Microsoft® CodeView™ and Utilities

Software Development Tools

for the MS-DOS® Operating System

Microsoft Corporation

If you have comments about the software, complete the Software Problem Report at the back of this manual and return it to Microsoft Corporation.

If you have comments about the software documentation, complete the Documentation Feedback reply card at the back of this manual and return it to Microsoft Corporation.

Document No. 510830017-100-O00-0587

Contents

Part 1 The CodeView Debugger 15

1 Getting Started 15

1.1 Restrictions 17
1.2 Preparing Programs for the CodeView Debugger 18
1.3 Starting the CodeView Debugger 27
1.4 Using CodeView Options 31
1.5 Debugging Large Programs 41
1.6 Working with Older Versions of MASM 42

2 The CodeView Display 45

2.1 Using Window Mode 48
2.2 Using Sequential Mode 77

3 Using Dialog Commands 79

3.1 Entering Commands and Arguments 81
3.2 Format for CodeView Commands and Arguments 83

4 CodeView Expressions 85

4.1 C Expressions 88
4.2 FORTRAN Expressions 92
4.3 BASIC Expressions 97
4.4 MASM Expressions 103
4.5 Line Numbers 104
4.6 Registers and Addresses 105
4.7 Memory Operators 109
4.8 Switching Expression Evaluators 112

5 Executing Code 115

5.1 Trace Command 118
5.2 Program Step Command 120
5.3 Go Command 123
5.4 Execute Command 126
5.5 Restart Command 127

6 Examining Data and Expressions 131

6.1 Display Expression Command 133
6.2 Examine Symbols Command 143
6.3 Dump Commands 148
6.4 Compare Memory Command 158
6.5 Search Memory Command 159
6.6 Port Input Command 161
6.7 Register Command 162
6.8 8087 Command 164

7 Managing Breakpoints 167

7.1 Breakpoint Set Command 169
7.2 Breakpoint Clear Command 172
7.3 Breakpoint Disable Command 174
7.4 Breakpoint Enable Command 175
7.5 Breakpoint List Command 176

8 Managing Watch Statements 179

8.1 Setting Watch-Expression and Watch-Memory Statements 182
8.2 Setting Watchpoints 186
8.3 Setting Tracepoints 191
8.4 Deleting Watch Statements 196
8.5 Listing Watchpoints and Tracepoints 197
8.6 C Examples 199
8.7 FORTRAN Examples 200

9 Examining Code 201

9.1 Set Mode Command 203
9.2 Unassemble Command 205
9.3 View Command 208

9.4 Current Location Command 211
9.5 Stack Trace Command 212

10 Modifying Code or Data 217

10.1 Assemble Command 219
10.2 Enter Commands 223
10.3 Fill Memory Command 234
10.4 Move Memory Command 236
10.5 Port Output Command 237
10.6 Register Command 238

11 Using System-Control Commands 243

11.1 Help Command 245
11.2 Quit Command 246
11.3 Radix Command 247
11.4 Redraw Command 249
11.5 Screen Exchange Command 250
11.6 Search Command 251
11.7 Shell Escape Command 253
11.8 Tab Set Command 256
11.9 Redirection Commands 257

Part2 Utilities 265

12 Linking Object Files with LINK 267

12.1 Specifying Files for Linking 269
12.2 Specifying Linker Options 279
12.3 Linker Operation 290
12.4 Using Overlays 294

13 Managing Libraries with LIB 297

13.1 Managing Libraries 300

13.2 Peforming Library Management Tasks with LIB 308

14 Automating Program Development with MAKE 315

14.1 Using MAKE 318
14.2 Creating a MAKE Description File 318
14.3 Automating Program Development 322
14.4 Running MAKE 323
14.5 Specifying MAKE Options 324
14.6 Using Macro Definitions with MAKE 325
14.7 Defining Inference Rules 329

15 Using EXEPACK, EXEMOD, SETENV, and ERROUT 331

15.1 Compressing Executable Files with the EXEPACK Utility 333
15.2 Modifying Program Headers with the EXEMOD Utility 335
15.3 Enlarging the DOS Environment with the SETENV Utility 338
15.4 Redirecting Error Output with the ERROUT Utility 339

A Regular Expressions 345

A.1 Introduction 347
A.2 Special Characters in Regular Expressions 347
A.3 Searching for Special Characters 348
A.4 Using the Period 348
A.5 Using Brackets 349
A.6 Using the Asterisk 350
A.7 Matching the Start or End of a Line 351

B Error Messages 353

B.1 CodeView Error Messages 355
B.2 Linker Error Messages 365
B.3 LIB Error Messages 377

B.4 MAKE Error Messages 382
B.5 EXEPACK Error Messages 385
B.6 EXEMOD Error Messages 387
B.7 SETENV Error Messages 389
B.8 ERROUT Error Messages 391

C Using Exit Codes 393

C.1 Exit Codes with MAKE 395
C.2 Exit Codes with DOS Batch Files 395
C.3 Exit Codes for Programs 396

Figures

Figure 1.1 CodeView Start-Up Screen in Window Mode 30
Figure 2.1 Elements of the CodeView Debugging Screen 48
Figure 2.2 The File Menu 62
Figure 2.3 The Search Menu 64
Figure 2.4 The View Menu 66
Figure 2.5 The Run Menu 67
Figure 2.6 The Watch Menu 68
Figure 2.7 The Options Menu 71
Figure 2.8 The Language Menu 73
Figure 2.9 The Calls Menu 74
Figure 8.1 Watch Statements in the Watch Window 186
Figure 8.2 Watchpoints in the Watch Window 190
Figure 8.3 Tracepoints in the Watch Window 195
Figure 8.4 C Watch Statements 199
Figure 8.5 FORTRAN Watch Statements 200

Tables

Table 1.1	Default Exchange and Display Modes	36
Table 4.1	CodeView C Operators	88
Table 4.2	C Radix Examples	91
Table 4.3	CodeView FORTRAN Operators	92
Table 4.4	FORTRAN Radix Examples	95
Table 4.5	FORTRAN Intrinsic Functions Supported by the CodeView Debugger	96
Table 4.6	CodeView BASIC Operators	97
Table 4.7	BASIC Radix Examples	100
Table 4.8	BASIC Instrinsic Functions Supported by the CodeView Debugger	102
Table 4.9	Registers	106
Table 6.1	CodeView Format Specifiers	134
Table 10.1	Flag-Value Mnemonics	239

Welcome to the Microsoft CodeView debugger and utilities. These are executable programs that help you develop software with the Microsoft BASIC, C, FORTRAN, and Pascal compilers, as well as the Macro Assembler (MASM). The utilities described here are designed to work with each language.

The Microsoft CodeView debugger is a powerful, window-oriented tool that helps you track down logical errors, for programs written in Microsoft BASIC, C, FORTRAN, or the Macro Assembler. Unlike its predecessors SYMDEB and DEBUG, the CodeView debugger can be easily learned and used by assembly and high-level-language programmers alike. The debugger provides a wide variety of techniques, and is presented in this manual in a substantially enhanced form.

The other utilities presented in this manual can be used for important parts of the development process. Some of them help you modify executable files, while others work with the MS-DOS environment. MAKE is particularly helpful for automating large software projects. One of the crucial steps in developing software is linking. Though the user's guide for each language provides some information on linking, you will find here the complete, authoritative reference on the LINK utility.

Here are the new features of the CodeView debugger:

- **Multi-language expression evaluation**. The CodeView debugger has a built-in language interpreter, which will evaluate either BASIC, C, or FORTRAN expressions. You can specify which language you want to evaluate; or you can select the **Auto** option, which tells the CodeView debugger to select the language for you. This flexibility lets you take advantage of the features of more than one language; you can enter binary numbers in FORTRAN, use pointers in C, and so on.
- **Source-level support for MASM users**. Assembly programmers will now be able to view source files directly, including comments.
- **386 support**. The CodeView debugger now fully supports debugging of code written specifically for the 386 processor. Users of the new chip can decode and assemble 386 instructions, as well as view the expanded 386 registers.
- **Expanded memory support**. If you have expanded memory, then you can substantially reduce the amount of main memory required to debug any given program. This may enable you to debug large programs that you could not debug before.

- **8087 emulator support**. If you link to a Microsoft emulator library instead of using an 8087 coprocessor, then you can now take advantage of the **7** command. The debugger will display pseudo-8087 registers, as if you *did* have a math coprocessor in your machine.
- **Fewer restrictions**. The debugger will now support debugging of library modules and overlays.
- **New commands**. The SYMDEB commands Compare, Fill, Move, Input and Output have all been added to the CodeView debugger's repertoire.

I. About This Manual

This manual is intended as a companion volume to Microsoft language manuals. It is not language- specific, except where examples are required; and in those cases there is an attempt to use examples from a variety of languages.

The manual is divided into two parts: the first part explains how to use the CodeView debugger to examine and locate program errors. The second part explains the use of each of the other utilities, including LINK, LIB, MAKE, EXEPACK, EXEMOD, and others. The Appendixes at the end of the manual list the error messages and return codes for the CodeView debugger as well as the other utilities.

Some aspects of the CodeView debugger may not be familiar if you do not have a background in assembly-language programming. However, you can ignore assembly-mode features and concentrate on using high-level features in source mode.

The following list tells how to find information on various aspects of the CodeView debugger:

For This Information:	See:
How to start learning about the debugger	remainder of this introduction
Compiling and linking programs in the special format required by the CodeView debugger, and invoking the debugger with various command-line options	Chapter 1, "Getting Started"

Using elements of the CodeView display, including windows, pop-up menus, and the mouse	Chapter 2, "The CodeView Display"
Specifying arguments for dialog commands	Chapter 3, "Using Dialog Commands"
Using the CodeView operators to create expressions	Chapter 4, "CodeView Expressions"
Executing all or part of your program	Chapter 5, "Executing Code"
Testing the value of expressions, or examining data of different sizes	Chapter 6, "Examining Data and Expressions"
Setting, enabling, disabling, clearing, and listing breakpoints	Chapter 7, "Managing Breakpoints"
Creating watch statements and managing the watch window	Chapter 8, "Managing Watch Statements"
Examining code and tracing function, procedure, or subroutine calls	Chapter 9, "Examining Code"
Modifying data or code in memory	Chapter 10, "Modifying Code or Data"
Controlling the operation of the CodeView debugger	Chapter 11, "Using System-Control Commands"

The following information is included in Part 2:

For This Information:	**See:**
How to create executable files from object modules	Chapter 12, "Linking Object Files with LINK"
How to manage object modules by organizing them into libraries	Chapter 13, "Managing Libraries with LIB"
How to have object files and executable files updated automatically	Chapter 14, "Automating Program Development with MAKE"
Use of the other utilities	Chapter 15, "Using EXEPACK, EXEMOD, SETENV, and ERROUT"

In addition to the information above, the following information is included

in the appendixes:

For This Information:	See:
How to use regular expressions to find variable text strings in a source file	Appendix A, "Regular Expressions"
A list of error messages	Appendix B, "Error Messages"
Codes returned to DOS by various utilities	Appendix C, "Exit Codes"

Important

There may be additional information about the CodeView debugger in the **README.DOC** file. This file will describe any additions to the documentation or changes made to the program after the manual was printed.

II. Notational Conventions

The following notational conventions are used throughout this manual:

Example of Convention	Description of Convention
KEYWORDS and other concepts	Bold capital letters are used for the names of files, directories, registers, environment variables, high-level language keywords, and intrinsic functions. Commands typed at the DOS level are also capitalized. These commands include built-in DOS commands such as **SET**, as well as program names such as **FL** and **LINK**. You are not required to use capital letters when you actually enter DOS commands or high-level-language keywords and intrinsic functions. Bold type sometimes indicates text that must be typed exactly as shown. Text that must be typed as shown includes compiler options and high-level-language operators.

Examples are shown below:

```
+                INT4
CONTINUE         FUNCTION
IF               /Zi
```

placeholders

Italics mark placeholders in command lines and option specifications. A placeholder represents a variable item that must appear at a specific point. Consider the command syntax for the Radix command:

N*number*

Note that *number* is italicized to indicate that it represents a general form for the Radix (**N**) command. In an actual command, the user supplies a particular number for the placeholder *number*.

Occasionally, italics may be used to emphasize particular words in the text.

`Examples`

Examples are displayed in a special nonproportional typeface so that they will look more like the programs you create with a text editor or the output of commonly used computer printers. If a command produces output, the input that you type is shown in boldface, while the output displayed by the CodeView debugger is shown in regular, nonboldface type, as in the following example:

```
>RAX
AX 0041
:43
>
```

`Program`

`.`
`.`
`.`

`Fragment`

Vertical ellipsis dots are used in program examples to indicate that a portion of the program has been omitted. For instance, in the following excerpt, three statements are shown. The ellipsis dots between the statements indicate that intervening program lines occur, but are not shown.

```
COUNT = 0
.
.
```

```
.
PASS = PASS + 1
.
.
.
COUNT = 0
```

[[optional items]]

Double brackets enclose optional fields in command-line and option syntax. Consider the following command-line syntax:

R *[[register]]* *[[=]value]*

The double brackets around the placeholders indicate that you may enter a *register* and you may enter a *value*. The equal sign (=) in the second set of brackets indicates that you may place an equal sign before the *value*, but only if you specify a *value*.

[[choice1 | choice2]]

The vertical bar indicates that you may enter one of the entries shown on either side of the bar. The following syntax block illustrates a vertical bar:

DB *[[address | range]]*

The bar indicates that following the Dump Bytes command (**DB**), you can specify either an *address* or a *range*. Since both are in double brackets, you can also give the command with no argument.

"Quotation marks"

Quotation marks set off terms defined in the text. For example, the term "highlight" appears in quotation marks the first time it is defined.

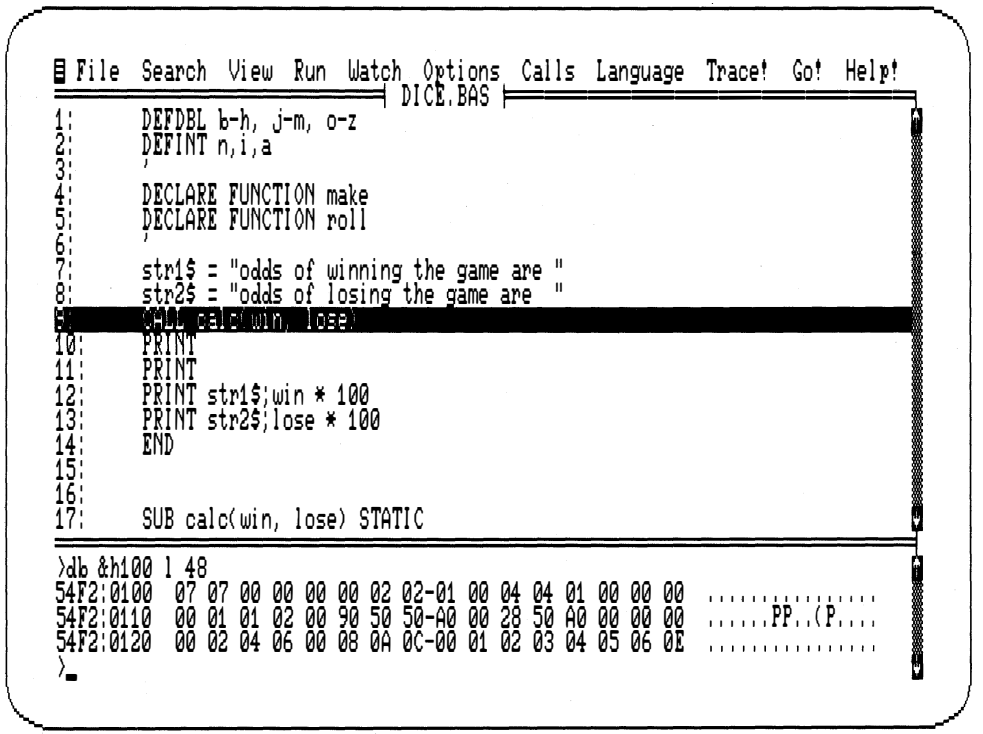
KEY NAMES

Small capital letters are used for the names of keys and key sequences, such as ENTER, CONTROL-C, and ALT-F.

Sample screens

Sample screens are shown in black and white. Your screens will look like this if you have a monochrome monitor, or if you use the **/B** option in the CodeView command line (see Section 1.4.1 in Chapter 1, "Getting Started"). The following figure shows an example. Screens will be slightly different if you use a color monitor in

color mode.



III. Definitions

The CodeView debugger deals with programs on both the source and assembly-language levels. Several terms used in this manual mean slightly different things, depending on whether source or assembly-language features are being discussed. Wherever possible, this manual will use an equivalent generic term rather than one of these language-dependent terms.

Terms that have a special use in this manual or are language dependent are listed below:

Term

	Meaning
DOS	A term used in Microsoft manuals to include both MS-DOS® and PC-DOS, except when noting features that are unique to one or the other.
Symbol	A name, often descriptive, that identifies a location anywhere in memory. In source languages, the names of variables and subprograms (subroutines or functions) are symbols.
Label	In source languages, a number that identifies a line in the source code (marking, for example, the end of a **DO** loop or the target of a **GOTO** statement). In assembly language, a label is a symbol that identifies an address in the code segment. The CodeView debugger does not recognize source-level line labels. Unless otherwise stated, all references to labels in this manual refer to assembly-language labels.
Routine	Any block of code that is called from somewhere else in a program. After the routine is executed, program control returns to the statement following the calling statement. In high-level languages, subprograms (subroutines or functions) are routines. Intrinsic functions are also routines (the call takes place on the assembly-language level to a routine in the language libraries). Assembly-language routines are also called "procedures."
Call	A statement that temporarily transfers program control to a routine, as described above.
CodeView format specifier	A one-letter code that may be appended to an expression to control how the value of that expression will be formatted for display. These specifiers come from **printf**, a function in the standard library of the C language.

IV. About the CodeView Debugger

The CodeView debugger can display and execute program code, control flow, and examine or change values in memory. Its window interface makes debugging easy. You can view your source code in one window, commands and responses in another, registers and flags in a third, and the values of variables or expressions in a fourth. You can examine the values of global or local variables, either by themselves or combined with other variables in expressions.

Important

You do *not* have to be an expert in assembly language to use the CodeView debugger; it brings the power of a full-featured debugging tool to high-level-language programmers. The debugger contains a full set of assembly-language-level features, but these are completely optional. When operating the CodeView debugger at the C, FORTRAN or BASIC source level, you use expressions from that language, and the debugger displays the values of the language's variables and expressions in decimal numbers.

If you wish to use only the debugger's source-level features, you will find some helpful suggestions in the next section. "Learning CodeView Features."

The window interface is designed for IBM® Personal Computers and IBM-compatible computers. However, you can also use the CodeView debugger with non-IBM-compatible computers using a sequential interface. Any debugging operation that can be performed with the window interface can also be performed with the sequential interface.

The CodeView debugger can access program locations through addresses, symbols, or line-number references. This makes it easy to locate and debug specific sections of code. You can debug programs at the source level, or you can examine code at the machine level in the debugger's assembly-language mode.

CodeView commands can be entered either from the keyboard or, in many cases, with the Microsoft Mouse (which is used with the window interface only). Once you learn the commands, you can work most efficiently using both the mouse and the keyboard. The mouse is not required; all commands can be entered from the keyboard.

Note

The CodeView debugger is designed specifically for the Microsoft Mouse. Many manufacturers advertise their pointing devices as being compatible with the Microsoft Mouse. The CodeView debugger may work with some of these devices if they are closely compatible.

The CodeView debugger is simple to learn and use. Its commands are logical and easy to understand, especially for programmers who are familiar with the Microsoft Symbolic Debug Utility (**SYMDEB**) or the **DEBUG** utility provided with DOS. The CodeView user interface shares some features with its predecessors, but also incorporates powerful new features such as pop-up menus, multiple windows, mouse support, and single-keystroke commands.

V. Learning CodeView Features

The CodeView debugger has many features, all of which are described in detail in this manual. If you are unfamiliar with source-level debuggers, some of these features will be new to you. If you are interested in learning to use the CodeView debugger only at the source level, you can simply ignore those features that don't apply to your needs. Assuming that you can use the CodeView window mode, here are a few suggestions on how to proceed:

1. Run through the sample session provided on the distribution disk. This will give you a broad idea of the CodeView debugger's power and flexibility.
2. When you begin to experiment with the CodeView debugger on your own, use only the menu commands. Their use rapidly becomes self evident, especially if you use the mouse. The menu commands are described in Chapter 3, "The CodeView Display."
3. When first reading through this manual, don't worry about the dialog commands or memory-level and assembly-level features. Pay special attention to any text that describes how to use a CodeView command with expressions. Most of your work with the CodeView debugger at the source level will involve high-level-language expressions and variable names.

4. Experiment with the sample-session program or another program you know well. Learn how to set and delete breakpoints, which stop program execution. Use the Go, Program Step, and Trace commands to execute parts of your program. Display some variables in the watch window. Watch the variables change as you execute different parts of your program. It is particularly instructive to set a breakpoint inside a loop and watch how the variables inside the loop change with each iteration. These tasks are explained in Chapters 5, 6 and 8.

5. Once you master breakpoints, learn how to use watchpoints and tracepoints, which stop program execution only when an expression becomes nonzero or changes. Watch statements are described in Chapter 9.

6. As you become more familiar with the CodeView debugger, try the Display Expression dialog command, which is entered on the CodeView command line with a question mark (`?`). The Display Expression command is described in Section 6.1.

7. Then graduate to the Dump commands. Learn how variables are stored in memory and how various Dump commands correspond to certain variable types. Experiment with the CodeView format specifiers. This knowledge will be valuable even at the source level, as it allows you to look simultaneously at all elements in an array. The Dump commands and format specifiers are explained in Chapter 7.

8. Learn the dialog commands that correspond to the menu commands you use frequently. They are often more powerful, and sometimes take less time to use than the menu commands. Dialog commands are described in Chapter 4.

Part 1
The CodeView Debugger

1 Getting Started 15
2 The CodeView Display 45
3 Using Dialog Commands 79
4 CodeView Expressions 85
5 Executing Code 115
6 Examining Data and Expressions 131
7 Managing Breakpoints 167
8 Managing Watch Statements 179
9 Examining Code 201
10 Modifying Code or Data 217
11 Using System-Control Commands 243

Chapter 1
Getting Started

1.1 Restrictions 17
1.2 Preparing Programs for the CodeView Debugger 18
1.2.1 Programming Considerations 18
1.2.2 CodeView Compile Options 19
1.2.3 CodeView Link Options 20
1.2.4 Preparing C Programs 21
1.2.5 Preparing FORTRAN Programs 23
1.2.6 Preparing BASIC Programs 24
1.2.7 Preparing MASM Programs 25
1.3 Starting the CodeView Debugger 27
1.4 Using CodeView Options 31
1.4.1 Starting with a Black-and-White Display 33
1.4.2 Specifying Start-Up Commands 34
1.4.3 Setting the Screen-Exchange Mode 35
1.4.4 Enabling Window or Sequential Mode 37
1.4.5 Turning Off the Mouse 38
1.4.6 Handling Interrupt Trapping 39
1.4.7 Using the Enhanced Graphics Adapter's 43-Line Mode 40
1.4.8 Using Expanded Memory (/E) 40
1.4.9 Using Two Video Adapters 41
1.5 Debugging Large Programs 41
1.6 Working with Older Versions of MASM 42

Getting started with the CodeView debugger requires several simple steps. You must prepare a special-format executable file for the program you wish to debug; then you invoke the debugger. You may also wish to specify options that will affect the debugger's operation.

This chapter describes how to produce executable files in the CodeView format using C, FORTRAN, BASIC, or the Macro Assembler, and how to load a program into the CodeView debugger. The chapter lists restrictions and programming considerations with regard to the debugger, which you may want to consult before compiling or assembling. Finally, the chapter describes how to use the debugger with Macro Assembler versions 1.0 through 4.0.

1.1 Restrictions

The following restrictions apply generally to the use of the CodeView debugger, regardless of the language being used. This list briefly describes kinds of files that are not directly supported by the debugger.

Restriction	Explanation
Include files	You will not be able to use the CodeView debugger to debug source code in include files.
Packed files	CodeView symbolic information cannot be put into a packed file.
.COM files	Files with the extension **.COM** can be debugged in assembly mode only; they can never contain symbolic information.
Memory-resident programs	The CodeView debugger can only work with disk-resident **.EXE** and **.COM** files. Debugging of memory-resident files such as device drivers or interrupt handlers is not supported.

Some of the features that are *no longer* disallowed by CodeView include: debugging of library modules, and debugging of overlayed code. CodeView users can now freely debug library modules and overlays.

1.2 Preparing Programs for the CodeView Debugger

You must compile and link with the correct options, in order to use a program with the CodeView debugger. These options direct the compiler and the linker to produce an executable file, which contains line-number information and a symbol table, in addition to the executable code.

Note

For the sake of brevity, this section and its three subsections use the term "compiling" to refer to the process of producing object modules. However, most everything said about compiling in this section applies equally well to assembling. Exceptions are noted in Section 1.2.7, "Preparing MASM Programs."

Not all compiler and linker versions support CodeView options. (Consult the section on the appropriate language below, for information about compiler versions. Also, you will need to use the Microsoft Overlay Linker, version 3.5 or later.) If you try to debug an executable file that was *not* compiled and linked with CodeView options, or if you use a compiler that does not support these options, then you will only be able to use the debugger in assembly mode. This means that the CodeView debugger will not be able to display source code or understand source-level symbols, such as symbols for functions and variables.

1.2.1 Programming Considerations

Any source code that is legal in C, FORTRAN, BASIC, or MASM can be compiled or assembled to create an executable file, and then debugged with the CodeView debugger. However, some programming practices make debugging more difficult.

Each of Microsoft languages listed above permit you to put code in separate include files, and read the files into your source file by using an include directive. However, you will not be able to use the CodeView debugger to debug source code in include files. The preferred method of developing programs is to create separate object modules, then link the object modules in with your program. The CodeView debugger supports

the debugging of separate object modules in the same session.

Also, the CodeView debugger will be more effective and easier to use if you put each source statement on a a separate line. A number of languages (C and BASIC in particular) permit you to place more than one statement on a single line of the source file. This practice does not prevent the CodeView debugger from functioning. However, the debugger must treat the line as a single unit; it cannot break the line down into separate statements. Therefore, if you have three statements on the same line, you will not be able to put a breakpoint or freeze execution on the individual statements. The best you will be able to do is freeze execution at the beginning of the three statements, or at the beginning of the next line.

Some languages (C and MASM in particular) support some kind of macro expansion. However, the CodeView debugger will not help you debug macros in source mode. You will need to expand the macros yourself before debugging them; otherwise, the debugger will treat them as simple statements or instructions.

Finally, your segments should be declared according to the standard Microsoft format. This is taken care of for you automatically, with each of the Microsoft high-level languages.

1.2.2 CodeView Compile Options

Note

Microsoft compilers will accept command-line options that are preceded by either a forward slash (/) or a dash (–). For brevity, this manual will list only the forward slash when describing options, but you may use either symbol.

Case is significant for options used with the C, FORTRAN, and BASIC compilers; you must type the letters exactly as given.

When you compile a source file for a program you want to debug, you must specify the **/Zi** option on the command line. The **/Zi** option instructs the compiler to include line-number and symbolic information in the object file.

If you do not need complete symbolic information in some modules, you can compile those modules with the **/Zd** option instead of **/Zi**. The **/Zd** option writes less symbolic information to the object file, so using this option will save disk space and memory. For example, if you are working on a program made up of five modules, but you only need to debug one module, you can compile that module with the **/Zi** option and the other modules with the **/Zd** option. You will be able to examine global variables and see source lines in modules compiled with the **/Zd** option, but local variables will be unavailable.

In addition, if you are working with a high-level language, you will probably want to use the **/Od** option, which turns off optimization. Optimized code may be rearranged for greater efficiency and, as a result, the instructions in your program may not correspond closely to the source lines. After debugging, you can compile a final version of the program with the optimization level you prefer.

You cannot debug a program until you compile it successfully. The CodeView debugger will not help you correct syntax or compiler errors. Once you successfully compile your program, you can then use the debugger to locate logical errors in the program.

Compiling examples are given in the sections on compiling and linking with specific languages.

1.2.3 CodeView Link Options

If you use **LINK** separately to link an object file or files for debugging, you should specify the **/CODEVIEW** option (it can be abbreviated as **/CO**). This instructs the linker to incorporate addresses for symbols and source lines into the executable file.

Note that when you use a Microsoft driver program which automatically invokes the linker (such as **CL** with C, or **FL** with FORTRAN), then the linker will automatically be invoked with the **/CO** option whenever you specify **/Zi** on the command line. You do not use **/CO** unless you are invoking the linker directly, by typing **LINK**.

Although executable files prepared with the **/CODEVIEW** option can be executed from the DOS command line like any other executable files, they are larger because of the extra symbolic information in them. To minimize program size, you will probably want to recompile and link your final version without the **/Zi** option when you finish debugging a program.

Linking expamples are given in the sections on compiling and linking with specific languages.

1.2.4 Preparing C Programs

In order to use the CodeView debugger with a program written in C, you will need to compile it with the Microsoft C Compiler, version 4.0 or later. Earlier versions of the compiler do not support the CodeView compile options. You will also need to link with the Microsoft Overlay Linker, version 3.5 or later.

Writing C Source

Microsoft C supports the use of include files, through use of the **#include** directive. However, you will not be able to debug source code put into include files. Therefore, you should reserve the use of include files for **#define** macros, and structure definitions.

The C language permits you to put more than one statement on a line. This practice makes it difficult for you to debug such lines of code. For example, the following code is legal in C:

```
code = buffer[count]; if (code == '\n') ++lines;
```

This code is made up of three separate source statements. When placed on the same line, the individual statements cannot be accessed during debugging. You could not, for example, stop program execution at `++lines;` The same code would be easier to debug if it were written in the following form:

```
code = buffer[count];
if (code == '\n')
        ++lines;
```

This makes code easier to read and corresponds with what is generally considered good programming practice.

You cannot easily debug macros with the CodeView debugger. The debugger will not break down the macro for you. Therefore, if you have complex macros with potential side effects, you may need to write them first as regular source statements.

Compiling and Linking C Programs

The **/Zi**, **/Zd**, and **/Od** options are all supported by the Microsoft C Compiler, version 4.0 and later. These options are accepted by both the **CL** driver, and the **MSC** driver which was supplied with version 4.0 of the compiler. (For a description of these options, see Section 1.2.2, "CodeView Compile Options.") Linking separately with **/CO** is only necessary when you compile with **MSC**. The CodeView debugger supports mixed-language programming; for an example of how to link a C module with modules from other languages, see Section 1.2.7, "Preparing MASM Programs."

■ Examples

```
CL /Zi /Od EXAMPLE.C

MSC /Zi /Od EXAMPLE;
LINK /CO EXAMPLE;

CL /Zi /Od /c MOD1.C
CL /Zd /Od /c MOD2.C
CL /Zi MOD1 MOD2
```

In the first example, **CL** is used to compile and link the source file **EXAMPLE.C**. **CL** creates an object file in the CodeView format, **EXAMPLE.OBJ**, and then automatically invokes the linker with the **/CO** option. The second example demonstrates how to compile and link the source file **EXAMPLE.C**, using the **MSC** program provided with version 4.0 of the compiler. Since **MSC** does not invoke the linker, linking must be done separately, with the **/CO** option given explicitly to the linker. Both examples result in an executable file, **EXAMPLE.EXE**, which has the line-number information, symbol table, and unoptimized code required by the CodeView debugger.

In the third example, the source module **MOD1.C** is compiled to produce an object file with full symbolic and line information, while **MOD2.C** is compiled to produce an object file with limited information. Then, **CL** is used again, to link the resulting object files. (This time, **CL** does not recompile, because the arguments have no **.C** extension.) Typing **/Zi** on the command line causes the linker to be invoked with the **/CO** option. The result is an executable file in which one of the modules, **MOD2.C**, will be harder to debug. However, the executable file will take up substantially less space on disk than it would if both modules were compiled with full symbolic information.

1.2.5 Preparing FORTRAN Programs

In order to use the CodeView debugger with a program written in FORTRAN, you will need to compile it with the Microsoft FORTRAN Optimizing Compiler, version 4.0 or later. Earlier versions of the compiler do not support the CodeView compile options. You will also need to link with the Microsoft Overlay Linker, version 3.5 or later.

Writing FORTRAN Source

The Microsoft FORTRAN compiler supports the use of include files, through use of the **$INCLUDE** directive. However, you will not be able to debug source code in an include file. If you have source code that you wish to put in separate files, then you should use the technique of separately compiled modules. The CodeView debugger does support this technique, by allowing you to trace through separate source files in the same session.

Compiling and Linking FORTRAN Programs

The **/Zi**, **/Zd**, and **/Od** options are all supported by the Microsoft FORTRAN Optimizing Compiler, version 4.0. For a description of these options, see Section 1.2.2, "CodeView Compile Options." The CodeView debugger supports mixed-language programming; for an example of how to link a FORTRAN module with modules from other languages, see Section 1.2.7, "Preparing MASM Programs."

- **Examples**

```
FL /Zi /Od EXAMPLE.FOR

FL /Zi /Od /c EXAMPLE.FOR
LINK /CO EXAMPLE;

FL /Zi /Od /c MOD1.FOR
FL /Zd /Od /c MOD2.FOR
FL /Zi MOD1 MOD2
```

In the first example, **FL** is used to compile and link the source file **EXAMPLE.FOR**. **FL** creates an object file in the CodeView format, **EXAMPLE.OBJ**, and then automatically invokes the linker with the **/CO** option. The second example demonstrates how to compile and link the source file **EXAMPLE.FOR**, using separate steps for compiling and

linking. In this case, the **/CO** option must be given explicitly to the linker. Both examples result in an executable file, **EXAMPLE.EXE**, which has the line-number information, symbol table, and unoptimized code required by the CodeView debugger.

In the third example, the source module **MOD1.FOR** is compiled to produce an object file with full symbolic and line information, while **MOD2.FOR** is compiled to produce an object file with limited information. Then **FL** is used again, to link the object files. (Note that this time, **FL** does not recompile, because the arguments have no **.FOR** extension.) Typing **/Zi** on the command line causes the linker to be invoked with the **/CO** option. The result is an executable file in which one of the modules, **MOD2.FOR**, will be harder to debug. However, the executable file will take up substantially less space on disk than it would if both modules will compiled with full symbolic information.

1.2.6 Preparing BASIC Programs

In order to use the CodeView debugger with a program written in BASIC, you will need to compile it with Microsoft QuickBASIC 3.0 or later, or the IBM BASIC Compiler version 3.0 or later. Earlier versions of each of these products do not support the CodeView compile options. You will also need to link with the Microsoft Overlay Linker, version 3.5 or later.

Writing BASIC Source

Microsoft BASIC supports the use of include files, through the use of the **REM $INCLUDE** directive. However, you will not be able to debug source code put into include files. The preferred practice for developing source code in separate files is to use separately compiled modules. The CodeView debugger does support this technique, by allowing you to trace through separate source files in the same session.

BASIC also permits you to put more than one statement on a line. This practice makes it difficult for you to debug such lines of code. For example, the following code is legal, even common, in BASIC:

```
SUM=0 : FOR I=1 TO N : SUM=SUM+ARRAY(I) : NEXT I
```

This code is actually made up of four separate BASIC statements. When placed on the same line, the individual statements cannot be accessed during debugging. You could not, for example, stop program execution at `SUM=SUM+ARRAY(I)`. The same code would be easier to debug if it were

written in the following form:

```
SUM=0
FOR I=1 TO N
    SUM=SUM+ARRAY(I)
NEXT I
```

This form is helpful anyway, because it makes programs easier to read.

Compiling and Linking BASIC Programs

Only the **/Zi** option is supported by **BASCOM**. **/Zd** and **/Od** are not supported. (For a description of these options, see Section 1.2.2, "CodeView Compile Options.") Consult the QuickBASIC manual for directions on how to produce QuickBASIC programs in the CodeView format. Linking separately with **/CO** is necessary when you compile with **BASCOM**. The CodeView debugger supports mixed-language programming; for an example of how to link a BASIC module with modules from other languages, see Section 1.2.7, "Preparing MASM Programs"

- **Example**

```
BASCOM /Zi EXAMPLE;
LINK /CO EXAMPLE;
```

The above example compiles the source file **EXAMPLE.BAS**, to produce an object file, **EXAMPLE.OBJ**, which contains the symbol and line-number information required by the CodeView debugger. Then the linker is invoked with the **/CO** option, to create an executable file that can be used with the debugger.

1.2.7 Preparing MASM Programs

In order to use all the features of the CodeView debugger with Macro Assembler programs, you will need to assemble with MASM version 5.0 or later. (Section 1.6 discusses how to use earlier versions of MASM with the debugger.) No matter what version of MASM you use, you will need to link with the Microsoft Overlay Linker, version 3.5 or later.

Writing MASM Source

If you have version 5.0 of MASM, then you should use the segment directives described in the *Macro Assembler Programmer's Guide.* Use of these directives ensures that segments will be declared in the correct way for use with the CodeView debugger. If you do not use these directives, then you need to make sure that the class type for the code segment is CODE.

Debugging of macros is not directly supported by the CodeView debugger. You can always see how a macro was expanded by choosing mixed or assembly mode (assembly mode displays straight, unassembled machine code), but then you will not see the macro as it was originally written. Particularly if you have complex conditionals, you may want to expand the macro yourself, for the purpose of debugging.

MASM also supports include files; as with the C language, you are better off reserving include files for macro and structure definitions. You will not be able to debug code in an include file.

Assembling and Linking

MASM supports the **/Zi** and **/Zd** options at assemble-time; however, the **/Od** option does not apply, since MASM does not optimize code for you. Unlike the compiler options that support the CodeView debugger, the MASM options are not case-sensitive. You may therefore enter **/ZI** or **/ZD** on the MASM command line, to produce an object file in the CodeView format.

After assembling, you will need to link with the **/CO** option.

■ Examples

```
MASM /ZI EXAMPLE;
LINK /CO EXAMPLE;

MASM /ZI MOD1;
MASM /ZD MOD2;
LINK /CO MOD1 MOD2;

CL /Zi /Od /c /AL prog.c
FL /Zi /Od /c sub1.for
BASCOM /Zi sub2;
MASM /ZI sub3;
LINK /CO prog sub1 sub2 sub3
```

The first example assembles the source file **EXAMPLE.ASM**, and produces the object file **EXAMPLE.OBJ**, which is in the CodeView format. The linker is then invoked with the **/CO** option, and produces an executable file with the symbol table and line-number information required by the debugger. The second example produces the object file **MOD1.OBJ**, which contains symbol and line-number information; and the object file **MOD2.OBJ**, which contains line-number information but no symbol table. The object files are then linked. The result is an executable file in which the second module will harder to debug. This executable file, however, will be smaller than it would be if both modules were assembled with the **/ZI** option.

The last example demonstrates how to create a mixed-language executable file that can be used with the CodeView debugger. The debugger will be able to trace through different source files in the same session, regardless of the language. Note that **LINK** is being used in prompt mode (there is no semi-colon at the end of the line), so that the user will be prompted for all the needed libraries.

1.3 Starting the CodeView Debugger

Before starting the debugger, make sure all the files it requires are available in the proper places. The following files are recommended for source-level debugging:

File	Location
CV.EXE	The CodeView program file can be in the current directory, or in any directory accessible with the **PATH** command. For example, if you set up your compiler files on a hard disk using the **SETUP** program provided on the distribution disk, you might put **CV.EXE** in the **\BIN** directory. If you have an older version of the debugger, take care to remove any copies of **CV.EXE** from directories in your **PATH**. The debugger has an overlay manager which reloads the file **CV.EXE** from time to time. If it reloads the wrong version of this file, then your machine will likely hang.
CV.HLP	If you want to have the on-line help available during your session, you should have this file

	either in the current directory, or in any directory accessible with the **PATH** command. For example, if you set up your compiler files on a hard disk using the **SETUP** program provided on the distribution disk, you might put **CV.HLP** in the **\BIN** directory. If the CodeView debugger cannot find the help file, you can still use the debugger, but you will see an error message if you try to use one of the help commands.
program.**EXE**	The executable file for the program you wish to debug must be in the current directory, or in a drive and directory you specify as part of the start-up file specification. The CodeView debugger will display an error message and refuse to start if the executable file is not found.
source.ext (extension depends on language)	Normally, source files should be in the current directory. However, if you specify a file specification for the source file during compilation, that specification will become part of the symbolic information stored in the executable file. For example, if you compiled with the command line argument DEMO, the CodeView debugger will expect the source file to be in the current directory. However, if you compiled with the command line argument \SOURCE\DEMO, then the debugger will expect the source file to be in directory \SOURCE. If the debugger cannot find the source file in the directory specified in the executable file (usually the current directory), the program will prompt you for a new directory. You can either enter a new directory, or you can press the ENTER key to indicate that you do not want a source file to be used for this module. If no source file is specified, you must debug in assembly mode.

If the appropriate files are in the correct directories, you can enter the CodeView command line at the DOS command prompt. The command line has the following form:

CV [*options*] *executablefile* [*arguments*]

The *options* are one or more of the options described in Section 1.4. The

executablefile is the name of an executable file to be loaded by the debugger. It must have the extension **.EXE** or **.COM**. If you try to load a nonexecutable file, the following message appears:

```
Not an executable file
```

Compiled programs and assembly-language programs containing CodeView symbolic information will always have the extension **.EXE**. Files with the extension **.COM** can be debugged in assembly mode, but they can never contain symbolic information.

The optional *arguments* are parameters passed to the *executablefile*. If the program you are debugging does not accept command-line arguments, you do not need to pass any arguments.

If you specify the *executablefile* as a file name with no extension, the CodeView debugger searches for a file with the given base name and the extension **.EXE**. Therefore, you must specify the **.COM** extension if you are debugging a **.COM** file. If the file you specify is not in the CodeView format, the debugger starts in assembly mode and displays the following message:

```
No symbolic information
```

You must specify an executable file when you start the CodeView debugger. If you omit the executable file, the debugger displays a message showing the correct command-line format.

When you give the debugger a valid command line, the executable program and the source file are loaded, the address data is processed, and the CodeView display appears. The initial display will be in window mode or sequential mode, depending on the options you specify and the type of computer you have.

For example, if you wanted to debug the program **BENCHMRK.EXE**, you could start the debugger with the following command line:

```
CV BENCHMRK
```

If you give this command line on an IBM Personal Computer, window mode will be selected automatically. The display will look like Figure 1.1.

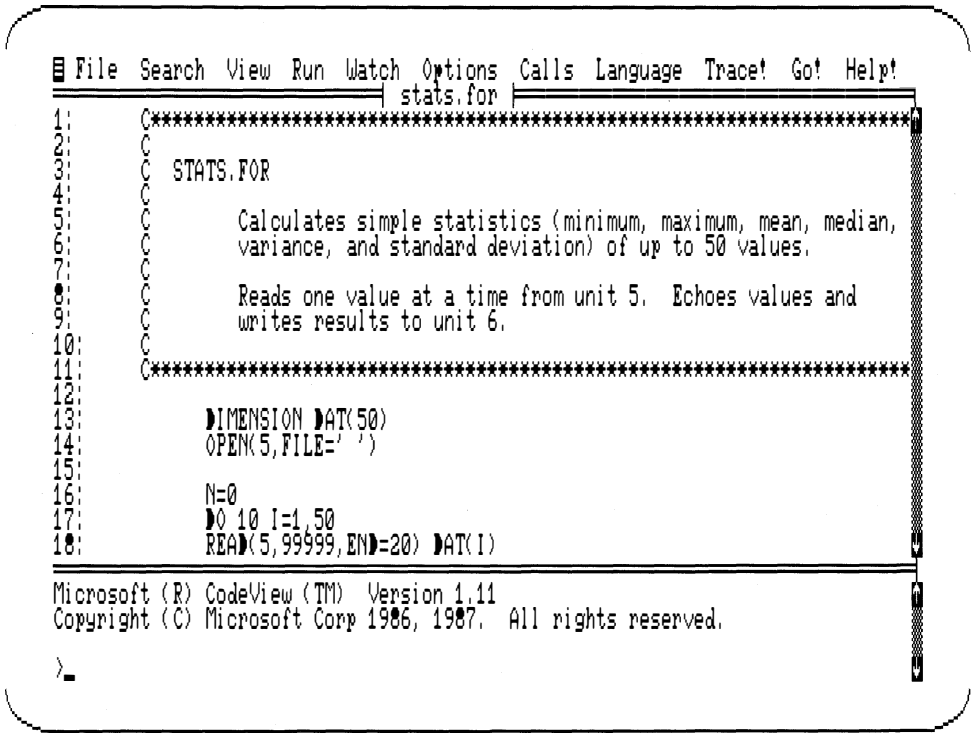


Figure 1.1 CodeView Start-Up Screen in Window Mode

If you give the same command line on most non-IBM computers, sequential mode will be selected. The following lines appear:

```
Microsoft (R) CodeView  Version 1.10
Copyright (C) Microsoft Corp 1986, 1987. All rights reserved.

>
```

You can use CodeView options, as described in Section 1.4, to override the default start-up mode.

If your program is written in a high-level language, the CodeView debugger is now at the beginning of the start-up code that precedes your program. In source mode, you can enter an execution command (such as Trace or Program Step) to automatically execute through the start-up code to the beginning of your program. At this point, you are ready to start debugging your program, as described in Chapters 3–11.

1.4 Using CodeView Options

You can change the start-up behavior of the debugger by specifying options in the command line.

An option is a sequence of characters preceded by either a forward slash (/) or a dash (–). For brevity, this manual will list only the forward slash when describing options, but you may use either. Unlike compiler command-line options, CodeView command-line options are not case sensitive.

A file whose name begins with a dash must be renamed before you use it with the CodeView debugger, so that the debugger will not interpret the dash as an option designator. You can use more than one option in a command line, but each option must have its own option designator and spaces must separate each option from other elements of the command line.

Note

The CodeView debugger's defaults are different for IBM Personal Computers than for other computers. However, the debugger may not always recognize the difference between computers. The debugger determines if you have an IBM computer by looking at certain locations in memory.

The following list suggests some situations in which you might want to use an option. If more than one condition applies, you can use more than one option (in any order). If none of the conditions applies, you need not use any options.

If:	Type:
You have an IBM-compatible computer and you want to use window mode	/W
You have a two-color monitor, a color graphics adapter (CGA), and an IBM or IBM-compatible computer	/B
You are debugging a graphics program and	

you want to be able to see the output screen	/S
You are debugging a program that uses multiple video-display pages and you want to be able to see the output screen	/S
You are using a non-IBM-compatible computer and you want to be able to see the output screen	/S
You are using an IBM-compatible computer to debug a program that does not use graphics or multiple video-display pages and you want to be able to see the output screen	/F
You are using an IBM-compatible computer that does not support certain IBM-specific interrupt trapping functions	/D
You are using a non-IBM-compatible computer and you want to enable CONTROL-C and CONTROL-BREAK	/I
You have an IBM computer, but you wish to debug in sequential mode (for example, with redirection)	/T
You have a mouse installed in your system, but you do not want to use it during the debugging session	/M
You want a 43-line display and you have an IBM or IBM-compatible computer with an enhanced graphics adapter (EGA) and an enhanced color display	/43
You want to use two monitors with the CodeView debugger	/2
You want the CodeView debugger to automatically execute a series of commands when it starts up	/C*commands*

For example, assume you are using an IBM-compatible computer with a CGA and a two-color monitor. The program you are debugging, which you could name GRAPHIX.EXE, plots points in graphics mode. You want to be able to see the output screen during the debugging session. Finally, you want to be able to start the debugger several times without having to remember all the options, and you want to execute the high-level language start-up code automatically each time. You could create a batch file called

STATSBUG.BAT consisting of the following line:

```
CV /W /B /S /CGmain GRAPHIX
```

The CodeView options are described in more detail in Sections 1.4.1–1.4.9.

1.4.1 Starting with a Black-and-White Display

■ Option

/B

The **/B** option forces the CodeView debugger to display in two colors even if you have a CGA. By default, the debugger checks on start-up to see what kind of display adapter is attached to your computer. If the debugger detects a monochrome adapter (MA), it displays in two colors. If it detects a CGA, it displays in multiple colors.

If you use a two-color monitor with a CGA, you may want to disable color. Monitors that display in only two colors (usually green and black, or amber and black) often attempt to show colors with different cross-hatching patterns, or in gray-scale shades of the display color. In either case, you may find the display easier to read if you use the **/B** option to force black-and-white display. Most two-color monitors still have four color distinctions: background (black), normal text, high-intensity text, and reverse-video text.

■ Example

```
CV /B CALC CALC.DAT
```

The above example starts the CodeView debugger in black-and-white mode. This is the only mode available if you have a monochrome adapter. The display is usually easier to read in this mode if you have a CGA and a two-color monitor.

1.4.2 Specifying Start-Up Commands

■ Option

/Ccommands

The **/C** option allows you to specify one or more *commands* that will be executed automatically upon start-up. You can use these options to invoke the debugger from a batch or **MAKE** file. Each command is separated from the previous command by a semicolon.

If one or more of your start-up commands has arguments that require spaces between them, you should enclose the entire option in double quotation marks. Otherwise, the debugger will interpret each argument as a separate CodeView command-line argument rather than as a debugging-command argument.

Warning

Any start-up option that uses the less-than (<) or greater-than (>) symbol must be enclosed in double quotation marks even if it does not require spaces. This ensures that the redirection command will be interpreted by the CodeView debugger rather than by DOS.

■ Examples

```
CV /CGmain CALC CALC.DAT
```

The above example loads the CodeView debugger with CALC as the executable file and CALC.DAT as the argument. Upon start-up, the debugger executes the high-level language start-up code with the command Gmain. Since no space is required between the CodeView command (**G**) and its argument (**main**), the option is not enclosed in double quotation marks.

```
CV "/C;S&;G INTEGRAL;DS ARRAYX L 20" CALC CALC.DAT
```

The above example loads the same file with the same argument as the first example, but the command list is more extensive. The debugger starts in mixed source/assembly mode (S&). It executes to the routine INTEGRAL (G

INTEGRAL), then dumps 20 short real numbers, starting at the address of the variable ARRAYX (DS ARRAYX L 20). Since several of the commands use spaces, the entire option is enclosed in double quotation marks.

```
CV "/C<INPUT.FIL" CALC CALC.DAT
```

The above example loads the same file and argument as the first example, but the start-up command directs the debugger to accept input from the file INPUT.FIL rather than from the keyboard. Although the option does not include any spaces, it must be enclosed in double quotation marks so that the less-than symbol will be read by the CodeView debugger rather than by DOS.

1.4.3 Setting the Screen-Exchange Mode

- **Options**

/F
/S

The CodeView debugger allows you to move quickly back and forth between the output screen, which contains the output from your program, and the debugging screen, which contains the debugging display. The debugger can handle this screen exchange in two ways: screen flipping or screen swapping. The **/F** option (screen flipping) and the **/S** option (screen swapping) allow you to choose the method from the command line.

If neither method is specified (possible only on non-IBM computers), the Screen Exchange command will not work. No screen exchange is the default for non-IBM computers. Screen flipping is the default for IBM computers with graphics adapters, and screen swapping is the default for IBM computers with monochrome adapters.

