invalid. This error value is sent by GDI.
0x6068	ERR_BAD_HPALETTE	Palette handle is invalid. This error value is sent by GDI.
0x6069	ERR_BAD_HMETAFILE	Metafile handle is invalid. This error value is sent by GDI.
0x7004	ERR_BAD_DVALUE	A 32-bit signed or unsigned value is invalid.
0x7005	ERR_BAD_DFLAGS	One or more 32-bit flags are invalid.
0x7006	ERR_BAD_DINDEX	A 32-bit index is invalid or out of range.
0x7007	ERR_BAD_PTR	Pointer is invalid.
0x7008	ERR_BAD_FUNC_PTR	Function pointer is invalid.

Value	Constant	Meaning
0x700A	ERR_BAD_STRING_PTR	Zero-terminated string pointer is invalid.
0x7060	ERR_BAD_COORDS	X- and y-coordinates are invalid. This error value is sent by GDI.

The following error values may have been combined with other values in the preceding table to identify the type of error:

Value	Constant	Meaning
0x4000	ERR_PARAM	Parameter is invalid. This flag is always set for parameter-validation error messages.
0x8000	ERR_WARNING	Nonfatal error occurred. An invalid parameter was detected, but the error was not serious enough to cause the function to fail. The invalid parameter is reported, but the function executes as usual.

To determine the size of an invalid parameter, you can combine ERR_SIZE_MASK (0x3000) with other error values by using the AND operator. The following table gives the possible results of this operation:

Value	Constant	Meaning
0x1000	ERR_BYTE	An 8-bit parameter is invalid.
0x2000	ERR_WORD	A 16-bit parameter is invalid.
0x3000	ERR_DWORD	A 32-bit parameter is invalid.

C.4 Common Programming Errors

The following list describes programming errors that sometimes appear in Windows applications:

- Passing invalid parameters.
- Accessing nonexistent window words. (In Windows 3.0, a call to the **SetWindowWord** or **SetWindowLong** function past the end of the allocated window words, as defined by the **RegisterClass** function, would damage internal window-management structures.)
- Using handles after they have been deleted or destroyed.
- Using a device context after it has been released.
- Deleting GDI objects before they are selected out of a device context.
- Neglecting to delete GDI or USER objects when an application terminates.

- Writing past the end of an allocated memory block.
- Reading or writing using a memory pointer after it has been freed.
- Neglecting to export window procedures and other callback functions.
- Neglecting to use the **MakeProcInstance** function with dialog procedures and other callback functions.

Many of these programming errors can cause unrecoverable application errors in Windows version 3.0. The debugging system can help you locate these types of problems.

Index

! (footnote character) in Help files, 23
(footnote character) in Help files, 23
$ (footnote character) in Help files, 23
.? command, 80386 Debugger, 128
? command, 80386 Debugger, 125
386 Instructions command, CVW, 85
80386 Debugger
  command keys, 118
  command parameters, 118–120
  command reference, 125–170
  common commands, listed, 123–125
  entering, 116–117
  MAPSYM command-line syntax, 114
  operators
    binary and unary, 121–122
    postfix, 122
  regular expressions, 122–123
  starting, 114
  symbol files, preparing, 113–114
  wdeb386 command-line syntax, 114–116
  wildcards, 122
8087 command, CVW, 83
CTRL+A key combination, 80386 Debugger, 118
CTRL+C key combination, 80386 Debugger, 118
CTRL+Q key combination, 80386 Debugger, 118
CTRL+S key combination, 80386 Debugger, 118
F1 key for choosing Help, 36
\bmc Help statement, 29
\bml Help statement, 29
\cf Help statement, 22
\colortbl Help statement, 22
\deff Help statement, 22
\f Help statement, 24
\fi Help statement, 24
\fonttbl Help statement, 22
\footnote Help statement
  defining help topics, 23
  keyword list, 27
\fs Help statement, 24
\keep Help statement, 25
\li Help statement, 24
\line Help statement, 25
\mbr Help statement, 29
\page Help statement, 23
\par Help statement, 23
\pard Help statement, 24
\pict Help statement, 28
\plain Help statement, 24
\ri Help statement, 24
\rtf Help statement, 22
\sa Help statement, 24
\sb Help statement, 24
\strike Help statement, 26
\tab Help statement, 25
\tx Help statement, 25
\ul Help statement, 26
\uldb Help statement, 26
\v Help statement, 26
\windows Help statement, 22