Screen flipping uses the video-display pages of the graphics adapter to store each screen of text. Video-display pages are a special memory buffer reserved for multiple screens of video output. This method is faster and uses less memory than screen swapping. However, screen flipping cannot be used with an MA, or to debug programs that produce graphics or use the video-display pages. In addition, the CodeView debugger's screen flipping works only with IBM and IBM-compatible microcomputers.

Screen swapping has none of the limitations of screen flipping, but is significantly slower and requires more memory. In the screen-swapping method, the CodeView debugger creates a buffer in memory and uses it to store the screen that is not being used. When the user requests the other screen, the debugger swaps the screen in the display buffer for the one in the storage buffer.

When you use screen swapping, the buffer size is 16K for all adapters. The amount of memory used by the CodeView debugger is increased by the size of the buffer.

Table 1.1 shows the default exchange mode (swapping or flipping) and the default display mode (sequential or window) for various configurations. Display modes are discussed in Section 1.4.4, "Enabling Window or Sequential Mode."

Table 1.1

Default Exchange and Display Modes

Computer	Display Adapter	Default Modes	Alternate Modes
IBM	CGA or EGA	/F /W	/S if your program uses video-display pages or graphics; /T for sequential mode
IBM compatible	CGA or EGA	/T	/W for window mode; /F for screen flipping with text programs, or /S for screen swapping with programs that use video-display pages or graphics
IBM	MA	/S /W	/T for sequential mode
IBM compatible	MA	/T	/W for window mode; /S for screen swapping
Noncompatible	Any	/T	/S for screen swapping

If you are not sure if your computer is completely IBM compatible, you can experiment. If the basic input/output system (BIOS) of your computer is not compatible enough, the CodeView debugger may not work with the /F option.

If you specify the **/F** option with an MA, the debugger will ignore the option and use screen swapping. If you try to use screen flipping to debug a program that produces graphics or uses the video-display pages, you may get unexpected results and have to start over with the **/S** option.

■ Examples

```
CV /F CALC CALC.DAT
```

The above example starts the CodeView debugger with screen flipping. You might use this command line if you have an IBM-compatible computer, and you want to override the default screen-exchange mode in order to use less memory and switch screens more quickly. The option would not be necessary on an IBM computer, since screen flipping is the default.

```
CV /S GRAFIX
```

The above example starts the debugger with screen swapping. You might use this command line if your program uses graphics mode.

1.4.4 Enabling Window or Sequential Mode

■ Options

/T
/W

The CodeView debugger can operate in window mode or in sequential mode. Window mode displays up to four windows, enabling you to see different aspects of the debugging-session program simultaneously. You can also use a mouse in window mode. Window mode requires an IBM or IBM-compatible microcomputer.

Sequential mode works with any computer, and is useful with redirection commands. Debugging information is displayed sequentially on the screen.

The behavior of each mode is discussed in detail in Chapter 3, "The CodeView Display." Refer back to Table 1.1 for the default and alternate modes for your computer. If you are not sure if your computer is completely IBM compatible, you can experiment with the options. If the BIOS of your computer is not compatible enough, you may not be able to use

window mode (the **/W** option).

Note

Although window mode is more convenient, any debugging operation that can be done in window mode can also be done in sequential mode.

■ Examples

```
CV /W SIEVE
```

The above example starts the CodeView debugger in window mode. You will probably want to use the **/W** option if you have an IBM-compatible computer, since the default sequential mode is less convenient for most debugging tasks.

```
CV /T SIEVE
```

The above example starts the debugger in sequential mode. You might want to use this option if you have an IBM computer and you have a specific reason for using sequential mode. For instance, sequential mode usually works better if you are redirecting your debugging output to a remote terminal.

1.4.5 Turning Off the Mouse

■ Option

/M

If you have a mouse installed on your system, you can tell the CodeView debugger to ignore it, using the **/M** option. You may need to use this option if you are debugging a program that uses the mouse and your mouse is not a Microsoft Mouse. This is due to a conflict between the program's use of the mouse and the debugger's use of it. If you use the **/M** option, the program you are debugging can still use the mouse, but the CodeView debugger cannot.

Important

The same conflict between program and debugger applies if you are not using the version of the Microsoft Mouse driver programs (**MOUSE.SYS** and **MOUSE.COM**) included on the distribution disks for certain Microsoft products. You may want to replace your old mouse driver program with the updated version. You will then be able to use the mouse with both the CodeView debugger and the program you are debugging. If you did not install a mouse driver when you set up Microsoft FORTRAN 4.0, Microsoft C 4.1, or Macro Assembler 5.0, see your User's Guide for information on installing **MOUSE.SYS** and **MOUSE.COM**. These programs will not work with pointing devices from other manufacturers.

1.4.6 Handling Interrupt Trapping

- **Options**

/D
/I

The **/D** option turns off nonmaskable interrupt (NMI) and 8259 interrupt trapping. If you are using an IBM PC Convertible, Tandy® 1000, or the AT&T® 6300 Plus and you are experiencing system crashes while using the CodeView debugger, try starting with the **/D** option. To enable window mode, use **/W** with **/D**; otherwise sequential mode is set automatically. Note that because this option turns off interrupt trapping, CONTROL-C and CONTROL-BREAK will not work, and an external interrupt may occur during a trace operation. If this happens you may find yourself tracing the interrupt handler instead of your program.

The **/I** option forces the debugger to handle NMI and 8259 interrupt trapping. Use this option to enable CONTROL-C and CONTROL-BREAK on computers the debugger does not recognize as being IBM compatible, such as the Eagle® PC. Window mode is set automatically with the **/I** option; you don't have to specify **/W**. Using the **/I** option lets you stop program execution at any point while you are using the CodeView debugger.

1.4.7 Using the Enhanced Graphics Adapter's 43-Line Mode

■ Option

■ Option

/43

If you have an EGA and a monochrome monitor or an enhanced color display monitor (or a compatible monitor), you can use the **/43** option to enable a 43-line-by-80-column text mode. You cannot not use this mode with other monitors, or if you have a CGA or an MA. The CodeView debugger will ignore the option if it does not detect an EGA.

The EGA's 43-line mode performs identically to the normal 80-column-by-25-line mode used by default on the EGA, CGA, and MA. The advantage of the 43-line mode is that more text fits on the CodeView display; the disadvantage is that the text is smaller and harder to read. If you have an EGA, you can experiment to see which size you prefer.

■ Example

```
CV /43 CALC CALC.DAT
```

The above example starts the CodeView debugger in 43-line mode if you have an EGA video adapter and an enhanced color or monochrome monitor. The option will be ignored if you do not have the hardware to support it.

1.4.8 Using Expanded Memory (/E)

■ Option

/E

The /E option enables the use of expanded memory. If expanded memory is present, the CodeView debugger will use it to store the symbolic information of the program. This may be as much as 40% of the size of the executable file for the program, and represents space that would otherwise be

taken up in main memory.

Note

This option enables exapanded memory only, not *extended.*

1.4.9 Using Two Video Adapters

- **Option**

/2

The **/2** option permits the use of two monitors with the CodeView debugger. The program display will appear on the current default monitor, while the CodeView display appears on the other monitor. You must have two monitors and two adapters to use the **/2** option. For example, if you have both a color graphics adapter and a monochrome adapter, you might want to set the CGA up as the default adapter. You could then debug a graphics program with the graphics display appearing on the graphics monitor and the debugging display appearing on the monochrome monitor. Microsoft Mouse support will be disabled on the debugging display if you use this option.

1.5 Debugging Large Programs

Because the CodeView debugger must reside in memory along with the program you are debugging, there may not be enough room to debug some large programs that could otherwise run in memory alone. However, there are at least two ways to get around memory limitations:

1. If you have expanded memory, use the **/E** option described earlier. This will enable CodeView to put the symbol table out on expanded memory, thus freeing up a good deal of main memory.
2. Since CodeView now supports the debugging of overlayed programs, you can substantially reduce the amount of memory required to run your program by using overlays when you link your program.

1.6 Working with Older Versions of MASM

You can use the CodeView debugger with files developed with the Microsoft (or IBM) Macro Assembler. Since the Microsoft Macro Assembler (Versions 1.0 through 4.0) does not write line numbers to object files, some of the CodeView debugger's features will not be used when you debug programs developed with the assembler.

The debugger can be used on either **.EXE** or **.COM** files, but you can only view symbolic information in **.EXE** files. The procedure for assembling and debugging **.EXE** files is summarized below:

1. In your source file, declare public any symbols, such as labels and variables, that you want to reference in the debugger. If the file is small, you may want to declare all symbols public.
2. Assemble as usual. No special options are required, and all assembly options are allowed.
3. Use LINK version 3.5 or later. Do not use the linker provided with versions 4.0 and earlier of the Macro Assembler. Use the **/CODEVIEW** option when linking.
4. Debug in assembly mode (this is the start-up default if the debugger doesn't find line-number information). You cannot use source mode for debugging, but you can load the source file into the display window and view it in source mode. You may find this convenient for referring to macros and comments. Any labels or variables that you declared public in the source file can be displayed and referenced by name instead of by address.

You can also use this procedure to debug assembly-language modules called by your program.

The CodeView debugger is more limited when debugging assembly-language programs than when debugging high-level-language programs. The following limitations apply:

- When writing stand-alone assembly-language programs that will be debugged with the CodeView debugger, the class type for the code segment must be CODE. For example, declare segments as follows:

```
sseg            SEGMENT para stack 'STACK'
sseg            ENDS

dseg            SEGMENT word public 'DATA'
```

```
dseg            ENDS

cseg            SEGMENT byte public 'CODE'
cseg            ENDS
```

- In programs assembled with Macro Assembler Versions 1.0 through 4.0, you cannot use variables in CodeView expressions because information on the type size of each variable is not written to the object file. You can still use constants. This means that the Display Expression command, the Watchpoint command, and the expression version of the Watch and Tracepoint commands do not work for such programs. Use the Dump command, and the memory versions of the Watch and Tracepoint commands instead.

  For example, assume your program has the following declaration:

  ```
  var_a  DW  30d
  ```

  You can use the following commands with this variable:

  ```
  DI var_a L 1 ;* Dump var_a as an integer
  WI var_a L 1 ;* Watch var_a as an integer
  TPI var_a    ;* Break when var_a changes (1 is default)
  ```

- Assembly-language symbol names are normally converted to uppercase when assembled. If symbols in your source file are lowercase, you may want to turn off Case Sense from the Options menu.

Chapter 2
The CodeView Display

2.1 Using Window Mode 48
2.1.1 Executing Window Commands with the Keyboard 50
2.1.1.1 Moving the Cursor with Keyboard Commands 50
2.1.1.2 Changing the Screen with Keyboard Commands 52
2.1.1.3 Controlling Program Execution with Keyboard Commands 53
2.1.1.4 Selecting from Menus with the Keyboard 54
2.1.2 Executing Window Commands with the Mouse 56
2.1.2.1 Changing the Screen with the Mouse 56
2.1.2.2 Controlling Program Execution with the Mouse 57
2.1.2.3 Selecting from Menus with the Mouse 60
2.1.3 Using Menu Selections 61
2.1.3.1 Using the File Menu 61
2.1.3.2 Using the Search Menu 63
2.1.3.3 Using the View Menu 66
2.1.3.4 Using the Run Menu 67
2.1.3.5 Using the Watch Menu 68
2.1.3.6 Using the Options Menu 70
2.1.3.7 Using the Language Menu 73
2.1.3.8 Using the Calls Menu 74
2.1.4 Using the Help System 75

The CodeView screen display can appear in two different modes: window and sequential. Either mode provides a useful debugging environment, but the window mode is the more powerful and convenient of the two.

Most users will prefer to use window mode, if they have the hardware to support it. In window mode, pop-up menus, function keys, and mouse support offer fast access to the most common commands. Different aspects of the program and debugging environment can be seen in different windows simultaneously. Window mode is described in Section 2.1.

Sequential mode should be familiar to most programmers. It is similar to the display mode of the CodeView debugger's predecessors, the Microsoft Symbolic Debug Utility (**SYMDEB**) and the DOS **DEBUG** utility. This mode is required if you do not have an IBM-compatible computer, and it is sometimes useful when redirecting command input or output. Sequential mode is described in Section 2.2.

2.1 Using Window Mode

Figure 2.1 shows the CodeView window-mode display with all windows open.

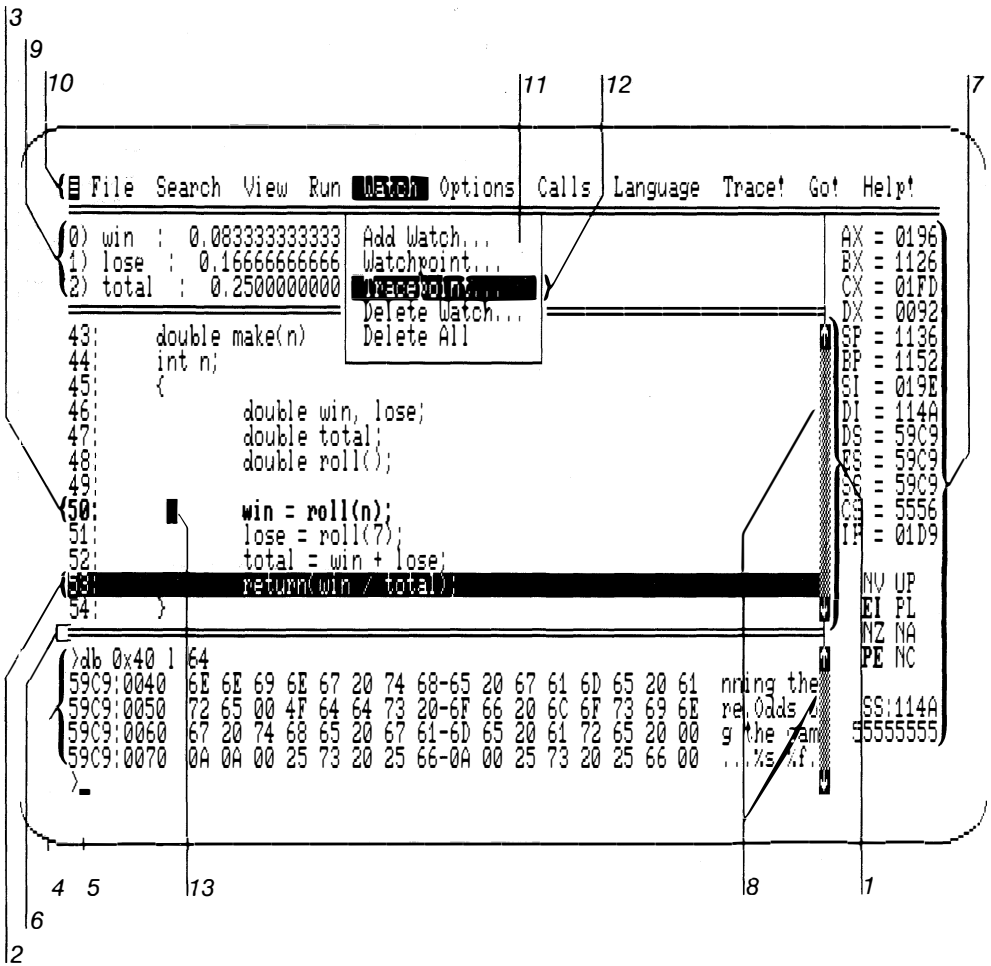


Figure 2.1 Elements of the CodeView Debugging Screen

The elements of the CodeView display marked in Figure 2.1 are explained below:

1. The display window shows the program being debugged. It can contain source code (as in the example), assembly-language instructions, or any specified text file.
2. The current location line (the next line the program will execute) is displayed in reverse video or in a different color. This line may not always be visible, since you can scroll to earlier or later parts of the program.
3. Lines containing previously set breakpoints are shown in high-intensity text.
4. The dialog window is where you enter dialog commands. These are the commands with optional arguments that you can enter at the CodeView prompt (>). You can scroll up or down in this window to view previous dialog commands and command output.
5. The cursor is a thin, blinking line that shows the location at which you can enter commands from the keyboard. You can move the cursor up and down, and put it in either the dialog or display window.
6. The display/dialog separator line divides the dialog window from the display window. You can move this line up or down to change the relative size of the two windows.
7. The register window shows the current status of the registers and flags. This is an optional window that can be opened or closed with one keystroke. If the 386 option is on, a much wider register window will be displayed, with 32-bit registers.
8. The scroll bars are the vertical bars on the right side of the screen. Each scroll bar has an up arrow and a down arrow that you can use to scroll through the display with a mouse.
9. The watch window is an optional window that shows the current status of specified variables or expressions. It appears automatically whenever you create watch statements. See Chapter 8, "Managing Watch Statements."
10. The menu bar shows titles of menus and commands that you can activate with the keyboard or the mouse. Titles followed by an exclamation point represent commands; other titles are menus.
11. Menus can be opened by specifying the appropriate title on the menu bar. On the sample screen, the Watch menu has been opened.
12. The menu "highlight" is a reverse-video or colored strip indicating the current selection in a menu. You can move the highlight up or down to change the current selection.

13. The mouse pointer indicates the current position of the mouse. It is shown only if you have a mouse installed on your system.
14. Dialog boxes (not shown) appear in the center of the screen when you choose a menu selection that requires a response. The box prompts you for a response and disappears when you enter your answer.
15. Message boxes (not shown) appear in the center of the screen to display errors or other messages.

The screen elements are described in more detail in the rest of this chapter.

2.1.1 Executing Window Commands with the Keyboard

The CodeView debugger accepts two kinds of commands: window commands and dialog commands. Dialog commands are entered as command lines following the CodeView prompt (>) in sequential mode. They are discussed in Chapter 3, "Using Dialog Commands."

The most common CodeView debugging commands and all the commands for managing the CodeView display are available with window commands. Window commands are one-keystroke commands that can be entered with function keys, CONTROL-key combinations, ALT-key combinations, or the direction keys on the numeric keypad.

Most window commands can also be entered with a mouse, as described in Section 2.1.2. The window commands available from the keyboard are described by category in Sections 2.1.1.1–2.1.1.4. For a table of commands by name, see ."

2.1.1.1 Moving the Cursor with Keyboard Commands

The following keys move the cursor or scroll text up or down in the display or dialog window:

Key	Function
F6	Moves the cursor between the display and dialog windows. If the cursor is in the dialog window when you press

F6, it will move to its previous position in the display window. If the cursor is in the display window, it will move to its previous position in the dialog window.

CONTROL-U — Moves the display/dialog separator line up one line.

This decreases the size of the display window and increases the size of the dialog window. If the cursor is in the dialog window, you can remove the display window entirely by moving the separator line to the top of the window.

CONTROL-D — Moves the display/dialog separator line down one line.

This increases the size of the display window and decreases the size of the dialog window. If the cursor is in the display window, you can remove the dialog window entirely by moving the separator line to the bottom of the screen.

UP ARROW — Moves the cursor up one line in either the display or dialog window.

DOWN ARROW — Moves the cursor down one line in either the display or dialog window.

PGUP — Scrolls up one page.

If the cursor is in the display window, the source lines or assembly-language instructions scroll up. If the cursor is in the dialog window, the buffer of commands entered during the session scrolls up. The cursor remains at its current position in the window. The length of a page is the current number of lines in the window.

PGDN — Scrolls down one page.

If the cursor is in the display window, the source lines or assembly-language instructions scroll down. If the cursor is in the dialog window, the buffer of commands entered during the session scrolls down. The cursor remains at its current position in the window. The length of a page is the current number of lines in the window.

HOME	Scrolls to the top of the file or command buffer. If the cursor is in the display window, the text scrolls to the start of the source file or program instructions. If the cursor is in the dialog window, the commands scroll to the top of the command buffer. The top of the command buffer may be blank if you have not yet entered enough commands to fill the buffer. The cursor remains at its current position in the window.
END	Scrolls to the bottom of the file or command buffer. If the cursor is in the display window, the text scrolls to the end of the source file or program instructions. If the cursor is in the dialog window, the commands scroll to the bottom of the command buffer and the cursor moves to the CodeView prompt (>) at the end of the buffer.

2.1.1.2 Changing the Screen with Keyboard Commands

The following keys change the screen or switch to a different screen:

Key	Function
F1	Displays initial on-line-help screen. The help system is discussed in Section 2.1.4. You can also get to the initial help screen by selecting Help from the View menu, as described in Section 2.1.3.3.
F2	Toggles the register window. The window disappears if present, or appears if absent. You can also toggle the register window with the Register selection from the Options menu, as described in Section 2.1.3.6.
F3	Switches between source, mixed, and assembly modes. Source mode shows source code in the display window, while assembly mode shows assembly-language instructions. Mixed mode shows both. You can also

change modes with the Source, Mixed, and Assembly selections from the View menu, as described in Section 2.1.3.3.

F4 — Switches to the output screen.

The output screen shows the output, if any, from your program. Press any key to return to the CodeView screen.

2.1.1.3 Controlling Program Execution with Keyboard Commands

The following keys set and clear breakpoints, trace through your program, or execute to a breakpoint:

Key	Function
F5	Executes to the next breakpoint, or to the end of the program if no breakpoint is encountered. This keyboard command corresponds to the Go dialog command when it is given without a destination breakpoint argument.
F7	Sets a temporary breakpoint on the line with the cursor, and executes to that line (or to a previously set breakpoint or the end of the program, if either is encountered before the temporary breakpoint). In source mode, if the line does not correspond to code (for example, data declaration or comment lines), the CodeView debugger sounds a warning and ignores the command. This window command corresponds to the Go dialog command when it is given with a destination breakpoint.
F8	Executes a Trace command. The CodeView debugger executes the next source line in source mode, or the next instruction in assembly mode. If the source line or instruction contains a call to a routine or interrupt, the debugger starts tracing through the call (enters the call and is ready to execute the first source line or instruction). This command will not trace into DOS function calls.

F9	Sets or clears a breakpoint on the line with the cursor. If the line does not currently have a breakpoint, one is set on that line. If the line already has a breakpoint, the breakpoint is cleared. If the cursor is in the dialog window, the CodeView debugger sounds a warning and ignores the command. This window command corresponds to the Breakpoint Set and Breakpoint Clear dialog commands.
F10	Executes the Program Step command. The CodeView debugger executes the next source line in source mode, or the next instruction in assembly mode. If the source line or instruction contains a call to a routine or interrupt, the debugger steps over the entire call (executes it to the return) and is ready to execute the line or instruction after the call.

Important

You can usually interrupt program execution by pressing CONTROL-BREAK or CONTROL-C. This can be used to exit endless loops, or it can interrupt loops that are slowed by the Watchpoint or Tracepoint commands (see Chapter 8, "Managing Watch Statements"). CONTROL-BREAK or CONTROL-C may not work if your program has a special use for one or both of these key combinations. If you have an IBM Personal Computer AT (or a compatible computer), you can use the SYSTEM-REQUEST key to interrupt execution regardless of your program's use of CONTROL-BREAK and CONTROL-C.

2.1.1.4 Selecting from Menus with the Keyboard

The CodeView debugger has seven pop-up menus. This section discusses how to make selections from menus. The effects of the selections are discussed in Section 2.1.3.

The menu bar at the top of the screen has nine titles: File, Search, View, Run, Watch, Options, Calls, Trace!, and Go!. The first seven titles are menus, and the last two are commands. The Trace! and Go! titles are provided primarily for mouse users, though you can activate them by pressing

ALT-T or ALT-G and then pressing the ENTER key. The exclamation point is a convention used to indicate that a title represents a command rather than a menu. The same commands are more easily accessible with the F5, F8, and F10 keys.

The steps for opening a menu and making a selection are described below:

1. To open a menu, press the ALT key and the first letter of the menu title. For example, press ALT-S to open the Search menu. The menu title is highlighted, and a menu box listing the selections pops up below the title.

2. There are two ways to make a selection from an open menu:

   a. Press the DOWN ARROW key on the numeric keypad to move down the menu. The highlight will follow your movement. When the item you want is highlighted, press the ENTER key to execute the command. For example, press the DOWN ARROW once to select Find from the Search menu.

      You can also press the UP ARROW key to move up the menu. If you move off the top or bottom of the menu, the highlight wraps around to the other end of the menu.

   b. While holding down the ALT key, press the first letter of the item you want to select. You do not have to press the ENTER key with this method. For example, press ALT-F to select Find from the Search menu. This selection method does not work with the Calls menu.

3. One of three things will happen at this point:

   a. For most menu selections, the choice is executed immediately.

   b. The items on the Options menu have small double arrows next to them if the option is on, or no arrows if the option is off. Choosing the item toggles the option. The status of the arrows will be reversed the next time you open the menu.

   c. Some items require a response. In this case, there is another step in the menu-selection process.

4. If the item you select requires a response, a dialog box opens when you select a menu item. Type your response to the prompt in the box and press the ENTER key. For example, the Find dialog box asks you to enter a regular expression (see Appendix B for a complete explanation of regular expressions).

If your response is valid, the command will be executed. If you enter an invalid response, a message box will appear, telling you the problem and asking you to press a key. Press any key to make the message box disappear.

At any point during the process of selecting a menu item, you can press the ESCAPE key to cancel the menu. While a menu is open, you can press the LEFT ARROW or RIGHT ARROW key to move from one menu to an adjacent menu, or to one of the command titles on the menu bar.

2.1.2 Executing Window Commands with the Mouse

The CodeView debugger is designed to work with the Microsoft Mouse (it also works with some compatible pointing devices). By moving the mouse on a flat surface, you can move the mouse pointer in a corresponding direction on the screen. The following terms refer to the way you select items or execute commands with the mouse:

Term	Definition
Point	To move the mouse until the mouse pointer rests on the item you want to select.
Click	To quickly press and release a mouse button while pointing at an item you want to select.
Drag	To press a mouse button while on a selected item, then hold the button down while moving the mouse. The item moves in the direction of the mouse movement. When the item you are moving is where you want it, release the button; the item will stay at that point.

The CodeView debugger uses two mouse buttons. The terms "click right," "click left," "click both," and "click either" are sometimes used to designate which buttons to use. When dragging, either button can be used.

2.1.2.1 Changing the Screen with the Mouse

You can change various aspects of the screen display by pointing to one of the following elements and then either clicking or dragging:

Item	Action
Double line	

separating display and dialog windows	Drag the separator line up to increase the size of the dialog window while decreasing the size of the display window, or drag the line down to increase the size of the display window while decreasing the size of the dialog window. You can eliminate either window completely by dragging the line all the way up or down (providing the cursor is not in the window you want to eliminate).
UP ARROW or DOWN ARROW on the scroll bar	Point and click on one of the four arrows on the scroll bars to scroll up or down. If you are in the display window, source code will scroll up or down. If you are in the dialog window, the buffer containing dialog commands entered during the session will scroll up or down. The distance moved is determined by which buttons you click, as follows:

Button	Action
Click left	Scroll up or down, one line at a time
Click right	Scroll up or down, one page at a time; the length of a page is the current size of the window
Click both	Scroll to the top or bottom of the file or command buffer

Some menu selections also change the screen display. See Section 2.1.3 for a description of the menu selections.

2.1.2.2 Controlling Program Execution with the Mouse

By clicking the following mouse items, you can set and clear breakpoints, trace through your program, execute to a breakpoint, or change the flag bits:

Item	Action
Source line or instruction	

Point and click on a source line in source mode or on an instruction in assembly mode to take one of the following actions:

Button	Action
Click left	If the line under the mouse cursor does not have a breakpoint, one is set there. If the line already has a breakpoint, the breakpoint is removed. Lines with breakpoints are shown in high-intensity text.
Click right	A temporary breakpoint is set on the line and the CodeView debugger executes until it reaches the line (or until it reaches a previously set breakpoint or the end of the program if either is encountered before the temporary breakpoint).

If you click on a line that does not correspond to code (for example, a declaration or comment), the CodeView debugger will sound a warning and ignore the command.

Trace! on menu bar

Point and click to trace the next instruction. The kind of trace is determined by the button clicked:

Button	Action
Click left	The Trace command is executed. The CodeView debugger executes the next source line in source mode, or the next instruction in assembly mode. If the source line or instruction contains a call to a routine or interrupt, the debugger starts tracing through the call (it enters the call and is ready to execute the first source line or instruction). This command will not trace into DOS

	function calls.
	Click right — The Program Step command is executed. The debugger executes the next source line in source mode, or the next instruction in assembly mode. If the source line or instruction contains a call to a routine or interrupt, the CodeView debugger steps over the entire call (it executes the call to the return) and is ready to execute the line or instruction after the call.
	These two commands are different only if the current location is the start of a procedure, interrupt, or call.
Go! on menu bar	Point and click either button to execute to the next breakpoint, or to the end of the program if no breakpoints are encountered.
Flag in register window	Point to a flag name and click either button to reverse the flag. If the flag bit is set, it will be cleared; if the flag bit is cleared, it will be set. The flag name is changed on the screen to match the new status. If you are using color mode, the color of the flag mnemonic will also change. This command can only be used when the register window is open. Use the command with caution, since changing flag bits can change program execution at the lowest level.

Important

You can usually interrupt program execution by pressing CONTROL-BREAK or CONTROL-C. See the note in Section 2.1.1.3 for more information.

2.1.2.3 Selecting from Menus with the Mouse

The CodeView debugger has seven pop-up menus. This section discusses how to make selections from these menus. The effect of each selection is discussed in Section 2.1.3.

The menu bar at the top of the screen has nine titles: File, Search, View, Run, Watch, Options, Calls, Trace!, and Go!. The first seven titles are menus and the last two are commands that you can execute by clicking with the mouse. The steps for opening a menu and making a selection are described below:

1. To open a menu, point to the title of the menu you want to select. For example, move the pointer onto File on the menu bar if you want to open the File menu.

2. With the mouse pointer on the title, press and hold down either mouse button. The selected title is highlighted and a menu box with a list of selections pops up below the title. For example, if you point to Search and press a button, the Search menu pops up.

3. With the button held down, move the mouse down. The highlight follows the mouse movement. You can move the highlight up or down in the menu box. For example, to select Find from the Search menu, move the highlight down the menu to Find.

   If you move off the box, the highlight will disappear. However, as long as you do not release the button, you can move the pointer back onto the menu to make the highlight reappear.

4. When the selection you want is highlighted, release the mouse button. For example, release the button with the highlight on Find.

   When you release the button, the menu selection is executed. One of three things will happen:

   a. For most menu selections, the choice is executed immediately.

   b. The items on the Options menu have small double arrows next to them if the option is on, or no arrows if the option is off. Choosing the item toggles the option. The status of the arrows will appear reversed the next time you open the menu.

   c. Some items require a response. In this case, there is another step in the menu-selection process.

5. If the item you select requires a response, a dialog box with a prompt appears. Type your response and press the ENTER key or a mouse button. For example, if you selected Find, the prompt will ask you to enter a regular expression (see Section 2.1.3.2, "Using the Search Menu," or Appendix A, "Regular Expressions," for an explanation of regular expressions).

   If your response is valid, the command will be executed. If you enter an invalid response in the dialog box, a message box will appear telling you the problem and asking you to press a key. Press any key or click a mouse button to make the message box disappear.

There are several shortcuts you can take when selecting menu items with the mouse. If you change your mind and decide not to select an item from a menu, just move off the menu and release the mouse button; the menu will disappear. You can move from one menu to another by dragging the pointer directly from any point on the current menu to the title of the new menu.

2.1.3 Using Menu Selections

This section describes the selections on each of the CodeView menus. These selections can be made with the keyboard, as described in Section 2.1.1, or with the mouse, as described in Section 2.1.2.

Note that while the Trace! and Go! commands appear on the menu bar, they are not menus. These titles are provided primarily for mouse users, though you can activate them by pressing ALT-T or ALT-G and then pressing the ENTER key, or by using the F5, F8, and F10 keys.

2.1.3.1 Using the File Menu

The File menu includes selections for working on the current source or program file. The File menu is shown in Figure 2.2, and the selections are explained below:

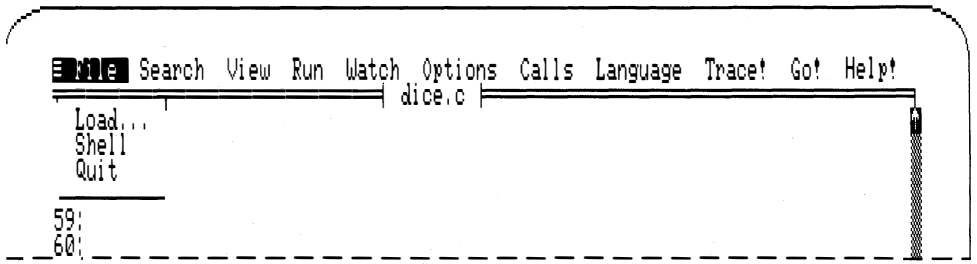


Figure 2.2 The File Menu

Selection	Action
Load...	Opens a new file. When you make this selection, a dialog box appears asking for the name of the new file you want to open. Type the name of a source file, an include file, or any other text file. The text of the new file replaces the current contents of the display window (if you are in assembly mode, the CodeView debugger will switch to source mode). When you finish viewing the file, you can reopen the original source file. The current location and breakpoints will still be marked when you return to the source file. You do not need to open a new file to see source files for a different module of your program. The CodeView debugger automatically switches to the source file of the other module when program execution enters the other module. While switching source files is never necessary, it may be desirable if you want to set breakpoints or execute to a line in another module.

Note

If the debugger cannot find the source file when it switches modules, a dialog box appears asking for a file specification for the source file. You can

either enter a new file specification if the file is in another directory, or press the ENTER key if no source file exists. If you press the ENTER key, the module can only be debugged in assembly mode.

Shell

Exits to a DOS shell. This brings up the DOS screen, where you can execute DOS commands or executable files. To return to the CodeView debugger, type `exit` at the DOS command prompt. The CodeView screen reappears with the same status it had when you left it.

The Shell command works by saving the current processes in memory and then executing a second copy of **COMMAND.COM**. This requires a significant amount of free memory (more than 200K), since the debugger, **COMMAND.COM**, symbol tables, and the debugged program must all be saved in memory. If you do not have enough memory to execute the Shell command, an error message appears. Even if you have enough memory to execute the shell, you may not have enough memory left to execute large programs from the shell.

The Shell command will not work unless you have executed your program's start-up code. You can do this after starting the debugger, or after restarting your program, by executing to any point within the program. For example, enter the dialog command `G main`, or simply press the F10 (Program Step) key.

Quit

Terminates the CodeView debugger and returns to DOS.

2.1.3.2 Using the Search Menu

The Search menu includes selections for searching through text files for text strings and for searching executable code for labels. The Search menu is shown in Figure 2.3 and the selections are explained below:

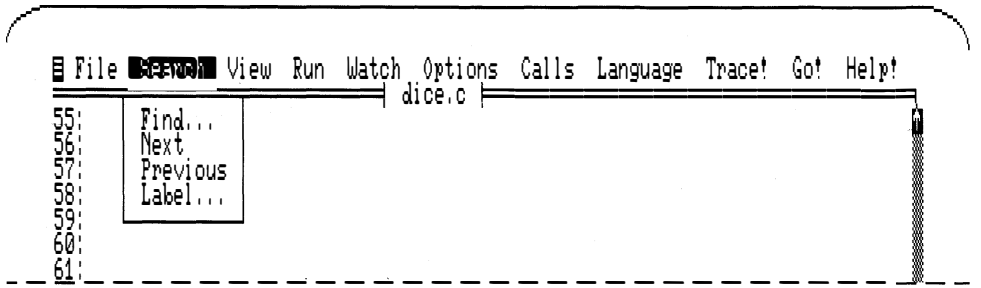


Figure 2.3 The Search Menu

Selection	Action
Find...	Searches the current source file or other text file for a specified regular expression. When you make this selection, a dialog box opens, asking you to enter a regular expression. Type the expression you want to search for and press the ENTER key. The CodeView debugger starts at the current or most recent cursor position in the display window and searches for the expression. If your entry is found, the cursor moves to the first source line containing the expression. If the entry is not found, a message box opens, telling you the problem and asking you to press a key (you can also click a mouse button) to continue. If you are in assembly mode, the debugger automatically switches to source mode when the expression is found. Regular expressions are a method of specifying variable text strings. This method is similar to the DOS method of using wild cards in file names. Regular expressions are explained in detail in Appendix B. You can use the Search selections without understanding regular expressions. Since text strings are the simplest form of regular expressions, you can simply enter a string of characters as the expression you want to find. For example, you could enter `count` if you wanted to search for the word "count."

The following characters have a special meaning in regular expressions: backslash (\), asterisk (*), left bracket ([), period (.), dollar sign ($), and caret (^). In order to find strings containing these characters, you must precede the characters with a backslash; this cancels their special meanings.

For example, the periods in FORTRAN relational and logical operators must be preceded by backslashes. You would use `\.EQ` to find the **.EQ.** operator. With C, you would use `\*ptr` to find `*ptr`; and with BASIC, you would use `NAME\$` to find `NAME$`.

The Case Sense selection from the Options menu has no effect on searching for regular expressions.

Next

Searches for the next match of the current regular expression.

This selection is meaningful only after you have used the Search command to specify the current regular expression. If the CodeView debugger searches to the end of the file without finding another match for the expression, it wraps around and starts searching at the beginning of the file.

Previous

Searches for the previous match of the current regular expression.

This selection is meaningful only after you have used the Search command to specify the current regular expression. If the debugger searches to the beginning of the file without finding another match for the expression, it wraps around and starts searching at the end of the file.

Label...

Searches the executable code for an assembly-language label.

If the label is found, the cursor moves to the instruction containing the label. If you start the search in source mode, the debugger will switch to assembly mode to show a label in a library routine or an assembly-language module.

2.1.3.3 Using the View Menu

The View menu includes selections for switching between source and assembly modes, and for switching among the debugging screen, the output screen, and the help screen. The corresponding function keys for menu selection are shown on the right side of the menu when appropriate. The View menu is shown in Figure 2.4, and the selections are explained below.

Note

The terms "source mode" and "assembly mode" apply to MASM programs as well as to high-level language programs. Source mode with MASM shows the source code as originally written, including comments and directives. Assembly mode displays unassembled machine code, without symbolic information.

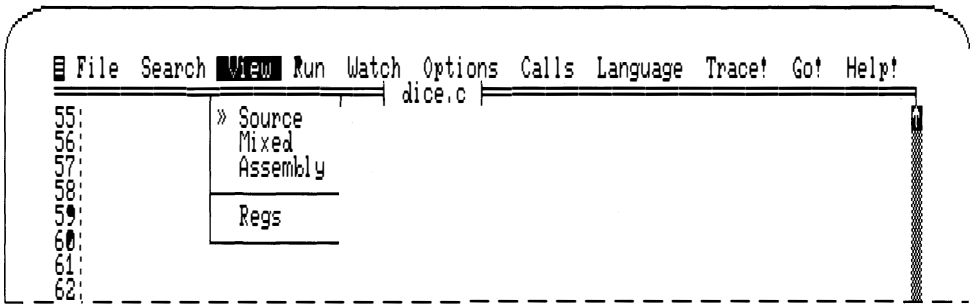


Figure 2.4 The View Menu

Selection	Action
Help	Opens the initial help menu. Section 2.1.4 tells how to move through the on-line-help system and return to the debugging screen.
Source	Changes to source mode (showing source lines only). If you select this mode when you are already in source mode, the selection will be ignored.

Mixed	Changes to mixed mode (showing both unassembled machine code and source lines). If you select this mode when you are already in mixed mode, the selection will be ignored.
Assembly	Changes to assembly mode (showing only unassembled machine code). If you select this mode when you are already in assembly mode, your selection will be ignored.
Registers	Selecting this option will toggle the register window on and off. You can also turn the register on and off by pressing the F2 key.

2.1.3.4 Using the Run Menu

The Run menu includes selections for running your program. The Run menu is shown in Figure 2.5, and the selections are explained below:

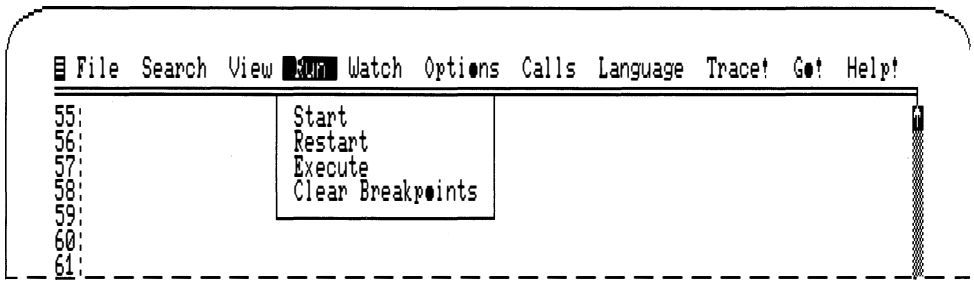


Figure 2.5 The Run Menu

Selection	Action
Start	Starts the program from the beginning and runs it. Any previously set breakpoints or watch statements will still be in effect. The CodeView debugger will run your program from the beginning to the first breakpoint, or to the end of the program if no breakpoint

is encountered. This has the same effect as selecting Restart (see the next selection) and then entering the Go command.

Restart — Restarts the current program, but does not begin executing it.

You can debug the program again from the beginning. Any previously set breakpoints or watch statements will still be in effect.

Execute — Executes in slow motion from the current instruction.

This is the same as the Execute dialog command (**E**). To stop animated execution, press any key or a mouse button.

Clear Breakpoints — Clears all breakpoints.

This selection may be convenient after selecting Restart if you don't want to use previously set breakpoints. Note that watch statements are not cleared by this command.

2.1.3.5 Using the Watch Menu

The Watch menu includes selections for managing the watch window. Selections on this menu are also available with dialog commands. The Watch menu is shown in Figure 2.6, and the selections are explained below:

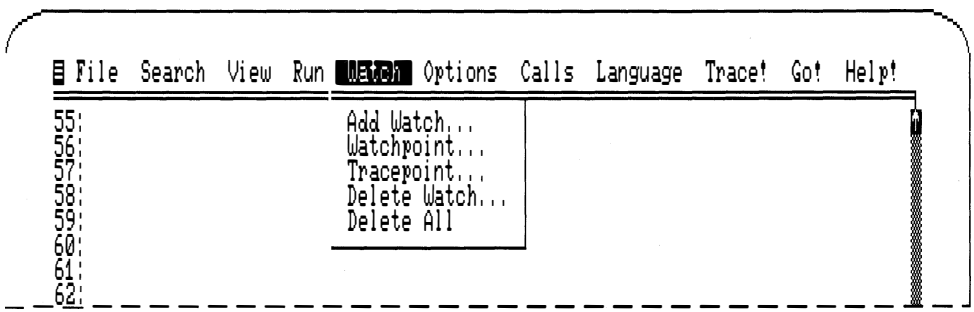


Figure 2.6 The Watch Menu

Selection	Action
Add Watch...	Adds a watch-expression statement to the watch window. A dialog window opens, asking for the source-level expression (which may be simply a variable) whose value you want to see displayed in the watch window. Type the expression and press the ENTER key or press a mouse button. The statement appears in the watch window in normal text. You cannot specify a memory range to be displayed with the Add Watch... selection as you can with the Watch dialog command. You can specify the format in which the value will be displayed. Type the expression, followed by a comma and a CodeView format specifier. If you do not give a format specifier, the CodeView debugger displays the value in a default format. See Chapter 6, "Examining Data and Expressions," for more information about format specifiers and the default format. See Section 8.1, "Setting Watch-Expression and Watch-Memory Statements," for more information about the Watch command.
Watchpoint...	Adds a watchpoint statement to the watch window. A dialog window opens, asking for the source-level expression whose value you want to test. The watchpoint statement appears in the watch window in high-intensity text when you enter the expression. A watchpoint is a conditional breakpoint that causes execution to stop when the expression becomes nonzero (true). See Section 8.2, "Setting Watchpoints," for more information.
Tracepoint...	Adds a tracepoint statement to the watch window. A dialog window opens, asking for the source-level expression or memory range whose value you want to test. The tracepoint statement appears in the watch window in high-intensity text when you enter the expression. A tracepoint is a conditional breakpoint that causes execution to stop when the value of a

given expression changes. You cannot specify a memory range to be tested with the Tracepoint... selection as you can with the Tracepoint dialog command.

When setting a tracepoint expression, you can specify the format in which the value will be displayed. Type the expression followed by a comma and a format specifier. If you do not give a format specifier, the CodeView debugger displays the value in a default format. See Chapter 6, "Examining Data and Expressions," for more information about format specifiers and the default format. See Section 8.3, "Setting Tracepoints," for more information about tracepoints.

Delete Watch... Deletes a watch statement from the watch window.

A dialog window opens, showing the current watch statements. If you are using a mouse, move the pointer to the statement you want to delete and click either button. If you are using the keyboard, press the UP ARROW or DOWN ARROW key to move the highlight to the statement you want to delete, then press the ENTER key.

Delete All Deletes all statements in the watch window.

All watch, watchpoint, and tracepoint statements are deleted, the watch window disappears, and the display window is redrawn to take advantage of the freed space on screen.

2.1.3.6 Using the Options Menu

The Options menu allows you to set options that affect various aspects of the behavior of the CodeView debugger. The Options menu is shown in Figure 2.7, and the selections are explained below:

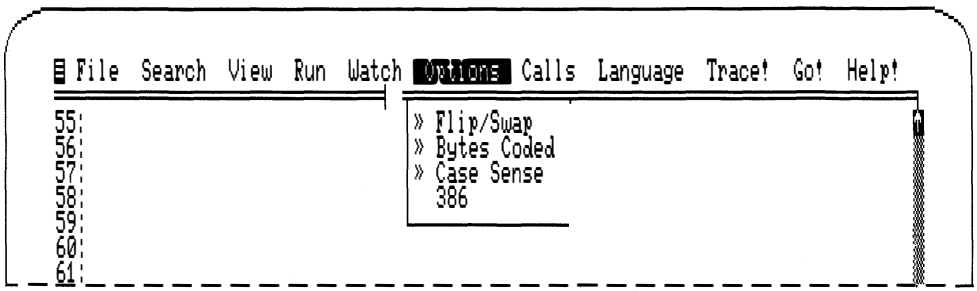


Figure 2.7 The Options Menu

Selections on the Options menu have small double arrows to the left of the selection name when the option is on. The status of the option (and the presence of the double arrows) is reversed each time you select the option. By default, the Flip/Swap and Bytes Coded options are on, and the 386 option is off, when you start the CodeView debugger. Depending on which language your main program is in, the debugger will automatically turn Case Sense on (if your program is in C) or off (if your program is in another language), when you start debugging.

The selections from the Options menu are discussed below:

Selection	Action
Flip/Swap	When on (the default), screen swapping or screen flipping (whichever the debugger was started with) is active; when off, swapping or flipping is disabled. Turning off swapping or flipping makes the screen scroll more smoothly. You will not see the program flip or swap each time you execute part of the program. This option has no effect if neither swapping nor flipping was selected during start-up.

Warning

Make sure that flipping or swapping is on any time your program writes to the screen. If swapping and flipping are off, your program will write the output at the location of the cursor. The

CodeView debugger will detect that the screen has changed and will redraw the screen, thus destroying the program output. The error message `Flip/Swap option off — application output lost` is also displayed.

Bytes Coded

When on (the default), both the instructions and the bytes for each instruction are shown; when off, only the instructions are shown.

This option affects only assembly mode. The following display shows the appearance of sample code when the option is off:

```
27:                name = gets(namebuf);
32AF:003E LEA          AX,Word Ptr [namebuf]
32AF:0041 PUSH         AX
32AF:0042 CALL         _gets (03E1)
32AF:0045 ADD          SP,02
32AF:0048 MOV          Word Ptr [name],AX
```

The following display shows the appearance of the same code when the option is on:

```
27:                name = gets(namebuf);
32AF:003E 8D46DE           LEA         AX,Word Ptr [namebuf]
32AF:0041 50               PUSH        AX
32AF:0042 E89C03           CALL        _gets (03E1)
32AF:0045 83C402           ADD         SP,02
32AF:0048 8946DA           MOV         Word Ptr [name],AX
```

Case Sense

When on, the CodeView debugger assumes that symbol names are case sensitive (each lowercase letter is different from the corresponding uppercase letter); when off, symbol names are not case sensitive.

This option is on by default for C programs, and off by default for FORTRAN, BASIC, and assembly programs. You will probably want to leave the option in its default setting.

386

When on, the CodeView debugger assembles and disassembles instructions in the 386 format, and the register window will display the wider 386-format registers.

To use this option, you should have a 386 processor, running in 386 mode.

2.1.3.7 Using the Language Menu

The Language menu allows you to either select the expression evaluator, or instruct the CodeView debugger to select if for you automatically. The Language menu is shown in Figure ??.?, and the selections are explained below:

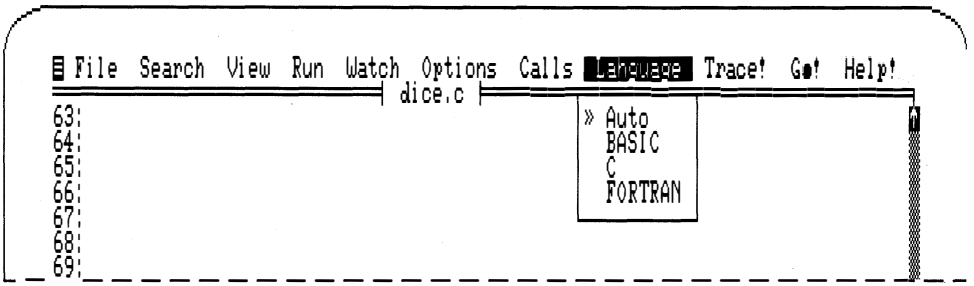


Figure 2.8 The Language Menu

As with the Options menu, any selection that is on is marked by double arrows. Unlike the Options menu, however, exactly one item (and no more) on the Language menu is selected at any given time.

Selection	Action
Auto	When selected, the debugger selects the expression evaluator for you. The Auto selection informs the debugger to change the expression evaluator, if needed, after every loading of a new source file. You will be able to tell what expression evaluator is in use by simply looking at the source file extension. The Auto selection will use the C expression evaluator if the current source file has neither a **.FOR** or **.BAS** extension. You can also tell which expression evaluator is in use at any time by executing the USE dialog command.
BASIC	When selected, the debugger uses the BASIC expression evaluator, regardless of what kind of program is being debugged.

C	When selected, the debugger uses the C expression evaluator, regardless of what kind of program is being debugged.
FORTRAN	When selected, the debugger uses the FORTRAN expression evaluator, regardless of what kind of program is being debugged.

2.1.3.8 Using the Calls Menu

The Calls menu is different from other menus in that its contents and size change, depending on the status of your program. The Calls menu is shown in Figure 2.8.

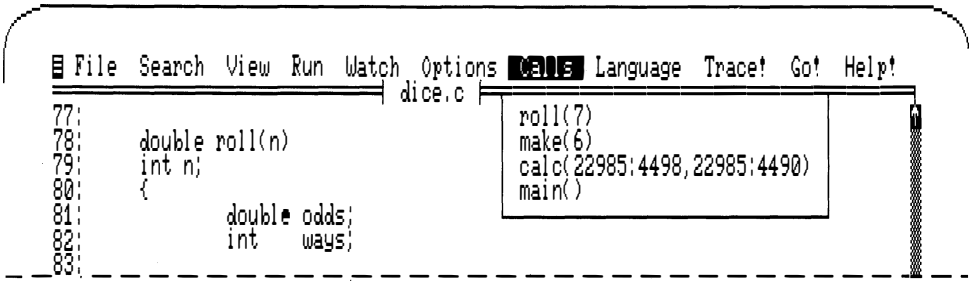


Figure 2.9 The Calls Menu

Like other menus, the Calls menu can be opened by pressing the ALT key and the first letter of the menu title (ALT-C). However, you cannot make selections from the Calls menu by pressing the ALT key with the first letter of your selection. You must use the UP ARROW or DOWN ARROW key to move to your selection, then press the ENTER key. You can also use the mouse to open and select from the Calls menu. Usually the menu is used to view the current functions rather than to actually make selections.