A

Accelerator key, creating, 37
ACCELERATORS statement, described, 10
Active Conversations command, DDESpy, 194
Active Links command, DDESpy, 194
Add command, CVW, 84
Add Watch command, CVW, 86
Address command, Heap Walker, 199, 202
Advise loops (DDE), tracking, 194
ALIAS section, Help project files, 45
Alloc Break command, DBWIN application, 247
Allocate All of Memory command, Heap Walker, 203
Animate command, CVW, 84, 104
Application performance, analyzing, 207
Arrays, displaying in CVW
  character arrays, 88
  described, 87
  dynamic array elements, 90
  multidimensional arrays, 89

B

.b command, 80386 Debugger, 128
BAGGAGE section, Help project files, 46
Bar, as a document convention, x
bc command, 80386 Debugger, 134
bd command, 80386 Debugger, 134
be command, 80386 Debugger, 135

Binary operators, 80386 Debugger, 121
BITMAP statement, described, 10
Bitmaps in Help files
  formats, 28
  hot spots, using as, 30
  inserting in text, 29
  multiple-resolution bitmaps, creating, 30–31
  wrapping text around, 29
BITMAPS section, Help project files, 47
bl command, 80386 Debugger, 135
BMROOT option, Help project files, 48
Bold type, as a document convention, x
bp command, 80386 Debugger, 136
br command, 80386 Debugger, 137
Brackets, as a document convention, x
Breakpoints
  selecting, 99
  setting on Windows messages, 101
  setting values, 101
  using when debugging, 102
Browse sequence, help topics, 27
BUILD option, Help project files, 48
BUILDTAGS section, Help project files, 50

C

c command, 80386 Debugger, 138
Callback activity, monitoring, 193
Case Sensitivity command, CVW, 85
Close command, CVW, 83
CodeView for MS-DOS (CV), 72
CodeView for Windows (CVW)
  advanced techniques
    multiple source windows, 108
    redirecting input and output, 109
    register variables, 109
    undefined pointers, 108
  animated application execution, 104
  vs. CodeView for MS-DOS, 72
  continuous application execution
    selecting breakpoint lines, 99
    setting breakpoint values, 101
    setting breakpoints on messages, 101
    using breakpoints, 102
  core DLLs, specifying, 74
  customizing, 110
  debugging version of Windows, 73
  display windows
    adjusting, 82
    opening, 81
CodeView for Windows (CVW) *(continued)*
  display windows *(continued)*
    selecting, 81
    types of, described, 80
  displaying application data
    arrays and structures, 87
    character arrays, 88
    dynamic array elements, 90
    expressions, 87
    heap, global or local, 93
    live expressions, 95
    memory, 92
    memory handles, dereferencing, 96
    modules, 98
    multidimensional arrays, 89
    Quick Watch command, using, 91
    register contents, 97
    tracing Windows messages, 91
    variables, 86
  ending a session, 108
  fatal exits, handling, 106
  general protection fault, handling, 107
  Help, accessing, 85
  interrupting application execution, 104
  jumping to a location, 104
  menu bar, using, 83
  modifying application data, 98
  monitors, using with CVW
    secondary monitor, 70
    single monitor, 70
  preparing applications for debugging, 73
  requirements for using, 69
  saving session information, 80
  single-step application execution, 103
  starting a debugging session
    command-line options, 78–79
    display options, 75
    dynamic-link libraries, 77
    multiple application instances, 76
    multiple applications, 76
    single application, 75
  vs. SYMDEB, 71
  TOOLS.INI file, modifying, 110
Color, specifying in Help files, 22
Command command, CVW, 83
COMPRESS option, Help project files, 51
COMPRESS.EXE (File Compression Utility), 215
Compressing files, 215
CONFIG section, Help project files, 52
CONTENTS option, Help project files, 52

Context-sensitive Help, defined, 34
Context-string error messages, Help Compiler, 237
Conversations (DDE), tracking, 194
Copy command, CVW, 83
COPYRIGHT option, Help project files, 53
Core dynamic-link libraries, debugging support, 73
CreateButton macro, 33
CSIPS.DAT file, 207, 209
CURRENT.STS file, CVW session information, 80
CURSOR statement, described, 10
CV (CodeView for MS-DOS), 72
CVW. *See* CodeView for Windows
cvw command options, CVW, 78–79

D

d command, 80386 Debugger, 139
D2N.BAT batch program, 74, 245
db command, 80386 Debugger, 139
DBWIN application, 245–247
dd command, 80386 Debugger, 140
DDE activity, monitoring. *See* DDESpy
DDESpy
  callback activity, monitoring, 193
  DDE errors, monitoring, 193
  DDE messages, monitoring, 192
  Monitor menu, 191
  Output menu, 191
  string-handle data, monitoring, 192
  tracking options
    active conversations, 194
    active links, 194
    registered servers, 194
    string handles, 194
DDESPY.EXE, Windows DDESpy, 191
DebugBreak function, 100
Debugger. *See* 80386 Debugger
Debugging. *See* CodeView for Windows; 80386 Debugger
Debugging version of Windows
  common programming errors, 254
  debugging functions, 249–250
  debugging messages
    error codes, listed, 251–254
    interpreting, 247–249
    logging, 245–247
  described, 245
  WIN.INI debug options, 250
DebugOutput function, 249
Decompressing files, 216
#define directive, 10
Delete Watch command, CVW, 87
Delete Watch, CVW, 84
.df command, 80386 Debugger, 128
.dg command, 80386 Debugger, 129
dg command, 80386 Debugger, 141
.dh command, 80386 Debugger, 131
di command, 80386 Debugger, 142
Dialog box, designing, 4
Dialog Editor
  described, 4
  illustrated, 5
DIALOG statement, described, 10
Directives, resource-definition files, 10
Discard command, Heap Walker, 200
DisLen entry, WIN.INI file, 174
DisStack entry, WIN.INI file, 176
dl command, 80386 Debugger, 143
DLGEDIT.EXE, Dialog Editor, 4
.dm command, 80386 Debugger, 131
Document conventions, x
dp command, 80386 Debugger, 144
.dq command, 80386 Debugger, 132
Dr. Watson
  configuring from the WIN.INI file
    DisLen entry, 174
    DisStack entry, 176
    GPContinue entry, 175
    LogFile entry, 176
    ShowInfo entry, 174
    SkipInfo entry, 173
    TrapZero entry, 175
  sample log file
    with comments, 179–182
    without comments, 177–178
DRWATSON.LOG file, 176
dt command, 80386 Debugger, 145
.du command, 80386 Debugger, 132
dw command
  80386 Debugger, 145
  CVW, 96
Dynamic-link libraries, debugging support, 73

E

e command, 80386 Debugger, 146
Edit Breakpoints command, CVW, 84
#elif directive, 10
Ellipses, as a document convention, x
#else directive, 10
#endif directive, 10
Error messages. *See* Messages

Error report, Dr. Watson, 173
ERRORLOG option, Help project files, 54
Executable file
  renaming, 15
  specifying in the rc command, 15
EXPAND.EXE (File Expansion Utility), 216
Expanding files, 216
Expressions, displaying in CVW, 87