The Calls menu shows the current routine and the trail of routines from which it was called. The current routine is always at the top. The routine from which the current routine was called is directly below. Other active routines are shown in the reverse order in which they were called. With C and FORTRAN programs, the bottom routine should always be **main**. (The only time when **main** will not be the bottom routine is when you are

tracing through the standard library's start-up or termination routines.)

The current value of each argument, if any, is shown in parentheses following the routine. The menu expands to accommodate the arguments of the widest routine. Arguments are shown in the current radix (the default is decimal). If there are more active routines than will fit on the screen, or if the routine arguments are too wide, the display will be truncated. The Stack Trace dialog command (**K**) shows all the routines and arguments regardless of the size of the display.

If you want to view code at the point where one of the routines was called, select the routine below the one you want to view. The cursor will move to the calling source line (in source mode) or to the calling instruction (in assembly mode). In other words, the cursor will indicate the calling location in the selected routine where the next-level routine was called. If you select the current (top-level) routine, the cursor moves to the current location in that routine.

For example, if the Calls menu shown in Figure 2.8 appears and you select `SAREA`, the cursor will move to the line in `main` where the routine `SAREA` is called.

Note

If you are using the CodeView debugger to debug assembly-language programs, routines will be shown in the Calls menu only if they use the Microsoft calling convention. This calling convention is explained in the user's guides for Microsoft C, FORTRAN, and BASIC.

2.1.4 Using the Help System

The CodeView on-line-help system uses tree-structured menus to give you quick access to help screens on a variety of subjects. The system is organized by subject so that you can reach a help screen on any topic with a minimum of keystrokes.

The help file is called **CV.HLP**. It must be present in the current directory or in one of the directories specified with the DOS **PATH** command. If the help file is not found, the CodeView debugger will still operate, but you will not be able to use the help system. An error message will appear if you try to use a help command.

When you request help, either by pressing the F1 key or by selecting Help from the View menu, the top-level help menu appears. You select one of the topics listed by pressing the highlighted first letter of the menu title. You can also select a topic by pointing to it with the mouse and clicking either button.

When you select a topic, the help screen on that subject may appear immediately, or in some cases, a second menu screen will appear. Keep selecting topics until you reach the screen you want.

In addition to menu titles, you can select four special commands from any help screen. The keys that select these commands are always shown at the top of the screen. You can also point and click with the mouse to select these commands. The commands are listed below:

Command	Action
PGUP	Returns to the previous help screen or menu.
PGDN	Proceeds to the next-level screen. This command is intended for help topics that consist of more than one screen of text. If you press this key on a menu screen, the top leftmost selection is chosen as the default. The CodeView debugger sounds a warning if there is no lower-level screen.
HOME	Returns to the top-level menu. The debugger sounds a warning if you are already at the top level.
END	Returns to the debugging screen.
ESC	Returns to the debugging screen.

When you use a selection letter to select a topic that has more than one screen of information, you can press the selection letter again to get the next screen. This has the same effect as pressing the PGDN key.

2.2 Using Sequential Mode

Sequential mode is required if you have neither an IBM Personal Computer nor a closely compatible computer. In sequential mode, the CodeView debugger works much like its predecessors, the Microsoft Symbolic Debug Utility (**SYMDEB**) and the DOS **DEBUG** utility.

In sequential mode, the CodeView debugger's input and output always move down the screen from the current location. When the screen is full, the old output scrolls off the top of the screen to make room for new output appearing at the bottom. You can never return to examine previous commands once they scroll off, but in many cases, you can reenter the command to put the same information on the screen again.

Most window commands cannot be used in sequential mode. However, the following function keys, which are used as commands in window mode, are also available in sequential mode:

Command	Action
F1	Displays a command-syntax summary. This is equivalent to the Help (**H**) dialog command. It is different from the on-line-help system accessed by the F1 key and the Help dialog command in window mode.
F2	Displays the registers. This is equivalent to the Register (**R**) dialog command.
F3	Toggles between source, mixed, and assembly modes. Pressing this key will rotate the mode between source, mixed, and assembly. You can achieve the same effect by using the Set Assembly (**S-**), Set Mixed (**S&**), and Set Source(**S+**) dialog commands.
F4	Switches to the output screen, which shows the output of your program. Press any key to return to the CodeView debugging screen. This is equivalent to the Screen Exchange (****) dialog command.

F5	Executes from the current instruction until a breakpoint or the end of the program is encountered. This is equivalent to the Go dialog command (**G**) with no argument.
F8	Executes the next source line in source mode, or the next instruction in assembly mode. If the source line or instruction contains a function, procedure, or interrupt call, the CodeView debugger executes the first source line or instruction of the call and is ready to execute the next source line or instruction within the call. This is equivalent to the Trace (**T**) dialog command.
F9	Sets or clears a breakpoint at the current program location. If the current program location has no breakpoint, one is set. If the current location has a breakpoint, it is removed. This is equivalent to the Breakpoint Set (**BP**) dialog command with no argument.
F10	Executes the next source line in source mode, or the next instruction in assembly mode. If the source line or instruction contains a function, procedure, or interrupt call, the call is executed to the end and the CodeView debugger is ready to execute the line or instruction after the call. This is equivalent to the Program Step (**P**) dialog command.

The CodeView Watch (**W**), Watchpoint (**WP**), and Tracepoint (**TP**) commands work in sequential mode, but since there is no watch window, the watch statements are not shown. You must use the Watch List command (**W**) to examine watch statements and watch values. See Chapter 8 for information on Watch Statement commands.

All the CodeView commands that affect program operation (such as Trace, Go, and Breakpoint Set) are available in sequential mode. Any debugging operation that can be done in window mode can also be done in sequential mode.

Chapter 3
Using Dialog Commands

3.1 Entering Commands and Arguments 81
3.1.1 Using Special Keys 81
3.1.2 Using the Command Buffer 82
3.2 Format for CodeView Commands and Arguments 83

CodeView dialog commands can be used in sequential mode or from the dialog window. In sequential mode, they are the primary method of entering commands. In window mode, dialog commands are used to enter commands that require arguments or that do not have corresponding window commands.

Many window commands have duplicate dialog commands. Generally, the window version of a command is more convenient, while the dialog version is more powerful. For example, to set a breakpoint on a source line in window mode, put the cursor on the source line and press F9, or point to the line and click the left mouse button. The dialog version of the command (BP) requires more keystrokes, but it allows you to specify an address, a pass count, and a string of commands to be taken whenever the breakpoint is encountered.

The rest of this chapter explains how to enter dialog commands.

3.1 Entering Commands and Arguments

Dialog commands are entered at the CodeView prompt (>). Type the command and arguments, then press the ENTER key.

In window mode, you can enter commands regardless of whether or not the cursor is at the CodeView prompt. If the cursor is in the display window, the text you type will appear after the prompt in the dialog window, even though the cursor remains in the display window.

3.1.1 Using Special Keys

While entering dialog commands or viewing output from commands, you can use the following special keys:

Key	Action
CONTROL-C	Stops the current output or cancels the current command line. For example, if you are watching a long display from a Dump command, you can press CONTROL-C to interrupt the output and return to the CodeView prompt. If you make a mistake while entering a command, you can press CONTROL-C to cancel the command without executing it. A new prompt appears

and you can reenter the command.

CONTROL-S — Pauses during output of a command. You can press any key to continue output. For example, if you are watching a long display from a Dump command, you can press CONTROL-S when a part of the display that you want to examine more closely appears. Then press any key when you are ready for the output to continue scrolling.

BACKSPACE — Deletes the previous character on the command line and moves the cursor back one space. For example, if you make an error while typing a command, you can use the BACKSPACE key to delete the characters back to the error, then retype the rest of the command.

3.1.2 Using the Command Buffer

In window mode, the CodeView debugger has a command buffer where the last 4K (4096 bytes) of commands and command output are stored. This amounts to approximately three screens of text, depending on the length of your commands and output. The command buffer is not available in sequential mode.

When the cursor is in the dialog window, you can scroll up or down to view the commands you have entered earlier in the session. The commands for moving the cursor and scrolling through the buffer are explained in sections 3.2.1.1 and 3.2.2.1.

Scrolling through the buffer is particularly useful for viewing the output from commands, such as Dump or Examine Symbols, whose output may scroll off the top of the dialog window.

If you have scrolled through the dialog buffer to look at previous commands and output, you can still enter new commands. When you type a command, it will appear to be overwriting the previous line where the cursor is located, but when you press the ENTER key, the new command will be entered at the end of the buffer. For example, if you enter a command while the cursor is at the start of the buffer, and then scroll to the end of the buffer, you will see the command you just entered. If you scroll back to the point where you entered the command, you will see the original characters rather than the characters you typed over the originals.

When you start the debugger, the buffer is empty except for the copyright message. As you enter commands during the session, the buffer is gradually filled from the bottom to the top. If you have not filled the entire buffer and you press the HOME key to go to the top of the buffer, you will not see the first commands of the session. Instead you will see blank lines, since there is nothing at the top of the buffer.

3.2 Format for CodeView Commands and Arguments

The CodeView command format is similar to the format of previous Microsoft debuggers, **SYMDEB** and **DEBUG**. However, some features, particularly operators and expressions, are different. The general format for CodeView commands is shown below:

command [[*arguments*]] [[;*command2*]]

The *command* is a one-, two-, or three-character command name, and *arguments* are expressions that represent values or addresses to be used by the command. The *command* is not case sensitive; any combination of uppercase and lowercase letters can be used. However, *arguments* consisting of source-level expressions may or may not be case sensitive. (For example, C expressions are normally case sensitive, while FORTRAN expressions are not. This can be affected by the language selected for expression evaluation, in the Options menu.) Usually, the first *argument* can be placed immediately after *command* with no space separating the two.

The number of arguments required or allowed with each command varies. If a command takes two or more arguments, you must separate the arguments with spaces. A semicolon (;) can be used as a command separator if you want to specify more than one command on a line.

■ Examples

```
>DB 100 200      ;* Example 1

>U Label1        ;* Example 2, C variable as argument

>U SUM           ;* Example 3, FORTRAN variable as argument
```

```
>U SUM; DB       ;* Example 4, multiple commands
```

In Example 1, DB is the first command (for the Dump Bytes command in this case). The arguments to the command are 100 and 200. The second command on this line is the Comment command (*). A semicolon is used to separate the two commands. The Comment command is used throughout the rest of the manual to number examples.

In Example 2, U is the command letter (for the Unassemble command) and the C language variable Label1 is a command argument. Again the Comment command labels the example.

In Example 3, U is again the command letter (for the Unassemble command) and the FORTRAN variable SUM is a command argument. Again the Comment command labels the example.

Expample 4 consists of three commands, separated by semi-colons. The first is the Unassemble command (U) with the FORTRAN variable SUM as an argument. The second is the Dump Bytes command (DB) with no arguments. The third is the Comment command.

Chapter 4
CodeView Expressions

4.1 C Expressions 88
4.1.1 C Symbols 89
4.1.2 C Constants 90
4.1.3 C Strings 91
4.2 FORTRAN Expressions 92
4.2.1 FORTRAN Symbols 93
4.2.2 FORTRAN Constants 94
4.2.3 FORTRAN Strings 95
4.2.4 FORTRAN Intrinsic Functions 95
4.3 BASIC Expressions 97
4.3.1 BASIC Symbols 99
4.3.2 BASIC Constants 99
4.3.3 BASIC Strings 100
4.3.4 BASIC Intrinsic Functions 101
4.4 MASM Expressions 103
4.5 Line Numbers 104
4.6 Registers and Addresses 105
4.6.1 Registers 105
4.6.2 Addresses 106
4.6.3 Address Ranges 107
4.7 Memory Operators 109
4.7.1 Extracting Segment Addresses (SEG) 109
4.7.2 Extracting Offset Addresses (OFF) 109

4.7.3 Accessing Bytes (BY) 110
4.7.4 Accessing Words (WO) 110
4.7.5 Accessing Double Words (DW) 111
4.8 Switching Expression Evaluators 112

CodeView command arguments are expressions that can include symbols, constant numbers, operators, and registers. Arguments can be simple machine-level expressions that directly specify an address or range in memory, or they can be source-level expressions that correspond to operators and symbols used in Microsoft C, FORTRAN, BASIC or the Macro Assembler.

Each of the three expression evaluators (C, FORTRAN, and BASIC) has a different set of operators and rules of precedence. However, the basic syntax for registers, addresses, and line numbers is the same regardless of the language. You can always specify the expression evaluator. If you specify a language other than the one used in the source, then the expression evaluator will still recognize your program symbols, if possible. (Exception: C and FORTRAN will likely not accept BASIC variables, because they do not understand BASIC type tags.) If you are debugging an assembly routine that is called from BASIC or FORTRAN, then you may want to choose the language of the main program, rather than C.

If the Auto option is on, then the debugger will examine the file extension of each new source file that you trace through. If the new module is in FORTRAN or BASIC, then the debugger will switch the expression evaluator to the appropriate language. Both C and assembly modules will cause the debugger to select C as the expression evaluator.

This chapter deals first with the expressions specific to each language. Line-number expressions are presented next. They work the same way regardless of the language. Then, register and address expressions are presented; generally, these do not have to be mastered unless you are doing assembly-level debugging. Finally, the chapter describes how to switch the expression evaluator.

Note

When you try to use a variable in an expression in a case where that variable is not defined, the CodeView debugger displays the message `UNKNOWN SYMBOL`. For example, the message appears if you try to reference a local variable outside the function where the variable is defined.

4.1 C Expressions

The C expression evaluator uses a subset of operators consisting of the most commonly used C operators and one additional CodeView operator, the colon (:). The CodeView C operators are listed in Table 4.1 in order of precedence.

Table 4.1
CodeView C Operators

Precedence	Operators
(Highest)	
1	( ) [ ] -> .
2	! ~ −[a] (*type*) ++ −− *[b] &[c] **sizeof**
3	*[b] / % :
4	+ −[a]
5	< > <= >=
6	== !=
7	&&
8	||
9	= += −= *= /= %=
(Lowest)	

[a] The minus sign with precedence 2 is the unary minus indicating the sign of a number, while the minus sign with precedence 4 is a binary minus indicating subtraction.

[b] The asterisk with precedence 2 is the pointer operator, while the asterisk with precedence 3 is the multiplication operator.

[c] The ampersand with precedence 2 is the address-of operator. The ampersand as a bitwise-AND operator is not supported by the CodeView debugger.

See the *Microsoft C Compiler Language Reference* for a description of how C operators can be combined with identifiers and constants to form expressions.

The colon operator (:) is the only CodeView C operator that is not a part of the C language itself. The colon acts as a *segment:offset* separator, as described in Section 4.6.2, "Addresses."

With the C expression evaluator, the period (.) has its normal use as a member selection operator, but it also has an extended use as a specifier of local variables in parent functions. The syntax is shown below:

function.variable

The *function* must be a higher-level function and the *variable* must be a local variable within the specified function. The *variable* cannot be a register variable. For example, you can use the expression `main.argc` to refer to the local variable `argc` when you are in a function that has been called by `main`.

The *type* operator (used in type casting) can be any of the predefined C types. The CodeView debugger limits pointer types to one level of indirection. For example, `(char *)sym` is accepted, but `(char **)sym` is not.

When a C expression is used as an argument with a command that takes multiple arguments, the expression should not have any internal spaces. For example, `count+6` is allowed, but `count + 6` may be interpreted as three separate arguments. Some commands (such as the Display Expression command) do permit spaces in expressions.

4.1.1 C Symbols

■ Syntax

name

A symbol is a name that represents a register, an absolute value, a segment address, or an offset address. At the C source level, a symbol is a variable name or the name of a function. Symbols (also called identifiers) follow the naming rules of the C compiler. Note that while CodeView command letters are not case sensitive, symbols given as arguments are case sensitive (unless you have turned off case sensitivity with the Case Sense selection from the Options menu).

In assembly-language output or in output from the Examine Symbols command, the CodeView debugger displays some symbol names in the object-code format produced by the Microsoft C Compiler. This format includes a leading underscore. For example, the function `main` is displayed as `_main`. Only global labels (such as procedure names) are shown in this format. You do not need to include the underscore when specifying such a symbol in CodeView commands. Labels within library routines are sometimes displayed with a double underscore (`__chkstk`). You must use leading underscores when accessing these labels with CodeView commands.

4.1.2 C Constants

■ Syntax

digits	Decimal format
0*digits*	Octal format
0x*digits*	Hexadecimal format
0n*digits*	Alternate decimal format

Numbers used in CodeView commands represent integer constants. They are made up of octal, decimal, or hexadecimal digits, and are entered in the current input radix. The C-language format for entering numbers of different radixes can be used to override the current input radix.

The default radix for the C expression evaluator is decimal. However, you can use the Radix command (N) to specify an octal or hexadecimal radix, as explained in Section 11.3, "Radix Command."

If the current radix is 16 (hexadecimal) or 8 (octal), you can enter decimal numbers in the special CodeView format **0n***digits*. For example, enter 21 decimal as `0n21`.

With radix 16, it is possible to enter a value or argument that could be interpreted either as a symbol or as a hexadecimal number. The CodeView debugger resolves the ambiguity by searching first for a symbol (identifier) with that name. If no symbol is found, the debugger interprets the value as a hexadecimal number. If you want to enter a number that overrides an existing symbol, use the hexadecimal format (0x*digits*).

For example, if you enter abc as an argument when the program contains a variable or function named abc, the CodeView debugger interprets the argument as the symbol. If you want to enter abc as a number, enter it as 0xabc.

Table 4.2 shows how a sample number (63 decimal) would be represented in each radix.

Table 4.2
C Radix Examples

Input Radix	Octal	Decimal	Hexadecimal
8	77	0n63	0x3F
10	077	63	0x3F
16	077	0n63	3F

4.1.3 C Strings

■ Syntax

"*null-terminated-string*"

Strings can be specified as expressions in the C format. You can use C escape characters within strings. For example, double quotation marks within a string are specified with the escape character \".

■ Example

```
>EA message "This \"string\" is okay."
```

The example uses the Enter ASCII command (EA) to enter the given string into memory starting at the address of the variable message.

4.2 FORTRAN Expressions

The FORTRAN expression evaluator uses a subset of operators consisting of the most commonly used FORTRAN operators and two additional operators, the period (.) and colon (:). A number of FORTRAN intrinsic functions, listed in Section 4.2.4, are also supported. FORTRAN function calls are permitted, but statement function names and COMMON block names are not. (Note that these limitations only apply to the arguments of CodeView commands. They do not apply to the source program, which can contain any valid FORTRAN expression. The CodeView FORTRAN operators are listed in Table 4.3 in order of precedence.

Table 4.3
CodeView FORTRAN Operators

Precedence	Operator
(Highest)	
1	()
2	. :
3	Unary + −
4	* /
5	Binary + −
6	.LT. .LE. .EQ. .NE. .GT. .GE.
7	.NOT.
8	.AND.
9	.OR.
10	.EQV. .NEQV.
11	=
(Lowest)	

The FORTRAN expression evaluator does not support the character concatenation operator (//) or the exponentiation operator (**).

The order and precedence with which the CodeView debugger evaluates FORTRAN expressions is the same as in the Microsoft FORTRAN language. See Chapter 2 of the *Microsoft FORTRAN Compiler Language Reference* for a description of how FORTRAN operators can be combined

with symbols and constants to form expressions.

The colon operator (:) may be used when specifying a memory address. It acts as a *segment:offset* separator, as described in Section 4.6.2, "Addresses."

In the CodeView debugger, the period (.) has an extended use as a specifier of local variables in parent routines. The syntax is shown below:

routine.variable

The *routine* must be a higher-level routine and the *variable* must be a local variable within the specified routine. For example, you can use the expression `main.X` to refer to the local variable `X` in the procedure `main` if you are in a routine that is called by `main`. Note that in this example, `main` refers to the main routine of the FORTRAN program being debugged, and is a symbol generated by the Microsoft FORTRAN Optimizing Compiler. It does not appear in FORTRAN source code.

4.2.1 FORTRAN Symbols

■ Syntax

name

A symbol is a name that represents a register, an absolute value, a segment address, or an offset address. At the FORTRAN source level, a symbol is simply a variable name or the name of a routine; you do not necessarily need to know what kind of address it represents. Note that when given as arguments, symbols are never case sensitive with the FORTRAN expression evaluator. If you have turned on case sensitivity with the Case Sense selection from the Options menu, it is turned off automatically when a symbol is used.

In assembly-language output or in output from the Examine Symbols command, the CodeView debugger displays some symbol names in the object-code format produced by the Microsoft FORTRAN Optimizing Compiler. This format includes a leading underscore. For example, the main routine in your program is displayed as `_main`. Only global labels (such as procedure names) are shown in this format. You do not need to include the underscore when specifying such a symbol in CodeView commands. Labels within library routines are sometimes displayed with a double underscore (`__chkstk`). You must use leading underscores when accessing these

labels with CodeView commands.

4.2.2 FORTRAN Constants

- Syntax

digits	Default radix
radix# digits	Specified radix
# digits	Hexadecimal

Numbers used in CodeView commands represent integer constants. They are entered in the current input radix (base). When you are using the FORTRAN expression evaluator, the debugger will recognize any radix between 2 and 36, inclusive, that is explicitly specified, as in `20#2G`. (If you switch to the C expression evaluator, the CodeView debugger will recognize only octal, decimal, and hexadecimal radixes.) The FORTRAN radix specifiers can be used to override the current radix. Note that a hexadecimal number may be entered in two ways. For example, 3F hex could be entered as either `#3F` or `16#3F`. In this manual, we will use the number sign alone to indicate hexadecimal numbers.

The default radix for the FORTRAN version of the CodeView debugger is decimal. However, you can use the Radix command (**N**) to specify an octal or hexadecimal radix, as explained in Section 11.3, "Radix Command."

With radix 16, it is possible to enter a value or argument that could be interpreted either as an identifier or as a hexadecimal number. The CodeView debugger resolves the ambiguity by searching first for a symbol (identifier) with that name. If no symbol is found, the debugger interprets the value as a hexadecimal number. If you want to enter a number that overrides an existing symbol, use the hexadecimal format (*# digits*).

For example, if you enter `ABC` as an argument when the program contains a variable or function named `ABC`, the CodeView debugger interprets the argument as the symbol. If you want to enter `ABC` as a number, enter it as `#ABC`.

Table 4.4 shows how a sample number (63 decimal) would be represented in the octal, decimal, and hexadecimal radixes.

Table 4.4
FORTRAN Radix Examples

Input Radix	Octal	Decimal	Hexadecimal
8	77	10#63	#3F
10	8#77	63	#3F
16	8#77	10#63	3F

4.2.3 FORTRAN Strings

■ Syntax

'string'

Strings can be specified as character expressions in the FORTRAN format. Single quotation marks within a string must be specified by two single quotation marks.

■ Example

```
>EA MESSAGE 'This ''string'' is okay.
```

The above example uses the Enter ASCII command (**EA**) to enter the given string into memory starting at the address of the variable MESSAGE. Note that the string includes embedded single quotation marks and trailing blanks.

4.2.4 FORTRAN Intrinsic Functions

When entering a FORTRAN expression, you may use a limited number of FORTRAN intrinsic functions. The primary use of these functions is to convert a FORTRAN variable or value from one type to another for purposes of calculation. The intrinsic functions recognized by the CodeView debugger's expression evaluator are listed in Table 4.5. See Chapter 3 of the *Microsoft FORTRAN Compiler Language Reference* for a complete description of the FORTRAN intrinsic functions.

Table 4.5
FORTRAN Intrinsic Functions Supported by the CodeView Debugger

Name	Definition	Argument Type	Function Type
CHAR(*int*)	Data-type conversion	int	char[1]
CMPLX(*genA*[,*genB*])	Data-type conversion	int, real, or cmp	cmp8
DBLE(*gen*)	Data-type conversion	int, real, or cmp	dbl
DCMPLX(*genA*[,*genB*])	Data-type conversion	int, real, or cmp	cmp16
DIMAG(*cmp16*)	Imaginary part of cmp16 number	cmp16	dbl
DREAL(*cmp16*)	Data-type conversion	cmp16	dbl
ICHAR(*char*)	Data-type conversion	char	int
IMAG(*cmp*)	Imaginary part of cmp number	cmp	real[2]
INT(*gen*)	Data-type conversion	int, real, or cmp	int
INT1(*gen*)	Data-type conversion	int, real, or cmp	int1
INT4(*gen*)	Data-type conversion	int, real, or cmp	int4
INTC(*gen*)	Data-type conversion	int, real, or cmp	INTEGER[C]
LOCFAR(*gen*)	Segmented address	int, real, or cmp	int4
LOCNEAR(*gen*)	Unsegmented address	int, real, or cmp	int2
REAL(*gen*)	Data-type conversion	int, real, or cmp	real4

[1] The abbreviations used for the different data types in this table are listed in Appendix B of the *Microsoft FORTRAN Compiler Language Reference.*

[2] If argument is COMPLEX*8, function is REAL*4. If argument is COMPLEX*16, function is DOUBLE PRECISION.

4.3 BASIC Expressions

The BASIC expression evaluator uses a subset of operators consisting of the most commonly used BASIC operators. It also supports one important BASIC command, the **LET** command, and one operator in addition to the BASIC operators: the colon (:). The CodeView BASIC operators are listed in Table 4.6 in order of precedence.

Table 4.6
CodeView BASIC Operators

Precedence	Operator
(Highest)	
1	()
2	. :
3	* /
4	\ MOD
5	+ -
6	= <> < > <= >=
7	NOT
8	AND
9	OR
10	XOR
11	EQV
12	IMP
13	LET...=
(Lowest)	

The BASIC expression evaluator does not support the exponentiation operator (^). Nor does it support string assignment, the string concatenation operator (+), or any of the relational operators (=, <, >, etc.) when used with strings. However, arrays, records, and user-defined types are all

supported.

The order and precedence with which the CodeView debugger evaluates BASIC expressions is the same as in the Microsoft BASIC language. See your BASIC documentation for a description of how BASIC operators can be combined with symbols and constants to form expressions.

Important

Earlier versions of Microsoft BASIC use the Microsoft Binary Real format for representing floating-point numbers. However, the CodeView debugger recognizes floating-point numbers (that is, all variables of type single or double) only when they are in the IEEE format. Always use the IEEE format with any program that you are going to use with the CodeView debugger. In some cases, this may require you to recompile with a different version.

The assignment operator (**LET**) is supported for numerical operations only. When you use **LET** in a BASIC expression, the return value will not be useful. However, an assignment will take place whenever the expression is evaluated. This gives you a convenient way for manipulating data. For example, after the expression `LET A = 5` is evaluated, the variable `A` will contain the value 5. You must use the keyword **LET** to specify assignment; otherwise, the BASIC expression evaluator will interpret the equal sign (=) as a test for equality.

The colon operator (`:`) may be used to specify a memory address. It acts as a *segment*`:` *offset* separator, as described in Section 4.6.2, "Addresses."

When a BASIC expression is used as an argument with a command that takes multiple arguments, the expression should not have any internal spaces. For examples, `COUNT+6` is allowed, but `COUNT + 6` may be interpreted as three arguments. Some commands (such as the Display Expression command) only take one argument; these commands do permit spaces in expressions.

4.3.1 BASIC Symbols

■ Syntax

name

A symbol is a name that represents a register, an absolute value, a segment address, or an offset address. At the BASIC source level, a symbol is simply a variable name or the name of a routine; you do not necessarily need to know what kind of address it represents. With the BASIC expression evaluator, symbols follow the naming rules of the BASIC Compiler. In particular, all the type specifiers used in BASIC ($, %, &, !, and #) are accepted by the BASIC expression evaluator. Note that symbols are never case sensitive to BASIC, whether the Case Sense option is on or not.

4.3.2 BASIC Constants

■ Syntax

fixed-point-string[[#¦!]]	Single or double, fixed-point format
floating-point-string[[#¦!]]	Single or double, floating-point format

digits	Integer, default radix
&O*digits*	Octal radix
&*digits*	Alternative octal radix
&H*digits*	Hexadecimal radix

With the BASIC expression evaluator, numbers can be entered as integer, long, single precision or double precision data objects. Constants are formed according to the rules of the Microsoft BASIC Compiler. A single or double precision constant must be entered in decimal radix, regardless of the current system radix. To enter a single or double use the Microsoft BASIC rules for forming fixed and floating point strings.

Constants which are integer or long are entered in the system radix, and are made up of octal, decimal, or hexidecimal digits. You may override the system radix by using the octal, or hexidecimal prefix. To enter integers in the decimal format, the system radix must be 10, and you use the default radix. There is no way enter decimal integers when the system radix is other than 10, unless you switch to another expression evaluator.

The default radix for the BASIC expression evaluator is decimal. However, you can use the Radix command (N) to specify an octal or hexadecimal radix, as explained in Section 11.3, "Radix Command."

With radix 16, it is possible to enter a value or argument that could be interpreted either as an identifier or as a hexadecimal number. The CodeView debugger resolves the ambiguity by searching first for a symbol (identifier) with that name. If no symbol is found, the debugger interprets the value as a hexadecimal number. If you want to enter a number that overrides an existing symbol, use the hexadecimal format (&H*digits*).

For example, if you enter ABC as an argument when the program contains a variable or function named ABC, the CodeView debugger interprets the argument as the symbol. If you want to enter ABC as a number, enter it as &HABC.

Table 4.7 shows how a sample number (63 decimal) would be represented in the octal, decimal, and hexadecimal radixes.

Table 4.7
BASIC Radix Examples

Input Radix	Octal	Decimal	Hexadecimal
8	77	-	&H3F
10	&O77	63	&H3F
16	&O77	-	3F

4.3.3 BASIC Strings

The BASIC expression evaluator does not allow you to input strings while debugging. However, it does recognize both fixed and variable length string variables, as defined by the BASIC compiler. (This includes arrays and records of strings.) Expressions that refer to strings will probably be quite simple, because string operators (concatenation and relational operators) are not supported by the BASIC expression evaluator.

By temporarily switching the expression evaluator to C or FORTRAN (see Section 4.8), you can enter a string literal at a given address. To use this technique effectively, however, you will need to understand how BASIC handles string variables. For more information, see Chapter 6, "Examining Data and Expressions."

4.3.4 BASIC Intrinsic Functions

When entering a BASIC expression, you may use a limited number of BASIC intrinsic functions. The primary use of these functions is to convert a BASIC variable or value from one type to another for purposes of calculation. The intrinsic functions recognized by the CodeView debugger's expression evaluator are listed in Table 4.8. See your BASIC Compiler manual for a complete description of the BASIC intrinsic functions.

Table 4.8
BASIC Instrinsic Functions Supported by the CodeView Debugger

Name	Definition	Argument Type	Function Type
ASC[1]	ASCII value of 1st character	string	integer
CDBL	Data-type conversion	numerical expression	double
CINT	Conversion, with rounding	numerical expression	integer
CSGN	Data-type conversion	numerical expression	single
CVI	Data-type conversion	2-byte string	integer
CVL	Data-type conversion	4-byte string	long
CVS	Data-type conversion	4-byte string	short
CVD	Data-type conversion	8-byte string	double
FIX	Conversion, with truncation	numerical expression	integer
INT	Conversion, with truncation	numerical expression	integer
LBOUND(*arr*[,*dim*])	Lowest index of array	array, dimension	integer
UBOUND(*arr*[,*dim*])	Highest index of array	array, dimension	integer
VAL	Numerical value of string	string	integer, long, single, or double
VARPTR	Offset of variable	variable name	integer
VARPTRSEG	Segment/offset of variable	variable name	long
VARSEG	Segment of variable	variable name	integer

[1] Except where noted, each of the functions in this table takes exactly one argument, of the type indicated in the third column.

4.4 MASM Expressions

Assembly modules use the C expression evaluator by default. However, you can always change the expression evaluator manually. You may want to do this, in fact, if your assembly module is called from FORTRAN or BASIC.

The **/ZI** assembly option directs MASM to provide size-attribute information for the CodeView debugger. This information, in turn, is used by each of the expression evaluators, to properly evaluate a variable. It may be necessary, in some cases, to explicitly declare a variable's type. (With the C expression evaluator, this can be done with a caste; with the BASIC and FORTRAN evaluators, it can be done with an intrinsic function.)

Note

Some versions of MASM will by default use the Microsoft Binary Real format for storing floating point numbers. However, this format is not recognized by the CodeView debugger, even if you are using the BASIC expression evaluator. Therefore, always use the IEEE format with programs that you are debugging. You can direct MASM to use this format by using the **.287** or **.8087** directive, or by using the **/R** assemble option.

Remember to use operators that are supported by the current expression evaluator. For example, to refer to `OFFSET X`, you type `&X` if C is selected, or `VARPTR(X)` if BASIC is selected. This applies only to expressions used in CodeView commands, not to the source code itself.

Stand-alone assembly programs will start up with Case Sense off, even though the C expression evaluator is selected by default.

4.5 Line Numbers

Line numbers are useful for source-level debugging. They correspond to the lines in source-code files (C, FORTRAN, BASIC, or MASM). In source mode, you see a program displayed with each line sequentially numbered. The CodeView debugger allows you to use these same numbers to access parts of a program.

■ Syntax

.[[*filename*:]]*linenumber*

The address corresponding to a source line number can be specified as *linenumber* prefixed with a period (.). The CodeView debugger assumes the source line is in the current source file unless you specify the optional *filename* followed by a colon and the line number.

The CodeView debugger displays an error message if *filename* does not exist, or if no source line exists for the specified number.

■ Examples

```
>V .100
```

The above example uses the View command (V) to display code starting at the source line 100. Since no file is indicated, the current source file is assumed.

```
>V .SAMPLE.FOR:10

>V .EXAMPLE.BAS:22

>V .DEMO.C:301
```

The above examples use the View command to display source code starting at line 10 of SAMPLE.FOR, line 22 of EXAMPLE.BAS, and line 301 of DEMO.C, respectively.

4.6 Registers and Addresses

Addresses are basic to each of the expression evaluators. A data symbol represents an address in a data segment; a procedure name represents an address in a code segment. This section presents alternative ways to refer to objects in memory. All of the syntax in this section can be considered as an extension to the C, FORTRAN, or BASIC expression evaluator.

4.6.1 Registers

- **Syntax**

[[@]]*register*

You can specify a register name if you want to use the current value stored in the register. Registers are rarely needed in source-level debugging, but they are used frequently for assembly-language debugging.

When you specify an identifier, the CodeView debugger first checks the symbol table for a symbol with that name. If the debugger does not find a symbol, it checks to see if the identifier is a valid register name. If you want the identifier to be considered a register, regardless of any name in the symbol table, use the "at sign" (@) as a prefix to the register name. For example, if your program has a symbol called AX, you could specify @AX to refer to the AX register. You can avoid this problem entirely by making sure that identifier names in your program do not conflict with register names.

The register names known to the CodeView debugger are shown in Table 4.9. Note that the 32-bit registers are available only if the 386 option is on, and the computer is running in 386 mode.

Table 4.9
Registers

Type	Names			
8-bit high-byte	AH	BH	CH	DH
8-bit low-byte	AL	BL	CL	DL
16-bit general purpose	AX	BX	CX	DX
16-bit segment	CS	DS	SS	ES
16-bit pointer	SP	BP	IP	
16-bit index	SI	DI		
32-bit general purpose	EAX	EBX	ECX	EDX
32-bit pointer	ESP	EBP		
32-bit index	ESI	EDI		

4.6.2 Addresses

■ Syntax

[[*segment*:]]*offset*

Addresses can be specified in the CodeView debugger through use of the colon operator as a *segment*:*offset* connector. Both the *segment* and the *offset* are made up of expressions.

A full address has a *segment* and an *offset*, separated by a colon. A partial address has just an *offset*; a default segment is assumed. The default *segment* varies, depending on the command with which the address is used. Commands that refer to data (Dump, Enter, Watch, and Tracepoint) use the contents of the **DS** register. Commands that refer to code (Assemble, Breakpoint Set, Go, Unassemble, and View) use the contents of the **CS** register.

Full addresses are seldom necessary in source-level debugging. Occasionally they may be convenient for referring to addresses outside the program, such as BIOS (basic input/output system) or DOS addresses.

■ **Examples**

```
>DB 100
```

In the above example, the Dump Bytes command (DB) is used to dump memory starting at offset address 100. Since no segment is given, the data segment (the default for Dump commands) is assumed.

```
>DB ARRAY(COUNT)        ;* FORTRAN/BASIC example
```

In the above example, the Dump Bytes command is used to dump memory starting at the address of the variable ARRAY(COUNT). In C, a similar variable might be denoted as `array[count]`.

```
>DB label+10
```

In the above example, the Dump Bytes command is used to dump memory starting at a point 10 bytes beyond the symbol `label`.

```
>DB ES:200
```

In the above example, the Dump Bytes command is used to dump memory at the address having the segment value stored in **ES** and the offset address 200.

4.6.3 Address Ranges

■ **Syntax**

startaddress endaddress
startaddress **L** *count*

A range is a pair of memory addresses that bound a sequence of contiguous memory locations.

You can specify a range in two ways. One way is to give the start and end points. In this case the range covers *startaddress* to *endaddress*, inclusive. If a command takes a range, but you do not supply a second address, the CodeView debugger usually assumes the default range. Each command has its own default range. (The most common default range is 128 bytes.)

You can also specify a range by giving its starting point and the number of objects you want included in the range. This type of range is called an object range. In specifying an object range, *startaddress* is the address of the first object in the list, **L** indicates that this is an object range rather than an ordinary range, and *count* specifies the number of objects in the range.

The size of the objects is the size taken by the command. For example, the Dump Bytes command (**DB**) has byte objects, the Dump Words command (**DW**) has words, the Unassemble (**U**) command has instructions, and so on.

■ Examples

```
>DB buffer
```

The above example dumps a range of memory starting at the symbol `buffer`. Since the end of the range is not given, the default size (128 bytes for the Dump Bytes command) is assumed.

```
>DB buffer buffer+20
```

The above example dumps a range of memory starting at `buffer` and ending at `buffer+20` (the point 20 bytes beyond `buffer`).

```
>DB buffer L 20
```

The above example uses an object range to dump the same range as in the previous example. The `L` indicates that the range is an object range, and `20` is the number of objects in the range. Each object has a size of 1 byte, since that is the command size.

```
>U funcname-30 funcname
```

The above example uses the Unassemble command (`U`) to list the assembly-language statements starting 30 instructions before `funcname` and continuing to `funcname`.

4.7 Memory Operators

The memory operators are unary operators that work in the same way regardless of the language selected, and return the result of a direct memory operation. They are chiefly of interest to programmers who debug in assembly mode, and are not necessary for high-level debugging.

All of the operators listed in this section are part of the CodeView expression evaluator, and should not be confused with CodeView commands. As operators, they can only build expressions, which in turn are used as arguments in commands.

4.7.1 Extracting Segment Addresses (SEG)

The segment portion of a 32-bit address can be obtained with the **SEG** operator.

■ Syntax

SEG *address*

The result of the **SEG** operation is a the 16-bit segment portion of *address*.

■ Example

```
>?SEG fardat
20556
>
```

In the above example, the **SEG** operator is used to display the segment portion of the address of `fardat`.

4.7.2 Extracting Offset Addresses (OFF)

The offset portion of a 32-bit address can be obtained with the **OFF** operator.

■ Syntax

OFF *address*

The result of the **OFF** operation is the 16-bit offset portion of *address.*

■ Example

```
>?OFF fardat
1790
>
```

In the above example, the **OFF** operator is used to display the offset portion of the address of `fardat`.

4.7.3 Accessing Bytes (BY)

You can access the byte at an address by using the **BY** operator.

■ Syntax

BY *address*

The result is a 2-byte integer, which contains the value of the first byte stored at *address.*

■ Example

```
>? BY sum
>101
```

The above example returns the first byte at the address of `sum`.

4.7.4 Accessing Words (WO)

You can access the word at an address by using the **WO** operator.

■ **Syntax**

WO *address*

The result is a 2-byte integer, which contains the value of the first 2 bytes stored at *address*.

■ **Example**

```
>? WO sum
>3120
```

The above example returns the first word at the address of sum.

4.7.5 Accessing Double Words (DW)

You can access the word at an address by using the DW operator.

■ **Syntax**

DW *address*

The result is a 4-byte integer, which contains the value of the first 4 bytes stored at *address*.

■ **Example**

```
>? DW sum
>32120365
```

The above example returns the value of the first double word at the address of sum.

4.8 Switching Expression Evaluators

The CodeView debugger allows you to specify a particular expression evaluator: C, FORTRAN, or BASIC. Unless you are debugging a multi-language program, specifying the expression evaluator is usually not necessary. (If you have such a program, you may just want to make use of the Auto option.) You may want to specify the expression evaluator manually if you are debugging a source module *without* the standard extension, or if you wanted to use a feature of a different language. For example, you might be debugging a C program, and want to evaluate a string of binary digits. The FORTRAN expression evaluator accepts base 2, so you might want to temporarily switch to the FORTRAN evaluator.

■ Mouse

To switch expression evaluators with the mouse, open the Language menu and click the appropriate language selection.

■ Keyboard

To switch expression evaluators with a keyboard command, press ALT-L to open up the Language menu, use the arrow keys to move to the appropriate language, then press RETURN.

■ Dialog

To switch expression evaluators using a dialog command, enter a command line with the syntax

USE [[*language*]]

where *language* is C, FORTRAN, or BASIC. The command is not case sensitive, and you can enter the language name in any combination of uppercase and lowercase letters. Entered on a line by itself, USE displays the name of the current expression evaluator.

■ Examples

```
>USE fortran
FORTRAN
```

The above example switches to the FORTRAN expression evaluator.

```
>USE
BASIC
```

The above example displays the name of the current expression evaluator, which in this case happens to be BASIC.

Chapter 5
Executing Code

5.1 Trace Command 118
5.2 Program Step Command 120
5.3 Go Command 123
5.4 Execute Command 126
5.5 Restart Command 127

Several commands execute code within a program. Among the differences between them is the size of step executed by each command. The commands and their step sizes are listed below:

Command	Action
Trace (**T**)	Executes the current source line in source mode, or the current instruction in assembly mode; traces into routines, procedures, or interrupts
Program Step (**P**)	Executes the current source line in source mode, or the current instruction in assembly mode; steps over routines, procedures, or interrupts
Go (**G**)	Executes the current program
Execute (**E**)	Executes the current program in slow motion
Restart (**L**)	Restarts the current program

In window mode, the screen is updated to reflect changes that occur when you execute a Trace, Program Step, or Go command. The highlight marking the current location is moved to the new current instruction in the display window. Values are changed, if appropriate, in the register and watch windows.

In sequential mode, the current source line or instruction is displayed after each Trace, Program Step, or Go command. The format of the display depends on the display mode. The three display modes available in sequential mode (source, assembly, and mixed) are discussed in Chapter 10, "Examining Code."

If the display mode is source (**S+**) in sequential mode, the current source line is shown. If the display mode is assembly (**S–**), the status of the registers and flags and the new current instruction are shown in the format of the Register command (see Chapter 7, "Examining Data and Expressions"). If the display mode is mixed (**S&**), the registers, the new source line, and the new instruction are all shown.

The commands that execute code are explained in Sections 5.1–5.5

Note

If you are executing a section of code with the Go or Program Step command, you can usually interrupt program execution by pressing

CONTROL-BREAK or CONTROL-C. This can terminate endless loops, or it can interrupt loops that are delayed by the Watchpoint or Tracepoint command (see Chapter 9, "Managing Watch Statements"). CONTROL-BREAK or CONTROL-C may not work if your program has a special use for either of these key combinations. If you have an IBM Personal Computer AT (or a compatible computer), you can use the SYSTEM-REQUEST key to interrupt execution regardless of your program's use of CONTROL-BREAK and CONTROL-C.

5.1 Trace Command

The Trace command executes the current source line in source mode, or the current instruction in assembly mode. The current source line or instruction is the one pointed to by the **CS** and **IP** registers. In window mode, the current instruction is shown in reverse video or in a contrasting color.

In source mode, if the current source line contains a call, the CodeView debugger executes the first source line of the called routine. In assembly mode, if the current instruction is **CALL**, **INT** or **REP**, the debugger executes the first instruction of the procedure, interrupt, or repeated string sequence.

Use the Trace command if you want to trace into calls. If you want to execute calls without tracing into them, you should use the Program Step command (**P**) instead. Both commands execute DOS function calls without tracing into them. There is no direct way to trace into DOS function calls.

In source mode, the CodeView debugger will only trace into functions and routines that have source code. For example, if the current line contains a call to an intrinsic function or a standard C library function, the debugger will simply execute the function if you are in source mode, since the source code for Microsoft standard libraries is not available. If you are in assembly or mixed mode, the debugger will trace into the function.

Note

The Trace command (**T**) uses the hardware trace mode of the 8086 family of processors. Consequently, you can also trace instructions

stored in ROM (read-only memory). However, the Program Step command (P) will not work in ROM. Using the Program Step command has the same effect as using the Go command (G).

■ Mouse

To execute the Trace command with the mouse, point to Trace on the menu bar and click the left button.

■ Keyboard

To execute the Trace command with a keyboard command, press the F8 key. This works in both window and sequential modes.

■ Dialog

To execute the Trace command using a dialog command, enter a command line with the following syntax:

T [[*count*]]

If the optional *count* is specified, the command executes *count* times before stopping.

■ Example

The following example shows the Trace command in sequential mode. In window mode, there would be no output from the commands, but the display would be updated to show changes caused by the command.

```
>S+         ;* FORTRAN example
source
>.
9:                CALL INPUT (DATA,N,INPFMT)
>T 3
34:               OPEN (1,FILE='EXAMPLE.DAT',STATUS='OLD')
35:               DO 100 I=1,N
36:               READ (1,'(BN,I10)',END=999) DATA(I)

>
```

The above example sets the display mode to source, then uses the Source Line command to display the current source line. (See Chapter 10, "Examining Code," for a further explanation of the Set Source and Source Line commands.) Note that the current source line calls the subroutine INPUT. The Trace command is then used to execute the next three source lines. These lines will be the first three lines of the subroutine INPUT.

Debugging C and BASIC source code is very similar. If you execute the Trace command when the current source line contains a C function call or a BASIC subprogram call, then the debugger will execute the first line of the called routine.

```
>S-
assembly
>T
AX=0058  BX=3050  CX=000B  DX=3FB0  SP=304C  BP=3056  SI=00CC  DI=40E0
DS=49B7  ES=49B7  SS=49B7  CS=3FB0  IP=0013  NV UP EI PL NZ AC PO NC
3FB0:0013 50              PUSH      AX
>
```

The above example sets the display mode to assembly and traces the current instruction. This example and the next example are the same as the examples of the Program Step command in Section 5.2. The Trace and Program Step commands behave differently only when the current instruction is a call, interrupt, or **REP** instruction.

```
>S&
mixed
>T
AX=0000  BX=319C  CX=0028  DX=0000  SP=304C  BP=3056  SI=00CC  DI=40E0
DS=49B7  ES=49B7  SS=49B7  CS=3FB0  IP=003C  NV UP EI PL NZ NA PO NC
8:                  IF (N.LT.1 .OR. N.GT.1000) GO TO 100
3FB0:003C 833ECE2101   CMP     Word Ptr [21CE],+01        DS:21CE=0028
>
```

The above example sets the display mode to mixed and traces the current instruction.

5.2 Program Step Command

The Program Step command executes the current source line in source mode, or the current instruction in assembly mode. The current source line or instruction is the one pointed to by the CS and IP registers. In window mode, the current instruction is shown in reverse video or in a contrasting

color.

In source mode, if the current source line contains a call, the CodeView debugger executes the entire routine and is ready to execute the line after the call. In assembly mode, if the current instruction is **CALL**, **INT**, or **REP**, the debugger executes the entire procedure, interrupt, or repeated string sequence.

Use the Program Step command if you want to execute over routine, function, procedure, and interrupt calls. If you want to trace into calls, you should use the Trace command (**T**) instead. Both commands execute DOS function calls without tracing into them. There is no direct way to trace into DOS function calls.

■ Mouse

To execute the Program Step command with the mouse, point to Trace on the menu bar and click the right button.

■ Keyboard

To execute the Program Step command with a keyboard command, press the F10 key. This works in both window and sequential modes.

■ Dialog

To execute the Program Step command using a dialog command, enter a command line with the following syntax:

P [[*count*]]

If the optional *count* is specified, the command executes *count* times before stopping.

■ Example

This example shows the Program Step command in sequential mode. In window mode, there would be no output from the commands, but the display would be updated to show changes caused by the command.

```
>S+          ;* FORTRAN/BASIC example
```

```
source
>.
9:              CALL INPUT (DATA,N,INPFMT)
>P 3
10:             CALL BUBBLE (DATA,N)
11:             CALL STATS (DATA,N)
12:             END
>
```

The above example sets the display mode to source, then uses the Source Line command to display the current source line. (See Chapter 10, "Examining Code," for a further explanation of the Set Source and Source Line commands.) Note that the current source line calls the subprogram INPUT. The Program Step command is then used to execute the next three source lines. The first program step executes the entire subprogram INPUT. The next two steps execute the subprograms BUBBLE and STATS, also in their entirety.

The same program, written in C, would behave exactly the same way with the Program Step command. The Program Step command will not trace into a C function call.

```
>S-
assembly
>P
AX=0058  BX=3050  CX=000B  DX=3FB0  SP=304C  BP=3056  SI=00CC  DI=40E0
DS=49B7  ES=49B7  SS=49B7  CS=3FB0  IP=0013  NV UP EI PL NZ AC PO NC
3FB0:0013 50              PUSH      AX
>
```

The above example sets the display mode to assembly and steps through the current instruction. This example and the next example are the same as the examples of the Trace command in Section 5.1. The Trace and Program Step commands behave differently only when the current instruction is a call, interrupt or **REP** instruction.

```
>S&
mixed
>P
AX=0000  BX=319C  CX=0028  DX=0000  SP=304C  BP=3056  SI=00CC  DI=40E0
DS=49B7  ES=49B7  SS=49B7  CS=3FB0  IP=003C  NV UP EI PL NZ NA PO NC
8:                IF (N.LT.1 .OR. N.GT.1000) GO TO 100
3FB0:003C 833ECE2101   CMP     Word Ptr [21CE],+01         DS:21CE=0028
>
```

The above example sets the display mode to mixed and steps through the current instruction.

5.3 Go Command

The Go command starts execution at the current address. There are two variations of the Go command. One simply starts execution and continues to the end of the program or until a breakpoint is encountered. The other variation is a Goto command, in which a destination is given with the command.

The Go command will stop execution when a breakpoint set earlier with the Breakpoint Set (**BP**), Watchpoint (**WP**), or Tracepoint (**TP**) command is encountered. If the command is given in the Goto form, the execution will stop before the destination is reached if a previously set breakpoint is encountered first.

If a destination address is given, but never encountered (for example, if the destination is on a program branch that is never taken), the CodeView debugger executes to the end of the program.

If you enter the Go command and the debugger does not encounter a breakpoint, the entire program is executed and the following message is displayed:

```
Program terminated normally (number)
```

The *number* in parentheses is the value returned by the program (sometimes called the exit or "errorlevel" code).

■ Mouse

To execute the Go command with no destination, point to Go on the menu bar and press either button.

To execute the Goto variation of the Go command, point to the source line or instruction you wish to go to; then press the right button. The highlight marking the current location will move to the source line or instruction you pointed to (unless a breakpoint is encountered first). The CodeView debugger will sound a warning and take no action if you try to go to a comment line or other source line that does not correspond to code.

If the line you wish to go to is in another module, you can use the Load command from the Files menu to load the source file for the other module. Then point to the destination line and press the right button.

■ Keyboard

To use a keyboard command to execute the Go command with no destination, press the F5 key. This works in both window and sequential modes.

To execute the Goto variation of the Go command, move the cursor to the source line or instruction you wish to go to. If the cursor is in the dialog window, first press the F6 key to move the cursor to the display window. When the cursor is at the appropriate line in the display window, press the F7 key. The highlight marking the current location will move to the source line or instruction you pointed to (unless a breakpoint is encountered first). The CodeView debugger will sound a warning and take no action if you try to go to a comment line or other source line that does not correspond to code.

If the line you wish to go to is in another module, you can use the Load command from the Files menu to load the source file for the other module. Then move the cursor to the destination line and press the F7 key.

■ Dialog

To execute the Go command using a dialog command, enter a command line with the following syntax:

G [[*breakaddress*]]

If the command is given with no argument, execution continues until a breakpoint or the end of the program is encountered.

The Goto form of the command can be given by specifying *breakaddress*. The *breakaddress* can be given as a symbol, a line number, or an address in the *segment*:*offset* format. If the offset address is given without a segment, the address in the **CS** register is used as the default segment. If you give *breakaddress* as a line number, but the corresponding source line is a comment, declaration, or blank line, the following message appears:

```
No code at this line number
```

■ Examples

The following examples show the Go command in sequential mode. In window mode, there would be no output from the commands, but the display would be updated to show changes caused by the command.

```
>G

Program terminated normally (0)
>
```

The above example passes control to the instruction pointed to by the current values of the **CS** and **IP** registers. No breakpoint is encountered, so the CodeView debugger executes to the end of the program, where it prints a termination message and the exit code returned by the program (0 in the example).

```
>S+      ;* FORTRAN/BASIC example (source mode)
source
>G BUBBLE
17:              A = B + C
>
```

In the above example, the display mode is first set to source (**S+**). See Chapter 10, "Examining Code," for information on setting the display mode. When the Go command is entered, the CodeView debugger starts program execution at the current address and continues until it reaches the start of the subprogram BUBBLE.

```
>S&      ;* C example (mixed mode)
mixed
>G .22
AX=02F4  BX=0002  CX=00A8  DX=0000  SP=3036  BP=3042  SI=0070  DI=40E0
DS=49B7  ES=49B7  SS=49B7  CS=3FB0  IP=0141  NV UP EI PL NZ NA PO NC
22:              x[i] = x[j];
3FB0:0141 8B7608        MOV     SI,Word Ptr [BP+08]          SS:304A=0070
>
```

The above example passes execution control to the program at the current address and executes to the address of source line 22. If the address with the breakpoint is never encountered (for example, if the program has less than 22 lines, or if the breakpoint is on a program branch that is never taken), the CodeView debugger executes to the end of the program.

Note

Mixed and source mode can be used equally well with all three languages. The examples alternate languages in this chapter simply to be accessible to more users.

```
>S-
assembly
>G #2A8
AX=0049  BX=0049  CX=028F  DX=0000  SP=12F2  BP=12F6  SI=04BA  DI=1344
DS=5DAF  ES=5DAF  SS=5DAF  CS=58BB  IP=02A8  NV UP EI PL NZ NA PE NC
58BB:02A8 98            CBW
>
```

The above example executes to address CS:#2A8. Since no segment address is given, the CS register is assumed.

5.4 Execute Command

The Execute command is similar to the Go command with no arguments, except that it executes in slow motion (several source lines per second). Execution starts at the current address and continues to the end of the program or until a breakpoint, tracepoint, or watchpoint is reached. You can also stop automatic program execution by pressing any key or a mouse button.

■ Mouse

To execute code in slow motion with the mouse, point to Run on the menu bar, press a mouse button and drag the highlight down to the Execute selection, then release the button.

■ Keyboard

To execute code in slow motion with a keyboard command, press ALT-R to open the Run menu, then press ALT-E to select Execute.

■ Dialog

To execute code in slow motion using a dialog command, enter a command line with the following syntax:

E

You cannot set a destination for the Execute command as you can for the Go command.

In sequential mode, the output from the Execute command depends on the display mode (source, assembly, or mixed). In assembly or mixed mode, the command executes one instruction at a time. The command displays the current status of the registers and the instruction. In mixed mode, it will also show a source line if there is one at the instruction. In source mode, the command executes one source line at a time, displaying the lines as it executes them.

Important

The Execute command has the same command letter (**E**) as the Enter command. If the command has at least one argument, it is interpreted as Enter; if not, it is interpreted as Execute.

5.5 Restart Command

The Restart command restarts the current program. The program is ready to be executed just as if you had restarted the CodeView debugger. Program variables are re-initialized, but any existing breakpoints or watch statements are retained. The pass count for all breakpoints is reset to 1. Any program arguments are also retained, though they can be changed with the dialog version of the command.

The Restart command can only be used to restart the current program. If you wish to load a new program, you must exit and restart the CodeView debugger with the new program name.

■ Mouse

To restart the program with the mouse, point to Run on the menu bar, press a mouse button and drag the highlight down to the Restart or Start selection, then release the button. The program will be restarted. If the Restart selection is chosen, the program will be ready to start executing from the beginning. If the Start selection is chosen, the program starts executing from the beginning and continues until a breakpoint or the end of the program is encountered.

■ Keyboard

To restart the program with a keyboard command, press ALT-R to open the Run menu, then press either ALT-R to select Restart or ALT-S to select Start. The program will be restarted. If the Restart selection is chosen, the program will be ready to start executing from the beginning. If the Start selection is chosen, the program starts executing from the beginning and continues until a breakpoint or the end of the program is encountered.

■ Dialog

To restart the program with a dialog command, enter a command line with the following syntax:

L [[*arguments*]]

When you restart using the dialog version of the command, the program will be ready to start executing from the beginning. If you want to restart with new program arguments, you can give optional *arguments*. You cannot specify new arguments with the mouse or keyboard version of the command.

Note

The command letter **L** is a mnemonic for Load, but the command should not be confused with the Load selection from the File menu, since that selection only loads a source file without restarting the program.

■ Examples

```
>L
>
```

The above example starts the current executable file, retaining any breakpoints, watchpoints, tracepoints, and arguments.

```
>L 6
>
```

The above example restarts the current executable file, but with 6 as the new program argument.

Chapter 6
Examining Data and Expressions

6.1 Display Expression Command 133
6.2 Examine Symbols Command 143
6.3 Dump Commands 148
6.3.1 Dump 150
6.3.2 Dump Bytes 151
6.3.3 Dump ASCII 151
6.3.4 Dump Integers 152
6.3.5 Dump Unsigned Integers 153
6.3.6 Dump Words 154
6.3.7 Dump Double Words 154
6.3.8 Dump Short Reals 155
6.3.9 Dump Long Reals 156
6.3.10 Dump 10-Byte Reals 157
6.4 Compare Memory Command 158
6.5 Search Memory Command 159
6.6 Port Input Command 161
6.7 Register Command 162
6.8 8087 Command 164

The CodeView debugger provides several commands for examining different kinds of data, including expressions, variables, memory, and registers. The data-evaluation commands discussed in this chapter are summarized below:

Command	Action
Display Expression (**?**)	Evaluates and displays the value of symbols or expressions
Examine Symbol (**X?**)	Displays the addresses of symbols
Dump (**D**)	Displays sections of memory containing data; there are several variations for examining different kinds of data
Compare Memory (**C**)	Compares two blocks of memory, byte-by-byte
Search Memory (**S**)	Scans memory for specified byte values
Port Input (**I**)	Reads a byte from a hardware port
Register (**R**)	Shows the current values of each register and each flag
8087 (**7**)	Shows the current value in the 8087 or 80287 register

6.1 Display Expression Command

The Display Expression command displays the value of CodeView expressions.

Each of the expression evaluators (C, FORTRAN, BASIC) accepts a different set of symbols, operators, functions, and constants, as explained in Chapter 5, "CodeView Expressions." The resulting expressions can contain the intrinsic functions listed for the FORTRAN and BASIC evaluators; they may also contain functions that are part of the executable file. The simplest form of expression is a symbol representing a single variable or routine.

Note

FORTRAN subroutines and BASIC subprograms do not return values as functions do. They can be used in expressions, and in fact may be useful for observing side-effects. However, the value returned by the expression will be meaningless.

In addition to its primary purpose of displaying values, the Display Expression command can also set values as a side effect. For example, with the C expression evaluator you can increment the variable n by using the expression `++n` with the Display Expression command. With the FORTRAN expression evaluator you would use `N=N+1` and with BASIC you would use `LET N=N+1`. After being incremented, the new value will be displayed.

You can specify the format in which the values of expressions are displayed by the Display Expression command. Type a comma after the expression, followed by a CodeView format specifier. The format specifiers used in the CodeView debugger are a subset of those used by the C **printf** function. They are listed in Table 6.1.

Table 6.1
CodeView Format Specifiers

Character	Output Format	Sample Expression	Sample Output
d	Signed decimal integer	`?40000,d`	`40000`
i	Signed decimal integer	`?40000,i`	`40000`
u[1]	Unsigned decimal integer	`?40000,u`	`40000`
o	Unsigned octal integer	`?40000,o`	`116100`
x or **X**[2]	Hexadecimal integer	`?40000,x`	`9c40`
f	Signed value in floating-point decimal format with six decimal places	`?3./2.,f`	`1.500000`
e or **E**[3]	Signed value in scientific-notation format with up to six decimal places (trailing zeros and decimal point are truncated)	`?3./2.,e`	`1.500000e+000`

g or **G**[3]	Signed value with floating-point decimal format (**f**) or scientific-notation format (**g** or **G**), whichever is more compact	`?3./2.,g`	`1.5`
c	Single character	`?65,c`	`A`
s[4]	Characters printed up to the first null character	`?'String',s` `?"String",s`	`String`

[1] FORTRAN and BASIC have no unsigned data types. Using an unsigned format specifier has no effect on the output of positive numbers, but causes negative numbers to be output as positive values.

[2] Hexadecimal letters are uppercase if the type is **X** and lowercase if the type is **x**.

[3] The "E" in scientific-notation numbers is uppercase if the type is **E** or **G**, lowercase if the type is **e** or **g**.

[4] Two examples are given: the first uses the single quotes required by FORTRAN; the second uses the double quotes required by C. Do not use the **s** type specifier with BASIC strings.

If no format specifier is given, single- and double-precision real numbers are displayed as if the format specifier had been given as **g**. (If you are familiar with the C language, you should note that the **n** and **p** format specifiers and the **F** and **H** prefixes are not supported by the CodeView debugger even though they are supported by the C **printf** function.)

The prefix **h** can be used with the integer format specifiers (**d**, **o**, **u**, **x**, and **X**) to specify a 2-byte integer. The prefix **l** can be used with the same types to specify a 4-byte integer. For example, the command `?100000,ld` produces the output `100000`. However, the command `?100000,hd` evaluates only the the low-order 2 bytes, producing the output `-31072`.

The Display Expression command does not work for programs assembled with Microsoft Macro Assembler versions 4.0 and earlier, because the assembler does not write information to the object file about the type size of each variable. Use the Dump command instead.

When calling a FORTRAN subroutine that uses alternate returns, the value of the return labels in the actual parameter list must be 0. For example, the subroutine call `CALL PROCESS (I,*10,J,*20,*30)` must be called from the debugger as `?PROCESS(IARG1,0,IARG2,0,0)`.

Note

Do *not* use a type specifier when evaluating strings in BASIC. Simply leave off the type specifier, and the BASIC expression evaluator will display the string correctly. The `s` type specifier assumes the C language string format, which BASIC conflicts with; if you use `s`, then the debugger will simply display characters at the given address until a null is encountered.

Using other values as return labels will cause the error `Type clash in function argument` or `Unknown symbol`

■ Mouse

The Display expression command cannot be executed with the mouse.

■ Keyboard

The Display expression command cannot be executed with a keyboard command.

■ Dialog

To display the value of an expression using a dialog command, enter a command line with the following syntax:

? *expression*[[,*format*]]

The *expression* is any valid CodeView expression, and the optional *format* is a CodeView format specifier.

■ C Examples

The following examples assume that the C source file contains the following variable declarations:

```
int     amount
char    *text
int     miles
char    hours
struct  {
```

```
        char name[20];
        int id;
        long class;
} student, *pstudent;

int     square (int);
```

Assume also that the program has been executed to the point where all these variables have been assigned values, and that the C expression evaluator is selected.

```
>? amount        ;* Example 1
500
>? amount,x
1f4
>? amount,o
764
>? &amount,X
1EE2
>
```

Example 1 displays the value of the variable amount, first in the default decimal format, then in hexadecimal and then in octal. Finally, the address-of operator is used to display the address where the value of amount is stored. Only the offset portion of the address is shown; the data segment is assumed.

```
>? 92,X              ;* Example 2
5C
>? 109*37,o
7701
>? 'T'
84
>? 118,c
v
>
```

Example 2 illustrates how the CodeView debugger can be used as a calculator. You can convert between radixes, calculate the value of constant expressions, or check ASCII equivalents.

```
>? text,X        ;* Example 3
13F3
>DA 0x13F3
3D83:13F0  Here is a string.
>? text,s
Here is a string.
>
```

Example 3 shows how to examine strings. One method is to evaluate the variable that points to the string, then dump the values at that address (the Dump command is explained in Section 6.3). A more direct method is to use the `s` type specifier.

```
>? miles          ;* Example 4
837
>? hours
14
>? miles/hours
59
>? (float)miles/hours,f
59.785714
>? (float)miles/hours,e
5.978571e+001
>
```

Example 4 displays the value of the symbols `miles` and `hours`. The two variables are then combined to calculate miles per hour. The value is calculated using integer division, then it is type cast so that real-number division can be performed. The real number is shown both in floating-point format using the `f` type specifier, and in scientific notation using the `e` type specifier.

```
>? student.id     ;* Example 5
19643
>? pstudent->id
19643
>
```

Example 5 illustrates how to display the values of members of a structure (or union).

```
>? amount         ;* Example 6
500
>? ++amount
501
>? amount=600
600
>
```

Example 6 shows how the Display Expression command can be used to change the values of variables.

```
>? square(9)      ;* Example 7
81
>
```

Example 7 shows how functions can be evaluated in expressions. The CodeView debugger executes the function `square` with an argument of 9 and displays the value returned by the function. You can only display the values of functions after you have executed into the `main` function.

■ FORTRAN Examples

These examples assume that the FORTRAN source file contains the following variable declarations:

```
INTEGER*2 MILES
REAL HOURS
CHARACTER*16 STR
MILES = 500
HOURS = 5.32
STR = 'Here is a string'
```

Assume also that the program has been executed to the point where all these variables have been assigned values, and that the FORTRAN expression evaluator is selected.

```
>? MILES
500
>? MILES,x
01f4
>? MILES,o
764
>
```

The above example displays the value of the variable `MILES`, first in the default decimal format, then in hexadecimal, and then in octal.

```
>? 92,X
005C
>? 109*37,o
7701
>? 'T'
84
>? 118,c
v
>
```

The above example illustrates how the CodeView debugger can be used as a calculator. You can convert between radixes, calculate the value of

constant expressions, or check ASCII equivalents.

```
>X?STRING
44CD:07EA
>DA #07EA
44CD:07EA  Here is a string...
>? STRING,s
'Here is a string'
>
```

The above example shows how to examine strings. One method is to evaluate the variable that points to the string, then dump the values at that address (the Dump command is explained in Section 6.3). A more direct method is to use the `s` format specifier.

```
>? MILES
500
>? HOURS
5.320000
>? MILES/HOURS
93.984962
>? MILES/HOURS,e
9.398496e+001
>
```

The above example displays the value of the symbols `MILES` and `HOURS`. The two variables are then combined to calculate miles per hour. The real number is shown both in real-number format and in scientific notation using the `e` format specifier.

```
>? MILES
500
>? MILES+1
501
>? MILES=600
600
>
```

The above example shows how the Display Expression command can be used to change the values of variables.

■ BASIC Examples

These examples assume that the BASIC source file contains the following variable declarations:

```
DEFINT M
DEFSNG H
MILES = 500
HOURS = 5.32
A$ = "ABC"
```

Assume also that the program has been executed to the point where all these variables have been assigned values, and that the BASIC expression evaluator is selected.

```
>? MILES
500
>? MILES,x
01f4
>? MILES,o
764
>
```

The above example displays the value of the variable MILES, first in the default decimal format, then in hexadecimal, and then in octal.

```
>? 92,X
005C
>? 109*37,o
7701
>? ASC(A$)
65
>? 118,c
v
>
```

The above example illustrates how the CodeView debugger can be used as a calculator. You can convert between radixes, calculate the value of constant expressions, or check ASCII equivalents.

```
>? MILES
500
>? HOURS
```

```
5.320000
>? MILES/HOURS
93.984962
>? MILES/HOURS,e
9.398496e+001
>
```

The above example displays the value of the symbols MILES and HOURS. The two variables are then combined to calculate miles per hour. The real number is shown both in real-number format and in scientific notation using the e format specifier. Note that the above examples will work correctly only if the BASIC program is using the IEEE format for floating-point numbers.

```
>DW S$ L 2
0003 22B4
>DA &H22B4
ABC
>? S$
"ABC"
>
```

The above example shows two methods for examining strings. The first method starts by dumping the first 2 words at the address of S$. (The Dump command is explained in Section 6.3.) The variable S$ is really a string descriptor, which consists of 2 words that describe the length and location of the string data itself. The first word (a word being 2 bytes of data) contains the length of the string, which in this case is 3; the second word contains the address (offset) of the string data, &H22B4. This is the address you would use if you wanted to manipulate the string directly. However, the address of the string data is subject to change as BASIC dynamically allocates memory space for strings.

The second method is much simpler. You merely use the Display Expression command, with no format descriptor. Do not use the s type specifier, because in that case the C string format is assumed. As long as the BASIC expression evaluator is selected, then BASIC strings will be properly displayed.

```
>? LET MILES=600
600
>
```

The above example shows how the Display Expression command can be used to change the values of variables.

6.2 Examine Symbols Command

The Examine Symbols command displays the names and addresses of symbols, and the names of modules, defined within a program. You can specify the symbol or group of symbols you want to examine by module, procedure, or symbol name.

■ Mouse

This command cannot be executed with the mouse.

■ Keyboard

This command cannot be executed with a keyboard command.

■ Dialog

To view the addresses of symbols using a dialog command, enter a command line in one of the following formats:

X*
X? [[*module*!]] [[*routine.*]] [[*symbol*]] [[*]]

in which *routine* is in program unit, such as C functions or BASIC subprograms, which is capable of having its own local variables.

The syntax combinations are listed in more detail below:

Syntax	Display
X?*module*!*routine.symbol*	The specified *symbol* in the specified *routine* in the specified *module*
X?*module*!*routine.**	All symbols in the specified *routine* in the specified *module*
X?*module*!*symbol*	The specified *symbol* in the specified *module*; symbols within routines are not found

X?*module*!*	All symbols in the specified *module*
X?*routine.symbol*	The specified *symbol* in the specified *routine*; looks for *routine* first in the current module, then in other modules from first to last
X?*routine.**	All symbols in the specified *routine*; looks for *routine* first in the current module, then in other modules from first to last
X?*symbol*	Looks for the specified *symbol* in this order: 1. In the current routine 2. In the current module 3. In other modules, from first to last
X?*	All symbols in the current routine
X*	All module names

To view global symbols when debugging assembly language modules, you must specify the module. For example, use `X?module` instead of `X?*`.

■ C Examples

For the following examples, assume that the program being examined is called `pi.exe`, and that it consists of two modules: `pi.c` and `math.c`. The `pi.c` module is a skeleton consisting only of the `main` function, while the `math.c` module has several functions. Assume that the current function is **div** within the *math* module.

```
>X*                ;*Example 1
PI.OBJ
MATH.OBJ
C:B(chkstk)
C:B(crt0)
.
.
.
C:B(itoa)
C:B(unlink)
```

```
>
```

Example 1 lists the two modules called by the program. The library file and each of the modules called by the program are also listed.

```
>X?*                ;*Example 2
              DI          int             b
              [BP-0006]   int             quotient
              SI          int             i
              [BP-0002]   int             remainder
              [BP+0004]   int             divisor
>
```

Example 2 lists the symbols in the current function (**div**). Local variables are shown as being stored either in a register (`b` in register `DI`), or at a memory location specified as an offset from a register (`divisor` at location `[BP+0004]`).

```
>X?pi!*             ;* Example 3
3D37:19B2 int            _scratch0        3D37:0A10 char          _p[]
3D37:2954 int            _scratch1        3D37:19B4 char          _t[]
3D37:2956 int            _scratch2        3D37:19B0 int           _q
3A79:0010 int            _main()           3A79:0010 int            main()
3D37:19B2 int             scratch0
3D37:0A10 char            p[]
3D37:2954 int             scratch1
3D37:19B4 char            t[]
3D37:2956 int             scratch2
3D37:19B0 int             q
>
```

Example 3 shows all the symbols in the the `pi.c` module.

```
>X?math!div.*    ;*Example 4
3A79:0264 int                 div()
              DI          int             b
              [BP-0006]   int             quotient
              SI          int             i
              [BP-0002]   int             remainder
              [BP+0004]   int             divisor
>
```

Example 4 shows the symbols in the `div` function in module `math.c`. You wouldn't need to specify the module if `math.c` were the current module, but you would if the current module were `pi.c`.

Variables that are local to a function are indented under that function.

```
>X?math!arctan.s ;* Example 5
3A79:00FA int                 arctan()
```

```
            [BP+0004] int                s
>
```

Example 5 shows one specific variable (s) within the arctan function.

■ FORTRAN Examples

For the following examples, assume that the program being examined is called FRUST.EXE, and that it consists of four modules: FRUST.FOR, FRUST1.FOR, FRUST2.FOR, and FRUST3.FOR. Assume that the current routine is main within the FRUST.FOR module.

```
>X*
FRUST.OBJ
FRUST1.OBJ
FRUST2.OBJ
FRUST3.OBJ
c:\lib\LLIBFORE.LIB(fixups)
c:\lib\LLIBFORE.LIB(crt0)
c:\lib\LLIBFORE.LIB(chkstk)
c:\lib\LLIBFORE.LIB(wr)
.
.
.
c:\lib\LLIBFORE.LIB(txtmode)
c:\lib\LLIBFORE.LIB(_creat)
```

The above example lists the three modules called by the program. The library files called by the program are also listed.

```
>X?T
        520D:0DE4 REAL*4         T
```

The above example shows the address of the variable T in the current module.

```
>X?FRUST3!MULTPI.*
4B28:0005 INTEGER*4         MULTPI()
                [BP+000A]                    V
                [BP+0006]                    X
                [BP-0004] INTEGER*4          MULTPI
```

The above example lists the symbols in the function MULTPI, located in module FRUST3. Variables that are local to the function are indented

under the function. You wouldn't need to specify the module if FRUST3 were the current module.

```
>X?FRUST2!SAREA.*
4B15:000E void                SAREA()
                [BP+0012]                       R1
                [BP+000E]                       R2
                [BP+000A]                       H
                [BP+0006]                       T
                520D:0DEC REAL*4                S12
                520D:0DE8 REAL*4                U
```

The above example shows all the symbols in the routine SAREA in the module FRUST2. Because SAREA is a subroutine instead of a function, the word void appears where function return value types are shown.

■ BASIC Examples

For the following examples, assume that the program being examined is called PROG.EXE, and that it consists of the following modules: PROG.BAS, and SORT.BAS. Assume that the current routine is the main program (which unlike subprograms, has no name in a BASIC program), and that the module SORT.BAS contains two subprograms, SORT and SWITCH.

```
>X*
PROG.OBJ
SORT.OBJ
BRUN303.LIB(ftmdata)
BRUN303.LIB(crt0)
BRUN303.LIB(crt0dat)
.
.
.
BRUN303.LIB(doexec)
BRUN303.LIB(execmsg)
```

The above example lists the two modules of the program, including PROG.OBJ, which is the main module. The BASIC library files called by the program are also listed.

```
>X?*
        5825:17BE integer           A%[array]
        5825:1780 single            HOURS!
        5825:1784 integer           I%
```

The above example lists the symbols in the current routine, which happens to be the main program. Although the main program has no label and therefore will not show up in a stack trace, it is still an independent routine and has its own local variables. In BASIC, local variables are not put on the stack unless they are subprogram parameters.

```
>X?*SORT!*
        572F:0033 integer           SORT()
        572F:00E1 integer           SWITCH()
```

The above example lists the routines in the module SORT.OBJ. This form of the Display Symbols command lists routines only, not variables. Note that SORT() and SWITCH() are given with the addresses of the two subprograms by that name.

```
>X?SORT!SWITCH.*
        [BP+0008] integer           B%
        [BP+0006] integer           C%
        5824:1798 integer           TEMP%
```

The above examples shows all the symbols in the routine SWITCH, which is in the SORT.OBJ module . Each represents an integer; however, B% and C% represent subprogram parameters that were passed on the stack, while TEMP% is a true subprogram variable. Therefore, TEMP% has an absolute address in memory, while B% and C% are addressed relative to the stack. (BP points to the value of the stack at the time the routine SWITCH was called.)

6.3 Dump Commands

The CodeView debugger has several commands for dumping data from memory to the screen (or other output device). The dump commands are listed below:

Command	Command Name
D	Dump (size is the default type)
DB	Dump Bytes
DA	Dump ASCII

DI	Dump Integers
DU	Dump Unsigned Integers
DW	Dump Words
DD	Dump Double Words
DS	Dump Short Reals
DL	Dump Long Reals
DT	Dump 10-Byte Reals

■ Mouse

The Dump commands cannot be executed with the mouse.

■ Keyboard

The Dump commands cannot be executed with keyboard commands.

■ Dialog

To execute any Dump command using a dialog command, enter a command line with the following syntax:

D[[*type*]] [[*address* | *range*]]

The *type* is a one-letter specifier that indicates the type of the data to be dumped. The dump commands expect either a starting *address* or a *range* of memory. If the starting *address* is given, the commands assume a default range (usually 128 bytes) starting at *address*. If *range* is given, the commands dump from the start to the end of *range*.

If neither *address* nor *range* is given, the commands assume the current dump address as the start of the range and the default size associated with the size of the object as the length of the range. Most Dump commands have a default range size of 128 bytes, but the Dump Real commands have a default range size of one real number.

The current dump address is the byte following the last byte specified in the previous Dump command. If no Dump command has been used during the session, the dump address is the start of the data segment (**DS**). For example, if you enter the Dump Words command with no argument as the

first command of a session, the CodeView debugger displays the first 64 words (128 bytes) of data declared in the data segment. If you repeat the same command, the debugger displays the next 64 words following the ones dumped by the first command.

Note

If the value in memory cannot be evaluated as a real number, the Dump commands that display real numbers (Dump Short Reals, Dump Long Reals, or Dump 10-Byte Reals) will display a number containing one of the following character sequences: `#NAN`, `#INF`, or `#IND`. NAN (not a number) indicates that the data cannot be evaluated as a real number. INF (infinity) indicates that the data evaluates to infinity. IND (indefinite) indicates that the data evaluates to an indefinite number.

Sections 6.3.1–6.3.10 discuss the variations of the Dump commands in order of the size of data they display.

6.3.1 Dump

■ Syntax

D [[*address* | *range*]]

The Dump command displays the contents of memory at the specified *address* or in the specified *range* of addresses. The command dumps data in the format of the default type. The default type is the last type specified with a Dump, Enter, Watch Memory, or Tracepoint Memory command. If none of these commands has been entered during the session, the default type is bytes.

The Dump command displays one or more lines, depending on the address or range specified. Each line displays the address of the first item displayed. The Dump command must be separated by at least one space from any *address* or *range* value. For example, to dump memory starting at symbol `a`, use the command `D a`, not `Da`. The second syntax would be interpreted as the Dump ASCII command.

6.3.2 Dump Bytes

■ Syntax

DB [*address* | *range*]

The Dump Bytes command displays the hexadecimal and ASCII values of the bytes at the specified *address* or in the specified *range* of addresses. The command displays one or more lines, depending on the address or range supplied.

Each line displays the address of the first byte in the line, followed by up to 16 hexadecimal byte values. The byte values are immediately followed by the corresponding ASCII values. The hexadecimal values are separated by spaces, except the eighth and ninth values, which are separated by a dash (–). ASCII values are printed without separation. Unprintable ASCII values (less than 32 or greater than 126) are displayed as dots. No more than 16 hexadecimal values are displayed in a line. The command displays values and characters until the end of the *range* or, if no *range* is given, until the first 128 bytes have been displayed.

■ Example

```
>DB 0 36
3D5E:0000 53 6F 6D 65 20 6C 65 74-74 65 72 73 20 61 6E 64 Some letters and
3D5E:0010 20 6E 75 6D 62 65 72 73-3A 00 10 EA 89 FC FF EF  numbers:.......
3D5E:0020 00 F0 00 CA E4          -                       .....
>
```

The above example displays the byte values from `DS:0` to `DS:36` (36 decimal is equivalent to 24 hexidecimal). The data segment is assumed if no segment is given. ASCII characters are shown on the right.

6.3.3 Dump ASCII

■ Syntax

DA [*address* | *range*]

The Dump ASCII command displays the ASCII characters at a specified *address* or in a specified *range* of addresses. The command displays one or more lines of characters, depending on the *address* or *range* specified.

If no ending address is specified, the command dumps either 128 bytes or all bytes preceding the first null byte, whichever comes first. Up to 64 characters per line are displayed. Unprintable characters, such as carriage returns and line feeds, are displayed as dots. ASCII characters less than 32 and greater than 126 are unprintable.

■ Examples

```
>DA 0
3D7C:0000  Some letters and numbers:
>
```

The above example displays the ASCII values of the bytes starting at DS:0. Since no ending address is given, values are displayed up to the first null byte.

```
>DA 0 36
3D7C:0000  Some letters and numbers:.....
>
```

In the above example, an ending address is given, so the characters from DS:0 to DS:36 (24 hexidecimal) are shown. Unprintable characters are shown as dots.

6.3.4 Dump Integers

■ Syntax

DI [[*address* | *range*]]

The Dump Integers command displays the signed decimal values of the words (2-byte values) starting at *address* or in the specified *range* of addresses. The command displays one or more lines, depending on the address or range specified. Each line displays the address of the first integer in the line, followed by up to eight signed decimal words. The values are separated by spaces. The command displays values until the end of the *range* or until the first 64 2-byte integers have been displayed, whichever comes first.

Note

In this manual an integer is considered a 2-byte value, since that is the integer size the CodeView debugger assumes. Note that a default FORTRAN integer is a 4-byte value.

■ Example

```
>DI 0 36
3D5E:0000   28499  25965  27680  29797  25972  29554  24864  25710
3D5E:0010   28192  28021  25954  29554     58  -5616   -887  -4097
3D5E:0020   -4096 -13824   2532
>
```

The above example displays the byte values from DS:0 to DS:36 (24 hexidecimal) Compare the signed decimal numbers at the end of this dump with the same values shown as unsigned integers in Section 6.3.5.

6.3.5 Dump Unsigned Integers

■ Syntax

DU [[*address* | *range*]]

The Dump Unsigned Integers command displays the unsigned decimal values of the words (2-byte values) starting at *address* or in the specified *range* of addresses. The command displays one or more lines, depending on the address or range specified. Each line displays the address of the first unsigned integer in the line, followed by up to eight decimal words. The values are separated by spaces. The command displays values until the end of the *range* or until the first 64 unsigned integers have been displayed, whichever comes first.

■ Example

```
>DU 0 36
3D5E:0000   28499  25965  27680  29797  25972  29554  24864  25710
3D5E:0010   28192  28021  25954  29554     58  59920  64649  61439
3D5E:0020   61440  51712   2532
>
```

The above example displays the byte values from DS:0 to DS:36 (24 hexidecimal) Compare the unsigned decimal numbers at the end of this dump with the same values shown as signed integers in Section 6.3.4.

6.3.6 Dump Words

■ Syntax

DW [[*address* | *range*]]

The Dump Words command displays the hexadecimal values of the words (2-byte values) starting at *address* or in the specified *range* of addresses. The command displays one or more lines, depending on the address or range specified. Each line displays the address of the first word in the line, followed by up to eight hexadecimal words. 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, whichever comes first.

■ Example

```
>DW 0 36
3D5E:0000  6F53 656D 6C20 7465 6574 7372 6120 646E
3D5E:0010  6E20 6D75 6562 7372 003A EA10 FC89 EFFF
3D5E:0020  F000 CA00 09E4
>
```

The above example displays the word values from DS:0 to DS:36 (24 hexidecimal). No more than eight values per line are displayed.

6.3.7 Dump Double Words

■ Syntax

DD [[*address* | *range*]]

The Dump Double Words command displays the hexadecimal values of the double words (4-byte values) starting at *address* or in the specified *range* of addresses.

The command displays one or more lines, depending on the address or range specified. Each line displays the address of the first double word in the line, followed by up to four hexadecimal double-word values. The words of each double word are separated by a colon. The values are separated by spaces. The command displays values until the end of the *range* or until the first 32 double words have been displayed, whichever comes first.

■ Example

```
>DD 0 36
3D5E:0000  656D:6F53 7465:6C20 7372:6574 646E:6120
3D5E:0010  6D75:6E20 7372:6562 EA10:003A EFFF:FC89
3D5E:0020  CA00:F000 6F73:09E4
>
```

The above example displays the double-word values from `DS:0` to `DS:36` (24 hexidecimal). No more than four double-word values per line are displayed.

6.3.8 Dump Short Reals

■ Syntax

DS [[*address* | *range*]]

The Dump Short Reals command displays the hexadecimal and decimal values of the short (4-byte) floating-point numbers at *address* or in the specified *range* of addresses.

The command displays one or more lines, depending on the address or range specified. Each line displays the address of the floating-point number in the first column. Next, the hexadecimal values of the bytes in the number are shown, followed by the decimal value of the number. The hexadecimal values are separated by spaces.

The decimal value has the following form:

[[–]] *digit.digits*E{+ | –} *exponent*

If the number is negative, it will have a minus sign; positive numbers have no sign. The first digit of the number is followed by a decimal point. Six

decimal places are shown following the decimal point. The letter **E** follows the decimal digits, and marks the start of a three-digit signed *exponent.*

The command displays at least one value. If a *range* is specified, all values in the range are displayed.

■ Example

```
>DS SPI
5E68:0100  DB 0F 49 40  3.141593E+000
>
```

The above example displays the short-real floating-point number at the address of the variable `SPI`. Only one value is displayed per line.

6.3.9 Dump Long Reals

■ Syntax

DL [[*address* | *range*]]

The Dump Long Reals command displays the hexadecimal and decimal values of the long (8-byte) floating-point numbers at the specified *address* or in the specified *range* of addresses.

The command displays one or more lines, depending on the address or range specified. Each line displays the address of the floating-point number in the first column. Next, the hexadecimal values of the bytes in the number are shown, followed by the decimal value of the number. The hexadecimal values are separated by spaces.

The decimal value has the following form:

[[–]] *digit.digits***E**{ + | – } *exponent*

If the number is negative, it will have a minus sign; positive numbers have no sign. The first digit of the number is followed by a decimal point. Six decimal places are shown following the decimal point. The letter **E** follows the decimal digits, and marks the start of a three-digit signed *exponent.*

The command displays at least one value. If a *range* is specified, all values in the range are displayed.

■ Example

```
>DL LPI
5E68:0200  11 2D 44 54 FB 21 09 40  3.141593E+000
>
```

The above example displays the long-real floating-point number at the address of the variable `LPI`. Only one value per line is displayed.

6.3.10 Dump 10-Byte Reals

■ Syntax

DT [*address* | *range*]

The Dump 10-Byte Reals command displays the hexadecimal and decimal values of the 10-byte floating-point numbers at the specified *address* or in the specified *range* of addresses.

The command displays one or more lines, depending on the address or range specified. Each line displays the address of the floating-point number in the first column. Next, the hexadecimal values of the bytes in the number are shown, followed by the decimal value of the number. The hexadecimal values are separated by spaces.

The decimal value has the following form:

[–]*digit.digits***E**{+ | –} *exponent*

If the number is negative, it will have a minus sign; positive numbers have no sign. The first digit of the number is followed by a decimal point. Six decimal places are shown following the decimal point. The letter **E** follows the decimal digits, and marks the start of a three-digit signed *exponent.*

The command displays at least one value. If a *range* is specified, all values in the range are displayed.

■ Example

```
>DT TPI
5E68:0300  DE 87 68 21 A2 DA 0F C9 00 40  3.141593E+000
>
```

The above example displays the 10-byte-real floating-point number at the address of the variable TPI. Only one number per line is displayed.

6.4 Compare Memory Command

The Compare Memory command provides a convenient way for comparing two blocks of memory, specified by absolute addresses. This command is primarily of interest to programmers using assembly mode; however, it can be useful to anyone who wants to efficiently compare two large areas of data, such as arrays.

■ Mouse

The Compare Memory command cannot be executed with the mouse.

■ Keyboard

The Compare Memory command cannot be executed with a keyboard command.

■ Dialog

To compare two blocks of memory, enter a command line with the following syntax:

C *range address*

The bytes in the memory locations specified by *range* are compared to the corresponding bytes in the memory locations beginning at *address*. If one or more corresponding bytes do not match, each pair of mismatched bytes is displayed.

■ **Examples**

```
>C 100,01FF 300      ;* hexidecimal radix assumed
39BB:0102 0A 00 39BB:0302
39BB:0108 0A 01 39BB:0308
>
```

The first example (in which hexidecimal is assumed to be the default radix), compares the block of memory from 100 to 1FF to the block of memory from 300 to 3FF. It indicates that the second and eigth bytes differ in the two areas of memory.

```
>C arr1(1) L 100  arr2(1)  ;* BASIC/FORTRAN notation used
>
```

The second example compares the 100 bytes starting at the address of `arr1(1)`, to the 100 bytes starting at address of `arr2(1)`. The CodeView debugger produces no output in response, so this indicates that the first 100 bytes of each array are identical. (This example would be entered as `C arr1[0] L 100 arr2[0]` in C.)

Note

You *can* enter the Compare Memory command using any radix you like; however, any output will still be in hexidecimal format.

6.5 Search Memory Command

The Search Memory command (not to be confused with the Search command discussed in Chapter 11), scans a specified area of memory, looking for specific byte values. It is primarily of interest to programmers using assembly mode, and users who want to test for the presence of specific values within a range of data.

■ Mouse

The Search Memory command cannot be executed with the mouse.

■ Keyboard

The Search Memory command cannot be executed with a keyboard command.

■ Dialog

To search a block of memory, enter the Search Memory command with the following syntax:

S *range list*

The debugger will search the specified *range* of memory locations for the byte values specified in the *list*. If bytes with the specified values are found, then the debugger displays the addresses of each occurrence of bytes in the list.

The *list* can have any number of bytes. Each byte value must be separated by a space or comma, unless the list is an ASCII string. If the list contains more than one byte, then the Search Memory command looks for a series of bytes that precisely match the order and value of bytes in *list*. If found, then the beginning address of each such series is displayed.

■ Examples

```
>S buffer 1 1500 "error"
2BBA:0404
2BBA:05E3
2BBA:0604
>
```

The first example displays the address of each memory location containing the string `error`. The command searches the first 1500 bytes at the address specified by `buffer`. The string was found at the three addresses displayed by the CodeView debugger.

```
>S DS:100 200 0A    ;* hexidecimal radix assumed
3CBA:0132
3CBA:01C2
```

```
>
```

The second example displays the address of each memory location in the range DS:0100 to DS:0200 (hexidecimal), which contains the byte value 0A. The value was found at two addresses.

6.6 Port Input Command

The Port Input Command read and displays a byte from a specified hardware port. It is primarily of interest to assembly-language programmers writing hardware-specific programs.

■ Mouse

The Port Input Command cannot be executed with the mouse.

■ Keyboard

The Port Input Command cannot be executed with a keyboard command.

I *port*

The byte is read and displayed from the specified *port*, which can be any 16-bit address.

■ Examples

```
>I 2f8   ;* hexidecimal radix assumed
E8
>
```

The preceding example reads input port number 2F8 and displays the result, E8. You may enter the port address using any radix you want, but the result will always be displayed in hexidecimal.

6.7 Register Command

The Register command has two functions. It displays the contents of the central processing unit (CPU) registers. It can also change the values of the registers. The display features of the Register command are explained here. The modification features of the command are explained in Chapter 11, "Modifying Code or Data."

■ Mouse

To display the registers with the mouse, point to View on the menu bar, press a mouse button and drag the highlight down to the Registers selection, then release the button. The register window will appear on the right side of the screen. If the register window is already on the screen, the same command removes it.

■ Keyboard

To display the registers using a keyboard command in window mode, press the F2 key. The register window will appear on the right side of the screen. If the register window is already on the screen, the same command will remove it.

In sequential mode, the F2 key will display the current status of the registers. (This produces the same effect as entering the Register dialog command with no argument.)

■ Dialog

To display the registers in the dialog window (or sequentially in sequential mode), enter a command line with the following syntax:

R

The current values of all registers and flags are displayed (in the dialog window in window mode). The instruction at the address pointed to by the current **CS** and **IP** register values is also shown. (The Register command can also be given with arguments, but only when used to modify registers, as explained in Chapter 11, "Modifying Code or Data.")

If the display mode is source (S+) or mixed (S&) (see Chapter 10, "Examining Code," for more information), the current source line is also displayed by the Register command. If an operand of the instruction contains memory expressions or immediate data, the CodeView debugger will evaluate operands and show the value to the right of the instruction. If the CS and IP registers are currently at a breakpoint location, the register display will indicate the breakpoint number.

In sequential mode, the Trace (T), Program Step (P), and Go (G) commands show registers in the same format as the Register command.

■ Examples

```
>S&
mixed
>R
AX=0005  BX=299E  CX=0000  DX=0000  SP=3800  BP=380E  SI=0070  DI=40D1
DS=5067  ES=5067  SS=5067  CS=4684  IP=014F  NV UP EI PL NZ NA PO NC
35:                        VARIAN = (N*SUMXSQ-SUMX**2)/(N-1)
4684:014F 8B5E06        MOV     BX,Word Ptr [BP+06]     ;BR1  SS:3814=299E
>
```

The above example displays all register and flag values, as well as the instruction at the address pointed to by the CS and IP registers. Since the mode has been set to mixed (S&), the current source line is also shown. The example is from a FORTRAN program, but applies equally well to BASIC and C programs.

```
>S-
assembly
>R
AX=0005  BX=299E  CX=0000  DX=0000  SP=3800  BP=380E  SI=0070  DI=40D1
DS=5067  ES=5067  SS=5067  CS=4684  IP=014F  NV UP EI PL NZ NA PO NC
4684:014F 8B5E06        MOV     BX,Word Ptr [BP+06]     ;BR1  SS:3814=299E
>
```

In the above example, the display mode is set to assembly (S–), so no source line is shown. Note the breakpoint number at the right of the last line, indicating that the current address is at Breakpoint 1.

6.8 8087 Command

The 8087 command dumps the contents of the 8087 registers. If you do not have an 8087 or 80287 coprocessor chip on your system, then this command will dump the contents of the psuedo-registers created by the compiler's emulator routines. This command is useful only if you have an 8087 chip installed, or if your executable file includes math routines from a Microsoft 8087-emulation library.

Note

This section does not attempt to explain how the registers of the Intel® 8087 and 80287 processors are organized or how they work. In order to interpret the command output, you must learn about the chip from an Intel reference manual or other book on the subject. Since the Microsoft emulator routines mimic the behavior of the 8087 coprocessor, these references will apply to emulator routines as well as the chips themselves.

■ Mouse

The 8087 command cannot be executed with the mouse.

■ Keyboard

The 8087 command cannot be executed with a keyboard command.

■ Dialog

To display the status of the 8087 or 80287 chip (or floating-point emulator) using a dialog command, enter a command line with the following syntax:

7

The current status of the chip is output when you enter the command. In window mode, the output is to the dialog window. If you do not have an 8087 or 80287 chip, and are not linking to an emulator library, all

registers in the output will contain 0.

■ Example

```
>7
Control 037F  (Projective closure, Round nearest, 64-bit precision)
                    iem=0 pm=1 um=1 om=1 zm=1 dm=1 im=1
Status  6004  cond=1000 top=4 pe=0 ue=0 oe=0 ze=1 de=0 ie=0
Tag     A1FF  instruction=59380  operand=59360  opcode=D9EE
Stack         Exp  Mantissa           Value
ST(3) special 7FFF 8000000000000000 = + Infinity
ST(2) special 7FFF 0101010101010101 = + Not a Number
ST(1) valid   4000 C90FDAA22168C235 = +3.141592265110390E+000
ST(0) zero    0000 0000000000000000 = +0.000000000000000E+000
>
```

In the example above, the first line of the dump shows the current closure method, rounding method, and precision. The number 037F is the hexadecimal value in the control register. The rest of the line interprets the bits of the number. The closure method can be either projective (as in the example) or affine. The rounding method can be either rounding to the nearest even number (as in the example), rounding down, rounding up, or using the chop method of rounding (truncating toward zero). The precision may be 64 bits (as in the example), 53 bits, or 24 bits.

The second line of the display indicates whether each exception mask bit is set or cleared. The masks are interrupt-enable mask (iem), precision mask (pm), underflow mask (um), overflow mask (om), zero-divide mask (zm), denormalized-operand mask (dm), and invalid-operation mask (im).

The third line of the display shows the hexadecimal value of the status register (6004 in the example), then interprets the bits of the register. The condition code (cond) in the example is the binary number 1000. The top of the stack (top) is Register 4 (shown in decimal). The other bits shown are precision exception (pe), underflow exception (ue), overflow exception (oe), zero-divide exception (ze), denormalized-operand exception (de), and invalid-operation exception (ie).

The fourth line of the display first shows the hexadecimal value of the tag register (A1FF in the example). It then gives the hexadecimal values of the instruction (59380), the operand (59360), and the operation code, or opcode, (D9EE).

The fifth line is a heading for the subsequent lines, which contain the contents of each 8087 or 80287 stack register. The registers in the example contain four types of numbers that may be held in these registers. Starting from the bottom, Register 0 contains zero. Register 1 contains a valid real

number. Its exponent (in hexadecimal) is 4000 and its mantissa is C90FDAA22168C235. The number is shown in scientific notation in the rightmost column. Register 2 contains a value that cannot be interpreted as a number, and Register 3 contains infinity.

Note

If you are using the floating-point emulator routines instead of the 8087 or 80287 chip, then the display of the stack registers will be altered slightly: the form ST(x) will be replaced with STe(x). In other words, the display for register 0 would begin with STe(0) instead of ST(0). If you see a mixture of ST and STe registers, then the emulator is being used as an overflow mechanism to the coprocessor.

Chapter 7
Managing Breakpoints

7.1 Breakpoint Set Command 169
7.2 Breakpoint Clear Command 172
7.3 Breakpoint Disable Command 174
7.4 Breakpoint Enable Command 175
7.5 Breakpoint List Command 176

The CodeView debugger enables you to control program execution by setting breakpoints. A breakpoint is an address that stops program execution each time the address is encountered. By setting breakpoints at key addresses in your program, you can "freeze" program execution and examine the status of memory or expressions at that point.

The commands listed below control breakpoints:

Command	Action
Breakpoint Set (**BP**)	Sets a breakpoint and, optionally, a pass count and break commands
Breakpoint Clear (**BC**)	Clears one or more breakpoints
Breakpoint Disable (**BD**)	Disables one or more breakpoints
Breakpoint Enable (**BE**)	Enables one or more breakpoints
Breakpoint List (**BL**)	Lists all breakpoints

In addition to these commands, the Watchpoint (**WP**) and Tracepoint (**TP**) commands can be used to set conditional breakpoints. These commands are explained in Chapter 9, "Managing Watch Statements." The Breakpoint commands are discussed in Sections 7.1–7.5.

7.1 Breakpoint Set Command

The Breakpoint Set command creates a breakpoint at a specified address. Any time a breakpoint is encountered during program execution, the program halts and waits for a new command.

The CodeView debugger allows up to 20 breakpoints (0 through 19). Each new breakpoint is assigned the next available number. Breakpoints remain in memory until you delete them (see Section 7.2 for more information) or until you quit the debugger. They are not canceled when you restart the program. This enables you to set up a complicated series of breakpoints, then execute through the program several times without resetting the breakpoints.

If you try to set a breakpoint at a comment line or other source line that does not correspond to code, the CodeView debugger displays the following message:

```
No code at this line number
```

■ Mouse

To set a breakpoint with the mouse, point to the source line or instruction where you want to set the breakpoint, then click the left button. The line will be displayed in high-intensity text, and will remain so until you remove or disable the breakpoint.

■ Keyboard

To set a breakpoint with a keyboard command in window mode, move the cursor to the source line or instruction where you want to set a breakpoint. You may have to press the F6 key to move the cursor to the display window. When the cursor is on the appropriate source line, press the F9 key. The line will be displayed in high-intensity text, and will remain so until you remove the breakpoint.

In sequential mode, the F9 key can be used to set a breakpoint at the current location. You must use the dialog version of the command to set a breakpoint at any other location.

■ Dialog

To set a breakpoint using a dialog command, enter a command line with the following syntax:

BP [[*address* [[*passcount*]] [["*commands*"]]]]

If no *address* is given, a breakpoint is created on the current source line in source mode, or on the current instruction in assembly mode. You can specify the *address* in the *segment*:*offset* format or as a source line, a routine name, or a label. If you give an offset address, the code segment is assumed.

The dialog version of the command is more powerful than the mouse or keyboard version in that it allows you to give a *passcount* and a string of

commands. The *passcount* specifies the first time the breakpoint is to be taken. For example, if the pass count is 5, the breakpoint will be ignored the first four times it is encountered, and taken the fifth time. Thereafter, the breakpoint is always taken.

The *commands* are a list of dialog commands enclosed in quotation marks (" ") and separated by semicolons (;). For example, if you specify the commands as "`? code;T`", the CodeView debugger will automatically display the value of the variable `code` and then execute the Trace command each time the breakpoint is encountered. The Trace and Display Expression commands are described in Chapter 7, "Examining Data and Expressions," and Chapter 6, "Executing Code," respectively.

In window mode, a breakpoint entered with a dialog command has exactly the same effect as one created with a window command. The source line or instruction corresponding to the breakpoint location is shown in high-intensity text.

In sequential mode, information about the current instruction will be displayed each time you execute to a breakpoint. The register values, the current instruction, and the source line may be shown, depending on the display mode. See Chapter 10, "Examining Code," for more information about display modes.

When a breakpoint address is shown in the assembly-language format, the breakpoint number will be shown as a comment to the right of the instruction. This comment appears even if the breakpoint is disabled (but not if it is deleted).