F

f command, 80386 Debugger, 147
Fatal exit, handling in CVW, 106
FatalExit function, 250
File Compression Utility, 215
File error messages, Help Compiler, 232
File Expansion Utility, 216
FILES section, Help project files, 54
Find command, CVW, 84
Flags command, Heap Walker, 202
Font
  designing, 5–6
  in Help files
    declaring, 22
    size, setting, 24
Font Editor
  designing fonts, 5–6
  Help, accessing, 5
  illustrated, 6
FONT statement, described, 10
FONTEDIT.EXE, Font Editor, 5
Footnote characters in Help files, 23
FORCEFONT option, Help project files, 55
Free memory commands, Heap Walker, 203
FreeProcInstance function, 39

G

g command, 80386 Debugger, 148
GC(-1) and Hit A command, Heap Walker, 199
GC(-1) and Walk command, Heap Walker, 199
GDI LocalWalk command, Heap Walker, 200
General protection (GP) fault
  continuing an application, 175
  handling in CVW, 107
GetWinDebugInfo function, 250
Global memory objects, displaying in CVW, 93
GPContinue entry, WIN.INI file, 175
Graphics files, using in Help files, 28

H

h command, 80386 Debugger, 149
Heap Walker
  described, 197
  displaying memory objects
    Object menu commands, 200
    Show command, 200
  illustrated, 197
  local heap, viewing
    Add! menu, 203
    Heap menu commands, 202
    Local Walk window, 201–203
    Sort menu commands, 202
  memory, allocating, 203
  saving heap listings in files, 198
  sorting memory objects, 199
  suggestions for using, 204
  walking the heap, 199
HEAPWALK.EXE, Windows Heap Walker, 197
Help command, CVW, 83
Help Compiler
  building help files, 21
  error messages
    context string errors, 237–239
    file errors, 232
    interpreting, 231
    macro errors, 237
    miscellaneous errors, 240–241
    project-file errors, 233–236
    topic-file errors, 239–240
Help file
  F1 key for choosing Help, 36
  SHIFT+F1 key, detecting, 39
  bitmaps
    hot spots, using as, 30
    inserting in text, 29
    multiple-resolution, creating, 30–31
    wrapping text around, 29
  building, 21
  canceling Help, 44
  context-sensitive Help, defined, 34
  error messages, 231–241
  filter procedure, 36
  footnote characters, 23
  graphics files, required formats, 28
  Help menu, 34–36
  keywords
    creating, 26–27
    searching for Help, 42

Help file *(continued)*
menu selections, monitoring, 37
mouse input, intercepting, 39–41
overview, 21
project files
ALIAS section, 45
BAGGAGE section, 46
BITMAPS section, 47
BMROOT option, 48
BUILD option, 48
BUILDTAGS section, 50
COMPRESS option, 51
CONFIG section, 52
CONTENTS option, 52
COPYRIGHT option, 53
ERRORLOG option, 54
FILES section, 54
FORCEFONT option, 55
ICON option, 56
LANGUAGE option, 56
macros in, 33
MAP section, 57
MAPFONTSIZE option, 58
MULTIKEY option, 59
OLDKEYPHRASE option, 60
OPTCDROM option, 60
OPTIONS section, 61
REPORT option, 62
ROOT option, 62
sample file, 33
sections, described, 32
TITLE option, 63
WARNING option, 64
WINDOWS section, 64
secondary windows, 43
topic files
browse sequence, creating, 27
character set, specifying, 22
colors, specifying, 22
context string, assigning, 23
described, 21
font size, setting, 24
fonts, declaring, 22
hot spots, creating, 26
indents, setting, 24
line breaks, controlling, 25
links, creating, 26
paragraph spacing, setting, 24
paragraphs, separating, 23
pop-up topics, creating, 26

Help file *(continued)*
topic files *(continued)*
tab stops, setting, 25
topic title, assigning, 23
HELPWININFO structure, 43
Hot spots in Help files
creating, 26
using bitmaps, 30
Hotspot Editor, 28

I

i command, 80386 Debugger, 150
ICON option, Help project files, 56
ICON statement, described, 10
#if directive, 11
#ifdef directive, 11
#ifndef directive, 11
Image Editor, 3
IMAGEDIT.EXE, Image Editor, 3
Images, designing, 3
#include directive, 11
INCLUDE environment variable, suppressing, 18
Indents, setting in Help files, 24
Infinite loop, locating, 104
Info command, Heap Walker, 202
InsertMenu macro, 33
Italic, as a document convention, x