■ Examples

```
>BP .19 10
>
```

The above example creates a breakpoint at Line 19 of the current source file (or if there is no executable statement at Line 19, at the first executable statement after Line 19). The breakpoint is passed over 9 times before being taken on the 10th pass.

```
>BP STATS 10 "?COUNTER = COUNTER + 1;G"
>
```

The above example creates a breakpoint at the address of the routine

STATS. The breakpoint is passed over 9 times before being taken on the 10th pass. Each time execution stops for the breakpoint, the quoted commands are executed. The Display Expression command increments COUNTER, then the Go command restarts execution. If COUNTER is set to 0 when the breakpoint is set, this has the effect of counting the number of times the breakpoint is taken.

```
>S-       ;* FORTRAN example - uses FORTRAN hexidecimal notation
assembly
>BP #0a94
>G
AX=0006  BX=304A  CX=000B  DX=465D  SP=3050  BP=3050  SI=00BB  DI=40D1
DS=5064  ES=5064  SS=5064  CS=46A2  IP=0A94  NV UP EI PL NZ NA PE NC
46A2:0A94 7205          JB      __chkstk+13 (0A9B)      ;BR1
>
```

The above example first sets the mode to assembly, then creates a breakpoint at the hexidecimal (offset) address #0A94 in the default (CS) segment. (The same address would be specified as 0x0A94 with the C expression evaluator, and as &H0A94 with the BASIC expression evaluator.) The Go command (G) is then used to execute to the breakpoint. Note that in the output to the Go command, the breakpoint number is shown as an assembly-language comment (;BR1) to the right of the current instruction. The Go command displays this output only in sequential mode; in window mode no assembly-language information appears.

7.2 Breakpoint Clear Command

The Breakpoint Clear command permanently removes one or more previously set breakpoints.

■ Mouse

To clear a single breakpoint with the mouse, point to the breakpoint line or instruction you want to clear. Breakpoint lines are shown in high-intensity text. Press the left mouse button. The line will be shown in normal text to indicate that the breakpoint has been removed.

To remove all breakpoints with the mouse, point to Run on the menu bar, press a mouse button and drag the highlight down to the Clear Breakpoints selection, then release the button.

■ Keyboard

To clear a single breakpoint with a keyboard command, move the cursor to the breakpoint line or instruction you want to clear. Breakpoint lines are shown in high-intensity text. Press the F9 key. The line will be shown in normal text to indicate that the breakpoint has been removed.

To remove all breakpoints using a keyboard command, press ALT-R to open the Run menu, then press ALT-C to select Clear Breakpoints.

■ Dialog

To clear breakpoints using a dialog command, enter a command line with the following syntax:

BC *list*
BC *

If *list* is specified, the command removes the breakpoints named in the list. The *list* can be any combination of integer values from 0 to 19. You can use the Breakpoint List command (**BL**) if you need to see the numbers for each existing breakpoint. If an asterisk (*) is given as the argument, all breakpoints are removed.

■ Examples

```
>BC 0 4 8
>
```

The above example removes breakpoints 0, 4, and 8.

```
>BC *
>
```

The above example removes all breakpoints.

7.3 Breakpoint Disable Command

The Breakpoint Disable command temporarily disables one or more existing breakpoints. The breakpoints are not deleted. They can be restored at any time using the Breakpoint Enable command (**BE**).

When a breakpoint is disabled in window mode, it is shown in the display window with normal text. When it is enabled, it is shown in high-intensity text.

Note

All disabled breakpoints are automatically enabled whenever you restart the program being debugged. The program can be restarted with the Start or Restart selection from the Run menu, or with the Restart dialog command (**L**). See Chapter 5, "Executing Code."

■ Mouse

The Breakpoint Disable command cannot be executed with the mouse.

■ Keyboard

The Breakpoint Disable command cannot be executed with a keyboard command.

■ Dialog

To disable breakpoints using a dialog command, enter a command line with the following syntax:

BD *list*
BD *

If *list* is specified, the command disables the breakpoints named in the list. The *list* can be any combination of integer values from 0 to 19. Use the Breakpoint List command (**BL**) if you need to see the numbers for each existing breakpoint. If an asterisk (*) is given as the argument, all

breakpoints are disabled.

The window commands for setting and clearing breakpoints can also be used to enable or clear disabled breakpoints.

■ Examples

```
>BD 0 4 8
>
```

The above example disables breakpoints 0, 4, and 8.

```
>BD *
>
```

The above example disables all breakpoints.

7.4 Breakpoint Enable Command

The Breakpoint Enable command enables breakpoints that have been temporarily disabled with the Breakpoint Disable command.

■ Mouse

To enable a disabled breakpoint with the mouse, point to the source line or instruction of the breakpoint, then click the left button. The line will be displayed in high-intensity text, and will remain so until you remove or disable the breakpoint. This is the same as creating a new breakpoint at that location.

■ Keyboard

To enable a disabled breakpoint using a keyboard command, move the cursor to the source line or instruction of the breakpoint, then press the F9 key. The line will be displayed in high-intensity text, and will remain so until you remove or disable the breakpoint. This is the same as creating a new breakpoint at that location.

■ Dialog

To enable breakpoints using a dialog command, enter a command line with the following syntax:

BE *list*
BE *

If *list* is specified, the command enables the breakpoints named in the list. The *list* can be any combination of integer values from 0 to 19. Use the Breakpoint List command (BL) if you need to see the numbers for each existing breakpoint. If an asterisk (*) is given as the argument, all breakpoints are enabled. The CodeView debugger ignores all or part of the command if you try to enable a breakpoint that is not disabled.

■ Examples

```
>BE 0 4 8
>
```

The above example enables breakpoints 0, 4, and 8.

```
>BE*
>
```

The above example enables all disabled breakpoints.

7.5 Breakpoint List Command

The Breakpoint List command lists current information about all breakpoints.

■ Mouse

The Breakpoint List command cannot be executed with the mouse.

■ Keyboard

The Breakpoint List command cannot be executed with a keyboard command.

■ Dialog

To list breakpoints using a dialog command, enter a command line with the following syntax:

BL

The command displays the breakpoint number, the enabled status, the address, the routine, and the line number. If the breakpoint does not fall on a line number, an offset is shown from the nearest previous line number. The pass count and break commands are shown if they have been set. The status can be e for enabled, d for disabled. If no breakpoints are currently defined, nothing is displayed.

■ Example

```
>BL
0 e 56C4:0105  _ARCTAN:10
1 d 56C4:011E  _ARCTAN:19                (pass = 10) "T;T"
2 e 56C4:00FD  _ARCTAN:9+6
>
```

In the above example, Breakpoint 0 is enabled at address 56C4:0105. This address is in routine ARCTAN and is at line 10 of the current source file. No pass count or break commands have been set.

Breakpoint 1 is currently disabled, as indicated by the d after the breakpoint number. It also has a pass count of 10, meaning that the breakpoint will not be taken until the 10th time it is encountered. The command string at the end of the line indicates that each time the breakpoint is taken, the Trace command will automatically be executed twice.

The line number for Breakpoint 2 has an offset. The address is #6 bytes beyond the address for line number 9 in the current source file. This indicates that the breakpoint was probably set in assembly mode, since it would be difficult to set a breakpoint anywhere except on a source line in source mode.

Chapter 8
Managing Watch Statements

8.1 Setting Watch-Expression and Watch-Memory Statements 182
8.2 Setting Watchpoints 186
8.3 Setting Tracepoints 191
8.4 Deleting Watch Statements 196
8.5 Listing Watchpoints and Tracepoints 197
8.6 C Examples 199
8.7 FORTRAN Examples 200

Watch Statement commands are among the Microsoft CodeView debugger's most powerful features. They enable you to set, delete, and list watch statements. Watch statements are specifications that describe expressions or areas of memory to watch. Some watch statements also specify conditional breakpoints that may or may not be taken, depending on the value of the expression or memory area.

Note

Syntax for each CodeView command is always the same, regardless of the expression evaluator; however, the method for specifying an *argument* may vary with the language. Therefore, each example in this chapter is repeated, with C, FORTRAN, and BASIC arguments. The sample screens that present these examples feature BASIC. At the end of this chapter are a C sample screen and a FORTRAN sample screen that each incorporate all the examples (except for Watch Delete and Watch List).

The Watch Statement commands are summarized below:

Command	Action
Watch (**W**)	Sets an expression or range of memory to be watched
Watchpoint (**WP**)	Sets a conditional breakpoint that will be taken when the expression becomes nonzero (true)
Tracepoint (**TP**)	Sets a conditional breakpoint that will be taken when a given expression or range of memory changes
Watch Delete (**Y**)	Deletes one or more watch statements
Watch List (**W**)	Lists current watch statements

Watch statements are like breakpoints in that they remain in memory until you specifically remove them or quit the CodeView debugger. They are not canceled when you restart the program being debugged. This enables you to set a complicated series of watch statements, then execute through the program several times without resetting the watch statements.

In window mode, Watch Statement commands can be entered either in the dialog window or with menu selections. Current watch statements are shown in a watch window that appears between the menu bar and the source window.

In sequential mode, the Watch, Tracepoint, and Watchpoint commands can be used, but since there is no watch window, you cannot see the watch statements and their values. You must use the Watch List command to examine the current watch statements.

Note

In order to set a watch statement containing a local variable, you must be in the function where the variable is defined. If the current line is not in the function, the CodeView debugger displays the message `UNKNOWN SYMBOL`. When you exit from a function containing a local variable referenced in a watch statement, the value of the statement is displayed as `UNKNOWN SYMBOL`. When you reenter the function, the local variable will again have a value. With the C and FORTRAN expression evaluators, you can avoid this limitation by using the period operator to specify both the function and the variable. For example, enter `main.x` instead of just `x`.

8.1 Setting Watch-Expression and Watch-Memory Statements

The Watch command is used to set a watch statement that describes an expression or a range of addresses in memory. The value or values described by this watch statement are shown in the watch window. The watch window is updated to show new values each time the value of the watch statement changes during program execution. Since the watch window does not exist in sequential mode, you must use the Watch List command to examine the values of watch statements.

When setting a watch expression, you can specify the format in which the value will be displayed. Type the expression followed by a comma and a type specifier. If you do not give a type specifier, the CodeView debugger displays the value in a default format. See Chapter 7, "Examining Data and Expressions," for more information about type specifiers and the

default format.

Note

If your program directly accesses absolute addresses used by IBM or IBM-compatible computers, you may sometimes get unexpected results with the Display Expression and Dump commands. However, the Watch command will usually show the correct values. This problem can arise if the CodeView debugger and your program try to use the same memory location.

This often occurs when a program reads data directly from the screen buffer of the display adapter. If you have an array called `screen` that is initialized to the starting address of the screen buffer, the command `DB screen L 16` will display data from the CodeView display rather than from the display of the program you are debugging. The command `WB screen L 16` will display data from the program's display (provided screen swapping or screen flipping was specified at start-up). This happens because watch-statement values are updated during program execution, and any values read from the screen buffer will be taken from the output screen rather than from the debugging screen.

■ Mouse

To set a watch-expression statement using the mouse, point to Watch on the menu bar, press a mouse button and drag the highlight down to the Add Watch selection, then release the button. A dialog box appears, asking for the expression to be watched. Type the expression and press the ENTER key or a mouse button.

You cannot use the mouse version of the command to specify a range of memory to be watched, as you can with the dialog version.

■ Keyboard

To set a watch-expression statement with a keyboard command, press ALT-W to open the Watch menu, then press ALT-A to select Add Watch. A dialog box appears, asking for the expression to be watched. Type the expression and press the ENTER key.

You cannot use the keyboard version of the command to specify a range of memory to be watched, as you can with the dialog version.

■ Dialog

To set a watch-expression statement or watch-memory statement using a dialog command, enter a command line with the following syntax:

W? *expression*[[,*format*]]	Watch expression
W[[*type*]] *range*	Watch memory

An *expression* used with the Watch command can be a simple variable, or a complex expression using several variables and operators. The expression should be no longer than the width of the watch window. You can specify *format* using a C **printf** type specifier. See Chapter 7, "Examining Data and Expressions," for more information.

When watching a memory location, *type* is a one-letter size specifier from the following list:

Specifier	Size
None	Default type
B	Byte
A	ASCII
I	Integer (signed decimal word)
U	Unsigned (unsigned decimal word)
W	Word
D	Double word
S	Short real
L	Long real
T	10-byte real

The default type used if no type size is specified is the last type used by a Dump, Enter, Watch Memory, or Tracepoint Memory command. If none of these commands has been used during the session, the default type is byte.

The data will be displayed in a format similar to that used by the Dump commands (see Chapter 7, "Examining Data and Expressions," for more information). The *range* can be any length, but only one line of data will be displayed in the watch window. If you do not specify an ending address for the range, the default range is one object.

■ Examples

The following commands display three watch statements in the watch window:

```
W? n
```

The above example displays the current value of the variable `n`.

```
W? higher * 100
```

The above example displays the value of the expression `higher * 100`.

```
WB arr(1) L 8     ;* BASIC/FORTRAN example
WB arr[0] L 8     ;* C example
```

The above examples each display the first 8 bytes at the address of the first element of the array `arr`. Though the two examples have the same function, they are entered differently because C uses a different syntax for indexing arrays.

Note

The examples in this chapter are based on C programs that use lowercase variables, and FORTRAN and BASIC programs that use uppercase variables. However, these examples will work perfectly well for FORTRAN and BASIC, as long as Case Sense is turned off. Otherwise, the FORTRAN and BASIC programs will work with uppercase variables only.

These commands, entered while debugging a BASIC program, produce the watch window in Figure 8.1. (Corresponding C and FORTRAN examples are included with other commands, at the end of the chapter.)

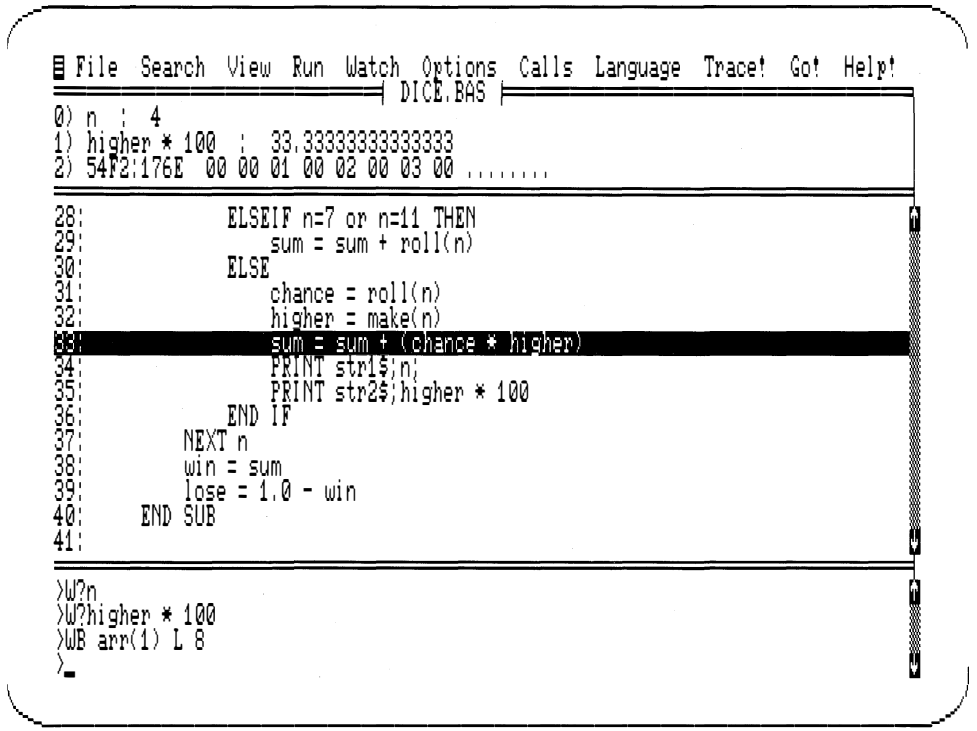


Figure 8.1 Watch Statements in the Watch Window

8.2 Setting Watchpoints

The Watchpoint command is used to set a conditional breakpoint called a watchpoint. A watchpoint breaks program execution when the expression described by its watch statement becomes true. You can think of watchpoints as "break when" points, since the break occurs when the specified expression becomes true (nonzero).

A watch statement created by the Watchpoint command describes the expression that will be watched and compared to 0. The statement remains in memory until you delete it or quit the CodeView debugger. Any valid CodeView expression can be used as the watchpoint expression as long as the expression is not wider than the watch window.

In window mode, watchpoint statements and their values are displayed in high-intensity text in the watch window. In sequential mode, there is no watch window, so the values of watchpoint statements can only be displayed with the Watch List command (see Section 8.5 for more information).

Although watchpoints can be any valid CodeView expression, the command works best with expressions that use the relational operators (such as < and > for C and BASIC, or `.LT.` and `.GT.` for FORTRAN). Relational expressions always evaluate to false (zero) or true (nonzero). Care must be taken with other kinds of expressions when used as watchpoints, because they will break execution whenever they do not equal precisely zero. For example, your program might use a loop variable `I`, which ranges from 1 to 100. If you entered `I` as a watchpoint, then it would *always* break execution, since `I` is never equal to 0. However, the relational expression `I>90` (or `I.GT.90`) would not break execution until `I` exceeded 90.

Note

An example of the problems involved with using nonrelational expressions occurs if you accidentally type the assignment operator when you mean to use the equality operator. Using the simple assignment operator is legal, but it has unexpected side effects. For example, the following expressions both set the value of `count` to 6:

```
count=6       (in C or FORTRAN)
LET count=6   (in BASIC)
```

Furthermore, both expressions evaluate to nonzero, so the break is always taken.

The expression `count=6` successfully tests for equality in BASIC. Note that the **LET** keyword is maintained in the BASIC expression evaluator, to preserve the distinction between assignment and test for equality.

■ Mouse

To set a watchpoint statement with the mouse, point to Watch on the menu bar, press a mouse button and drag the highlight down to the Watchpoint selection, then release the button. A dialog box appears, asking for the expression to be watched. Type the expression and press the ENTER key or a mouse button.

■ Keyboard

To execute the Watchpoint command with a keyboard command, press ALT-W to open the Watch menu, then press ALT-W to select Watchpoint. A dialog box appears, asking for the expression to be watched. Type the expression and press the ENTER key.

■ Dialog

To set a watchpoint using a dialog command, enter a command line with the following syntax:

WP? *expression*[[,*format*]]

The *expression* can be any valid CodeView expression (usually a relational expression). You can enter *format* as a C **printf** type specifier, but there is little reason to do so, since the expression value is normally either 1 or 0.

■ Examples

The following dialog commands display two watch statements (watchpoints) in the watch window:

```
WP? higher > chance       ;* BASIC/C example
WP? higher .gt. chance    ;* FORTRAN example
```

The above examples instruct the CodeView debugger to break execution when the variable `higher` is greater than the variable `chance`. (Note that BASIC and C happen to use the same syntax in this case, but FORTRAN uses its own.) After setting this watchpoint, you could use the Go command to execute until the condition becomes true.

```
WP? n=7 or n=11           ;* BASIC example
WP? n==7 || n==11         ;* C example
WP? n.eq.7 .or. n.eq.11   ;* FORTRAN example
```

The above examples instruct the CodeView debugger to break execution when the variable `n` is equal to 7 or 11.

Note

BASIC and C will each display a numerical result in response to a boolean expression (0 being equivalent to false, nonzero to true) However, corresponding FORTRAN condition will be displayed with either **.TRUE.** or **.FALSE.** in the watch window.

These commands, entered while debugging a BASIC program, produce the watch window in Figure 8.2. (Corresponding C and FORTRAN examples are included with other commands, at the end of the chapter.)

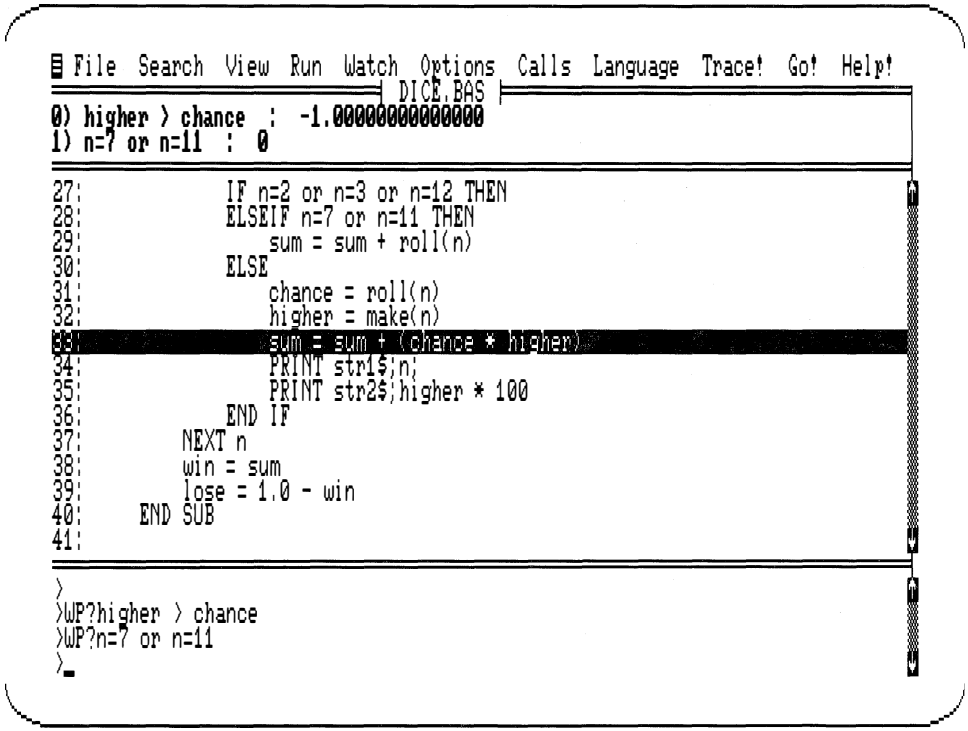


Figure 8.2 Watchpoints in the Watch Window

Note

Setting watchpoints significantly slows execution of the program being debugged. The CodeView debugger has to check to see if the expression is true each time a source line is executed in source mode, or each time an instruction is executed in assembly mode. Be careful when setting watchpoints near large or nested loops. A loop that executes almost instantly when run from MS-DOS can take many minutes if executed from within the debugger with several watchpoints set.

Tracepoints do not slow CodeView execution as much as watchpoints, so you should use tracepoints when possible. For example, although you can set a watchpoint on a Boolean variable (`WP? moving`), a tracepoint on the same variable (`TP? moving`) has essentially the same effect and does not slow execution as much.

If you enter a seemingly endless loop, press CONTROL-BREAK or CONTROL-C to exit. You will soon learn the size of loop you can safely execute when watchpoints are set.

8.3 Setting Tracepoints

The Tracepoint command is used to set a conditional breakpoint called a tracepoint. A tracepoint breaks program execution when there is a change in the value of a specified expression or range of memory.

The watch statement created by the Tracepoint command describes the expression or memory range to be watched and tested for change. The statement remains in memory until you delete it or quit the CodeView debugger.

In window mode, tracepoint statements and their values are shown in high-intensity text in the watch window. In sequential mode, there is no watch window, so the values of tracepoint statements can only be displayed with the Watch List command (see Section 8.5, "Listing Watchpoints and Tracepoints," for more information).

An expression used with the Tracepoint command must evaluate to an "lvalue". In other words, the expression must refer to an area of memory not more than 128 bytes in size. For example, `i==10` (which is similar to `I.EQ.10` in FORTRAN and `I=10` in BASIC) would be invalid because it is either 1 (true) or 0 (false) rather than a value stored in memory. The expression `sym1+sym2` is invalid because it is the calculated sum of the value of two memory locations. The expression `buffer` would be invalid if `buffer` is an array of 130 bytes, but valid if the array is 120 bytes. Note that if buffer is declared as an array of 64 bytes, then the Tracepoint command given with the expression `buffer` checks all 64 bytes of the array. The same command given with the C expression `buffer[32]`, or `BUFFER(33)` in FORTRAN or BASIC, means that only one byte (the 33rd) will be checked.

Note

The following is relevant only to C programs.

Register variables are not considered lvalues. Therefore, if `i` is declared as `register int i`, the command `TP? i` is invalid. However, you can still check for changes in the value of `i`. Use the Examine Symbols command to learn which register contains the value of `i`. Then learn the value of `i`. Finally, set up a watchpoint to test the value. For example, use the following sequence of commands:

```
>X? i
3A79:0264 int              div()
            SI       int              i
>?i
10
>WP? @SI!=10
>
```

When setting a tracepoint expression, you can specify the format in which the value will be displayed. Type the expression followed by a comma and a type specifier. If you do not give a type specifier, the CodeView debugger displays the value in a default format. See Chapter 7, "Examining Data and Expressions," for more information about type specifiers and the default format.

■ Mouse

To set a tracepoint-expression statement with the mouse, point to Watch on the menu bar, press a mouse button and drag the highlight down to the Tracepoint selection, then release the button. A dialog box appears, asking for the expression to be watched. Type the expression and press the ENTER key or a mouse button.

You cannot specify a range of memory to be watched with the mouse version of the command as you can with the dialog version.

■ Keyboard

To set a tracepoint-expression statement with a keyboard command, press ALT-W to open the Watch menu, then press ALT-T to select Tracepoint. A dialog box appears, asking for the expression to be watched. Type the expression and press the ENTER key.

You cannot use the keyboard version of the command to specify a range of memory to be watched, as you can with the dialog version.

■ Dialog

To set a tracepoint using a dialog command, enter a command line with the following syntax:

TP? *expression*,[[*format*]]	Expression tracepoint
TP[[*type*]] *range*	Memory tracepoint

An *expression* used with the Tracepoint command can be a simple variable or a complex expression using several variables and operators. The expression should not be longer than the width of the watch window. You can specify *format* using a C **printf** type specifier if you do not want the value to be displayed in the default format (decimal for integers or floating point for real numbers). See Section 6.1, "Display Expression Command," for more information.

In the memory-tracepoint form, *range* must be a valid address range and *type* must be a one-letter memory-size specifier. If you specify only the start of the range, the CodeView debugger displays one object as the default.

Although no more than one line of data will be displayed in the watch window, the range to be checked for change can be any size up to 128 bytes. The data will be displayed in the format used by the Dump commands (see Chapter 7, "Examining Data and Expressions," for more information). The valid memory-size specifiers are listed below:

Specifier	Size
None	Default type
B	Byte
A	ASCII
I	Integer (signed decimal word)
U	Unsigned (unsigned decimal word)
W	Word
D	Double word
S	Short real
L	Long real
T	10-byte real

The default type if no type size is specified is the last type used by a Dump, Enter, Watch Memory, or Tracepoint Memory command. If none of these commands has been used during the session, the default type is byte.

■ Examples

The following dialog commands display three watch statements (tracepoints) in the watch window:

```
TP? sum
```

The above example instructs the CodeView debugger to stop whenever the value of the variable `sum` changes.

```
TPB arr(1) L 8     ;* BASIC, FORTRAN example
TPB arr[0] L 8     ;* C example
```

The above examples instruct the debugger to stop whenever any of the first 8 bytes, starting with the address of the first element of `arr`, change in value.

These commands, entered while debugging a BASIC program, produce the watch window in Figure 8.3. (Corresponding C and FORTRAN examples are included with other commands, at the end of the chapter.)

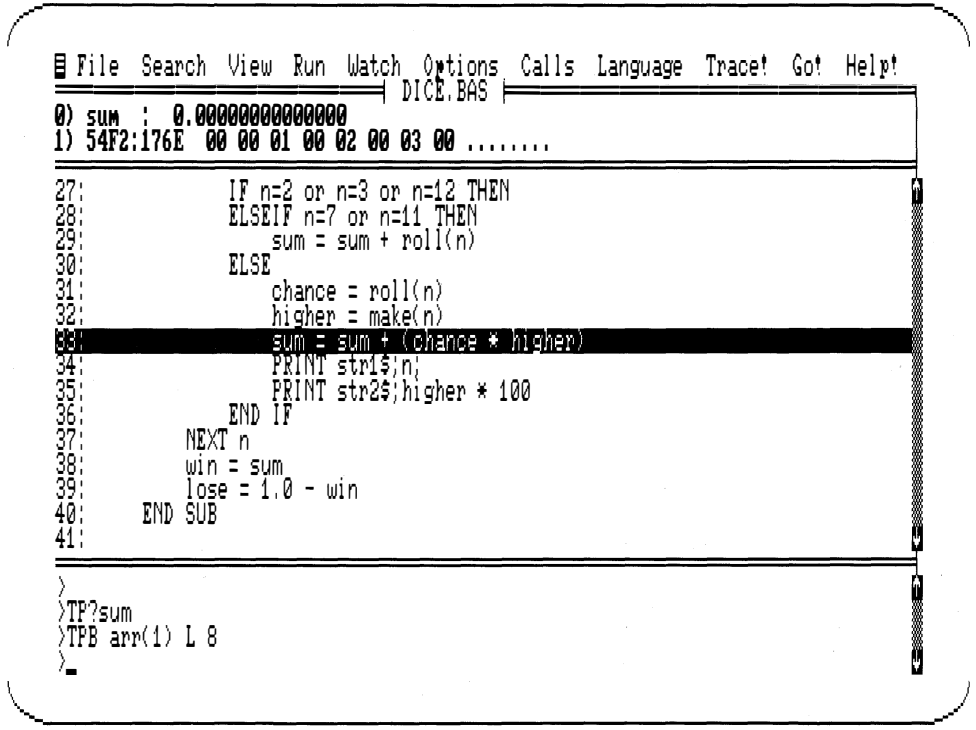


Figure 8.3 Tracepoints in the Watch Window

Note

Setting tracepoints significantly slows execution of the program being debugged. The CodeView debugger has to check to see if the expression or memory range has changed each time a source line is executed in source mode or each time an instruction is executed in assembly mode. However, tracepoints do not slow execution as much as watchpoints. Be careful when setting tracepoints near large or nested loops. A loop that executes almost instantly when run from MS-DOS can take many minutes if executed from within the debugger with several tracepoints set.

If you enter a seemingly endless loop, press CONTROL-BREAK or CONTROL-C to exit. Often you can tell how far you went in the loop by the value of the tracepoint when you exited.

8.4 Deleting Watch Statements

The Watch Delete command enables you to delete watch statements that were set previously with the Watch, Watchpoint, or Tracepoint command.

When you delete a watch statement in window mode, the statement disappears and the watch window closes around it. For example, if there are three watch statements in the window and you delete statement 1, the window is redrawn with one less line. Statement 0 remains unchanged, but statement 2 becomes statement 1. If there is only one statement, the window disappears.

■ Mouse

To delete a watch statement with the mouse, point to Watch on the menu bar, press a mouse button and drag the highlight down to the Delete Watch selection, then release the button. A dialog box appears, containing all the watch statements. Point to the statement you want to delete and press the ENTER key or a mouse button. The dialog box disappears and the watch window is redrawn without the deleted watch statement.

You can also delete all the statements in the watch window at once, simply by selecting the Delete All selection.

■ Keyboard

To execute the Execute command with a keyboard command, press ALT-W to open the Watch menu, then press ALT-D to select Delete Watch. A dialog box appears, containing all the watch statements. Use the UP and DOWN ARROW keys to move the cursor to the statement you want to delete, then press the ENTER key. The dialog box disappears and the watch window is redrawn without the deleted watch statement.

You can also delete all the statements in the watch window at once, simply by selecting the Delete All selection. Do this by pressing ALT-D twice after the Watch menu is open, and then pressing ENTER.

■ Dialog

To delete watch statements using a dialog command, enter a command line with the following syntax:

Y *number*

When you set a watch statement, it is automatically assigned a number (starting with 0). In window mode, the number appears to the left of the watch statement in the watch window. In sequential mode, you can use the Watch List (**W**) command to view the numbers of current watch statements.

You can delete existing watch statements by specifying the *number* of the statement you want to delete with the Delete Watch command. (The **Y** is a mnemonic for Yank.)

You can use the asterisk (*) to represent all watch statements.

■ Examples

```
>Y 2 ;* Example 1
>

>Y * ;* Example 2
>
```

Example 1 deletes watch statement 2. Example 2 deletes all watch statements and closes the watch window.

8.5 Listing Watchpoints and Tracepoints

The Watch List command lists all previously set watchpoints and tracepoints with their assigned numbers and their current values.

This command is the only way to examine current watch statements in sequential mode. The command has little use in window mode, since watch statements are already visible in the watch window.

■ Mouse

The Watch List command cannot be executed with the mouse.

■ Keyboard

The Watch List command cannot be executed with a keyboard command.

■ Dialog

To list watch statements using a dialog command, enter a command line with the following syntax:

W

The display is the same as the display that appears in the watch window in window mode.

Note

The command letter for the Watch List command is the same as the command letter for the memory version of the Watch command when no memory size is given. The difference between the commands is that the Watch List command never takes an argument. The Watch command always requires at least one argument.

■ Example

```
>W
0) code,c  :  I
1) (float)letters/words,f  :  4.777778
2) 3F65:0B20  20 20 43 4F 55 4E 54 COUNT
3) lines==11 :  0
>
```

8.6 C Examples

The seven examples shown previously in a BASIC screen would be entered in a C debugging session as follows:

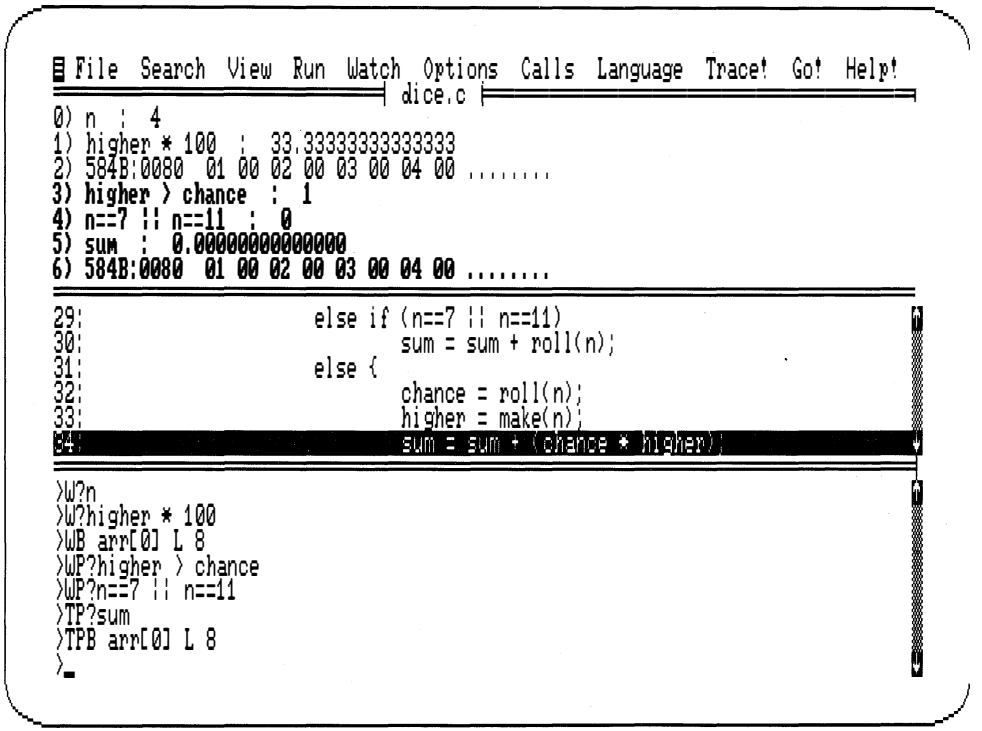


Figure 8.4 C Watch Statements

The first three items in the watch window are simple watch statements. They display values but never cause execution to break.

The next two items are watchpoints; they cause execution to break whenever they evaluate to true (nonzero). The fourth item will break execution whenever `higher` is greater than `chance`, and the fifth item will break execution whenever `n` is equal to 7 or 11.

The last two items are tracepoints, which cause execution to break whenever any bytes within a specified area of memory change. The sixth item breaks execution whenever the value of `sum` changes; the seventh item breaks execution whenever there is a change in any of the 8 bytes in memory starting at the location of `arr[0]`.

8.7 FORTRAN Examples

The seven examples shown previously in a BASIC screen would be entered in a FORTRAN debugging session as follows:

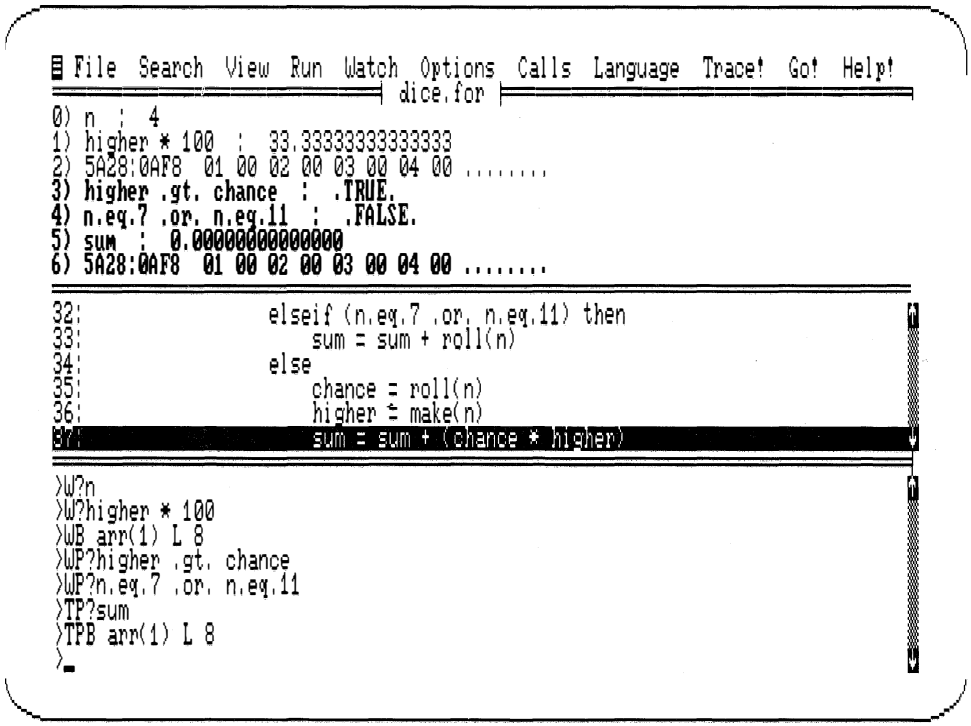


Figure 8.5 FORTRAN Watch Statements

The first three items in the watch window are simple watch statements. They display values but never cause execution to break.

The next two items are watchpoints; they cause execution to break whenever they evaluate to true (nonzero). The fourth item will break execution whenever HIGHER is greater than CHANCE, and the fifth item will break execution whenever N is equal to 7 or 11.

The last two items are tracepoints, which cause execution to break whenever any bytes within a specified area of memory change. The sixth item breaks execution whenever the value of SUM changes; the seventh item breaks execution whenever there is a change in any of the 8 bytes in memory starting at the location of ARR(1).

Chapter 9
Examining Code

9.1 Set Mode Command 203
9.2 Unassemble Command 205
9.3 View Command 208
9.4 Current Location Command 211
9.5 Stack Trace Command 212

Several CodeView commands allow you to examine program code, or data related to code. The following commands are discussed in this chapter:

Command	Action
Set Mode (**S**)	Sets format for code displays
Unassemble (**U**)	Displays assembly instructions
View (**V**)	Displays source lines
Current Location (**.**)	Displays the current location line
Stack Trace (**K**)	Displays routines or procedures

9.1 Set Mode Command

The Set Mode command sets the mode in which code is displayed. The two basic display modes are source mode, in which the program is displayed as source lines, and assembly mode, in which the program is displayed as assembly-language instructions. These two modes can be combined in mixed mode, in which the program is displayed with both source lines and assembly-language instructions.

In sequential mode, there are three display modes: source, assembly, and mixed. These modes affect the output of commands that display code (Register, Trace, Program Step, Go, Execute, and Unassemble).

In window mode, there are two display modes: source and assembly, but there are additional options for mixing source and assembly modes and controlling the display of assembly-language instructions. The display mode affects the way the program is shown in the display window.

Source and mixed modes are only available if the executable file contains symbols in the CodeView format. Programs that do not contain symbolic information (including all **.COM** files) are displayed in assembly mode.

■ Mouse

To set the display mode with the mouse, point to View on the menu bar, press a mouse button and drag the highlight to either the Source selection for source mode, the Mixed selection for mixed mode, or the Assembly selection for assembly mode. Then release the button.

You can further control the display of assembly-language instructions by making selections from the Options menu. See Section 2.1.3.6, "Using the Options Menu," for more information.

■ Keyboard

To change the display mode with a keyboard command, press the F3 key. This will rotate the mode to the next setting; you may need to press F3 twice to get the desired mode. This command works in either window or sequential mode. In sequential mode, the word `source`, `mixed`, or `assembly` is displayed to indicate the new mode.

In window mode you can further control the display of assembly-language instructions by making selections from the Options menu. See Section 2.1.3.6, "Using the Options Menu," for more information.

■ Dialog

To set the display mode from the dialog window, enter a command line with the following syntax:

S[[+ | – | &]]

If the plus sign is specified (S+), source mode is selected, and the word `source` is displayed.

If the minus sign is specified (S–), assembly mode is selected, and the word `assembly` is displayed. In window mode, the display will include any assembly options, except the Mixed Source option, previously toggled on from the Options menu. The Mixed Source option is always turned off by the S– command.

If the ampersand is specified (S&), mixed mode is selected, and the word `mixed` is displayed. In window mode, the display will include any assembly options previously toggled on from the Options menu. In addition, the Mixed Source option will be turned on by the S& command.

If no argument is specified (S), the current mode (`source`, `assembly`, or `mixed`) is displayed.

The Unassemble command in sequential mode is an exception in that it displays mixed source and assembly with both the source (S+) and mixed (S&) modes. When you enter the dialog version of the Set Mode command, the CodeView debugger outputs the name of the new display mode: `source`, `assembly`, or `mixed`.

Note

80286 protected-mode mnemonics cannot be displayed in assembly or mixed mode. They will not be shown in the display window in window mode.

■ Examples

```
>S+
source
>S-
assembly
>S&
mixed
>
```

The above examples show the source mode being changed to `source`, `assembly`, and `mixed`. In window mode, the commands change the format of the display window. In sequential mode, the commands change the output from the commands that display code (Register, Trace, Program Step, Go, Execute, and Unassemble). See the sections on individual commands for examples of how they are affected by the display mode.

9.2 Unassemble Command

The Unassemble command displays the assembly-language instructions of the program being debugged. It is most useful in sequential mode, where it is the only method of examining a sequence of assembly-language instructions. In window mode it can be used to display a specific portion of assembly-language code in the display window.

Note

Occasionally, code similar to the following will be displayed:

```
FE30   ???   Byte Ptr  [BX + SI]
```

These are instructions Intel recommends against using because equivalent opcodes exist and the instruction may not work properly on all machines, or because the opcodes are reserved. You should not see this type of code unless you are unassembling data.

■ Mouse

The Unassemble command has no direct mouse equivalent, but you can view unassembled code at any time by changing the mode to assembly or mixed (see Section 9.1, "Set Mode Command," for more information).

■ Keyboard

The Unassemble command has no direct keyboard equivalent, but you can view unassembled code at any time by changing the mode to assembly or mixed (see Section 9.1, "Set Mode Command," for more information).

■ Dialog

To display unassembled code using a dialog command, enter a command line with the following syntax:

U [[*address* | *range*]]

The effect of the command varies depending on whether you are in sequential or window mode.

In sequential mode, if you do not specify *address* or *range*, the disassembled code begins at the current unassemble address and shows the next eight lines of instructions. The unassemble address is the address of the instruction after the last instruction displayed by the previous Unassemble command. If the Unassemble command has not been used during the session, the unassemble address is the current instruction.

If you specify an *address*, the disassembly starts at that address and shows the next eight lines of instructions. If you specify a *range*, the instructions within the range will be displayed.

The format of the display depends on the current display mode (see Section 9.1, "Set Mode Command," for more information). If the mode is source (S+) or mixed (S&), the CodeView debugger displays source lines mixed with unassembled instructions. One source line is shown for each corresponding group of assembly-language instructions. If the display mode is assembly, only assembly-language instructions are shown.

In window mode, the Unassemble command changes the mode of the display window to assembly. The display format will reflect any options previously set from the Options menu. There is no output to the dialog window. If *address* is given, the instructions in the display window will begin at the specified address. If *range* is given, only the starting address will be used. If no argument is given, the debugger scrolls down and displays the next screen of assembly-language instructions.

Note

80286 protected-mode mnemonics cannot be displayed with the Unassemble command.

■ Examples

```
>S&
mixed
>U 0x11
49D0:0011 35068E        XOR   AX,__sqrtjmptab+8cd4 (8E06)
49D0:0014 189A2300      SBB   Byte Ptr [BP+SI+0023],BL
49D0:0018 FC            CLD
49D0:0019 49            DEC   CX
49D0:001A CD351ED418    INT   35 ;FSTP    DWord Ptr [__fpinit+ee (18D4)]
49D0:001F CD3D          INT   3D ;FWAIT
7:                A = 0.0
49D0:0021 CD35EE        INT   35 ;FLDZ
```

The above example sets the mode to mixed and unassembles eight lines of disassembled code, plus whatever source lines are encountered within those lines. The display would be the same if the mode were source.

The example is taken from a FORTRAN debugging session, but produces results similar to what the same commands would produce with a C or BASIC program.

```
>S-
assembly
>U 0x11
49D0:0011 35068E        XOR   AX,__sqrtjmptab+8cd4 (8E06)
49D0:0014 189A2300      SBB   Byte Ptr [BP+SI+0023],BL
49D0:0018 FC            CLD
49D0:0019 49            DEC   CX
49D0:001A CD351ED418    INT   35 ;FSTP    DWord Ptr [__fpinit+ee (18D4)]
49D0:001F CD3D          INT   3D ;FWAIT
49D0:0021 CD35EE        INT   35 ;FLDZ
>
```

The above example sets the mode to assembly and repeats the same command.

9.3 View Command

The View command displays the lines of a text file (usually a source module or include file). It is most useful in sequential mode, where it is the only method of examining a sequence of source lines. In window mode, the View command can be used to page through the source file or to load a new source file.

■ Mouse

To load a new source file with the button, point to File on the menu bar, press a mouse button and drag the highlight to the Load selection, then release the button. A dialog box appears, asking for the name of the file you wish to load. Type the name of the file, and press the ENTER key or a mouse button. The new file appears in the display window.

The paging capabilities of the View command have no direct mouse equivalent, but you can move about in the source file by pointing to the up or down arrows on the scroll bars and then clicking different mouse buttons. See Chapter 3, "The CodeView Display," for more information about paging with the mouse.

■ Keyboard

To load a new source file with a keyboard command, press ALT-F to open the File menu, then press ALT-L to select Load. A dialog box appears, asking for the name of the file you wish to load. Type the name of the file, and press the ENTER key. The new file appears in the display window.

The paging capabilities of the View command have no direct keyboard equivalent, but you can move about in the source file by first putting the cursor in the display window with the F6 key, then pressing the PGUP, PGDN, HOME, END, and UP ARROW and DOWN ARROW keys. See Chapter 3, "The CodeView Display," for more information about paging with keyboard commands.

■ Dialog

To display source lines using a dialog command, enter a command line with the following syntax:

V [[*expression*]]

Since addresses for the View command are often specified as a line number (with an optional source file), a more specific syntax for the command would be as follows:

V [[.[[*filename*:]]*linenumber*]]

The effect of the command varies, depending on whether you are in sequential or window mode.

In sequential mode, the View command displays eight source lines. The starting source line is one of the following:

- The current source line if no argument is given.
- The specified *linenumber*. If *filename* is given, the specified file is loaded, and the *linenumber* refers to lines in it.
- The address that *expression* evaluates to. For example, *expression* could be a procedure name or an address in the *segment*:*offset* format. The code segment is assumed if no segment is given.

In sequential mode, the View command is not affected by the current display mode (source, assembly, or mixed); source lines are displayed regardless of the mode.

In window mode, if you enter the View command while the display mode is assembly, the CodeView debugger will automatically switch back to source mode. If you give *linenumber* or *expression*, the display window will be redrawn so that the source line corresponding to the given *address* will appear at the top of the source window. If you specify a *filename* with a *linenumber*, the specified file will be loaded.

If you enter the View command with no arguments, the display will scroll down one line short of a page; that is, the source line that was at the bottom of the window will be at the top.

Note

The View command with no argument is similar to pressing the PGDN key, or clicking right on the down arrow with the mouse. The difference is that pressing the PGDN key enables you to scroll down one more line.

■ Examples

```
>V BUBBLE                ;* Example 1, FORTRAN source code
51:           IF (N .LE. 1) GOTO 101
52:           DO 201 I = 1,N-1
53:           DO 301 J = I + 1,N
54:           IF (X(I) .LE. X(J)) GOTO 301
55:           TEMP = X(I)
56:           X(I) = X(J)
57:           X(J) = TEMP
58:       301 CONTINUE
```

Example 1 (shown in sequential mode) displays eight source lines, beginning at routine BUBBLE.

```
>V .math.c:30    ;* Example 2, C source code
30:                 register int j;
31:
32:                 for (j = q; j >= 0; j--)
33:                         if (t[j] + p[j] > 9) {
34:                                 p[j] += t[j] - 10;
35:                                 p[j-1] += 1;
36:                         } else
37:                                 p[j] += t[j];
>
```

Example 2 loads the source file `math.c` and displays eight source lines starting at line 30.

All forms of the View command are supported with all languages that work with the CodeView debugger. The above examples vary languages simply to be more accessible to all users.

9.4 Current Location Command

The Current Location command displays the source line or assembly-language instruction corresponding to the current program location.

■ Mouse

The Current Location command cannot be executed with the mouse.

■ Keyboard

The Current Location command cannot be executed with a keyboard command.

■ Dialog

To display the current location line using a dialog command, enter a command line with the following syntax:

•

In sequential mode, the command displays the current source line. The line is displayed regardless of whether the current debugging mode is source or assembly. If the program being debugged has no symbolic information, the command will be ignored.

In window mode, the command puts the current program location (marked with reverse video or a contrasting color) in the center of the display window. The display mode (source or assembly) will not be affected. This command is useful if you have scrolled through the source code or assembly-language instructions so that the current location line is no longer visible.

For example, if you are in window mode and have executed the program being debugged to somewhere near the start of the program, but you have scrolled the display to a point near the end, the Current Location command returns the display to the current program location.

■ Example

```
>.
MINDAT = 1.0E6
>
```

The above example illustrates how to display the current source line in sequential mode. The same command in window mode would not produce any output, but it could change the text shown in the display window.

9.5 Stack Trace Command

The Stack Trace command allows you to display routines that have been called during program execution. The first line of the display shows the name of the current routine. The succeeding lines (if any) list any other routines that were called to reach the current address.

For each routine, the values of any arguments are shown in parentheses after the routine name. Values are shown in the current radix (the default is decimal).

The term "stack trace" is used because, as each routine is called, its address and arguments are stored (pushed) onto the program stack. Therefore, tracing through the stack shows the currently active routines. With C and FORTRAN programs, the **main** routine will always be at the bottom of the stack. With BASIC programs, the main program is not listed on the stack, because BASIC programs have no standard label (such as **main**) corresponding to the first line of a program. Only routines *called* by the main program will be displayed. In assembly-language programs, the bottom routine displayed in the stack trace is **astart** instead of **main**.

The Stack Trace command also enables you to find and view the source lines where individual routines were called.

Note

This discussion uses the term "routines", which is a general term for functions, subroutines, subprograms, and procedures—each of which uses the stack to transfer control to an independent program unit. In assembly mode, the term "procedure" may be more accurate. If you are using the CodeView debugger to debug assembly-language programs, the Stack Trace command will work only if procedures were called with the calling convention used by Microsoft languages. This calling convention is explained in the User's Guide to each language.

■ Mouse

To view a stack trace with the mouse, point to Calls on the menu bar and press a mouse button. The Calls menu will appear, showing the current routine at the top and other routines below it in the reverse order in which they were called; for example, the first routine called will be at the bottom. The values of any routine arguments will be shown in parentheses following the routines.

If you want to view code at the point where one of the routines was called, hold the mouse button down and drag the highlight to the routine below the one you want to view, then release the button. The cursor will move to the calling source line (in source mode) or the calling instruction (in assembly or mixed mode). In other words, the cursor will indicate the calling location in the selected routine where the next-level routine was called. If you select the current (top-level) routine, the cursor moves to the current location in that routine.

■ Keyboard

To view a stack trace with a keyboard command, press ALT-C to open the Calls menu. The menu will show the current routine at the top, and other routines below it in the reverse order in which they were called; for example, the first routine called will be at the bottom. The values of any routine arguments will be shown in parentheses following the routine.

If you want to view code at the point where one of the routines was called, press the down arrow to move the highlight down to the routine below the one you want to view, then press the ENTER key. The cursor will move to the calling source line (in source mode) or the calling instruction (in assembly or mixed mode). In other words, the cursor will indicate the calling

location in the selected routine where the next-level routine was called. If you select the current (top-level) routine, the cursor moves to the current location in that routine.

■ Dialog

To display a stack trace with a dialog command, enter a command line with the following syntax:

K

The output from the Stack Trace dialog command lists the routines in the reverse order in which they were called. The arguments to each routine are shown in parentheses. Finally, the line number from which the routine was called is shown.

You can enter the line number as an argument to the View or Unassemble command if you want to view code at the point where the routine was called.

In window mode, the output from the Stack Trace dialog command appears in the dialog window. You may need the dialog version rather than the menu version, since the Calls menu can be truncated if there are too many routines or routine arguments. The dialog display wraps around if necessary, so that you can see all routines and all arguments.

■ FORTRAN Example

```
>K
ANALYZE(67,0), line 94
COUNTWORDS(0,512), line 73
MAIN(2,5098), line 42
>
```

In the above example, the first line of output indicates that the current routine is ANALYZE. Its first argument currently has a decimal value of 67, and its second argument has a value of 0. The current location in this routine is `line 94`.

The second line indicates that ANALYZE was called by COUNTWORDS, and that its arguments have the values 0 and 512. Routine ANALYZE was called from `line 73` of routine COUNTWORDS.

Likewise, COUNTWORDS was called from `line 42` of MAIN, and its arguments have the values 2 and 5098.

If the radix had been set to 16 or 8 using the Radix (N) command, the arguments would be shown in that radix. For example, the last line would be shown as `MAIN(2,13ea)` in hexadecimal or `MAIN(2,11752)` in octal.

■ C Example

```
>K
analyze(67,0), line 94
countwords(0,512), line 73
main(2,5098)
>
```

As with the FORTRAN example, the above example shows the routines on the stack in the reverse order in which they were called. Since `analyze` is on the top, it has been called most recently; in other words, it is the current routine.

Each routine is shown with the arguments it was called with, along with the last source line that it had been executing. Note that `main` is shown with the command line arguments `argc` (which is equal to 2), and `argv` (which is a pointer equal to 5098 decimal). Since the language is C, `main` will always be on the bottom of the stack.

■ BASIC Example

```
>K
ROLL#(19122:6040)
MAKE#(19122:6040)
CALC(19122:5982, 19122:5990)
>
```

As with the FORTRAN example, the above example shows the routines on the stack in the reverse order in which they were called. Since ROLL# is on the top, it has been called most recently; in other words, it is the current routine.

Each routine is displayed along with the arguments that it was passed. In BASIC, arguments passed to routines are always addresses.

This example shows some features peculiar to BASIC. First of all, there is no MAIN displayed, because the BASIC compiler does not produce any such symbol. Furthermore, each routine will have a type tag if it is a function; this indicates what the function returns. ROLL# and MAKE# are both functions returning a double precision floating-point. A function that returned a short integer would have a % type tag. CALC has no type tag since it is a subprogram, and therefore does not return a value of any type.

Chapter 10
Modifying Code or Data

10.1 Assemble Command 219
10.2 Enter Commands 223
10.2.1 Enter Command 226
10.2.2 Enter Bytes Command 227
10.2.3 Enter ASCII Command 228
10.2.4 Enter Integers Command 229
10.2.5 Enter Unsigned Integers Command 229
10.2.6 Enter Words Command 230
10.2.7 Enter Double Words Command 231
10.2.8 Enter Short Reals Command 232
10.2.9 Enter Long Reals Command 233
10.2.10 Enter 10-Byte Reals Command 234
10.3 Fill Memory Command 234
10.4 Move Memory Command 236
10.5 Port Output Command 237
10.6 Register Command 238

The CodeView debugger provides the following commands for modifying code or data in memory:

Command	Action
Assemble (**A**)	Modifies code
Enter (**E**)	Modifies memory, usually data
Register (**R**)	Modifies registers and flags
Fill Memory (**F**)	Fills a block of memory
Move Memory (**M**)	Copies one block of memory to another
Port Output (**O**)	Outputs a byte to a hardware port

Changes to code are temporary. You can use them for testing in the CodeView debugger, but you cannot save them or permanently change the program. To make permanent changes, you must modify the source code and recompile.

10.1 Assemble Command

The Assemble command assembles 8086-family (8086, 8087, 8088, 80186, 80287, and 80286 unprotected) instruction mnemonics and places the resulting instruction code into memory at a specified address. The only 8086-family mnemonics that cannot be assembled are 80286 protected-mode mnemonics.

■ Mouse

The Assemble command cannot be executed with the mouse.

■ Keyboard

The Assemble command cannot be executed with a keyboard command.

■ Dialog

To assemble code using a dialog command, enter a command line with the following syntax:

A[[*address*]]

If *address* is specified, the assembly starts at that address; otherwise the current assembly address is assumed.

The assembly address is normally the current address (the address pointed to by the **CS** and **IP** registers). However, when you use the Assemble command, the assembly address is set to the address immediately following the last instruction where you assembled an instruction. When you enter any command that executes code (Trace, Program Step, Go, or Execute), the assembly address is reset to the current address.

When you type the Assemble command, the assembly address is displayed. The CodeView debugger then waits for you to enter a new instruction in the standard 8086-family instruction-mnemonic form. You can enter instructions in uppercase, lowercase, or both.

To assemble a new instruction, type the desired mnemonic and press the ENTER key. The CodeView debugger assembles the instruction into memory and displays the next available address. Continue entering new instructions until you have assembled all the instructions you want. To conclude assembly and return to the CodeView prompt, press the ENTER key only.

If an instruction you enter contains a syntax error, the debugger displays the message `^ Syntax error`, redisplays the current assembly address, and waits for you to enter a correct instruction. The caret symbol in the message will point to the first character the CodeView debugger could not interpret.

The following nine rules govern entry of instruction mnemonics:

1. The far-return mnemonic is **RETF**.

2. String mnemonics must explicitly state the string size. For example, use **MOVSW** to move word strings and **MOVSB** to move byte strings.

3. The CodeView debugger automatically assembles short, near, or far jumps and calls, depending on byte displacement to the destination address. These may be overridden with the **NEAR** or **FAR** prefix, as shown in the following examples:

```
JMP     0x502
JMP     NEAR 0x505
JMP     FAR  0x50A
```

   The **NEAR** prefix can be abbreviated to **NE**, but the **FAR** prefix cannot be abbreviated. The above examples use the C notation for hexidecimal numbers. If the FORTRAN option was selected, then you would enter the operands as #502, #505, and #50A; if the BASIC option was selected, you would enter them as &H502, &H505, and &H50A.

4. The CodeView debugger cannot determine whether some operands refer to a word memory location or to a byte memory location. In these cases, the data type must be explicitly stated with the prefix **WORD PTR** or **BYTE PTR**. Acceptable abbreviations are **WO** and **BY**. Examples are shown below:

```
MOV     WORD PTR [BP],1
MOV     BYTE PTR [SI-1],symbol
MOV     WO PTR [BP],1
MOV     BY PTR [SI-1],symbol
```

5. The CodeView debugger cannot determine whether an operand refers to a memory location or to an immediate operand. The debugger uses the convention that operands enclosed in square brackets refer to memory. Two examples are shown below:

```
MOV     AX,#21
MOV     AX,[#21]
```

   The first statement moves 21 hexidecimal into AX. The second statement moves the data at offset 21 hexidecimal into AX. Both statements use the FORTRAN notation for the hexidecimal number 21. If the C option was selected, then this number would be represented as 0x21, and if the BASIC option was selected, then the number would be represented as &H21.

6. The **DB** instruction assembles byte values directly into memory. The **DW** instruction assembles word values directly into memory, as shown in the following examples:

```
DB      1,2,3,4,"This is an example."
DW      1000,2000,3000,"Bach"
```

7. The CodeView debugger supports all forms of indirect register instructions, as shown in the following examples:

```
ADD     BX,[BP+2].[SI-1]
POP     [BP+DI]
PUSH    [SI]
```

8. All instruction-name synonyms are supported, as shown in the following examples:

```
LOOPZ   &H100
LOOPE   &H100
JA      &H200
JNBE    &H200
```

   If you assemble instructions and then examine them with the Unassemble command (**U**), the CodeView debugger may show synonymous instructions, rather than the ones you assembled. The above examples use the BASIC hexidecimal notation. Instead of using the &H prefix, you would use 0x with the C option selected, and # with the FORTRAN option selected.

9. Do not assemble and execute 8087 or 80287 instructions if your system is not equipped with one of these math coprocessor chips. The **WAIT** instruction, for example, will cause your system to hang up if you try to execute it without the appropriate chip.

■ Example (FORTRAN notation)

```
>U #40 L 1
39B0:0040 89C3            MOV         BX,AX
>A #40
39B0:0040 MOV     CX,AX
39B0:0042
>U #40 L 1
39B0:0040 89C1            MOV         CX,AX
>
```

The above example modifies the instruction at address 40 hexidecimal so that it moves data into the CX register instead of the BX register. (40 hexidecimal is notated as 0x40 in C, and as &H40 in BASIC.) The Unassemble command (**U**) is used to show the instruction before and after the assembly.

You can modify a portion of code for testing, as in the example, but you cannot save the modified program. You must modify your source code and recompile.

10.2 Enter Commands

The CodeView debugger has several commands for entering data to memory. You can use these commands to modify either code or data, though code can usually be modified more easily with the Assemble command (**A**). The Enter commands are listed below:

Command	Command Name
E	Enter (size is the default type)
EB	Enter Bytes
EA	Enter ASCII
EI	Enter Integers
EU	Enter Unsigned Integers
EW	Enter Words
ED	Enter Double Words
ES	Enter Short Reals
EL	Enter Long Reals
ET	Enter 10-Byte Reals

■ Mouse

The Enter commands cannot be executed with the mouse.

■ Keyboard

The Enter commands cannot be executed with keyboard commands.

■ Dialog

To enter data to memory with a dialog command, enter a command line with the following syntax:

E[[*type*]] *address* [[*list*]]

The *type* is a one-letter specifier that indicates the type of the data to be entered. The *address* indicates where the data will be entered. If no segment is given in the address, the data segment (**DS**) is assumed.

The *list* can consist of one or more expressions that evaluate to data of the size specified by *type*. This data will be entered to memory at *address*. If one of the values in the list is invalid, an error message will be displayed. The values preceding the error are entered; values at and following the error are not entered.

The expressions in the list are evaluated in the current radix, regardless of the size and type of data being entered. For example, if the radix is 10 and you give the value 10 in a list with the Enter Words command, the decimal value 10 will be entered even though word values are normally entered in hexadecimal. This means that the Enter Words, Enter Integers, and Enter Unsigned Integers commands are identical when used with the list method, since 2-byte data is being entered for each command.

If *list* is not given, the CodeView debugger will prompt for values to be entered to memory. Values entered in response to prompts are accepted in hexadecimal for the Enter Bytes, Enter ASCII, Enter Words, and Enter Double Words commands. The Enter Integers command accepts signed decimal integers, while the Enter Unsigned Integers command accepts unsigned decimal integers. The Enter Short Reals, Enter Long Reals, and Enter 10-Byte Reals commands accept decimal floating-point values.

With the prompting method of data entry, the CodeView debugger prompts for a new value at *address* by displaying the address and its current value. You can then replace the value, skip to the next value, return to a previous value, or exit the command, as explained below:

- To replace the value, type the new value after the current value.
- To skip to the next value, press the SPACEBAR. Once you have skipped to the next value, you can change its value or skip to the following value. If you pass the end of the display, the CodeView debugger displays a new address to start a new display line.

- To return to the preceding value, type a backslash (\). When you return to the preceding value, the debugger starts a new display line with the address and value.
- To stop entering values and return to the CodeView prompt, press the ENTER key. You can exit the command at any time.

Sections 10.2.1–10.2.10 discuss the Enter commands in order of the size of data they accept.

■ Examples

```
>EW PLACE 16 32
```

The above example shows how to enter two word-sized values at the address PLACE. If the default radix (decimal) is in effect, this command enters the hexadecimal values 10 and 20.

```
>EW PLACE

3DA5:0B20  00F3._
```

The above example illustrates the prompting method of entering data. When you supply the address where you want to enter data but supply no data to be entered there, the CodeView debugger displays the current value of the address and waits for you to enter a new value. The underscore in these examples represents the CodeView cursor. You could change the value F3 to the new value 16 (10 hexidecimal) by typing 10 (but don't press the ENTER key yet). The value must be typed in hexadecimal for the Enter Words command, as shown below:

```
>EW PLACE

3DA5:0B20  00F3.10_
```

You could then skip to the next value by pressing the SPACEBAR. The CodeView debugger responds by displaying the next value, as shown below:

```
>EW PLACE

3DA5:0B20  00F3.10  4F20._
```

You could then type another hexadecimal value, such as 30:

```
>EW PLACE

3DA5:0B20  00F3.10  4F20.30_
```

Press the SPACEBAR to move to the next value:

```
>EW PLACE

3DA5:0B20  00F3.10  4F20.30  3DC1._
```

Assume you realize that the last value entered, 30, is incorrect. You really wanted to enter 20. You could return to the previous value by typing a backslash. The CodeView debugger starts a new line, starting with the previous value. Note that the backslash is not echoed:

```
>EW PLACE

3DA5:0B20  00F3.10  4F20.30  3DC1.
3DA5:0B22  0030._
```

Type the correct value, 20:

```
>EW PLACE

3DA5:0B20  00F3.10  4F20.30  3DC1.
3DA5:0B22  0030.20_
```

If this is the last value you want to enter, press the ENTER key to stop. The CodeView prompt reappears, as shown below:

```
>EW PLACE

3DA5:0B20  00F3.10  4F20.30  3DC1.
3DA5:0B22  0030.20
>_
```

10.2.1 Enter Command

■ Syntax

E *address* [[*list*]]

The Enter command enters one or more values into memory at the specified *address*. The data is entered in the format of the default type, which is the last type specified with a Dump, Enter, Watch Memory, or Tracepoint Memory command. If none of these commands has been entered during the session, the default type is bytes.

Use this command with caution when entering values in the list format; values will be truncated if you enter a word-sized value when the default type is actually bytes. If you are not sure of the current default type, specify the size in the command.

Important

The Execute command and the Enter command have the same command letter (**E**). The difference is that the Execute command never takes an argument; the Enter command always requires at least one argument.

10.2.2 Enter Bytes Command

■ Syntax

EB *address* [[*list*]]

The Enter Bytes command enters one or more byte values into memory at *address*. The optional *list* can be entered as a list of expressions separated by spaces. The expressions are evaluated and entered in the current radix. If *list* is not given, the CodeView debugger prompts for new values, which must be entered in hexadecimal.

The Enter Bytes command can also be used to enter strings, as described in Section 10.2.3, "Enter ASCII Command."

■ Examples

```
>EB 256 10 20 30
>
```

If the current radix is 10, the above example replaces the three bytes at DS:256, DS:257, and DS:258 with the decimal values 10, 20, and 30. (These three bytes correspond to the hexidecimal addresses DS:0100, DS:0101, and DS:0102.)

```
>EB 256

3DA5:0100  130F.A
>
```

The above example replaces the byte at DS:256 (DS:0100 hexidecimal) with 10 (0A hexidecimal).

10.2.3 Enter ASCII Command

■ Syntax

EA *address* [[*list*]]

The Enter ASCII command works in the same way as the Enter Bytes command (**EB**) described in Section 10.2.2. The *list* version of this command can be used to enter a string expression. You can include escape sequences in strings.

■ Example

```
>EA message "Can\'t open file"
>
```

In the example above, the string `Cannot open file` is entered starting at the symbolic address `message`. (Note that the double quotation marks are CodeView string delimiters.)

You can also use the Enter Bytes command to enter a string expression, or you can enter nonstring values using the Enter ASCII command.

10.2.4 Enter Integers Command

■ Syntax

EI *address* [[*list*]]

The Enter Integers command enters one or more word values into memory at *address* using the signed-integers format. With the CodeView debugger, a signed integer can be any decimal integer between –32,768 and 32,767.

The optional *list* can be entered as a list of expressions separated by spaces. The expressions are entered and evaluated in the current radix. If *list* is not given, the CodeView debugger prompts for new values, which must be entered in decimal.

■ Examples

```
>EI 256 -10 10 -20
>
```

If the current radix is 10, the above example replaces the three integers at DS:256, DS:258, and DS:260 with the decimal values -10, 10, and -20. (The three addresses correspond to the three hexidecimal addresses DS:0100, DS:0102, and DS:104.)

```
>EI 256

3DA5:0100  130F.-10
>
```

The above example replaces the integer at DS:256 (hexidecimal address DS:0100) with -10.

10.2.5 Enter Unsigned Integers Command

■ Syntax

EU *address* [[*list*]]

The Enter Unsigned Integers command enters one or more word values into memory at *address* using the unsigned-integers format. With the CodeView debugger, an unsigned integer can be any decimal integer between 0 and 65,535.

The optional *list* can be entered as a list of expressions separated by spaces. The expressions are entered and evaluated in the current radix. If *list* is not given, the CodeView debugger prompts for new values, which must be entered in decimal.

■ Examples

```
>EU 256 10 20 30
>
```

If the current radix is 10, the above example replaces the three unsigned integers at DS:256, DS:258, and DS:260 with the decimal values 10, 20, and 30. (These addresses correspond to the hexidecimal addresses DS:0100, DS:0102, and DS:0104.)

```
>EU 256

3DA5:0100  130F.10
>
```

The above example replaces the integer at DS:256 (DS:0100 hexidecimal) with 10.

10.2.6 Enter Words Command

■ Syntax

EW *address* [[*list*]]

The Enter Words command enters one or more word values into memory at *address*.

The optional *list* can be entered as a list of expressions separated by spaces. The expressions are entered and evaluated in the current radix. If *list* is not given, the CodeView debugger prompts for new values, which must be entered in hexadecimal.

■ Examples

```
>EW 256 10 20 30
>
```

If the current radix is 10, the above example replaces the three words at DS:256, DS:258, and DS:260 with the hexadecimal values A, 14, and 1E. (These addresses correspond to the hexidecimal addresses DS:0100, DS:0102, and DS:0104.)

```
>EW 256

3DA5:0100  130F.A
>
```

The above example replaces the integer at DS:256 (DS:0100 hexidecimal) with 10 (0A hexidecimal).

10.2.7 Enter Double Words Command

■ Syntax

ED *address* [[*list*]]

The Enter Double Words command enters one or more double-word values into memory at *address*. Double words are displayed and entered in the *segment*:*offset* address format; that is, two words separated by a colon (:). If the colon is omitted and only one word entered, only the offset portion of the address will be changed.

The optional *list* can be entered as a list of expressions separated by spaces. The expressions are entered and evaluated in the current radix. If *list* is not given, the CodeView debugger prompts for new values, which must be entered in hexadecimal.

■ Examples

```
>ED 256 8700:12008
>
```

If the current radix is 10, the above example replaces the double words at DS:256 (DS:0100 hexidecimal) with the hexadecimal address 21FC:2EE8 (8700:12008).

```
>ED 256

3DA5:0100  21FC:2EE8.2EE9
>
```

The above example replaces the offset portion of the double word at DS:256 (DS:0100 hexidecimal) with 2EE9 hexidecimal. Since the segment portion of the address is not provided, the existing segment (21FC hexidecimal) is unchanged.

10.2.8 Enter Short Reals Command

■ Syntax

ES *address* [[*list*]]

The Enter Short Reals command enters one or more short-real values into memory at *address*.

The optional *list* can be entered as a list of real numbers separated by spaces. The numbers must be entered in decimal, regardless of the current radix. If *list* is not given, the CodeView debugger prompts for new values, which must be entered in decimal. Short-real numbers can be entered either in floating-point format or in scientific-notation format.

■ Examples

```
>ES 256 23.479 1/4 -1.65E+4 235
>
```

The above example replaces the four numbers at DS:256, DS:260, DS:264, and DS:268 with the real numbers 23.479, 0.25, -1650.0, and 235.0. (These addresses correspond to the hexidecimal addresses DS:0100,

DS:0104, DS:0108, and DS:0112.)

```
>ES PI
3DA5:0064  42 79 74 65   7.215589E+022  3.141593
>
```

The above example replaces the number at the symbolic address `PI` with `3.141593`.

10.2.9 Enter Long Reals Command

■ Syntax

EL *address* [[*list*]]

The Enter Long Reals command enters one or more long-real values into memory at *address*.

The optional *list* can be entered as a list of real numbers separated by spaces. The numbers must be entered in decimal, regardless of the current radix. If *list* is not given, the CodeView debugger prompts for new values, which must be entered in decimal. Long-real numbers can be entered either in floating-point format or in scientific-notation format.

■ Examples

```
>EL 256 23.479 1/4 -1.65E+4 235
>
```

The above example replaces the four numbers at DS:256, DS:264, DS:272, and DS:280 with the real numbers `23.479`, `0.25`, `-1650.0`, and `235.0` (These addresses correspond to the hexidecimal addresses DS:0100, DS:0108, DS:0110, and DS:0118.)

```
>EL PI
3DA5:0064  42 79 74 65 DC 0F 49 40   5.012391E+001  3.141593
>
```

The above example replaces the number at the symbolic address `PI` with `3.141593`.

10.2.10 Enter 10-Byte Reals Command

■ Syntax

ET *address* [[*list*]]

The Enter 10-Byte Reals command enters one or more 10-byte-real values into memory at *address*.

The optional *list* can be entered as a list of real numbers separated by spaces. The numbers must be entered in decimal, regardless of the current radix. If *list* is not given, the CodeView debugger prompts for new values, which must be entered in decimal. The numbers can be entered either in floating-point format or in scientific-notation format.

■ Examples

```
>ET 256 23.479 1/4 -1.65E+4 235
>
```

The above example replaces the four numbers at DS:256, DS:266, DS:276, and DS:286 with the real numbers `23.479`, `0.25`, `-1650.0`, and `235.0`. (These addresses correspond to the hexidecimal addresses DS:0100, DS:010A, DS:0114, and DS:011E.)

```
>ET PI
3DA5:0064  42 79 74 65 DC 0F 49 40 7F BD  -3.292601E-193  3.141593
>
```

The above example replaces the number at the symbolic address `PI` with `3.141593`.

10.3 Fill Memory Command

The Fill Memory command provides an efficient way of filling up a large or small block of memory, with any values you specify. It is primarily of interest to assembly programmers, because the command enters values directly into memory. However, you may find it useful for initializing large data areas such as an array or structure.

■ Mouse

The Fill Memory command cannot be executed with a mouse.

■ Keyboard

The Fill Memory command cannot be executed with a keyboard command.

■ Dialog

To fill an area of memory with values you specify, enter the Fill Memory command as follow:

F *range list*

The Fill Command fills the addresses in the specified *range* with the byte values specified in *list.* The values in the list are repeated until the whole range is filled. (Thus, if you specify only one value, the entire range is filled with that same value.) If the *list* has more values than the number of bytes in the *range,* then the command ignores any extra values.

■ Examples

```
>F 100 L 100 0    ;* hexidecimal radix assumed
>
```

The first example fills 255 (100 hexidecimal) bytes of memory starting at DS:0100 with the value 0. This example might possibly be used to reintialize the program's data without having to restart the program.

```
>F table L 64 42 79 74 ;* hexidecimal radix assumed
>
```

The second example fills the 100 (64 hexidecimal) bytes starting at `table` with the following hexidecimal byte values: 42, 79, 74. These three values are repeated until all 100 bytes are filled.

Note

You can enter arguments to the Fill Command using any radix which is most convenient.

10.4 Move Memory Command

The Move Memory command enables you to copy all the values in one block of memory, directly to another block of memory of the same size. This command is of most interest to assembly programmers, but can be used by anyone who wants to do large data transfers efficiently. For example, you can use this command to copy all the values in one array to the elements of another.

■ Mouse

The Move Memory command cannot be executed with the mouse.

■ Keyboard

The Move Memory command cannot be executed with a keyboard command.

■ Dialog

To copy the values in one block of memory to another, enter the Move Memory command with the following syntax:

M *range address*

The values in the block of memory specified by *range* are copied to a block of the same size beginning at *address*. All data in *range* is guaranteed to be copied completely over to the destination block, even if the two blocks overlap. However, if they do overlap then some of the original data in *range* will be altered.

To prevent loss of data, the Move Memory command copies data starting at the source block's lowest address whenever the source is at a higher address than the destination. If the source is at a lower address, then the Move Memory command copies data beginning at the source block's highest address.

■ Example

```
>M arr1(1) L arsize arr2(1)  ;* BASIC/FORTRAN example
>
```

In the above example, the block of memory beginning with the first element of `arr1`, and `arsize` bytes long, is copied directly to a block of the same size beginning at the address of the first element of `arr2`. In C, this command would be entered as `M arr1[0] L arsize arr2[0]`.

10.5 Port Output Command

The Port Ouput command sends specific byte values to hardware ports. It is primarily of use to assembly programmers writing code that interacts directly with hardware.

■ Mouse

The Port Output command cannot be executed with a mouse.

■ Keyboard

The Port Output command cannot be executed with a keyboard command.

■ Dialog

To output to a hardware port, enter the Port Output command with the following syntax:

O *port byte*

The specified *byte* is sent to the specified *port*, in which *port* is a 16-bit port address.

■ Example

```
>O 2F8 4F       ;* hexidecimal default radius assumed
>
```

The byte value 4F hexidecimal is sent to output port 2F8.

As with all other CodeView commands, you can enter the Port Output command using any radius you prefer.

10.6 Register Command

The Register command has two functions: it displays the contents of the central processing unit (CPU) registers, and it can also change the values of those registers. The modification features of the command are explained in this section. The display features of the Register command are explained in Chapter 7, "Examining Data and Expressions."

■ Mouse

The only register that can be changed with the mouse is the flags register. The register's individual bits (called flags) can be set or cleared. To change a flag, first make sure the register window is open. The window can be opened by selecting Registers from the Options menu, or by pressing the F2 key.

The flag values are shown as mnemonics in the bottom of the window. Point to the flag you want to change and click either button. The mnemonic word representing the flag value will change. The mnemonics for each flag are shown in the third and fifth columns of Table 10.1. The color or highlighting of the flag will also be reversed when you change a flag. Set flags are shown in red on color monitors and in high-intensity text on two-color monitors. Cleared flags are shown in light blue or normal text.

■ Keyboard

The registers cannot be changed with keyboard commands.

■ Dialog

To change the value of a register with a dialog command, enter a command line with the following syntax:

R [[*registername*[[=]*expression*]]]

To modify the value in a register, type the command letter **R** followed by *registername*. The CodeView debugger displays the current value of the register and prompts for a new value. Press the ENTER key if you only want to examine the value. If you want to change it, type an expression for the new value and press the ENTER key.

As an alternative, you can type both *registername* and *expression* in the same command. You can use the equal sign (=) between *registername* and *expression*, but a space has the same effect.

The register name can be any of the following names: **AX**, **BX**, **CX**, **DX**, **CS**, **DS**, **SS**, **ES**, **SP**, **BP**, **SI**, **DI**, **IP**, or **F** (for flags).

To change a flag value, supply the register name **F** when you enter the Register command. The command displays the current value of each flag as a two-letter name. The flag values are shown in Table 10.1.

Table 10.1
Flag-Value Mnemonics

Flag Name	Set Dialog	Set Window	Clear Dialog	Clear Window
Overflow	**OV**	overflow	**NV**	novrflow
Direction	**DN**	down	**UP**	up
Interrupt	**EI**	enable	**DI**	disable
Sign	**NG**	negative	**PL**	positive
Zero	**ZR**	zero	**NZ**	not zero
Auxiliary carry	**AC**	auxcarry	**NA**	no auxcy
Parity	**PE**	even	**PO**	odd
Carry	**CY**	carry	**NC**	no carry

At the end of the list of values, the command displays a dash (–). Enter new values after the dash for the flags you wish to change, then press the

ENTER key. You can enter flag values in any order. Flags for which new values are not entered remain unchanged. If you do not want to change any flags, simply press the ENTER key.

If you enter an illegal flag name, an error message will be displayed. The flags preceding the error are changed; flags at and following the error are not changed.

■ Examples

```
>R IP 256
>
```

The above example changes the IP register to the value 256 (0100 hexidecimal).

```
>R AX
AX 0E00
:_
```

The above example displays the current value of the AX register and prompts for a new value (the underscore represents the CodeView cursor).

```
>R AX
AX 0E00
:256
>_
```

You can now type any 16-bit value after the colon. As the above example shows, if the current radix is 10, you can enter 256 to change the AX value to 256 (256).

```
>R F UP EI PL
```

The example above shows the command-line method of changing flag values.

```
>R F
NV(OV) UP(DN) EI(DI) PL(NG) NZ(ZR) AC(NA) PE(PO) NC(CY) -OV DI ZR
>R F
OV(NV) UP(DN) DI(EI) PL(NG) ZR(NZ) AC(NA) PE(PO) NC(CY) -
>
```

With the prompting method of changing flag values (shown above), the first mnemonic for each flag is the current value, and the second mnemonic (in parentheses) is the alternate value. You can enter one or more mnemonics at the dash prompt. In the example, the command is given a second time to show the results of the first command.

Chapter 11
Using System-Control Commands

11.1 Help Command 245
11.2 Quit Command 246
11.3 Radix Command 247
11.4 Redraw Command 249
11.5 Screen Exchange Command 250
11.6 Search Command 251
11.7 Shell Escape Command 253
11.8 Tab Set Command 256
11.9 Redirection Commands 257
11.9.1 Redirecting CodeView Input 258
11.9.2 Redirecting CodeView Output 259
11.9.3 Redirecting CodeView Input and Output 260
11.9.4 Commands Used with Redirection 261
11.9.4.1 Comment Command 261
11.9.4.2 Delay Command 263
11.9.4.3 Pause Command 263

This chapter discusses commands that control the operation of the CodeView debugger. The commands in this category are listed below:

Command	Action
Help (**H**)	Displays help
Quit (**Q**)	Returns to DOS
Radix (**N**)	Changes radix
Redraw (@)	Redraws screen
Screen Exchange (\)	Switches to output screen
Search (**/**)	Searches for regular expression
Shell Escape (**!**)	Starts new DOS shell
Tab Set (#)	Sets tab size
Redirection and related commands	Control redirection of CodeView output or input

The system-control commands are discussed in the following sections.

11.1 Help Command

The CodeView debugger has two help systems: a complete on-line-help system available only in window mode, and a syntax summary available with sequential mode.

■ Mouse

To enter the complete on-line-help system with the mouse, point to View on the menu bar, press a mouse button and drag the highlight down to the Help selection, then release the button. The initial help screen appears. See Chapter 3, "The CodeView Display," for details on using the on-line-help system.

■ Keyboard

If you are in window mode, press the F1 key to enter the complete on-line-help system. See Chapter 3, "The CodeView Display," for details on using the on-line-help system. If you are in sequential mode, a syntax-summary screen appears when you press F1.

■ Dialog

If you are in window mode, you can view the complete on-line-help system with the following command:

H

If you are in sequential mode, this command displays a screen containing all CodeView dialog commands with the syntax for each. This screen is the only help available in sequential mode.

11.2 Quit Command

The Quit command terminates the CodeView debugger and returns control to DOS.

■ Mouse

To quit the CodeView debugger with the mouse, point to File on the menu, press a mouse button and drag the highlight down to the Quit selection, then release the button. The CodeView screen will be replaced by the DOS screen, with the cursor at the DOS prompt.

■ Keyboard

To quit the CodeView debugger with a keyboard command, press ALT-F to open the File menu, then press ALT-Q to select Quit. The CodeView screen will be replaced by the DOS screen, with the cursor at the DOS prompt.

■ Dialog

To quit the CodeView debugger using a dialog command, enter a command line with the following syntax:

Q

When the command is entered, the CodeView screen will be replaced by the DOS screen, with the cursor at the DOS prompt.

11.3 Radix Command

The Radix command changes the current radix for entering arguments and displaying the value of expressions. The default radix when you start the CodeView debugger is 10 (decimal). Radixes 8 (octal) and 16 (hexadecimal) can also be set. Binary and other radixes are not allowed.

The following seven conditions are exceptions; they are not affected by the Radix command:

1. The radix for entering a new radix is always decimal.
2. Format specifiers given with the Display Expression command or any of the Watch Statement commands override the current radix.
3. Addresses output by the Assemble, Dump, Enter, Examine Symbol, and Unassemble commands are always shown in hexadecimal.
4. In assembly mode, all values are shown in hexadecimal.
5. The display radix for Dump, Watch Memory, and Tracepoint Memory commands is always hexadecimal if the size is bytes, words, or double words, and always decimal if the size is integers, unsigned integers, short reals, long reals, or 10-byte reals.
6. The input radix for the Enter commands with the prompting method is always hexadecimal if the size is bytes, words, or double words, and always decimal if the size is integers, unsigned integers, short reals, long reals, or 10-byte reals. The current radix is used for all values given as part of a list, except real numbers, which must be entered in decimal.
7. The register display is always in hexadecimal.

■ Mouse

You cannot change the input radix with the mouse.

■ Keyboard

You cannot change the input radix using a keyboard command.

■ Dialog

To change the input radix using a dialog command, enter a command line with the following syntax:

N[[*radixnumber*]]

The *radixnumber* can be 8 (octal), 10 (decimal), or 16 (hexadecimal). The default radix when you start the CodeView debugger is 10 (decimal), unless your main program is written with MASM, in which case the default radix is 16 (hexidecimal). If you give the Radix command with no argument, the debugger displays the current radix.

■ Examples

```
>N10
>N
10
>? prime
107
>

>N8     ;* C example
>? prime
0153
>

>N16    ;* FORTRAN example
>? prime
#006b
>

>N8     ;* BASIC example
>? prime
&153
>
```

```
The above examples show how
107
decimal, stored in the variable
prime,
would be displayed with different radices. Examples are
taken from different languages; there is no logical connection
between the radix and the language used in each example.


>N8
>? 34,i
28
>N10
>? 28,i
28
>N16
>? 1C,i
28
>
```

In the above example, the same number is entered in different radixes, but the `i` format specifier is used to display the result as a decimal integer in all three cases. See Chapter 7, "Examining Data and Expressions," for more information on format specifiers.

11.4 Redraw Command

The Redraw command can be used only in window mode. It redraws the CodeView screen. This command is seldom necessary, but you might need it if the output of the program being debugged temporarily disturbs the CodeView display.

■ Mouse

You cannot redraw the screen using the mouse.

■ Keyboard

You cannot redraw the screen using a keyboard command.

■ Dialog

To redraw the screen using a dialog command, enter a command line with the following syntax:

@

11.5 Screen Exchange Command

The Screen Exchange command allows you to temporarily switch from the debugging screen to the output screen.

The CodeView debugger will use either screen flipping or screen swapping to store the output and debugging screens. See Chapter 2, "Getting Started," for an explanation of flipping and swapping.

■ Mouse

The Screen Exchange command cannot be executed with the mouse.

■ Keyboard

The Screen Exchange command cannot be executed with a keyboard command.

■ Dialog

To execute the Screen Exchange command from the dialog window, enter a command line with the following syntax:

\

The output screen appears. Press any key when you are ready to return to the debugging screen.

11.6 Search Command

The Search command allows you to search for a regular expression in a source file. The expression to be found is specified either in a dialog box or as an argument to a dialog command. Once you have found an expression, you can also search for the next or previous occurrence of the expression.

Regular expressions are a method of specifying variable text patterns. A pattern can be used to search for text strings that match the pattern. This method is similar to the DOS method of using wild-card characters in file names. Regular expressions are explained in detail in Appendix B.

You can use the Search command without understanding regular expressions. Since text strings are the simplest form of regular expressions, you can simply enter a string of characters as the expression you want to find. For example, you could enter `COUNT` if you wanted to search for the word "COUNT" in the source file.

The following characters have special meanings in regular expressions: backslash (\), asterisk (*), left bracket ([), period (.), dollar sign ($), and caret (^). To find strings containing these characters, you must precede the characters with a backslash; this cancels their special meanings.

For example, you would use `\*` to find `x*y`. The periods in the relational operators must also be preceded by a backslash.

The Case Sense selection from the Options menu has no effect on searches for regular expressions.

Important

When you search for the next occurrence of a regular expression, the CodeView debugger searches to the end of the file, then wraps around and begins again at the start of the file. This can have unexpected results if the expression occurs only once. When you give the command repeatedly, nothing seems to happen. Actually, the debugger is repeatedly wrapping around and finding the same expression each time.

■ Mouse

To find a regular expression with the mouse, point to Search on the menu bar, press a mouse button and drag the highlight down to the Find selection, then release the button. A dialog box appears, asking for the regular expression to be found. Type the expression, and press either the ENTER key or a mouse button. The CodeView debugger starts searching at the current cursor position and puts the cursor at the next line containing the regular expression. An error message appears if the expression is not found. If you are in assembly mode, the debugger automatically switches to source mode when the expression is found.

After you have found a regular expression, you can search for the next or previous occurrence of the expression. Point to Search on the menu bar, press a mouse button and drag the highlight down to the Next or Previous selection, then release the button. The cursor will move to the next or previous match of the expression.

You can also search the executable code for a label (such as a routine name or an assembly-language label). Point to Search on the menu bar, press a mouse button and drag the highlight down to the Label selection, then release the button. A dialog box appears, asking for the label to be found. Type the label name, and press either the ENTER key or a mouse button. The cursor will move to the line containing the label. This selection differs from other search selections because it searches executable code rather than source code. The CodeView debugger will switch to assembly mode, if necessary, to display a label in a library routine or assembly-language module.

■ Keyboard

To find a regular expression with a keyboard command, press ALT-S to open the Search menu, then press ALT-F to select Find. A dialog box appears, asking for the regular expression to be found. Type the expression, and press the ENTER key. The CodeView debugger starts searching at the current cursor position and puts the cursor at the next line containing the regular expression. An error message appears if the expression is not found. If you are in assembly mode, the debugger automatically switches to source mode when the expression is found.

After you have found a regular expression, you can search for the next or previous occurrence of the expression. Press ALT-S to open the Search menu, then press ALT-N to select Next or ALT-P to select Previous. The cursor will move to the next or previous match of the expression.

You can also search the executable code for a label (such as a routine name or an assembly-language label). Press ALT-S to open the Search menu, then press ALT-L to select Label. A dialog box appears, asking for the label to be found. Type the label name, and press the ENTER key. The cursor will move to the line containing the label. This selection differs from other search selections because it searches executable code rather than source code. The CodeView debugger will switch to assembly mode, if necessary, to display a label in a library routine or assembly-language module.

■ Dialog

To find a regular expression using a dialog command, enter a command line with the following syntax:

/[[regularexpression]]

If *regularexpression* is given, the CodeView debugger searches the source file for the first line containing the expression. If no argument is given, the debugger searches for the next occurrence of the last regular expression specified.

In window mode, the CodeView debugger starts searching at the current cursor position and puts the cursor at the next line containing the regular expression. In sequential mode, the debugger starts searching at the last source line displayed. It puts the source line where the expression is found on the screen. An error message appears if the expression is not found. If you are in assembly mode, the CodeView debugger automatically switches to source mode when the expression is found.

You cannot search for a label with the dialog version of the Search command, but using the View command with the label as an argument has the same effect.

11.7 Shell Escape Command

The Shell Escape command allows you to exit from the CodeView debugger to a DOS shell. You can execute DOS commands or programs from within the debugger, or you can exit from the debugger to DOS while retaining your current debugging context.

The Shell Escape command works by saving the current processes in memory and then executing a second copy of **COMMAND.COM**. The **COMSPEC** environment variable is used to locate a copy of **COMMAND.COM**.

Opening a shell requires a significant amount of free memory (usually more than 200K). This is because the CodeView debugger, the symbol table, **COMMAND.COM**, and the program being debugged must all be saved in memory. If you do not have enough memory, an error message will appear. Even if you have enough memory to start a new shell, you may not have enough memory left to execute large programs from the shell.

If you change directories while working in the shell, make sure you return to the original directory before returning to the CodeView debugger. If you don't, the debugger may not be able to find and load source files when it needs them.

Note

In order to use the Shell Escape command, the executable file being debugged must release the memory it does not need. Programs compiled with the Microsoft FORTRAN Optimizing Compiler do this automatically if the FORTRAN start-up code has been executed. You must execute into the program before using the Shell Escape command; for example, enter `G main` after starting the CodeView debugger.

You cannot use the Shell Escape command with assembler programs unless the program specifically releases memory using the DOS function call #4A (Set Block). The same thing can be accomplished by linking the assembler program with the **/CPARMAXALLOC** link option. If the program has not released memory, the CodeView debugger will print this message: `Not enough memory`.

■ Mouse

To open a DOS shell with the mouse, point to File on the menu bar, press a mouse button and drag the highlight down to the Shell selection, then release the button. If there is enough memory to open the shell, the DOS screen will appear. You can execute any DOS internal command or any program. When you are ready to return to the debugging session, type the command `exit` (in any combination of uppercase and lowercase letters).

The debugging screen will appear with the same status it had when you left it.

■ Keyboard

To open a DOS shell using a keyboard command, press ALT-F to open the File menu, then press ALT-S to select Shell. If there is enough memory to open the shell, the DOS screen will appear. You can execute any DOS internal command or any program. When you are ready to return to the debugging session, type the command `exit` (in any combination of uppercase and lowercase letters). The debugging screen will appear with the same status it had when you left it.

■ Dialog

To open a DOS shell using a dialog command, enter a command line with the following syntax:

![[*command*]]

If you want to exit to DOS and execute several programs or commands, enter the command with no arguments. The CodeView debugger executes a new copy of **COMMAND.COM** and the DOS screen appears. You can run programs or DOS internal commands. When you are ready to return to the debugger, type the command `exit` (in any combination of uppercase and lowercase letters). The debugging screen will appear with the same status it had when you left it.

If you want to execute a program or DOS internal command from within the CodeView debugger, enter the Shell Escape command (**!**) followed by the name of the command or program you want to execute. The output screen appears and the debugger executes the command or program. When the output from the command or program is finished, the message `Press any key to continue...` appears at the bottom of the screen. Press a key to make the debugging screen reappear with the same status it had when you left it.

■ Examples

```
>!
```

In the above example, the CodeView debugger saves the current debugging context and executes a copy of **COMMAND.COM**. The DOS screen appears and you can enter any number of commands. To return to the debugger, enter `exit`.

```
>!DIR a:*.for
```

In the above example, the DOS internal command `DIR` is executed with the argument `a:*.for`. The directory listing will be followed by a prompt telling you to press any key to return to the CodeView debugging screen.

```
>!CHKDSK a:
```

In the above example, the DOS external command `CHKDSK` is executed, and the status of the disk in Drive A is displayed in the dialog window. The program name specified could be for any executable file, not just for a DOS external program.

11.8 Tab Set Command

The Tab Set command sets the width in spaces that the CodeView debugger fills for each tab character. The default tab is eight spaces. You might want to set a smaller tab size if your source code has so many levels of indentation that source lines extend beyond the edge of the screen. This command has no effect if your source code was written with an editor that indents with spaces rather than with tab characters.

■ Mouse

You cannot set the tab size using the mouse.

■ Keyboard

You cannot set the tab size using a keyboard command.

■ Dialog

To set the tab size using a dialog command, enter a command line with the following syntax:

 # number

The *number* is the new number of characters for each tab character. In window mode, the screen will be redrawn with the new tab width when you enter the command. In sequential mode, any output of source lines will reflect the new tab size.

■ Example

```
>.
32:                 IF (X(I)) .LE. X(J)) GOTO 301
>#4
>.
32:         IF (X(I)) .LE. X(J)) GOTO 301
>
```

In the above example, the Source Line (.) command is used to show the source line with the default tab width of eight spaces. Next the Tab Set command is used to set the tab width to four spaces. The Source Line command then shows the same line.

11.9 Redirection Commands

The CodeView debugger provides several options for redirecting commands from or to devices or files. In addition to the redirection commands, several other commands are relevant only when used with redirected files. The redirection commands and related commands are discussed in Sections 11.9.1–11.9.4.3.

■ Mouse

None of the redirection or related commands can be executed with the mouse.

■ Keyboard

None of the redirection or related commands can be executed with keyboard commands.

■ Dialog

The redirection commands are entered with dialog commands, as shown in Sections 11.9.1–11.9.4.3.

11.9.1 Redirecting CodeView Input

■ Syntax

< *devicename*

The Redirected Input command causes the CodeView debugger to read all subsequent command input from a device, such as another terminal or a file. The sample session supplied with most versions of the debugger is an example of commands being redirected from a file.

■ Examples

```
><COM1
```

The above example redirects commands from the device (probably a remote terminal) designated as `COM1` to the CodeView terminal.

```
><INFILE.TXT
```

The above example redirects command input from file `INFILE.TXT` to the CodeView debugger. You might use this command to prepare a CodeView session for someone else to run. You create a text file containing a series of CodeView commands separated by carriage-return–line-feed combinations

or semicolons. When you redirect the file, the debugger will execute the commands to the end of the file. One way to create such a file is to redirect commands from the CodeView debugger to a file (see Section 11.9.3) and then edit the file to eliminate the output and add comments.

11.9.2 Redirecting CodeView Output

■ Syntax

[[**T**]>[[>]] *devicename*

The Redirected Output command causes the CodeView debugger to write all subsequent command output to a device, such as another terminal, a printer, or a file. The term "output" includes not only the output from commands, but the command characters that are echoed as you type them.

The optional **T** indicates that the output should be echoed to the CodeView screen. If you do not use the **T**, you will not be able to see your commands as you type them. Normally, you will want to use the **T** if you are redirecting output to a file, so that you can see what you are typing. However, if you are redirecting output to another terminal, you may not want to see the output on the CodeView terminal.

The optional second greater-than symbol appends the output to an existing file. If you redirect output to an existing file without this symbol, the existing file will be replaced.

■ Examples

```
>>COM1
```

The above example's output is redirected to the device designated as `COM1` (probably a remote terminal). One situation in which you might want to do this is when you are debugging a graphics program and want CodeView commands to be displayed on a remote terminal at the same time that the program display appears on the originating terminal.

```
>T>OUTFILE.TXT
.
.
```

```
.
>>CON
.
.
.
```

The above example's output is redirected to the file OUTFILE.TXT. You might want to do this in order to keep a permanent record of a CodeView session. Note that the optional **T** is used so that the session will be echoed to the CodeView screen as well as to the file. After redirecting some commands to a file, output is returned to the console (terminal) with the command >CON.

```
>T>>OUTFILE.TXT
```

If, later in the session, you want to redirect more commands to the same file, use the double greater-than symbol, as in the above example, to append the output to the existing file.

11.9.3 Redirecting CodeView Input and Output

■ Syntax

= *devicename*

The Redirected Input and Output command causes the CodeView debugger to write all subsequent command output to a device and to simultaneously receive input from the same device. This is practical only if the device is a remote terminal.

Redirecting input and output works best if you start in sequential mode (using the /**T** option), since this eliminates unnecessary screen exchanges. The CodeView debugger's window interface has little purpose in this situation, since the remote terminal can act only as a sequential (nonwindow) device.

■ **Example**

```
>=COM1
```

In the above example, output and input are redirected to the device designated as COM1. This would be useful if you wanted to enter debugging commands and see the debugger output on a remote terminal, while entering program commands and viewing program output on the terminal where the debugger is running.

11.9.4 Commands Used with Redirection

The following commands are intended for use when redirecting commands to or from a file. Although they are always available, these commands have little practical use during a normal debugging session.

Command	Action
Comment (*)	Displays comment
Delay (:)	Delays execution of commands from a redirected file
Pause (")	Interrupts execution of commands from a redirected file until a key is pressed

11.9.4.1 Comment Command

■ **Syntax**

**comment*

The Comment command is an asterisk (*) followed by text. The CodeView debugger echoes the text of the comment to the screen (or other output device). This command is useful in combination with the redirection commands when saving a commented session, or when writing a commented session that will be redirected to the debugger.

■ Examples

```
>T>OUTPUT.TXT
>* Dump first 20 bytes of screen buffer
>D #B800:0 L 20
B800:0000 54 17 6F 17 20 17 72 17 65 17 74 17 75 17 72 17 T.o. .r.e.t.u.r.
B800:0010 6E 17 20 17                                     n. .
>
```

In the above example, the user is sending a copy of a CodeView session to file OUTPUT.TXT. Comments are added to explain the purpose of the command. The text file will contain commands, comments, and command output.

```
* Dump first 20 bytes of screen buffer
D #B800:0 L 20
.
.
.
< CON
```

The above example illustrates another way to use the Comment command. You can put comments into a text file of commands that will be executed automatically when you redirect the file into the CodeView debugger. In this example, an editing program was used to create the text file called INPUT.TXT.

```
><INPUT.TXT
>* Dump first 20 bytes of screen buffer
>D #B800:0 L 20
B800:0000 54 17 6F 17 20 17 72 17 65 17 74 17 75 17 72 17 T.o. .r.e.t.u.r.
B800:0010 6E 17 20 17                                     n. .
.
.
.
>< CON
>
```

When you read the file into the debugger, using the Redirected Input command, you will see the comment, then the output from the command, as in the above example.

11.9.4.2 Delay Command

■ **Syntax**

```
:
```

The Delay command interrupts execution of commands from a redirected file and waits about half a second before continuing. You can put multiple Delay commands on a single line to increase the length of the delay. The delay is the same length, regardless of the processing speed of the computer.

■ **Example**

```
: ;* That was a short delay...
::::: ;* That was a longer delay...
```

In the above example from a text file that might be redirected into the CodeView debugger, the Delay command is used to slow execution of the redirected file.

11.9.4.3 Pause Command

■ **Syntax**

```
"
```

The Pause command interrupts execution of commands from a redirected file and waits for the user to press a key. Execution of the redirected commands begins as soon as a key is pressed.

■ **Example**

```
* Press any key to continue
"
```

In the above example from a text file that might be redirected into the CodeView debugger, a Comment command is used to prompt the user to press a key. Then the Pause command is used to halt execution until the user responds.

```
>* Press any key to continue
>"
```

The above example shows the output when the text is redirected into the debugger. The next CodeView prompt will not appear until the user presses a key.