J

j command, 80386 Debugger, 150
Jumps in Help files
creating, 26
using bitmaps, 30

K

K (footnote character) in Help files, 26–27
k command, 80386 Debugger, 151
ka command, 80386 Debugger, 151
Keywords for help topics
creating, 26–27
searching for Help, 42
kt command, 80386 Debugger, 152

L

la command, 80386 Debugger, 152
Label/Function command, CVW, 84
LANGUAGE option, Help project files, 56

LC(-1) and LocalWalk command, Heap Walker, 200
lg command, 80386 Debugger, 153
Line breaks in Help files, controlling, 25
Links in Help files
  creating, 26
  using bitmaps, 30
Live expression, creating in CVW, 95
lm command, 80386 Debugger, 153
ln command, 80386 Debugger, 154
LoadAccelerators function, 37
Local command, CVW, 83
Local heap, viewing, 201–203
Local memory objects, displaying in CVW, 93
LocalWalk command, Heap Walker, 200–203
Log file, Dr. Watson, 173
LogFile entry, WIN.INI file, 176
ls command, 80386 Debugger, 154

M

m command, 80386 Debugger, 155
Macros, in Help files
  adding, 23
  bitmaps, using as hotspots, 30
  error messages, Help Compiler, 237
  project files, using in, 33
MAP file, 113
MAP section, Help project files, 57
MAPFONTSIZE option, Help project files, 58
MAPSYM, command-line syntax, 114–116
Margins, overriding in Help files, 24
Maximize command, CVW, 83
Memory
  allocating by using Heap Walker, 203
  displaying in CVW
    display options, 92
    heap, global or local, 93
    live expressions, 95
    memory handles, dereferencing, 96
    modifying memory locations, 98
  size, determining, 203
Memory command, CVW, 83, 92
Memory object
  displaying information about, 199
  failure to delete objects, tracing, 204
  hexadecimal dump, displaying, 200
  local heap, viewing, 201, 203
  sorting memory objects, 199
  viewing selectively, 200
Memory Window command, CVW, 85
MENU statement, described, 10
Messages
  debugging messages
    error codes, listed, 251–254
    interpreting, 247–249
    logging, 245–247
  Help Compiler error messages
    context string errors, 237–239
    file errors, 232
    interpreting, 231
    macro errors, 237
    miscellaneous errors, 240–241
    project-file errors, 233–136
    topic-file errors, 239–240
  monitoring
    DDE messages, 192
    frequency of message output, 186
    message types, selecting, 185
    output device, selecting, 186
    window to be monitored, selecting, 187
  Resource Compiler diagnostic messages, 219–228
  setting breakpoints on, 101
  tracing, 91
Metafile, using in Help files, 28
Module command, Heap Walker, 199
Modules, displaying in CVW, 98
MONCBSTRUCT structure, 193
MONERRSTRUCT structure, 193
MONHSZSTRUCT structure, 192
MONMSGSTRUCT structure, 192
MRBC (Multiple Resolution Bitmap Compiler), 30
mrbc command-line syntax, 30
MULTIKEY option, Help project files, 59
MULTIKEYHELP structure, 42
Multiple Resolution Bitmap Compiler (MRBC), 30

N

N2D.BAT batch program, 74, 245

O

o command, 80386 Debugger, 156
Oldest command, Heap Walker, 200
OLDKEYPHRASE option, Help project files, 60
Open Module command, CVW, 83
Open Source command, CVW, 83
Operators, 80386 Debugger
  binary and unary operators, 121
  postfix operators, 122
OPTCDROM option, Help project files, 60