Part 2
Utilities

12 Linking Object Files with LINK 267

13 Managing Libraries with LIB 297

14 Automating Program Development with MAKE 315

15 Using EXEPACK, EXEMOD, SETENV, and ERROUT 331

Chapter 12
Linking Object Files with LINK

12.1 Specifying Files for Linking 269
12.1.1 Specifying File Names 269
12.1.2 Linking with the LINK Command Line 270
12.1.3 Linking with the LINK Prompts 272
12.1.4 Linking with a Response File 274
12.1.5 How LINK Searches for Libraries 275
12.1.6 LINK Memory Requirements 277
12.1.7 Terminating the LINK Session 278
12.2 Specifying Linker Options 279
12.2.1 Viewing the Options List (/HE) 280
12.2.2 Pausing during Linking (/P) 280
12.2.3 Displaying Linker Process Information (/I) 281
12.2.4 Packing Executable Files (/E) 282
12.2.5 Listing Public Symbols (/M) 283
12.2.6 Including Line Numbers in the Map File (/LI) 283
12.2.7 Preserving Case Sensitivity (/NOI) 284
12.2.8 Ignoring Default Libraries (/NOD) 284
12.2.9 Controlling Stack Size (/ST) 284
12.2.10 Setting the Maximum Allocation Space (/CP) 285
12.2.11 Setting Maximum Number of Segments (/SE) 286
12.2.12 Setting the Overlay Interrupt (/O) 287

12.2.13 Ordering Segments (/DO) 287
12.2.14 Controlling Data Loading (/DS) 288
12.2.15 Controlling Executable-File Loading (/HI) 288
12.2.16 Preserving Compatibility (/NOG) 289
12.2.17 Preparing for Debugging (/CO) 289
12.3 Linker Operation 290
12.3.1 Alignment of Segments 291
12.3.2 Frame Number 291
12.3.3 Order of Segments 291
12.3.4 Combined Segments 292
12.3.5 Groups 293
12.3.6 Fixups 293
12.4 Using Overlays 294
12.4.1 Restrictions on Overlays 295
12.4.2 Overlay-Manager Prompts 295

The Microsoft Overlay Linker (**LINK**) is used to combine object files into a single executable file. It can be used with object files compiled or assembled on 8086/8088 or 80286 machines. The format of input to the linker is the Microsoft Relocatable Object-Module Format (OMF), which is based on the Intel® 8086 OMF.

The output file from **LINK** (that is, the executable file) is not bound to specific memory addresses. Thus, the operating system can load and execute this file at any convenient address. **LINK** can produce executable files containing up to 1 megabyte of code and data.

The following sections explain how to run the linker and specify options that control its operation.

12.1 Specifying Files for Linking

Instead of using high-level language commands to invoke the linker, you can use the **LINK** command to invoke **LINK** directly. You can specify the input required for this command in one of three ways:

1. By placing it on the command line.
2. By responding to prompts.
3. By specifying a file containing responses to prompts. This type of file is known as a "response file."

12.1.1 Specifying File Names

You can use any combination of uppercase and lowercase letters for the file names you specify on the **LINK** command line or give in response to the **LINK** command prompts. For example, **LINK** considers the following three file names to be equivalent:

```
abcde.fgh
AbCdE.FgH
ABCDE.fgh
```

If you specify file names without extensions, **LINK** uses the following default file-name extensions:

File

Type	Default Extension
Object	.OBJ
Executable	.EXE
Map	.MAP
Library	.LIB

You can override the default extension for a particular command-line field or prompt by specifying a different extension. To enter a file name that has *no* extension, type the name followed by a period.

■ Examples

Consider the following two file specifications:

```
ABC.
ABC
```

If you use the first file specification, **LINK** assumes that the file has no extension. If you use the second file specification, **LINK** uses the *default* extension for that prompt.

12.1.2 Linking with the LINK Command Line

Use the following form of the **LINK** command to specify input on the command line:

LINK [[*options*]] *objectfiles*[[,[[*executablefile*]][,[[*mapfile*]][,[[*libraryfiles*]]]]]][;]

The *objectfiles* field allows you to specify the names of the object files you are linking. At least one object-file name is required. A space or plus sign (+) must separate each pair of object-file names. **LINK** automatically supplies the **.OBJ** extension when you give a file name without an extension. If your object file has a different extension, or if it appears in another directory or on another disk, you must give the full name—including the extension and path name—for the file to be found. If **LINK** cannot find a given object file, it displays a message and waits for you to change disks.

The *executablefile* field allows you to specify the name of the executable file. If the file name you give does not have an extension, **LINK** automatically adds **.EXE** as the extension. You can give any file name you like; however, if you are specifying an extension, you should always use **.EXE**,

because DOS expects executable files to have either this extension or the **.COM** extension.

The *mapfile* field allows you to specify the name of the map file, if you are creating one. To include public symbols and their addresses in the map file, specify the **/MAP** option on the **LINK** command line. See Section ?.?, "Listing Public Symbols," for a description of the **/MAP** option and Section ?.?, "Formats for Listings," for a description of map-file formats. If you specify a map-file name without an extension, **LINK** automatically adds an extension of **.MAP**. **LINK** creates the map file in the current working directory unless you specify a path name for the map file.

The *libraryfiles* field allows you to specify the name of a library that you want linked to the other object file(s). When you compile a source file for a high-level language, the compiler places the name of a library in the object file that it creates. The library name corresponds to the memory-model and floating-point options that you chose on the compiler command line, or the defaults for options you did not explicitly choose. The linker automatically searches for a library with this name. Because of this, you do not need to give library names on the **LINK** command line unless you want to add the names of other libraries, search for libraries in different locations, or override the use of the library named in the object file.

The *options* field allows you to specify the linker options described in Sections ?.? through ?.?. You do not have to give any *options* when you run the linker. If you specify *options*, you can put them anywhere on the command line.

You can select the default for any command-line field by omitting the file name or names before the commas. The only exception to this is the default for *mapfile*: if you use a comma as a placeholder for the map file on the command line, **LINK** creates a map file. This file has the same base name as the executable file. Use NUL for the map-file name if you do not want to produce a map file.

You can also select default responses by using a semicolon (;). The semicolon tells **LINK** to use the defaults for all remaining fields.

If you do not give all file names on the command line, or if you do not end the command line with a semicolon, the linker prompts you for the files you omitted, using the prompts described in Section ?.?, "Linking with the LINK Prompts."

If you do not specify a drive or directory for a file, the linker assumes that the file is on the current drive and directory. If you want the linker to create files in a different location than the current drive and directory, you must specify the new drive and directory for each such file on the command line.

See Sections ?.? through ?.? for a description of the input you give in each command-line field. See Section ?.? (old 4.5.2) for a description of the rules for entering file names in the **LINK** command fields.

■ Examples

```
LINK FUN+TEXT+TABLE+CARE, ,FUNLIST, XLIB.LIB
```

The command line above causes **LINK** to load and link the object modules FUN.OBJ, TEXT.OBJ, TABLE.OBJ, and CARE.OBJ, and search for unresolved references in the library file XLIB.LIB and the default libraries. By default, the executable file produced by **LINK** is named FUN.EXE. **LINK** also produces a map file named FUNLIST.MAP.

```
LINK FUN,,;
```

This command line produces a map file named FUN.MAP, since a comma appears as a placeholder for the *mapfile* specification on the command line.

```
LINK FUN,;
LINK FUN;
```

These command lines do not produce a map file, since commas do not appear as placeholders for the *mapfile* specification.

12.1.3 Linking with the LINK Prompts

If you want to use the LINK prompts to specify input to the linker, start the linker by typing **LINK** at the DOS command level. **LINK** prompts you for the input it needs by displaying the following lines, one at a time:

```
Object Modules [.OBJ]:
Run File [basename.EXE]:
List File [NUL.MAP]:
Libraries [.LIB]:
```

LINK waits for you to respond to each prompt before printing the next one. Section ?.? (old 4.5.2) gives the rules for specifying file names in response to these prompts.

The responses you give to the **LINK** command prompts correspond to the fields on the **LINK** command line. (See Section 4.4.1 for a discussion of the **LINK** command line.) The following list shows these correspondences:

Prompt	Command-Line Field
"Object Modules"	*objectfiles*
"Run File"	*executablefile*
"List File"	*mapfile*
"Libraries"	*libraryfiles*

If a plus sign (+) is the last character that you type on a response line, the prompt appears on the next line, and you can continue typing responses. In this case, the plus sign must appear at the end of a complete file or library name, path name, or drive name.

Default Responses

To select the default response to the current prompt, type a carriage return without giving a file name. The next prompt will appear.

To select default responses to the current prompt and all remaining prompts, type a semicolon (;) followed immediately by a carriage return. After you enter a semicolon, you cannot respond to any of the remaining prompts for that link session. Use this option to save time when you want to use the default responses. Note, however, that you cannot enter a semicolon in response to the "Object Modules" prompt, because there is no default response for that prompt.

The following list shows the defaults for the other linker prompts:

Prompt	Default
"Run File"	The name of the first object file submitted for the "Object Modules" prompt, with the **.EXE** extension replacing the **.OBJ** extension

"List File"	The special file name **NUL.MAP**, which tells **LINK** *not* to create a map file
"Libraries"	The default libraries encoded in the object module (see Section 4.4.1.5, "Specifying Libraries."

12.1.4 Linking with a Response File

To operate the linker with a response file, you must set up the response file and then type the following:

LINK @ *responsefile*

Here *responsefile* specifies the name or pathname of the response file that starts the linker.

A response file contains responses to the **LINK** prompts. You may give options at the end of any response or place them on one or more separate lines. The responses must be in the same order as the **LINK** prompts discussed in Section ?.? (old 4.4.2). Each new response must appear on a new line or must begin with a comma; however, you can extend long responses across more than one line by typing a plus sign (+) as the last character of each incomplete line.

You can also enter the name of a response file after any **LINK** command prompt or at any position in the **LINK** command line.

LINK treats the input from the response file just as if you had entered it in response to prompts or in a command line. It treats any carriage-return–line-feed combination in the response file the same as if you had pressed the ENTER key in response to a prompt or included a comma in a command line.

You can use options and command characters in the response file in the same way as you would use them in responses you type at the keyboard. For example, if you type a semicolon on the line of the response file corresponding to the "Run File" prompt, **LINK** uses the default responses for the executable file and for the remaining prompts.

When you enter the **LINK** command with a response file, each **LINK** prompt is displayed on your screen with the corresponding response from your response file. If the response file does not include a line with a file name, semicolon, or carriage return for each prompt, **LINK** displays the missing prompts and waits for you to enter responses. When you type an

acceptable response, **LINK** continues the link session.

■ Example

Assume that the following response file is named `FUN.LNK`:

```
FUN TEXT TABLE CARE
/PAUSE /MAP
FUNLIST
GRAF.LIB
```

You can type the following command to run **LINK** and tell it to use the responses in `FUN.LNK`:

```
LINK @FUN.LNK
```

The response file tells **LINK** to load the four object modules `FUN`, `TEXT`, `TABLE`, and `CARE`. **LINK** produces an executable file named `FUN.EXE` and a map file named `FUNLIST.MAP`. The `/PAUSE` option tells **LINK** to pause before it produces the executable file so that you can swap disks, if necessary. The `/MAP` option tells **LINK** to include public symbols and addresses in the map file. **LINK** also links any needed routines from the library file `GRAF.LIB`. See the discussions of the **/PAUSE** and **/MAP** options in Section 4.6.2 and 4.6.5, respectively, for more information about these options.

12.1.5 How LINK Searches for Libraries

LINK searches for the default library first in the current working directory, then in any directory specified in the **LIB** environment variable.

Since the object file already contains the name of the correct library, you are not required to specify a library on the **LINK** command line or in response to the **LINK** "Libraries" prompt unless you want to do one of the following:

- Add the names of additional libraries to be searched
- Search for libraries in different locations
- Override the use of one or more default libraries

Searching Additional Libraries

You can tell **LINK** to search additional libraries by specifying one or more library files on the command line or in response to the "Libraries" prompt. **LINK** searches these libraries *before* it searches default libraries. It searches these libraries in the order in which you specify them.

If the library name you give includes a path specification, **LINK** searches only that directory for the library.

If you specify only a library name (without a path specification), **LINK** searches in the following locations to find the given library file:

- The current working directory
- Any path specifications or drive names that you give on the command line or type in response to the "Libraries" prompt, in the order in which they appear (see Section 4.5.3.2)
- The locations given by the **LIB** environment variable

LINK automatically supplies the **.LIB** extension if you omit it from a library-file name. If you want to link a library file that has a different extension, be sure to specify the extension.

Searching Different Locations for Libraries

You can tell **LINK** to search additional locations for libraries by giving a drive name or path specification in the *libfield* on the command line or in response to the "Libraries" prompt. You can specify up to 16 additional paths. If you give more than 16 paths, **LINK** ignores the additional paths without displaying an error message.

LINK searches for the default libraries in the same order as for libraries given on the command line. See Section 4.5.3.1, "Searching Additional Libraries," for more information.

Overriding Libraries Named in Object Files

If you do not want to link with the library whose name is included in the object file, you can give the name of a different library instead. You might want to specify a different library name in the following cases:

- If you assigned a "custom" name to a standard library when you set up your libraries,
- If you want to link with a library that supports a different math package than the math package you gave on the compiler command line (or the default).

If you specify a new library name on the **LINK** command line, the linker searches the new library to resolve external references before it searches the library specified in the object file.

If you want the linker to ignore the library whose name is included in the object file, you must use the **/NOD** option. This option tells **LINK** to ignore the default-library information that is encoded in the object files created by high-level language compilers. Use this option with caution; see the discussion of the **/NOD** option in Section 4.6.8 for more information.

■ Example

```
LINK

Object Modules [.OBJ]: FUN TEXT TABLE CARE
Run File [FUN.EXE]:
List File [NUL.MAP]:
Libraries [.LIB]: C:\TESTLIB\ NEWLIBV3
```

This example links four object modules to create an executable file named `FUN.EXE`. **LINK** searches `NEWLIBV3.LIB` before searching the default libraries to resolve references. To locate `NEWLIBV3.LIB` and the default libraries, the linker searches the current working directory, then the `C:\TESTLIB\` directory, and finally, the locations given by the **LIB** environment variable.

12.1.6 LINK Memory Requirements

LINK uses available memory for the link session. If the files to be linked create an output file that exceeds available memory, **LINK** creates a temporary disk file to serve as memory. This temporary file is handled in one of the following ways, depending on the DOS version:

- If the linker is running on DOS Version 3.0 or later, it uses a DOS system call to create a temporary file with a unique name in the current working directory.

- If the linker is running on a version of DOS prior to 3.0, it creates a temporary file named **VM.TMP**.

When the linker creates a temporary disk file, you will see the message

```
Temporary file tempfile has been created.
Do not change diskette in drive, letter
```

Here, *tempfile* is ".\" followed by either **VM.TMP** or a name generated by DOS, and *letter* is the current drive.

The message *Do not change diskette in drive* will not appear if the current drive is a hard disk. After this message appears, do not remove the disk from the drive specified by *letter* until the link session ends. If the disk is removed, the operation of **LINK** is unpredictable, and you may see the following message:

```
unexpected end-of-file on scratch file
```

When this happens, rerun the link session. The temporary file created by **LINK** is a working file only. **LINK** deletes it at the end of the link session.

Note

Do not give any of your own files the name **VM.TMP**. The linker displays an error message if it encounters an existing file with this name.

12.1.7 Terminating the LINK Session

To terminate a link session, press CONTROL-C while **LINK** is working or while you are entering responses to **LINK** prompts. If you realize that you entered an incorrect response to a previous prompt, press CONTROL-C to exit **LINK** and begin again. You can use the normal DOS editing keys to correct entries at the current prompt.

12.2 Specifying Linker Options

This section explains how to use linker options to specify and control the tasks performed by **LINK**. All options begin with the linker's option character, the forward slash (/).

When you use the **LINK** command line to invoke **LINK**, options can appear at the end of the line or after individual fields on the line. However, they must precede the comma that separates each field from the next.

If you respond to the individual prompts for the **LINK** command, you can specify linker options at the end of any response. When you specify more than one option, you can either group the options at the end of a single response or distribute the options among several responses. Every option must begin with the slash character (/), even if other options precede it on the same line. Similarly, in a response file, options can appear on a line by themselves or after individual response lines.

Abbreviations

Since linker options are named according to their functions, some of these options are quite long. You can abbreviate the options to save space and effort. Be sure that your abbreviation is unique so that the linker can determine which option you want. (The minimum legal abbreviation for each option is indicated in the description of the option.)

For example, since several options begin with the letters "NO," abbreviations for those options must be longer than "NO" to be unique. You cannot use "NO" as an abbreviation for the **/NOIGNORECASE** option, since **LINK** cannot tell which of the options beginning with "NO" you intend. The shortest legal abbreviation for this option is **/NOI**.

Abbreviations must be begin with the first letter of the option and must be continuous through the last letter typed. No gaps or transpositions are allowed.

Numerical Arguments

Some linker options take numeric arguments. A numeric argument can be any of the following:

- A decimal number from 0 to 65,535.
- An octal number from 0 to 177777. A number is interpreted as octal if it starts with 0. For example, the number `10` is a decimal number, but the number `O10` is an octal number, equivalent to 8 in decimal.
- A hexadecimal number from 0 to FFFF. A number is interpreted as hexadecimal if it starts with **0X** (with no base specifier before the pound sign). For example, `0X10` is a hexadecimal number, equivalent to 16 in decimal.

12.2.1 Viewing the Options List (/HE)

■ Option

/HE[LP]

The **/HELP** option causes **LINK** to display a list of the available options on the screen. This gives you a convenient reminder of the available options. Do not give a file name when using the **/HELP** option.

12.2.2 Pausing during Linking (/P)

■ Option

/P[AUSE]

Unless you instruct it otherwise, **LINK** performs the linking session from beginning to end without stopping. The **/PAUSE** option tells **LINK** to pause in the link session before it writes the executable (**.EXE**) file to disk. This option allows you to swap disks before **LINK** writes the executable file.

If you specify the **/PAUSE** option, **LINK** displays the following message before it creates the run file:

```
About to generate .EXE file
Change diskette in drive letter and press <ENTER>
```

The *letter* corresponds to the current drive. **LINK** resumes processing when you press the ENTER key.

Note

Do not remove the disk that will receive the list file or the disk used for the temporary file.

If a temporary file is created on the disk you plan to swap, you should press CONTROL-C to terminate the **LINK** session. Rearrange your files so that the temporary file and the executable file can be written to the same disk. Then try linking again.

12.2.3 Displaying Linker Process Information (/I)

- **Option**

/I[NFORMATION]

The **/INFORMATION** option tells the linker to display information about the linking process, including the phase of linking and the names of the object files being linked. This option is useful if you want to determine the locations of the object files being linked and the order in which they are linked.

Output from this option is sent to the standard error output. You can use the **ERROUT** utility, described in Section ?.?, to redirect output to any file or device.

The following example shows a sample of the linker output when the **/I** and **/MAP** options are specified on the **LINK** command line:

```
**** PASS ONE ****
TEST.OBJ(test.for)
**** LIBRARY SEARCH ****
LLIBFOR7.LIB(wr)
LLIBFOR7.LIB(fmtout)
LLIBFOR7.LIB(ldout)
    .
    .
    .
**** ASSIGN ADDRESSES ****
  1 segment "TEST_TEXT" length 122H bytes
  2 segment "_DATA" length 912H bytes
  3 segment "CONST" length 12H bytes
```

```
    .
    .
    .
**** PASS TWO ****
TEST.OBJ(test.for)
LLIBFOR7.LIB(wr)
LLIBFOR7.LIB(fmtout)
LLIBFOR7.LIB(ldout)
    .
    .
    .
**** WRITING EXECUTABLE ****
```

12.2.4 Packing Executable Files (/E)

■ Option

/E[XEPACK]

The **/EXEPACK** option directs **LINK** to remove sequences of repeated bytes (typically null characters) and optimize the load-time relocation table before creating the executable file. (The load-time relocation table is a table of references, relative to the start of the program, each of which changes when the executable image is loaded into memory and an actual address for the entry point is assigned.) Executable files linked with this option may be smaller, and thus load faster, than files linked without this option. However, you cannot use the Symbolic Debug Utility (SYMDEB) or the CodeView window-oriented debugger to debug with packed files. The **EXEPACK** option strips symbolic information from the input file and notifies you of this with a warning message.

The **/EXEPACK** option does not always give a significant saving in disk space, and may sometimes actually increase file size. Programs that have a large number of load-time relocations (about 500 or more) and long streams of repeated characters are usually shorter if packed. If you're not sure whether your program meets these conditions, link it both ways and compare the results.

12.2.5 Listing Public Symbols (/M)

- **Option**

/M[AP]

You can list all public (global) symbols defined in an object file or files by using the **/MAP** option. When you invoke LINK with the **/MAP** option, a list of public symbols will be added to *mapfile*. If you do not use this option, then *mapfile* will contain only a list of segments.

If no map file was specified when you invoke LINK (for example, you take the default of NUL), then the **/MAP** option has no effect.

12.2.6 Including Line Numbers in the Map File (/LI)

- **Option**

/LI[NENUMBERS]

You can include the line numbers and associated addresses of your source program in the map file by using the **/LI** option. Ordinarily the map file does not contain line numbers.

To produce a map file with line numbers, you must give **LINK** an object file (or files) with line-number information. You can use the **/Zd** option with any Microsoft compiler to include line numbers in the object file. If you give **LINK** an object file without line-number information, the **/LI** option has no effect.

The **/LI** option forces **LINK** to create a map file, even if you did not explicitly tell the linker to create a map file. By default, the file is given the same base name as the executable file, plus the extension **.MAP**. You can override the default name by specifying a new map file on the **LINK** command line or in response to the "List File" prompt.

12.2.7 Preserving Case Sensitivity (/NOI)

- **Option**

/NOI[GNORECASE]

By default, **LINK** treats uppercase letters and lowercase letters as equivalent. Thus ABC, abc, and Abc are considered the same name. When you use the **/NOI** option, the linker distinguishes between uppercase letters and lowercase letters, and considers ABC, abc, and Abc to be three separate names. Since names in some high-level languages are not case sensitive, this option can have minimal importance. However, in some languages, such as C, case is significant. If you plan to link your files from other high-level language with C routines, you may want to use this option.

12.2.8 Ignoring Default Libraries (/NOD)

- **Option**

/NOD[EFAULTLIBRARYSEARCH]

The **/NOD** option tells **LINK** *not* to search any library specified in the object file to resolve external references.

In general, FORTRAN programs do not work correctly without a standard FORTRAN library. Thus, if you use the **/NOD** option, you should explicitly specify the name of a standard library.

12.2.9 Controlling Stack Size (/ST)

- **Option**

/ST[ACK]:*number*

The **/ST** option allows you to specify the size of the stack for your program. The *number* is any positive value (decimal, octal, or hexadecimal) up to 65,535 (decimal). It represents the size, in bytes, of the stack.

If you get a stack-overflow message, you may need to increase the size of the stack. In contrast, if your program uses the stack very little, you may save some space by decreasing the stack size.

Note

You can also use the **EXEMOD** utility, described in Section ?.?, to change the default stack size in executable files by modifying the executable-file header. The format of the executable-file header is discussed in the *Microsoft MS-DOS Programmer's Reference* and in other reference books on DOS.

12.2.10 Setting the Maximum Allocation Space (/CP)

■ Option

/CP[ARMAXALLOC]:*number*

The **/CP** option sets the maximum number of 16-byte paragraphs needed by the program when it is loaded into memory. The operating system uses this value when allocating space for the program before loading it. The option is useful when you want to execute another program from within your program and you need to reserve space for the executed program.

LINK normally requests the operating system to set the maximum number of paragraphs to 65,535. Since this represents more memory than could be available under DOS 3.X, the operating system always denies the request and allocates the largest contiguous block of memory it can find. If the **/CP** option is used, the operating system allocates no more space than the option specified. This means any additional space in memory is free for other programs.

The *number* can be any integer value in the range 1 to 65,535. If *number* is less than the minimum number of paragraphs needed by the program, **LINK** ignores your request and sets the maximum value equal to the minimum. The minimum number of paragraphs needed by a program is never less than the number of paragraphs of code and data in the program. To free more memory for programs compiled in the medium and large memory models, link with **/CP:1**; this leaves no space for the near heap.

Note

You can change the maximum allocation after linking by using the **EXEMOD** utility, which modifies the executable-file header, as described in Section ?.?. The format of the executable-file header is discussed in the *Microsoft MS-DOS Programmer's Reference* and in other reference books on DOS.

12.2.11 Setting Maximum Number of Segments (/SE)

■ Option

/SE[GMENTS]:*number*

The **/SE** option controls the number of segments that the linker allows a program to have. The default is 128, but you can set *number* to any value (decimal, octal, or hexadecimal) in the range 1 to 1024 (decimal).

For each segment, the linker must allocate some space to keep track of segment information. By using a relatively low segment limit as a default (128), the linker avoids having to allocate a large amount of storage space for all programs.

When you set the segment limit higher than 128, the linker allocates more space for segment information. This option allows you to raise the segment limit for programs with a large number of segments. For programs with fewer than 128 segments, you can keep the storage requirements of the linker at the lowest level possible by setting the segment *number* to reflect the actual number of segments in the program.

If the number of segments allocated is too high for the amount of memory **LINK** has available to it, you will see the following error message:

```
segment limit too high
```

To specify a number of segments that will fit in the amount of memory available, set the segment lower and relink the object files.

12.2.12 Setting the Overlay Interrupt (/O)

■ Option

/O[VERLAYINTERRUPT]:*number*]

By default, the interrupt number used for passing control to overlays is 63 (3F hexadecimal). The **/OVERLAYINTERRUPT** option allows the user to select a different interrupt number.

The *number* can be a decimal number from 0 to 255, an octal number from octal 0 to octal 0377, or a hexadecimal number from hexadecimal 0 to hexadecimal FF. Numbers that conflict with DOS interrupts can be used; however, their use is not advised.

In general, you should not use **/OVERLAYINTERRUPT** with programs. The exception to this guideline would be a program that uses overlays and spawns another program using overlays; in this case, each program should use a separate overlay-interrupt number, meaning at least one of the programs should be compiled with **/OVERLAYINTERRUPT**.

12.2.13 Ordering Segments (/DO)

■ Option

/DO[SSEG]

The **/DOSSEG** option is automatically enabled by a special object module record in Microsoft language libraries. If you are linking to one of these libraries, then you do not need to specify this option.

The **/DOSSEG** option forces segments to be ordered as follows:

1. All segments with a class name ending in **CODE**
2. All other segments outside **DGROUP**
3. **DGROUP** segments, in the following order:

a. Any segments of class **BEGDATA** (this class name is reserved for Microsoft use)

b. Any segments not of class **BEGDATA**, **BSS**, or **STACK**

c. Segments of class **BSS**

d. Segments of class **STACK**

12.2.14 Controlling Data Loading (/DS)

■ Option

/DS[ALLOCATE]

By default, **LINK** loads all data starting at the low end of the data segment. At run time, the **DS** (data segment) register is set to the lowest possible address to allow the entire data segment to be used.

Use the **/DSALLOCATE** option to tell **LINK** to load all data starting at the high end of the data segment instead. In this case, the **DS** register is set at run time to the lowest data-segment address that contains program data.

The **/DSALLOCATE** option is typically used with the **/HIGH** option, discussed in the next section, to take advantage of unused memory within the data segment. You can allocate any available memory below the area specifically allocated for **DGROUP**, using the same **DS** register.

Warning

This option should be used only with assembly-language programs.

12.2.15 Controlling Executable-File Loading (/HI)

■ **Option**

/HI[GH]

The executable file can be placed either as low or as high in memory as possible. Use of the **/HIGH** option causes **LINK** to place the executable file as high as possible in memory. Without the **/HIGH** option, **LINK** places the executable file as low as possible.

Note

This option should be used only with assembly-language programs.

12.2.16 Preserving Compatibility (/NOG)

■ **Option**

/NOG[ROUPASSOCIATION]

The **/NOG** option causes the linker to ignore group associations when assigning addresses to data and code items. It is provided primarily for compatibility with previous versions of the linker (Versions 2.02 and earlier) and early versions of Microsoft language compilers.

Note

This option should be used only with assembly-language programs.

12.2.17 Preparing for Debugging (/CO)

■ **Option**

/CO[DEVIEW]

The **/CO** option is used to prepare for debugging with the CodeView window-oriented debugger. This option tells the linker to prepare a special executable file containing symbolic data and line-number information.

You can run this executable file outside the CodeView debugger; the extra data in the file will be ignored. However, to keep file size to a minimum, use the special-format executable file only for debugging; then you can link a separate version without the **/CO** option after the program is debugged.

12.3 Linker Operation

LINK performs the following steps to combine object modules and produce an executable file:

1. Reads the object modules submitted
2. Searches the given libraries, if necessary, to resolve external references
3. Assigns addresses to segments
4. Assigns addresses to public symbols
5. Reads code and data in the segments
6. Reads all relocation references in object modules
7. Performs fixups
8. Outputs an executable file (executable image and relocation information)

Steps 5, 6, and 7 are performed concurrently: in other words, **LINK** will move back and forth between these steps before it progresses to Step 8.

The "executable image" contains the code and data that constitute the executable file. The "relocation information" is a list of references, relative to the start of the program, each of which changes when the executable image is loaded into memory and an actual address for the entry point is assigned.

The following sections explain the process **LINK** uses to concatenate segments and resolve references to items in memory.

12.3.1 Alignment of Segments

LINK uses a segment's alignment type to set the starting address for the segment. The alignment types are **BYTE**, **WORD**, **PARA**, and **PAGE**. These correspond to starting addresses at byte, word, paragraph, and page boundaries, representing addresses that are multiples of 1, 2, 16, and 256, respectively. The default alignment is **PARA**.

When **LINK** encounters a segment, it checks the alignment type before copying the segment to the executable file. If the alignment is **WORD**, **PARA**, or **PAGE**, **LINK** checks the executable image to see if the last byte copied ends at an appropriate boundary. If not, **LINK** pads the image with extra null bytes.

12.3.2 Frame Number

LINK computes a starting address for each segment in a program. The starting address is based on a segment's alignment and the sizes of the segments already copied to the executable file. The address consists of an offset and a "canonical frame number." The canonical frame number specifies the address of the first paragraph in memory that contains one or more bytes of the segment. A frame number is always a multiple of 16 (a paragraph address). The offset is the number of bytes from the start of the paragraph to the first byte in the segment. For **BYTE** and **WORD** alignments, the offset may be nonzero. The offset is always zero for **PARA** and **PAGE** alignments.

The frame number of a segment can be obtained from the map file created by **LINK**. You determine the frame number simply by converting the last digit of the segment's "Start" address to "0". For example, a "Start" address of 0C0A6 gives us a frame number of 0C0A0.

12.3.3 Order of Segments

LINK copies segments to the executable file in the same order that it encounters them in the object files. This order is maintained throughout the program unless **LINK** encounters two or more segments having the same class name. Segments having identical class names belong to the same class type, and are copied as a contiguous block to the executable

file.

The **/DOSSEG** option may change the way in which segments are ordered.

12.3.4 Combined Segments

LINK uses combine types to determine whether or not two or more segments sharing the same segment name should be combined into one large segment. The valid combine types are **PUBLIC**, **STACK**, **COMMON**, and **PRIVATE**.

If a segment has combine type **PUBLIC**, **LINK** automatically combines it with any other segments having the same name and belonging to the same class. When **LINK** combines segments, it ensures that the segments are contiguous and that all addresses in the segments can be accessed using an offset from the same frame address. The result is the same as if the segment were defined as a whole in the source file.

LINK preserves each individual segment's alignment type. This means that even though the segments belong to a single, large segment, the code and data in the segments do not lose their original alignment. If the combined segments exceed 64K, **LINK** displays an error message.

If a segment has combine type **STACK**, **LINK** carries out the same combine operation as for **PUBLIC** segments. The only exception is that **STACK** segments cause **LINK** to copy an initial stack-pointer value to the executable file. This stack-pointer value is the offset to the end of the first stack segment (or combined stack segment) encountered.

If a segment has combine-type **COMMON**, **LINK** automatically combines it with any other segments having the same name and belonging to the same class. When **LINK** combines **COMMON** segments, however, it places the start of each segment at the same address, creating a series of overlapping segments. The result is a single segment no larger than the largest segment combined.

A segment has combine type **PRIVATE** only if no explicit combine type is defined for it in the source file. **LINK** does not combine private segments.

12.3.5 Groups

Groups allow segments to be addressed relative to the same frame address. When **LINK** encounters a group, it adjusts all memory references to items in the group so that they are relative to the same frame address.

Segments in a group do not have to be contiguous, belong to the same class, or have the same combine type. The only requirement is that all segments in the group fit within 64K.

Groups do not affect the order in which the segments are loaded. Unless you use class names and enter object files in the right order, there is no guarantee that the segments will be contiguous. In fact, **LINK** may place segments that do not belong to the group in the same 64K of memory. Although **LINK** does not explicitly check that all segments in a group fit within 64K of memory, **LINK** is likely to encounter a fixup overflow error if this requirement is not met.

12.3.6 Fixups

Once the starting address of each segment in a program is known and all segment combinations and groups have been established, **LINK** can "fix up" any unresolved references to labels and variables. To fix up unresolved references, **LINK** computes an appropriate offset and segment address and replaces the temporary values generated by the assembler with the new values.

LINK carries out fixups for the types of references shown in the following list:

Type of Reference	Description
Short	Occurs in **JMP** instructions that attempt to pass control to labeled instructions in the same segment or group. The target instruction must be no more than 128 bytes from the point of reference. **LINK** computes a signed, 8-bit number for this reference. It displays an error message if the target instruction belongs to a different segment or group (has a different frame address), or if the target is more than 128 bytes distant in either direction.

Near self relative	Occurs in instructions that access data relative to the same segment or group. **LINK** computes a 16-bit offset for this reference. It displays an error if the data are not in the same segment or group.
Near segment relative	Occurs in instructions that attempt to access data in a specified segment or group, or relative to a specified segment register. **LINK** computes a 16-bit offset for this reference. It displays an error message if the offset of the target within the specified frame is greater than 64K or less than 0, or if the beginning of the canonical frame of the target is not addressable.
Long	Occurs in **CALL** instructions that attempt to access an instruction in another segment or group. **LINK** computes a 16-bit frame address and 16-bit offset for this reference. **LINK** displays an error message if the computed offset is greater than 64K or less than 0, or if the beginning of the canonical frame of the target is not addressable.

The size of the value to be computed depends on the type of reference. If **LINK** discovers an error in the anticipated size of a reference, it displays a fixup overflow message. This can happen, for example, if a program attempts to use a 16-bit offset to reach an instruction which is more than 64K away. It can also occur if all segments in a group do not fit within a single 64K block of memory.

12.4 Using Overlays

You can direct **LINK** to create an overlaid version of a program. In an overlaid version of a program, specified parts of the program (known as "overlays") are loaded only if and when they are needed. These parts share the same space in memory. Only code is overlaid; data are never overlaid. Programs that use overlays usually require less memory, but they run more slowly because of the time needed to read and reread the code from

disk into memory.

You specify overlays by enclosing them in parentheses in the list of object files that you submit to the linker. Each module in parentheses represents one overlay. For example, you could give the following object-file list in the *objectfiles* field of the **LINK** command line:

```
a + (b+c) + (e+f) + g + (i)
```

In this example, the modules `(b+c)`, `(e+f)`, and `(i)` are overlays. The remaining modules, and any drawn from the run-time libraries, constitute the resident part (or root) of your program. Overlays are loaded into the same region of memory, so only one can be resident at a time. Duplicate names in different overlays are not supported, so each module can appear only once in a program.

The linker replaces calls from the root to an overlay, and calls from an overlay to another overlay with an interrupt (followed by the module identifier and offset). By default, the interrupt number is 63 (3F hexadecimal). You can use the **/OVERLAYINTERRUPT** option of the **LINK** command to change the interrupt number.

12.4.1 Restrictions on Overlays

You can overlay only modules to which control is transferred and returned by a standard 8086 long (32-bit) call/return instruction. However, calls to subroutines modified with the **NEAR** attribute are short (16-bit) calls. This means that you cannot overlay modules containing **NEAR** subroutines if other modules call those subroutines.

12.4.2 Overlay-Manager Prompts

The overlay manager is part of the language's run-time library. If you specify overlays during linking, the code for the overlay manager is automatically linked with the other modules of your program.

When the executable file is run, the overlay manager searches for that file whenever another overlay needs to be loaded. The overlay manager first searches for the file in the current directory; then, if it does not find the file, the manager searches the directories listed in the **PATH** environment variable. When it finds the file, the overlay manager extracts the overlay modules specified by the root program. If the overlay manager cannot find an overlay file when needed, it prompts the user to enter the file name.

Even with overlays, the linker produces only *one* **.EXE** file. This file is opened again and again, as long as the overlay manager needs to extract new overlay modules.

For example, assume that an executable program called PAYROLL.EXE, which does not exist in either the current directory or the directories specified by **PATH**, uses overlays. If the user runs it by entering a complete path specification, the overlay manager displays the following message when it attempts to load overlay files:

```
Cannot find PAYROLL.EXE
Please enter new program spec:
```

The user can then enter the drive or directory, or both, where PAYROLL.EXE is located. For example, if the file is located in directory \EMPLOYEE\DATA\ on Drive B, the user could enter B:\EMPLOYEE\DATA\ or simply \EMPLOYEE\DATA\ if the current drive is B.

If the user later removes the disk in Drive B and the overlay manager needs to access the overlay again, it does not find PAYROLL.EXE, and displays the following message:

```
Please insert diskette containing B:\EMPLOYEE\DATA\PAYROLL.EXE
in drive B: and strike any key when ready.
```

After the overlay file has been read from the disk, the overlay manager displays the following message:

```
Please restore the original diskette.
Strike any key when ready.
```

Chapter 13
Managing Libraries with LIB

13.1 Managing Libraries 300
13.1.1 Managing Libraries with the LIB Command Line 300
13.1.1.1 Specifying the Library File 300
13.1.1.2 Specifying a Page Size 301
13.1.1.3 Giving LIB Commands 301
13.1.1.4 Specifying a Cross-Reference-Listing File 303
13.1.1.5 Specifying an Output Library 304
13.1.2 Managing Libraries with the LIB Prompts 305
13.1.2.1 Extending Lines 306
13.1.2.2 Using Default Responses 306
13.1.3 Managing Libraries with a Response File 307
13.1.4 Terminating the LIB Session 308
13.2 Peforming Library Management Tasks with LIB 308
13.2.1 Creating a Library File 309
13.2.2 Changing a Library File 310
13.2.3 Adding Library Modules 310
13.2.4 Deleting Library Modules 311
13.2.5 Replacing Library Modules 311
13.2.6 Extracting Library Modules 311

13.2.7 Moving Library Modules 311
13.2.8 Combining Libraries 312
13.2.9 Creating a Cross-Reference-Listing File 312
13.2.10 Performing Consistency Checks 313
13.2.11 Setting the Library-Page Size 313

The Microsoft Library Manager (**LIB**) is a utility designed to help you create, organize, and maintain run-time libraries. Run-time libraries are collections of compiled or assembled functions that provide a common set of useful routines. After you have linked a program with a run-time-library file, that program can call a run-time routine exactly as if the function were included in the program. The call to the run-time routine is resolved by finding that routine in the library file.

Run-time libraries are created by combining separately compiled object files into one library file. Library files are usually identified by their **.LIB** extension, although other extensions are allowed.

In addition to accepting DOS object files and library files, **LIB** can read the contents of 286 XENIX archives and Intel-style libraries and combine their contents with DOS libraries. To see how you can add the contents of a 286 XENIX archive or an Intel-style library to a DOS library, use the command symbol described in Section ?.?.

Once an object file is incorporated into a library, it becomes an object "module." **LIB** makes a distinction between object files and object modules: an object "file" exists as an independent file, while an object "module" is part of a larger library file. An object file can have a full path name, including a drive designation, directory-path name, and file-name extension (usually **.OBJ**). Object modules have only a name. For example, `B:\RUN\SORT.OBJ` is an object-file name, while `SORT` is the corresponding object-module name.

Using **LIB**, you can create a new library file, add object files to an existing library, delete library modules, replace library modules, and create object files from library modules. **LIB** also lets you combine the contents of two libraries into one library file.

The command syntax is straightforward; you can give **LIB** all the input it requires directly from the command line. Once you have learned how **LIB** works and what input it needs, you can use one of the two alternative methods of invoking **LIB**, described in Sections ?.? and ?.?. The alternative methods allow you to enter input in response to prompts instead of having to enter the input on the **LIB** command line.

13.1 Managing Libraries

You run **LIB** by typing the **LIB** command on the DOS command line. You can specify the input required for this command in one of three ways:

1. By placing it on the command line.
2. By responding to prompts.
3. By specifying a file containing responses to prompts. (This type of file is known as a "response file.")

13.1.1 Managing Libraries with the LIB Command Line

You can start **LIB** and specify all the input it needs from the command line. In this case, the **LIB** command line has the following form:

LIB *oldlibrary* [[**/PAGESIZE:***number*]] [[*commands*]][,[*listfile*]][,[*newlibrary*]]][;]

To tell **LIB** to use the default responses for the remaining fields, use a semicolon (**;**) after any field except the *oldlibrary* field. The semicolon should be the last character on the command line.

Sections ?.? through ?.? describe the input that you give in each command-line field.

13.1.1.1 Specifying the Library File

■ **Field**

oldlibrary[;]

The *oldlibrary* field allows you to specify the name of the existing library to be used. Usually library files are named with the **.LIB** extension. You can omit the **.LIB** extension when you give the library-file name since **LIB** assumes that the file-name extension is **.LIB**. If your library file does not have the **.LIB** extension, be sure to include the extension when you give the library-file name. Otherwise, **LIB** cannot find the file.

Path names are allowed with the library-file name. You can give **LIB** the path name of a library file in another directory or on another disk. There is no default for this field. **LIB** produces an error message if you do not give a file name.

If you give the name of a library file that does not exist, **LIB** displays the following prompt:

```
Library file does not exist. Create?
```

Type `y` to create the library file, or `n` to terminate **LIB**. This message is suppressed if the nonexistent library name you give is followed immediately by commands, a comma, or a semicolon.

If you type an *oldlibrary* name and follow it immediately with a semicolon (**;**), **LIB** performs only a consistency check on the given library. A consistency check tells you whether all the modules in the library are in usable form. **LIB** prints a message only if it finds an invalid object module; no message appears if all modules are intact.

13.1.1.2 Specifying a Page Size

- **Option**

[[**/PAGESIZE:***number*]]

The **/PAGESIZE** option allows you to specify the library-page size of a new library or change the library-page size of an existing library. The page size of a library affects the alignment of modules stored in the library. Modules in the library are always aligned to start at a position that is a multiple of the page size (in bytes) from the beginning of the file. The default page size for a new library is 16 bytes. See Section ?.? "Setting the Library-Page Size," for more information.

13.1.1.3 Giving LIB Commands

■ Field

[[*commands*]]

The *commands* field allows you to specify the command symbols for manipulating modules. To use this field, type a command symbol such as (+, −, −+, *, or −*), followed immediately by a module name or an object-file name. You can specify more than one operation in this field, in any order. If you leave this field blank, **LIB** does not make any changes to *oldlibrary*.

Command Symbol	Meaning
+	The add command symbol. A plus sign makes an object file the last module in the library file. Immediately following the plus sign, give the name of the object file. You can use path names for the object file. **LIB** automatically supplies the **.OBJ** extension, so you can omit the extension from the object-file name. You can also use the plus sign to combine two libraries. When you give a library name following the plus sign, a copy of the contents of the given library is added to the library file being modified. You must include the **.LIB** extension when you give a library-file name. Otherwise, **LIB** uses the default **.OBJ** extension when it looks for the file.
−	The delete command symbol. A minus sign deletes a module from the library file. Immediately following the minus sign, give the name of the module to be deleted. A module name has no path name and no extension.
−+	The replace command symbol. A minus sign followed by a plus sign replaces a module in the library. Following the replacement symbol, give the name of the module to be replaced. Module names have no path names and no extensions. To replace a module, **LIB** deletes the given module, then appends the object file having the same name as the module. The object file is assumed to have an **.OBJ** extension and to reside in the current working directory.

*	The copy command symbol. An asterisk followed by a module name copies a module from the library file into an object file of the same name. The module remains in the library file. When **LIB** copies the module to an object file, it adds the **.OBJ** extension and the drive designation and path name of the current working directory to the module name to form a complete object-file name. You cannot override the **.OBJ** extension, drive designation, or path name given to the object file. However, you can later rename the file or copy it to whatever location you like.
–*	The move command symbol. A minus sign followed by an asterisk moves an object module from the library file to an object file. This operation is equivalent to copying the module to an object file, as described above, then deleting the module from the library.

13.1.1.4 Specifying a Cross-Reference-Listing File

■ Field

[[*listfile*]]

The *listfile* field allows you to specify a file name for a cross-reference-listing file. You can specify a full path name for the listing file to cause it to be created outside your current working directory. You can give the listing file any name and any extension. **LIB** does not supply a default extension if you omit the extension.

A cross-reference-listing file contains the following two lists:

1. An alphabetical list of all public symbols in the library.

   Each symbol name is followed by the name of the module in which it is referenced.

2. A list of the modules in the library.

   Under each module name is an alphabetical listing of the public symbols defined in that module. The default when you omit the response to this prompt is the special file name **NUL**, which tells **LIB** *not* to create a listing file.

13.1.1.5 Specifying an Output Library

■ Field

[[*newlib*]]

The *newlibrary* field allows you to specify the name of the new library file that will contain the specified changes. This prompt appears only if you specify changes to the library in the *commands* field. The default is the current library-file name.

If you do not specify a new library-file name, the original, unmodified library is saved in a library file with the same name but with a **.BAK** extension replacing the **.LIB** extension.

■ Examples

```
LIB LANG-+HEAP;
```

The example above uses the replace command symbol (–+) to instruct **LIB** to replace the HEAP module in the library LANG.LIB. **LIB** deletes the HEAP module from the library, then appends the object file HEAP.OBJ as a new module in the library. The semicolon at the end of the command line tells **LIB** to use the default responses for the remaining prompts. This means that no listing file is created and that the changes are written to the original library file instead of creating a new library file.

```
LIB LANG-HEAP+HEAP;

LIB LANG+HEAP-HEAP;
```

The examples above perform the same function as the first example in this section, but in two separate operations, using the add (+) and delete (–) command symbols. The effect is the same for these examples because delete operations are always carried out before add operations, regardless of the order of the operations in the command line. This order of execution prevents confusion when a new version of a module replaces an old version in the library file.

```
LIB FOR;
```

The example above causes **LIB** to perform a consistency check of the library file FOR.LIB. No other action is performed. **LIB** displays any consistency errors it finds and returns to the operating-system level.

```
LIB LANG,LCROSS.PUB
```

This example tells **LIB** to perform a consistency check of the library file LANG.LIB and then create a cross-reference-listing file named LCROSS.PUB.

```
LIB FIRST -*STUFF *MORE, ,SECOND
```

The last example instructs **LIB** to move the module STUFF from the library FIRST.LIB to an object file called STUFF.OBJ. The module STUFF is removed from the library in the process. The module MORE is copied from the library to an object file called MORE.OBJ; the module remains in the library. The revised library is called SECOND.LIB. It contains all the modules in FIRST.LIB except STUFF, which was removed by using the move command symbol (-*). The original library, FIRST.LIB, remains unchanged.

13.1.2 Managing Libraries with the LIB Prompts

If you want to respond to individual prompts to give input to **LIB**, start **LIB** at the DOS command level by typing **LIB**. **LIB** prompts you for the input it needs by displaying the following four messages, one at a time:

```
Library name:
Operations:
List file:
Output library:
```

LIB waits for you to respond to each prompt, then prints the next prompt.

The responses you give to the **LIB** command prompts correspond to the fields on the **LIB** command line. (See Sections ?.? and ?.? for a discussion of the **LIB** command line.) The following list shows these correspondences:

Prompt	Command-Line Field

"Library name"	The *oldlibrary* field and the optional **PAGESIZE:***number* option (see Sections ?.? and ?.?, respectively). If you want to perform a consistency check on the library, type a semicolon (**;**) immediately after the library name.
"Operations"	Any of the commands allowed in the *commands* field (see Section 5.3.1.3).
"List file"	The *listfile* field.
"Output library"	The *newlib* field.

13.1.2.1 Extending Lines

If you have many operations to perform during a library session, use the ampersand command symbol (&) to extend the operations line. Give the ampersand symbol after an object-module or object-file name; do not put the ampersand between an operation's symbol and a name.

The ampersand causes **LIB** to repeat the "Operations" prompt, allowing you to type more operations.

13.1.2.2 Using Default Responses

After any entry but the first, use a single semicolon (**;**) followed immediately by a carriage return to select default responses to the remaining prompts. You can use the semicolon command symbol with the command-line and response-file methods of invoking **LIB**, but it is not necessary since **LIB** supplies the default responses wherever you omit responses.

The following list shows the defaults for **LIB** prompts:

Prompt	Default
"Operations"	No operation; no change to library file.
"List file"	The special file name **NUL**, which tells **LIB** not to create a listing file.
"Output library"	The current library name. This prompt appears only if you specify at least one operation at the "Operations" prompt.

13.1.3 Managing Libraries with a Response File

To operate **LIB** with a response **file,** you **must** first **set** response file and then type the following at the DOS command line:

LIB @ *responsefile*

The *responsefile* is the name of a response file. The response-file name can be qualified with a drive and directory specification to name a response file from a directory other than the current working directory.

You can also enter the name of a response file at any position in a command line or after any of the linker prompts. The input from the response file will be treated exactly as if it had been entered in command lines or after prompts. A carriage-return–line-feed combination in the response file is treated the same as pressing the ENTER key in response to a prompt, or using a comma in a command line.

Before you use this method, you must set up a response file containing responses to the **LIB** prompts. This method lets you conduct the library session without typing responses to prompts at the keyboard.

A response file has one text line for each prompt. Responses must appear in the same order as the command prompts appear. Use command symbols in the response file the same way you would use responses typed on the keyboard. You can type an ampersand at the end of the response to the "Operations" prompt and continue typing operations on the next line.

When you run **LIB** with a response file, the prompts are displayed with the responses from the response file. If the response file does not contain responses for all the prompts, **LIB** uses the default responses.

■ Example

```
LIBFOR
+CURSOR+HEAP-HEAP*FOIBLES
CROSSLST
```

The contents of the above response file cause **LIB** to delete the module HEAP from the `LIBFOR.LIB` library file, extract the module `FOIBLES` and place it in an object file named `FOIBLES.OBJ`, and append the object files `CURSOR.OBJ` and `HEAP.OBJ` as the last two modules in the library. Finally, **LIB** creates a cross-reference-listing file named `CROSSLST`.

13.1.4 Terminating the LIB Session

You can press CONTROL-C at any time during a library session to terminate the session and return to DOS. If you notice that you have entered an incorrect response at a previous prompt, you should press CONTROL-C to exit **LIB** and begin again. You can use the normal DOS editing keys to correct errors at the current prompt.

13.2 Peforming Library Management Tasks with LIB

You can perform a number of library-management functions with **LIB**, including the following tasks:

- Create a library file
- Delete modules
- Extract a module and place it in a separate object file
- Extract a module and delete it
- Append an object file as a module of a library, or append the contents of a library
- Replace a module in the library file with a new module
- Produce a listing of all public symbols in the library modules

For each library session, **LIB** reads and interprets the user's commands as listed below. It determines whether a new library is being created or an existing library is being examined or modified.