Optimizing application performance, 207
OPTIONS section, Help project files, 61

P

p command, 80386 Debugger, 156
Paste command, CVW, 83
Performance analysis, 207
Placeable metafile, using in Help files, 28
Pop-up help topics, creating, 26
ProfClear function, Profiler, 208
ProfFinish function, Profiler, 208
ProfFlush function, Profiler, 208
Profiler
  displaying sampled data, 209
  functions, described, 208
  overview, 207
  preparing to run, 208
  sampling code, 209
  VPROD.386 device driver, installing, 209
ProfInsChk function, Profiler, 208
ProfSampRate function, Profiler, 208
ProfSetup function, Profiler, 208
ProfStart function, Profiler, 208
ProfStop function, Profiler, 208
Project file, Help
  error messages, Help Compiler, 233–236
  macros in, 33
  options
    BMROOT, 48
    BUILD, 48
    COMPRESS, 51
    CONTENTS, 52
    COPYRIGHT, 53
    ERRORLOG, 54
    FORCEFONT, 55
    ICON, 56
    LANGUAGE, 56
    MAPFONTSIZE, 58
    MULTIKEY, 59
    OLDKEYPHRASE, 60
    OPTCDROM, 60
    REPORT, 62
    ROOT, 62
    TITLE, 63
    WARNING, 64
  sample file, 33
  sections
    ALIAS, 45
    BAGGAGE, 46
    BITMAPS, 47
Project file, Help *(continued)*
  sections *(continued)*
    BUILDTAGS, 50
    CONFIG, 52
    described, 32
    FILES, 54
    MAP, 57
    OPTIONS, 61
    WINDOWS, 64

Q

q command, CVW, 106, 108
Quick Watch command, CVW, 84, 91

R

r command, 80386 Debugger, 157
rc command, Resource Compiler
  compiling resources separately, 15
  conditional branching, specifying, 16
  messages, displaying, 18
  options, described, 13–15
  renaming compiled resource files, 17
  renaming the executable file, 15
  resource-definition file, specifying, 15
  searching directories
    adding a directory, 17
    INCLUDE variable, suppressing, 18
  specifying the executable file, 15
  syntax, 12
RCDATA statement, described, 10
.reboot command, 80386 Debugger, 133
Refresh Seg Names command, Heap Walker, 199
Register command, CVW, 83, 97
Register variables, handling in CVW, 109
Registered Servers command, DDESpy, 194
RegisterRoutine macro, 33
Registers
  displaying contents in CVW, 97
  modifying the values of, 98
Regular expression, 80386 Debugger, 122–123
Repeat Last Find command, CVW, 84
REPORT option, Help project files, 62
Resource Compiler (RC)
  compiling resources separately, 15
  conditional branching, specifying, 16
  described, 12
  messages
    diagnostic, 219–228
    displaying, 18

Resource Compiler (RC) *(continued)*
rc command
executable file, specifying, 15
options, described, 13–15
resource-definition file, specifying, 15
syntax, 12
renaming compiled resource files, 16–17
resource-definition files, creating
described, 9
directives, 10
multiline statements, 10
sample file, 11
single-line statements, 10
resources, including in applications, 9
searching directories
adding a directory, 17
INCLUDE variable, suppressing, 18
Resource-definition file. *See* Resource Compiler
Resource file
compiling resources separately, 15
renaming, 16–17
Resources
dialog boxes, designing, 4
fonts, designing, 5
images, designing, 3
including in applications, 9
Rich-text format (RTF) for help topics, 22
rip command, CVW, 104
ROOT option, Help project files, 62
RTF (rich-text format) for help topics, 22

S

s command, 80386 Debugger, 159
Save command, Heap Walker, 198, 202
SEGENTRY.DAT file, 207, 209
Segmentation Test command, Heap Walker, 199
Segmented-graphics bitmap, 28
Selected Text command, CVW, 84
Set Breakpoint command, CVW, 84, 100, 102
Set Swap Area command, Heap Walker, 199
Settings command, DBWIN application, 246
SetWinDebugInfo function, 250
SetWindowsHook function, 36
Show command, Heap Walker, 200
SHOWHITS.EXE application
command-line syntax, 210
described, 209
information categories, described, 210
ShowInfo entry, WIN.INI file, 174
Size command
CVW, 83
Heap Walker, 199, 202
SkipInfo entry, WIN.INI file, 173
Sorting memory objects, 199
Source command, CVW, 83, 104
Spy
See also CodeView for Windows
frequency of message output, 186
message types, selecting, 185
output device, selecting, 186
starting and stopping, 187
window to be monitored, selecting, 187
SPY.EXE, Windows Spy, 185
Step command, CVW, 103
String Handles command, DDESpy, 194
String-handle data, monitoring, 192
STRINGTABLE statement, described, 10
Structures, displaying in CVW, 87
Symbol File Generator (MAPSYM), 113
Symbol file, preparing for 80386 Debugger, 113
Symbolic Debugger (SYMDEB), 71

T

t command, 80386 Debugger, 160
Tab stops, setting in Help files, 25
TITLE option, Help project files, 63
TOOLHELP.DLL library, used by Dr. Watson, 173
TOOLS.INI file, customizing CVW, 110
Topic file
See also Help file
creating, 21–28
error messages, Help Compiler, 239–240
Trace command, CVW, 103
Trace Speed command, CVW, 85
TranslateAccelerator function, 37
TrapZero entry, WIN.INI file, 175
Type command, Heap Walker, 199

U

u command, 80386 Debugger, 162
Unary operators, 80386 Debugger, 121
#undef directive, 11
Undo command, CVW, 83
UnhookWindowsHook function, 39
Unrecoverable application errors, debugging, 254
User LocalWalk command, Heap Walker, 200

V

v command
  CVW, 106
  80386 Debugger, 162
Variables
  displaying in CVW, 86
  modifying the values of, 98
vc command, 80386 Debugger, 163
Version stamp, 12
Vertical bar, as a document convention, x
VGASYS.FNT file, 5
vl command, 80386 Debugger, 163
vo command, 80386 Debugger, 164
VPROD.386 device driver, installing, 209
vs command, 80386 Debugger, 164
vt command, 80386 Debugger, 165

W

w command, 80386 Debugger, 165
wa command, 80386 Debugger, 166
Walk Free List command, Heap Walker, 199
Walk Heap command, Heap Walker, 199
Walk LRU List command, Heap Walker, 199
WARNING option, Help project files, 64
Watch command, CVW, 83
wbm command, CVW, 101
wdeb386 command, 80386 Debugger, 114–116
WDEB386.EXE. *See* 80386 Debugger
wdg command, CVW, 93, 105
wdl command, CVW, 93
wdm command, CVW, 98
Wildcards, 80386 Debugger, 122
WIN.INI debug options, 250
WIN.INI file, configuring Dr. Watson
  DisLen entry, 174
  DisStack entry, 176
  GPContinue entry, 175
  LogFile entry, 176
  ShowInfo entry, 174
  SkipInfo entry, 173
  TrapZero entry, 175
WINDEBUGINFO structure, 250
Windows debugging version
  common programming errors, 254
  debugging functions, 249–250
  debugging messages
    error codes, listed, 251–254
    interpreting, 247–249
    logging, 245
Windows debugging version *(continued)*
  described, 245–247
  WIN.INI debug options, 250
Windows messages
  setting breakpoints on, 101
  tracing by using CVW, 91
WINDOWS section, Help project files, 64
WinHelp function
  canceling Help, 44
  context-sensitive Help, 34
  mouse input, 39
  searching keyword tables, 42
  secondary Help windows, 43
  supporting the Help menu, 35
wka command, CVW, 105
wr command, 80386 Debugger, 166
wwm command, CVW, 91

Y

y command, 80386 Debugger, 167

Z

z command, 80386 Debugger, 168
zd command, 80386 Debugger, 169
zl command, 80386 Debugger, 169
zs command, 80386 Debugger, 170

Microsoft Corporation
One Microsoft Way
Redmond, WA 98052-639

Microsoft®

0392 Part No. 28918