1. **LIB** processes deletion and extraction commands (if any).

   LIB does not actually delete modules from the existing file. Instead, it marks the selected modules for deletion, creates a new library file, and copies only the modules *not* marked for deletion into the new library file.

2. **LIB** processes any addition commands. Like deletions, additions are not performed on the original library file. Instead, the additional modules are appended to the new library file. (If there were no deletion or extraction commands, a new library file is created in the addition stage by copying the original library file.)

As **LIB** carries out these commands, it reads the object modules in the library, checks them for validity, and gathers the information necessary to build a library index and a listing file. The linker uses the library index to search the library.

The listing file contains a list of all public symbols in the index and the names of the modules in which they are defined. **LIB** produces the listing file only if you ask for it during the library session.

LIB never makes changes to the original library; it copies the library and makes changes to the copy. Therefore, when you terminate **LIB** for any reason, you do not lose your original file. It also means that when you run **LIB**, enough space must be available on your disk for both the original library file and the copy.

When you change a library file, **LIB** lets you specify a different name for the file containing the changes. If you use this option, the modified library is stored under the name you give, and the original, unmodified version is preserved under its own name. If you choose not to give a new name, **LIB** gives the modified file the original library name, but keeps a backup copy of the original library file. This copy has the extension **.BAK** instead of **.LIB**.

13.2.1 Creating a Library File

To create a new library file, give the name of the library file you want to create in the *oldlibrary* field of the command line or at the "Library name" prompt. **LIB** supplies the **.LIB** extension.

The name of the new library file must not be the name of an existing file. If it is, **LIB** assumes that you want to change the existing file. When you give the name of a library file that does not currently exist, **LIB** displays the following prompt:

```
Library file does not exist. Create?
```

Type `y` to create the file, or `n` to terminate the library session. This message is suppressed if the nonexistent library name you give is followed immediately by commands, a comma, or a semicolon.

You can specify a page size for the library when you create it. The default page size is 16 bytes. See the Section ?.?, "Setting the Library-Page Size," for a discussion of this option.

Once you have given the name of the new library file, you can insert object modules into the library by using the add command symbol (+) in the *commands* field of the command line or at the "Operations" prompt. You can also add the contents of another library, if you wish. See Section ?.?, "Adding Library Modules," and Section ?.?, "Combining Libraries," for a discussion of these options.

13.2.2 Changing a Library File

You can change an existing library file by giving the name of the library file at the "Library name" prompt. All operations you specify in the *oldlibrary* field of the command line or at the "Operations" prompt are performed on that library.

However, **LIB** lets you keep both the unchanged library file and the newly changed version, if you like. You can do this by giving the name of a new library file in the *newlibrary* field of the command line or at the "Output library" prompt. The changed library file is stored under the new library-file name, while the original library file remains unchanged.

If you don't give a new file name, the changed version of the library file replaces the original library file. Even in this case, **LIB** saves the original, unchanged library file with the extension **.BAK** instead of **.LIB**. Thus, at the end of the session you have two library files: the changed version with the **.LIB** extension and the original, unchanged version with the **.BAK** extension.

13.2.3 Adding Library Modules

Use the add command symbol (+) in the *commands* field of the command line or at the "Operations" prompt to add an object module to a library. Give the name of the object file to be added, without the **.OBJ** extension, immediately following the plus sign.

LIB strips the drive designation and the extension from the object-file specification, leaving only the base name. This becomes the name of the object module in the library. For example, if the object file `B:\CURSOR` is added to a library file, the name of the corresponding object module is `CURSOR`.

Object modules are always added to the end of a library file.

13.2.4 Deleting Library Modules

Use the delete command symbol (–) in the *commands* field of the command line or at the "Operations" prompt to delete an object module from a library. After the minus sign, give the name of the module to be deleted. A module name does not have a path name or extension; it is simply a name, such as CURSOR.

13.2.5 Replacing Library Modules

Use the replace command symbol (–+) in the *commands* field to replace a module in the library. Following the replace command symbol, give the name of the module to be replaced. Remember that module names do not have path names or extensions.

To replace a module, **LIB** deletes the given module, then appends the object file having the same name as the module. The object file is assumed to have an **.OBJ** extension and to reside in the current working directory.

13.2.6 Extracting Library Modules

Use the copy command symbol (*) followed by a module name in the *commands* field to extract a module from the library file into an object file of the same name. The module remains in the library file. When **LIB** copies the module to an object file, it adds the **.OBJ** extension and the drive designation and path name of the current working directory to the module name. This forms a complete object-file name. You cannot override the **.OBJ** extension, drive designation, or path name given to the object file, but you can later rename the file or extract it to any location you like.

13.2.7 Moving Library Modules

Use the move command symbol (–*) in the *commands* field to move an object module from the library file to an object file. This operation is equivalent to copying the module to an object file, then deleting the module from the library.

13.2.8 Combining Libraries

You can add the contents of a library to another library by using the add command symbol (+) with a library-file name instead of an object-file name in the *commands* field. In the *commands* field of the command line or at the "Operations" prompt, give the add command symbol (+) followed by the name of the library whose contents you wish to add to the library being changed. When you use this option, you must include the **.LIB** extension of the library-file name. Otherwise, **LIB** assumes that the file is an object file and looks for the file with an **.OBJ** extension.

In addition to allowing DOS libraries as input, **LIB** also accepts 286 XENIX archives and Intel-format libraries. Therefore, you can use **LIB** to convert libraries from either of these formats to the DOS format.

LIB adds the modules of the library to the end of the library being changed. Note that the added library still exists as an independent library. **LIB** copies the modules without deleting them.

Once you have added the contents of a library or libraries, you can save the new, combined library under a new name by giving a new name in the *newlibrary* field of the command line or at the "Output library" prompt. If you omit the "Output library" response, **LIB** saves the combined library under the name of the original library being changed. The original library is saved with the same base name and the extension **.BAK**.

13.2.9 Creating a Cross-Reference-Listing File

Create a cross-reference-listing file by giving a name for the listing file in the *listfile* field of the command line or at the "List file" prompt. If you do not give a listing-file name, **LIB** uses the special file name **NUL**, which means that no listing file is created.

You can give the listing file any name and any extension. To cause the listing file to be created outside your current working directory, you can specify a full path name, including drive designation. **LIB** does not supply a default extension if you omit the extension.

A cross-reference-listing file contains two lists. The first is an alphabetical listing of all public symbols in the library. Each symbol name is followed by the name of the module in which it is referenced.

The second list is an alphabetical list of the modules in the library. Under each module name is an alphabetical listing of the public symbols referenced in that module.

13.2.10 Performing Consistency Checks

When you give only a library name followed by a semicolon in the *oldlibrary* field of the command line or at the "Library name" prompt, **LIB** performs a consistency check, displaying messages about any errors it finds. No changes are made to the library. It is not usually necessary to perform consistency checks, since **LIB** automatically checks object files for consistency before adding them to the library.

To produce a cross-reference-listing file with a consistency check, invoke **LIB**, specify the library name followed by a semicolon, and give the name of the listing file. **LIB** then performs the consistency check and creates the cross-reference-listing file.

13.2.11 Setting the Library-Page Size

You can set the library-page size while you are creating a library or change the page size of an existing library by adding a page-size option after the library-file name in the **LIB** command line or after the new library-file name at the "Library name" prompt. The option has the following form:

/PAGESIZE:*number*

The *number* specifies the new page size. It must be an integer value representing a power of 2 between the values 16 and 32,768. The option name can be abbreviated to **/P:***number*.

The page size of a library affects the alignment of modules stored in the library. Modules in the library are always aligned to start at a position that is a multiple of the page size (in bytes) from the beginning of the file. The default page size is 16 bytes for a new library or the current page size for an existing library.

Note

Because of the indexing technique used by **LIB**, a library with a large page size can hold more modules than a library with a smaller page size. However, for each module in the library, an average of *pagesize*/2 bytes of storage space is wasted. In most cases, a small page size is advantageous; you should use a small page size unless you need to put a very large number of modules in a library.

Another consequence of this indexing technique is that the page size determines the maximum possible size of the **.LIB** file. Specifically, this limit is *number* * 65,536. For example, `/P:16` means that the **.LIB** file has to be smaller than 1 megabyte (16 * 65,536 bytes).

Chapter 14
Automating Program Development with MAKE

14.1 Using MAKE 318
14.2 Creating a MAKE Description File 318
14.3 Automating Program Development 322
14.4 Running MAKE 323
14.5 Specifying MAKE Options 324
14.6 Using Macro Definitions with MAKE 325
14.6.1 Defining and Specifying Macros 325
14.6.2 Using Macros within Macro Definitions 327
14.6.3 Using Special Macros 328
14.7 Defining Inference Rules 329

The Microsoft Program Maintenance Utility (**MAKE**) automates program development. **MAKE** can update an executable file automatically whenever changes are made to one of its source or object files and it can update *any* file whenever changes are made to other, related files.

Before you run **MAKE**, you must create a file containing the information that **MAKE** needs in order to run. This type of file is known as a **MAKE** "description file." The following example shows a **MAKE** description file named SAMPLE:

```
#SAMPLE IS THE NAME OF THIS FILE
SAMPLE.EXE: SAMPLE.OBJ
      LINK SAMPLE;
```

This description file has the following characteristics:

- SAMPLE.EXE is the name of the "outfile." The outfile is the file that you want **MAKE** to update.
- SAMPLE.OBJ is the name of an "infile." An infile is a file that **MAKE** examines in order to determine whether the outfile should be updated. If the infile has changed more recently than the outfile has changed, then **MAKE** will update the outfile.
- LINK SAMPLE; is the command which tells **MAKE** to update the outfile. In the example above, **MAKE** updates SAMPLE.EXE (the outfile) whenever SAMPLE.OBJ (the infile) has been changed.

To update SAMPLE, you would type the following command:

```
MAKE SAMPLE
```

MAKE then compares the last-modification dates of SAMPLE.EXE and SAMPLE.OBJ. If the date for SAMPLE.OBJ is more recent than the date for SAMPLE.EXE, **MAKE** carries out the **LINK** command, LINK SAMPLE;, specified in the description file. This **LINK** command links the SAMPLE.OBJ file, so that the corresponding executable file, SAMPLE.EXE, is updated automatically to reflect the changes to SAMPLE.OBJ.

14.1 Using MAKE

The general procedure for using **MAKE** is as follows:

1. Create a file in which you give **MAKE** the following information:
   a. The name of each outfile that you want it to update
   b. For each outfile, the infiles that must change to cause **MAKE** to update the outfile
   c. The commands that you want **MAKE** to perform when any of the infiles change
2. Run **MAKE**. On the DOS command line, you must specify the name of the **MAKE** description file you have created. (You can also specify options that affect the way in which **MAKE** operates; see Section 6.6 for a description of these options.)

After you invoke **MAKE**, it compares the last-modification date of the infiles with the last-modification date of the corresponding outfiles. If any infile date is more recent than the outfile date, **MAKE** automatically carries out the commands given in the description file and updates the outfile.

The following sections explain how to create a **MAKE** description file and run **MAKE**.

14.2 Creating a MAKE Description File

Since a **MAKE** description file is just a text file, you can use any text editor to create one. You will usually want to give the **MAKE** description file the same file name as the program it updates (with no extension); however, you can use any valid file name.

A **MAKE** description file consists of one or more description blocks, each with the following general form:

```
[[macrodefinition]]
.
.
.
outfile : infile[[,infile...]][[# comment]]
[[# comment]]
    command [[# comment]]
    [[command]] [[# comment]]
    .
    .
    .
```

The following list defines how the fields appearing in a description block are used:

Field	Usage
macrodefinition	Defines one or more **MAKE** macro definitions. See Section 6.7 for an explanation of how to use macro definitions in a **MAKE** description file.
outfile	Specifies the name of a file that you want **MAKE** to update automatically. A colon must separate this field from the *infile* fields.
infile	Specifies the names of any files that the outfile depends on. For example, if the outfile is an executable file, the infiles might be object files; if the outfile is an object file, the infiles might be source files. The line containing the *outfile* and *infile* fields is known as the "dependency line."
command	Specifies the name of an executable file (for example, **LINK**) or a DOS internal command.

Note

One way to remember the **MAKE** description-file format is to think of it in terms of an "if-then" form: if an *outfile* is out of date with respect to any *infile*, or if an *outfile* does not exist, then do *commands*.

The following sections define the rules for using outfile and infile names, commands, comments, and description blocks in a description file.

Outfiles and Infiles

The *outfile* and *infile* fields must contain valid file names. If any file is not on the same drive and in the same directory as the description file, you must include a path specification with the file name.

In any description block, you can give any number of *infile* names, but only one *outfile* name. At least one space must separate each pair of *infile* names. If you have more *infile* names than can fit on one line, type a backslash (\) at the end of the current line, and then continue typing names on the next line.

Commands

The *command* field in a description block can contain any valid DOS command line, consisting of the base name of an **.EXE**, **.COM**, or **.BAT** file or a DOS internal command. You can give any number of commands, but each must begin on a new line and each must appear immediately after a tab or after at least one space.

MAKE carries out this command only if one or more of the infiles in the description block has been changed since the *outfile* was created or most recently updated.

Comments

The *comment* field must contain a number sign #. The number sign (#) is a comment character. **MAKE** ignores all characters that follow the comment character on the same line.

If a comment appears on the same line as the outfile name, it must appear after the *infile* name(s). If a comment appears on a line where a command is expected, the comment character (#) must be the first character on the line; no leading spaces are allowed.

Description Blocks

You can give any number of description blocks in a description file. You must make sure, however, that a blank line appears between the last line of one description block and the first line of the next description block.

The order in which you place the description blocks is important. **MAKE** examines each description block in turn and makes its decision to carry out the command in that block based on the last-modification date of the outfile and infile. If a command in a later description block changes a file used in an earlier description block, **MAKE** has no way to return to that earlier description block to update files that depend on the changed files.

■ Example

```
MOD1.OBJ:      MOD1.ASM
        MASM MOD1;

MOD2.OBJ:      MOD2.C  #Comment allowed after infile
#Comment before command must start in first column
        CL /c /AL MOD2.C  #Comment allowed here

MOD3.OBJ:      MOD3.FOR
        FL /c MOD3.FOR

EXAMPLE.EXE:  MOD1.OBJ MOD2.OBJ MOD3.OBJ
        LINK MOD1+MOD2+MOD3,EXAMPLE,EXAMPLE;
```

The sample description file tells **MAKE** how to update or create four outfiles: `MOD1.OBJ`, `MOD2.OBJ`, `MOD3.OBJ`, and `EXAMPLE.EXE`. To update or create an object file, **MAKE** invokes the appropriate assembler or compiler. To update or create `EXAMPLE.EXE`, **MAKE** will link the three object files.

Note that the description blocks appear in the order in which the outfiles are updated or created. Thus, **MAKE** updates `MOD1.OBJ`, `MOD2.OBJ`, and `MOD3.OBJ` (or creates them, if necessary) before it updates or creates `EXAMPLE.EXE`. Thus, after **MAKE** is run, any changes to the source files will be reflected in `EXAMPLE.EXE`.

The next section further describes how **MAKE** processes files.

14.3 Automating Program Development

Consider a test program called WORK.EXE that is made from two object files, WORK1.OBJ and WORK2.OBJ, and where both object modules must be linked with a library file named LIBV3.LIB. During development, you will sometimes recompile WORK1, and sometimes recompile WORK2; however, WORK.EXE needs to be updated every time you alter the program.

The following block descriptions in a **MAKE** description file named WORK allow you to update WORK.EXE automatically:

```
WORK1.OBJ:  WORK1.C
    CL /c /AL WORK1.C

WORK2.OBJ:  WORK2.FOR
    FL /c WORK2.FOR

WORK.EXE:   WORK1.OBJ WORK2.OBJ \LIB\LIBV3.LIB
    LINK /CO WORK1.OBJ+WORK2.OBJ,WORK,,\LIB\LIBV3.LIB
```

Each time you finish debugging the program's files, invoke **MAKE** with the following command line:

```
MAKE WORK
```

MAKE carries out the following steps (where each step corresponds to a description block):

1. Checks to see if WORK1.C has been changed since the last time WORK1.OBJ was changed (in other words, you've made a change to the source file since the last compile). If so, it carries out the given **CL** command to recompile WORK1.C.
2. Checks WORK2.FOR in the same way it checked WORK1.C in Step 1. Note that if only one of the files has been changed, then only that file is recompiled. For example, if you change WORK1.C but not WORK2.FOR, then only the first file is recompiled; but if each one has been changed since its last compile, then each are now recompiled.
3. Checks to see if either of the object files WORK1.OBJ or WORK2.OBJ, or the library file LIBV3.LIB has been changed since the last time the modules were linked. If either of the object files has been recompiled, or if the library file has been changed, then **MAKE** relinks the program.

If you run **MAKE** with this description file immediately after you create the source files `WORK1.C` and `WORK2.FOR`, **MAKE** carries out Steps 1 and 2 to compile these source files (since in each case the outfile does not exist), then links them in Step 3.

If you invoke **MAKE** again without changing any of the infiles, **MAKE** does not execute any commands.

If you change *one* of the ojbect files `FIRST_WORK1.OBJ` or `FIRST_WORK2.OBJ`, **MAKE** relinks that file and then relinks the program in Step 3.

If you change the library file `LIBV3.LIB`, but make no other changes, **MAKE** skips Steps 1 and 2, but relinks the program in Step 3 (as specified in the last description block).

14.4 Running MAKE

■ Syntax

MAKE [[*options*]] [[*macrodefinitions*]] *filename*

The following list describes the options you can give on the **MAKE** command line:

Option	Meaning
options	One or more of the **MAKE** options described in Section 6.6.
macrodefinitions	One or more **MAKE** macro definitions. The use of macro definitions is described in Section 6.7.
filename	The name of a **MAKE** description file.

Once you start **MAKE**, it reads the line in each description block that specifies the outfile and infiles and checks the modification dates of those files. If any of the infiles has a modification date later than the outfile's modification date, or if the outfile does not exist, **MAKE** displays the commands specified in the block and then executes the given commands. Otherwise, it skips to the next description block.

If **MAKE** cannot find a file, it displays a message informing you that the file was not found. If the missing file is an outfile, **MAKE** continues running since, in many cases, the missing file will be created by later commands.

If the missing file is an infile or a command file (that is, an executable or batch file), **MAKE** stops running. **MAKE** also stops running and displays an exit code if any command in the description block returns an error, unless a minus sign (–) precedes the command line in the **MAKE** description file.

MAKE executes any commands in the environment in which the **MAKE** command itself is invoked. Thus, you can include environment variables such as **PATH** for the commands specified in the description file.

14.5 Specifying MAKE Options

To invoke a **MAKE** option, type the option on the **MAKE** command line in the *options* field. The following list describes each **MAKE** option available with **MAKE** and how the option effects how **MAKE** operates:

Option	Action
/D	Displays the last modification date of each file as the file is scanned.
/I	Ignores exit codes (also called return or "errorlevel" codes) returned by programs called from the **MAKE** description file. **MAKE** continues executing the rest of the description file despite the errors.
/N	Displays commands in the description file that **MAKE** would execute but does not execute these commands. This option is useful if you are debugging a **MAKE** description file.
/S	Does not display lines as they are executed.

14.6 Using Macro Definitions with MAKE

Macro definitions let you associate a name with text used in a description file, then use the name instead of the text wherever the text appears in a description file. This feature makes it easier to update a description file when one of the names used in the file changes: when you update a macro definition, the corresponding text is updated wherever the macro appears in the definition file. Thus, you can change the text used throughout the description file without having to edit every line that uses the particular text.

You might want to use macro definitions to perform operations such as the following:

1. Specifying the base names of source, object, and executable files under development. If the program name changes, you only need to change the base name in the macro definition; then the base name is changed automatically for the source, object, and executable files given in the description file.
2. Specifying the set of default options for a command such as **FL** or **LINK**. If the options change, changing the macro definition changes the options wherever the macro appears in the description file.

14.6.1 Defining and Specifying Macros

The following defines the form of a macro definition:

name= text

After you define a macro, use the following to include the macro in the description file:

$(*name***)**

Wherever the pattern $(*name*) appears in the description file, that pattern is replaced by *text*. The *name* is converted to uppercase; for example, the names `flags` and `FLAGS` are equivalent. If you define a macro name but leave *text* blank, *text* will be a null string.

For *name*, you can also use any environment variable that is defined in the current environment in a macro definition. For example, if the environment variable **PATH** is defined in the current environment, the value of **PATH** will replace any occurrences of `$(PATH)` in the description file.

You can give macro definitions in either of the following two places:

1. In the **MAKE** description file. Each macro definition must appear on a separate line. Any white space (tab or space characters) between *name* and the equal sign (=) or between the equal sign and *text* is ignored. Any other white space is considered part of *text*.
2. On the **MAKE** command line.

To include white space in a macro definition, enclose the entire definition in double quotation marks (`" "`).

If the same *name* is defined in more than one place, the following order of precedence applies:

1. Command-line definition
2. Description-file task definition
3. Environment definition

■ Example

Assume the following **MAKE** description file named `LINKER`:

```
base=ABC
debug="/CO"

$(base).OBJ:        $(base).OBJ
        LINK $(base).OBJ

$(base).exe:        $(base).obj \lib\libv3.lib
        LINK $(debug),$(base),$(base),$(base);
```

In this description file, macro definitions are given for the names `base` and `debug`.

The `base` macro defines the base name of the object and executable files being maintained. **MAKE** replaces each occurrence of `$(base)` with the text `ABC`. If the program name changes, you would only have to replace `ABC` in the macro definition with the new program name to change the

base name of the two files.

The debug macro tells the linker to prepare a special executable file containing symbolic data and line-number information.

If you want to override one of the macro values in this description file, you can give a new macro definition on the **MAKE** command line, as shown in the following example:

```
MAKE base=DEF linker
```

This command-line definition of base overrides the definition of base in the description file. This causes base to be replaced with DEF instead of ABC.

If you do not want the special executable file created during linking, you could run **MAKE** with the following command line:

```
MAKE debug= linker
```

Since you give a blank value for debug (note the white space between the equal sign and the **MAKE** description-file name), it will be treated as a null string. Because definition on the command line has higher precedence than the definition in the description file, the $(debug) macro becomes a null string. Thus, the linker does not prepare the special executable file for debugging.

14.6.2 Using Macros within Macro Definitions

Macros can be used within macro definitions. For example, you could have the following macro definition in a **MAKE** description file named PICTURE:

```
LIBS=$(DLIB)\LIBV3.LIB $(DLIB)\GRAPHICS.LIB
```

You could then run **MAKE** and specify the definition for the macro named $(DLIB) on the command line, as shown in the following example:

```
MAKE DLIB=C:\LIB PICTURE
```

In this case, every occurrence of the macro $(DLIB) in the description file would be expanded to C:\LIB, so the definition of the LIBS macro in the description file would be expanded to the following:

```
LIBS=C:\LIB\LIBV3.LIB C:\LIB\GRAPHICS.LIB
```

Be careful to avoid infinitely recursive macros such as the following:

```
A = $(B)
B = $(C)
C = $(A)
```

In the example above, if the macro `$(B)` is undefined, all of these macros will be undefined, as well.

14.6.3 Using Special Macros

MAKE recognizes the following special macro names and automatically substitutes the corresponding text for each:

Name	Value Substituted
$*	Base name of the outfile (without the extension)
$@	Complete outfile name
$**	Complete list of infiles

- **Example**

```
TEST.EXE: MOD1.OBJ MOD2.OBJ MOD3.OBJ
        LINK $**, $@;
        $*
```

In the **LINK** command in the example above, `$**` represents all of the infiles that correspond to the outfile `TEST.EXE`, and `$@` specifies the complete name of `TEST.EXE` as the executable-file name on the **LINK** command line. The final line uses `$*` to specify the base name of `TEST.EXE`, `TEST`, as the next command to be carried out. Thus, this example is equivalent to the following:

```
TEST:EXE: MOD1.OBJ MOD2.OBJ MOD3.OBJ
        LINK MOD1.OBJ MOD2.OBJ MOD3.OBJ, TEST.EXE;
        TEST
```

14.7 Defining Inference Rules

Often, you use **MAKE** to perform updates on one type of file when a file of another type is changed. For example, you often use **MAKE** to update object files when source files change or update executable files when object files change.

MAKE allows you to define rules, known as "inference rules," that allow you to give a single command to convert all files with a given extension to files with a different extension. For example, you can use inference rules to specify a single **LINK** command that changes any object file (which has an extension of **.OBJ**) to an executable file (which has an extension of **.EXE**). You would not have to include the **LINK** command in each block in which you link a object file.

Inference rules have the following form:

.inextension.outextension **:**
    command
    [[*command*]]
    .
    .
    .

In this format, *command* specifies one of the commands that you must use to convert files with extension *inextension* to files with extension *outextension*. Using the earlier example of converting source files to object files, *inextension* would be **.OBJ**, *outextension* would be **.EXE**, and *command* would be the **LINK** command with any appropriate command-line options.

If **MAKE** finds a description block without an explicit command, it looks for an inference rule that matches both the outfile extension and the infile extension. If it finds such a rule, **MAKE** carries out any commands given in the rule.

You can include inference rules in one of two places:

1. In a **MAKE** description file
2. In a file named **TOOLS.INI**. This file is known as the "tools-initialization file." A line beginning with the tag **[make]** must appear before any dependency rules in **TOOLS.INI.**

MAKE searches for dependency rules in the following order:

1. In the current description file
2. In **TOOLS.INI**. **MAKE** looks for **TOOLS.INI** on the current drive and directory, then searches any directories given in the DOS **PATH** command. If **MAKE** finds **TOOLS.INI**, it looks through the file for a line beginning with the tag **[make]**. It applies any appropriate inference rules following this line.

■ Example

```
.OBJ.EXE:
        LINK $*.OBJ;

EXAMPLE1.EXE:  EXAMPLE1.OBJ

EXAMPLE2.EXE:  EXAMPLE2.OBJ
        LINK /CO ,,,LIBV3.LIB
```

In the sample description file above, line 1 defines an inference rule that executes the **LINK** command on line 2 to create an object file whenever a change is made in the corresponding ojbect file. The file name in the inference rule is specified with the special macro name `$*` so that the rule applies to any base name with the `.OBJ` extension.

When **MAKE** encounters a line containing an outfile and one or more infiles, it first looks for commands on the next line. When it does not find any commands, **MAKE** checks for a rule that may apply and finds the rule defined in lines 1 and 2 of the description file. **MAKE** applies the rule, replacing the `$*` macro with `EXAMPLE1` when it executes the command, so that the **LINK** command becomes

```
LINK EXAMPLE1.OBJ;
```

When **MAKE** reaches the line containing the `EXAMPLE2.EXE` outfile, it does not search for a dependency rule, since a command is explicitly given for this outfile/infile relationship.

Chapter 15

Using EXEPACK, EXEMOD, SETENV, and ERROUT

15.1 Compressing Executable Files with the EXEPACK Utility 333

15.2 Modifying Program Headers with the EXEMOD Utility 335

15.3 Enlarging the DOS Environment with the SETENV Utility 338

15.4 Redirecting Error Output with the ERROUT Utility 339

The following utilities allow you to modify files and change the operating environment:

Utility	Function
Microsoft EXE File Compression Utility (**EXEPACK**)	Compresses executable files by removing sequences of repeated characters from the file and by optimizing the relocation table.
Microsoft EXE File Header Utility (**EXEMOD**)	Modifies header information in executable files.
Microsoft Environment Expansion Utility (**SETENV**)	Enlarges the DOS environment table in IBM PC-DOS Versions 3.1, 3.0, 2.1, and 2.0. **SETENV** allows you to use more and/or larger environment variables.
Microsoft STDERR Redirection Utility (**ERROUT**)	Redirects standard error output from any command to a given file or device.

The following sections explain how to use the **EXEPACK**, **EXEMOD**, **SETENV**, and **ERROUT** utilities.

15.1 Compressing Executable Files with the EXEPACK Utility

The **EXEPACK** utility compresses sequences of identical characters from a specified executable file. It also optimizes the relocation table, whose entries are used to determine where modules are loaded into memory when the program is executed. Using **EXEPACK**, you can reduce the size of some files and decrease the time required to load them.

EXEPACK does not always give a significant saving in disk space, and may sometimes actually increase file size because of an enhanced **.EXE** loader. However, programs that have approximately 500 or more entries in the relocation table and long streams of repeated characters are usually shorter and take less time to load if packed.

The **EXEPACK** program has exactly the same function as the **LINK /EXEPACK** option, except that **EXEPACK** works on files that have already been linked. One use for this utility is to pack the executable files provided with the product distribution. If you have floppy disks, you may want to pack all programs in order to make more room on your disks.

The **EXEPACK** command-line format is as follows:

EXEPACK *executablefile packedfile*

The *executablefile* is the file to be packed and *packedfile* is the name for the packed file. The *packedfile* should have a different name or be on a different drive or directory. **EXEPACK** will not pack a file onto itself.

When using **EXEPACK** to pack an executable overlay file or a file that calls overlays, the packed file should always be renamed with the original name to avoid the overlay-manager prompt.

Note

Using **EXEPACK** removes all symbolic debug information for executable files.

■ Example

```
EXEPACK WORK.EXE WORK.TMP
DEL WORK.EXE
RENAME WORK.TMP WORK.EXE
```

In the example above, the executable file `WORK.EXE` is packed to a temporary file. The original is then deleted and the new packed version is renamed with the original name.

15.2 Modifying Program Headers with the EXEMOD Utility

The **EXEMOD** utility allows you to modify fields in the header of an executable file. Some of the options available with **EXEMOD** are the same as **LINK** options, except that they work on files that have already been linked. Unlike the **LINK** options, the **EXEMOD** options require that values be specified as hexadecimal numbers.

To display the current status of the header fields, type the following:

EXEMOD *executablefile*

To modify one or more of the fields in the file header, type the following:

EXEMOD *executablefile* [[**/H**]] | [[**/STACK** *hexnum*]][[**/MIN** *hexnum*]][[**/MAX** *hexnum*]]

hexnum is a number entered using hexdecimal digits (uppercase or lowercase); no prefix is needed. **EXEMOD** expects the *executablefile* to be the name of an existing file
with the **.EXE** extension. If the file name is given without an extension, **EXEMOD** appends **.EXE** and searches for that file. If you supply a file with an extension other than **.EXE**, **EXEMOD** displays the following error message:

```
exemod: file not .EXE
```

The **EXEMOD** options are shown with the forward slash (/) designator, but a dash (-) may also be used. Options can be given in either uppercase or lowercase, but they cannot be abbreviated. The **EXEMOD** options and their effects are described in the following list:

Option	Effect
/H	Displays the current status of the DOS program header. Its effect is the same as entering **EXEMOD** with an *executablefile* but without options. The **/H** option should not be used with other options.
/STACK *hexnum*	Allows you to set the size of the stack (in bytes) for your program by setting the initial **SP** (stack pointer) value to *hexnum*. The minimum allocation value is adjusted upward, if

necessary. This option has the same effect as the **LINK /STACK** option, except that it works on files that are already linked.

/MIN *hexnum* — Sets the minimum allocation value (that is, the minimum number of 16-byte paragraphs needed by the program when it is loaded into memory) to *hexnum*. The actual value set may be different from the requested value if adjustments are necessary to accommodate the stack.

/MAX *hexnum* — Sets the maximum allocation value (that is, the maximum number of 16-byte paragraphs used by the program when it is loaded into memory) to *hexnum*. The maximum allocation value must be greater than or equal to the minimum allocation value. This option has the same effect as the **LINK /CPARMAXALLOC** option.

Note

Use of the **/STACK** option on programs developed with other than Microsoft compilers or assemblers may cause the programs to fail, or **EXEMOD** may return an error message.

EXEMOD works on packed files. When it recognizes a packed file, it will print the following message:

```
packed file
```

It will then continue to modify the file header.

When packed files are loaded, they are expanded to their unpacked state in memory. If the **EXEMOD /STACK** option is used on a packed file, the value changed is the value that **SP** will have after expansion. If either the **/MIN** or the **/STACK** option is used, the value is corrected as necessary to accommodate unpacking of the modified stack. The **/MAX** option operates as it would for unpacked files.

If the header of a packed file is displayed, the **CS:IP** and **SS:SP** values are displayed as they are after expansion. These values are not the same as the actual values in the header of the packed file.

■ Examples

```
>EXEMOD TEST.EXE

Microsoft (R) EXE File Header Utility  Version 4.00
Copyright (C) Microsoft Corp 1985-1987.  All rights reserved.

TEST.EXE                              (hex)          (dec)

Minimum load size (bytes)             419D          16797
Overlay number                           0              0
Initial CS:IP                    0403:0000
Initial SS:SP                    0000:0000              0
Minimum allocation (para)                0              0
Maximum allocation (para)             FFFF          65535
Header size (para)                      20             32
Relocation table offset                 1E             30
Relocation entries                       1              1
```

The example above shows how to use **EXEMOD** to display the current file header for file TEST.EXE.

```
EXEMOD TEST.EXE /STACK FF /MIN FF /MAX FFF
```

The above example shows how to use the **EXEMOD** command line to modify the header fields in TEST.EXE.

```
>EXEMOD TEST.EXE

Microsoft (R) EXE File Header Utility  Version 4.00
Copyright (C) Microsoft Corp 1985-1987.  All rights reserved.

TEST.EXE                              (hex)          (dec)

Minimum load size (bytes)             528D          20877
Overlay number                           0              0
Initial CS:IP                    0403:0000
Initial SS:SP                    0000:00FF            256
Minimum allocation (para)                FF            256
Maximum allocation (para)               FFF           4095
Header size (para)                      20             32
Relocation table offset                 1E             30
Relocation entries                       1              1
```

The last example shows how you would determine the current status of the header for FILE.EXE after using the command in the previous example to modify the header.

15.3 Enlarging the DOS Environment with the SETENV Utility

The **SETENV** utility allows you to allocate more operating-environment space to DOS by modifying a copy of **COMMAND.COM**.

Normally, DOS Versions 2.0 and later allocate 160 bytes (10 paragraphs) for the environment table. This may not be enough if you want to set numerous environment variables using the **SET** or **PATH** command. For example, if you have a hard disk with several levels of subdirectories, a single environment variable might take 40 or 50 characters. Since each character uses 1 byte, you could easily require more than 160 bytes if you want to set several environment variables.

Note

SETENV is guaranteed to work only with IBM PC-DOS Versions 2.0, 2.1, 3.0, and 3.1. **SETENV** may or may not work with other versions of DOS. Moreover, you should not use **SETENV** with versions of DOS later than Version 3.1. Consult your DOS manual for information on how to increase environment size in these later versions.

To enlarge the environment table, you must use **SETENV** to modify a copy of **COMMAND.COM**. Make sure you work on a copy and retain an unmodified version of **COMMAND.COM** for backup.

The command line for modifying the environment table is as follows:

SETENV *filename* [[*environmentsize*]]

Normally *filename* specifies **COMMAND.COM**. It must be a valid, unmodified copy of **COMMAND.COM**, though it could have a different name if you renamed it. The optional *environmentsize* is a decimal number specifying the size in bytes of the new allocation; *environmentsize* must be

a number greater than or equal to 160, and less than or equal to 65,520. The specified *environmentsize* is rounded up to the nearest multiple of 16 (the size of a paragraph).

If *environmentsize* is not given, **SETENV** reports the value that the **COMMAND.COM** file is currently allocating.

After modifying **COMMAND.COM**, you must reboot so that the environment table is set to the new size.

■ Examples

```
>SETENV COMMAND.COM

Microsoft (R) Environment Expansion Utility  Version 2.00
Copyright (C) Microsoft Corp 1985-1987.  All rights reserved.

command.com: Environment allocation = 160
```

In the example above, no environment size is specified, so **SETENV** reports the current size of the environment table.

```
SETENV COMMAND.COM 605
```

In the example above, an environment size of 605 bytes is requested. Since 605 bytes is not on a paragraph boundary (a multiple of 16), **SETENV** rounds the request up to 608 bytes. **COMMAND.COM** is modified so that it will automatically set an environment table of 608 bytes (38 paragraphs). You must reboot to set the new environment-table size.

15.4 Redirecting Error Output with the ERROUT Utility

By default, standard output and standard error output from a DOS program are both directed to the terminal. The **ERROUT** utility allows you to execute any legal DOS command-line (including an executable or batch file, as well as arguments), and redirect standard error output to a specified file or device.

The **ERROUT** command-line format is as follows:

ERROUT [**/f** *standarderrorfile*] *doscommandline*

The *doscommandline* is simply the entire command line you would type in if you were not using **ERROUT**. This includes the **.EXE**, **.COM**, or **.BAT** file you are invoking, as well as any options, arguments, and spaces that you would normally use. The *doscommandline* runs to the end of the **ERROUT** command line.

The **/f** *standarderrorfile* option is the name of the file or device to which standard error output is redirected. The **f** must be lower case, and at least one space must separate it from the beginning of *standarderrorfile*. Without the use of this option, **ERROUT** has no effect; *doscommandline* is simply executed as it would normally be executed by DOS.

Note

With **ERROUT**, you may use the DOS redirection operators **>** and **>>** just as you normally would. However, their effects change somewhat; only standard output is redirected to the file indicated by **>** or **>>**. Standard error is redirected to *standarderrorfile*.

■ Examples

```
ERROUT /f ERR.FIL TYPE READ.ME > OUT.FIL
```

In the above example, the standard output of the command `TYPE READ.ME` is redirected to the file `OUT.FIL`, while the standard error output, if any, is redirected to the file `ERR.FIL`. If there is no error output, then **ERROUT** will still create a file called `ERR.FIL`, and this file will be 0 bytes long.

```
ERROUT /f C_ERRORS.DOC CL /AL /Zi /Od demo.c
```

In the above example, the entire command line beginning with **CL** is executed; all of the command-line arguments `/AL`, `/Zi`, `/CO`, and `demo.c` modify the **CL** command as they normally would. Error output, if any, is sent to `C_ERRORS.DOC`.

```
ERROUT /f PRN MASM /ZI TEST,,;
```

In the above example, the DOS command line `MASM /ZI TEST,,;` is executed, and standard error output is sent to the printer (which is the device indicated by `PRN`).

Appendixes

A Regular Expressions 345
B Error Messages 353
C Using Exit Codes 393

Appendix A
Regular Expressions

A.1 Introduction 347
A.2 Special Characters in Regular Expressions 347
A.3 Searching for Special Characters 348
A.4 Using the Period 348
A.5 Using Brackets 349
A.5.1 Using the Dash within Brackets 349
A.5.2 Using the Caret within Brackets 350
A.5.3 Matching Brackets within Brackets 350
A.6 Using the Asterisk 350
A.7 Matching the Start or End of a Line 351

A.1 Introduction

Regular expressions are used to specify text patterns in searches for variable text strings. Special characters can be used within regular expressions to specify groups of characters to be searched for.

This appendix explains all the special characters you can use to form regular expressions, but you do not need to learn them all to use the CodeView Search commands. The simplest form of regular expression is simply a text string. For example, if you want to search for all instances of the symbol `COUNT`, you can specify `COUNT` as the string to be found.

If you only want to search for simple strings, you do not need to read this entire appendix, but you should know how to search for strings containing the special characters used in regular expressions. See Section B.3 for more information.

A.2 Special Characters in Regular Expressions

The following characters have special meanings in regular expressions:

Character	Purpose
Backslash (\)	Removes the special characteristics of the following characters: backslash (\), period (.), caret (^), dollar sign ($), asterisk (*), and left bracket ([)
Period (.)	Matches any character
Caret (^)	Matches beginning of line
Dollar sign ($)	Matches end of line
Asterisk (*)	Matches any number of repetitions of the previous character
Brackets ([ ])	Matches characters specified within the brackets; the following special characters may be used inside brackets:

Caret (^)	Reverses the function of the brackets; that is, matches any character except those specified within the brackets
Dash (–)	Matches characters in ASCII order between (inclusive) the characters on either side of the dash

A.3 Searching for Special Characters

If you need to match one of the special characters used in regular expressions, you must precede it with a backslash when you specify a search string. The special characters are the asterisk (*), backslash (\), left bracket ([), caret (^), dollar sign ($), and period (.).

For example, the regular expression `I*J` matches such combinations as `IJ`, `I+J`, and `I-J`. The regular expression `I\*J` matches only `I*J`. The backslash is necessary because the asterisk (*) is a special character in regular expressions.

A.4 Using the Period

A period in a regular expression matches any single character. This corresponds to the question mark (?) used in specifying DOS file names.

For example, you could use the regular expression `AMAX.` to search for either of the intrinsic functions `AMAX0` and `AMAX1`. You could use the expression `X.Y` to search for strings such as `X+Y`, `X-Y`, or `X*Y`. If your programming style is to put a space between variables and operators, you could use the regular expression `X . Y` for the same purpose.

Note that when you use the period as a wild card, you will find the strings you are looking for, but you may also find other strings that you aren't interested in. You can use brackets to be more exact about the strings you want to find.

A.5 Using Brackets

You can use brackets to specify a character or characters you want to match. Any of the characters listed within the brackets is an acceptable match. This method is more exact than using a period to match any character.

For example, the regular expression `x[-+/*]y` matches `x+y`, `x-y`, `x/y`, or `x*y`, but not `x=y` or `xzy`. The regular expression `COUNT[12]` matches `COUNT1` and `COUNT2`, but not `COUNT3`.

Most regular-expression special characters have no special meaning when used within brackets. The only special characters within brackets are the dash (–), caret (^), and right bracket (]). Even these characters only have special meanings in certain contexts, as explained in Sections B.5.1–B.5.3.

A.5.1 Using the Dash within Brackets

The dash can be used within brackets to specify a group of sequential ASCII characters. For example, the regular expression `[0-9]` matches any digit; it is equivalent to `[0123456789]`. Similarly, `[a-z]` matches any lowercase letter, and `[A-Z]` matches any uppercase letter.

You can combine ASCII ranges of characters with other listed characters. For example, `[A-Za-z ]` matches any uppercase or lowercase letter or a space.

The dash has this special meaning only if you use it to separate two ASCII characters. It has no special meaning if used directly after the starting bracket or directly before the ending bracket. This means that you must be careful where you place the dash (minus sign) within brackets.

For example, you might use the regular expression `[+-/*]` to match the characters `+`, `-`, `/`, and `*`. However, this does not give the intended result. Instead it matches the characters between `+` and `/` and also the character `*`. To specify the intended characters, put the dash first or last in the list: `[-+/*]` or `[+/*-]`.

A.5.2 Using the Caret within Brackets

If used as the first character within brackets, the caret (^) reverses the meaning of the brackets. That is, any character except the ones in brackets will be matched. For example, the regular expression `[^0-9]` matches any character that is not a digit. Specifying the characters to be excluded is often more concise than specifying the characters you want to match.

If the caret is not in the first position within the brackets, it is treated as an ordinary character. For example, the expression `[0-9^]` matches any digit or a caret.

A.5.3 Matching Brackets within Brackets

Sometimes you may want to specify the bracket characters as characters to be matched. This is no problem with the left bracket; it is treated as a normal character. However, the right bracket is interpreted as the end of the character list rather than as a character to be matched.

If you want the right bracket to be matched, you must make it the first character after the initial left bracket. For example, the regular expression `[]#![@%]` matches either bracket character or any of the other characters listed within the brackets. However, if you changed the order of just one of the characters (to `[#]![@%]`), the meaning would be changed so that you would be specifying two groups of characters in brackets: `[#]` and `[@%]`.

A.6 Using the Asterisk

The asterisk matches zero or more occurrences of the character preceding the asterisk.

For example, the regular expression `IF *(TEST` will match any of the following strings:

```
IF (TEST
IF        (TEST
IF(TEST
```

Note that the last example contains zero repetitions of the space character.

The asterisk is convenient if the text you are searching for might contain some spaces, but you don't know the exact number. (Be careful in this situation: you can't be sure if the text contains a series of spaces or a tab.)

You might also use the asterisk to search for a symbol when you aren't sure of the spelling. For example, you could use `first*ime` if you aren't sure if the identifier you are searching for is spelled `firsttime` or `firstime`.

One particularly powerful use of the asterisk is to combine it with the period (`.*`). This combination searches for any group of characters, and is similar to the asterisk used in specifying DOS file names. For example, the expression `(.*)` matches `(test)`, `(response .EQ. 'Y')`, `(x=0;x .LE. 20;x=x+1)`, or any other string that starts with a left parenthesis and ends with a right parenthesis.

You can use brackets with the asterisk to search for a sequence of repeated characters of a given type. For example, `\[[0-9]*]` matches number strings within brackets (`[1353]` or `[3]`), but does not match character strings within brackets (`[count]`). Empty brackets (`[ ]`) are also matched, since the characters in the brackets are repeated zero times.

A.7 Matching the Start or End of a Line

In regular expressions, the caret (`^`) matches the start of a line, while the dollar sign ($) matches the end of a line.

For example, the regular expression `^C` matches any uppercase C that starts a line. Similarly, `)$` matches a right parenthesis at the end of a line, but not a right parenthesis within a line.

You can combine both symbols to search for entire lines. For example, `^{$` matches any line consisting of only a left curly brace in the left margin, and `^$` matches blank lines.

Appendix B
Error Messages

B.1 CodeView Error Messages 355
B.2 Linker Error Messages 365
B.3 LIB Error Messages 377
B.4 MAKE Error Messages 382
B.5 EXEPACK Error Messages 385
B.6 EXEMOD Error Messages 387
B.7 SETENV Error Messages 389
B.8 ERROUT Error Messages 391

B.1 CodeView Error Messages

The CodeView debugger displays an error message whenever it detects a command it cannot execute. Most errors (start-up errors are the exception) terminate the CodeView command under which the error occurred, but do not terminate the debugger. You may see any of the following error messages:

`Argument to IMAG/DIMAG must be simple type`

You specified an argument to an **IMAG** or **DIMAG** function that is not permitted, such as an array with no subscripts.

`Array must have subscript`

You specified an array without any subscripts in an expression: for example, `IARRAY+2`.

`Bad address`

You specified the address in an invalid form.

For instance, you may have entered an address containing hexadecimal characters when the radix is decimal.

`Bad breakpoint command`

You typed an invalid breakpoint number with the Breakpoint Clear, Breakpoint Disable, or Breakpoint Enable command.

The number must be in the range 0 to 19.

`Bad flag`

You specified an invalid flag mnemonic with the Register dialog command (**R**).

Use one of the mnemonics displayed when you enter the command **RF**.

`Bad format string`

You used an invalid format specifier following an expression.

Expressions used with the Display Expression, Watch, Watchpoint, and Tracepoint commands can have CodeView format specifiers set off from the expression by a comma. The valid format specifiers are **d**, **i**, **u**, **o**, **x**, **X**, **f**, **e**, **E**, **g**, **G**, **c**, and **s**. Some format specifiers can be preceded by the prefix **h** or **l**. See Chapter 7, "Examining Data and

Expressions," for more information about format specifiers.

`Bad integer or real constant`

You specified an illegal numeric constant in an expression.

`Bad intrinsic function`

You specified an illegal intrinsic function name in an expression.

`Bad radix (use 8, 10, or 16)`

With the **N** command you can use only octal, decimal, and hexadecimal radixes.

`Bad register`

You typed the Register command (**R**) with an invalid register name.

Use **AX**, **BX**, **CX**, **DX**, **SP**, **BP**, **SI**, **DI**, **DS**, **ES**, **SS**, **CS**, **IP**, or **F**.

`Bad subscript`

You entered an illegal subscript expression for an array, such as `IARRAY(3.3)` or `IARRAY((3,3))`.

`Bad type cast`

The types of the operands in an expression are incompatible.

`Bad type (use one of 'ABDILSTUW')`

The valid dump types are ASCII (**A**), Byte (**B**), Integer (**I**), Unsigned (**U**), Word (**W**), Double Word (**D**), Short Real (**S**), Long Real (**L**), and 10-Byte Real (**T**).

`Badly formed type`

The type information in the symbol table of the file you are debugging is incorrect.

If this message occurs, please note the circumstances of the error and inform Microsoft Corporation, using the Software Problem Report at the back of this manual.

`Breakpoint # or '*' expected`

You entered the Breakpoint Clear (**BC**), Breakpoint Disable (**BD**), or Breakpoint Enable (**BE**) command with no argument.

These commands require that you specify the number of the

breakpoint to be acted on, or that you specify the asterisk (*), indicating that all breakpoints are to be acted on.

`Cannot use struct or union as scalar`

A struct or union variable cannot be used as a scalar value in a C expression.

Such variables must be followed by a file specifier or preceded with the address-of operator.

`Cannot cast complex constant component into REAL`

Both the real and imaginary components of a **COMPLEX** constant must be compatible with type **REAL**.

`Cannot cast IMAG/DIMAG argument to COMPLEX`

Arguments to **IMAG** and **DIMAG** must be simple numeric types.

`Can't find` *filename*

The CodeView debugger could not find the executable file you specified when you started.

You probably misspelled the file name, or the file is in a different directory.

`Character constant too long`

You specified a character constant that is too long for the FORTRAN expression evaluator.

The limit is 126 bytes.

`Character too big for current radix`

In a constant, you specified a radix that is larger than the current CodeView radix.

Use the **N** command to change the radix.

`Constant too big`

The CodeView debugger cannot accept an unsigned constant number larger than 4,294,967,295 (16#FFFFFFFF).

`Divide by zero`

An expression in an argument of a dialog command attempts to divide by zero.

`Expression too complex`

An expression given as a dialog-command argument is too complex.

Try simplifying the expression.

`Extra input ignored`

You specified too many arguments to a command.

The CodeView debugger evaluates the valid arguments and ignores the rest. Often, in this situation, the debugger may not evaluate the arguments the way you intended.

`Flip/Swap option off — application output lost`

The program you are debugging is writing to the screen, but the output cannot be displayed because you have turned off the flip/swap option.

`Floating point error`

This message should not occur, but if it does, please note the circumstances of the error and inform Microsoft Corporation, using the Software Problem Report at the back of this manual.

`Index out of bound`

You specified a subscript value that is outside the bounds declared for the array.

`Internal debugger error`

If this message occurs, please note the circumstances of the error and inform Microsoft Corporation, using the Software Problem Report at the back of this manual.

`Invalid argument`

One of the arguments you specified is not a valid CodeView expression.

`Missing '"'`

You specified a string as an argument to a dialog command, but you did not supply a closing double quotation mark.

`Missing '('`

An argument to a dialog command was specified as an expression containing a right parenthesis, but no left parenthesis.

`Missing ')'`

An argument to a dialog command was specified as an expression containing a left parenthesis, but no right parenthesis.

`Missing ']'`

An argument to a dialog command was specified as an expression containing a left bracket, but no right bracket.

This error can also occur if a regular expression is specified with a right bracket but no left bracket.

`Missing '(' in complex constant`

The debugger is expecting the opening parenthesis of a complex constant in an expression, but it is missing.

`Missing ')' in complex constant`

The debugger expects the closing parenthesis of a complex constant in an expression.

`Missing ')' in substring`

The debugger expects the closing parenthesis of a substring expression.

`Missing '(' to intrinsic`

The debugger expects an opening parenthesis for an intrinsic function.

`Missing ')' to intrinsic`

The debugger expects a closing parenthesis for an intrinsic function.

`No closing single quote`

You specified a character in an expression used as a dialog-command argument, but the closing single quotation mark is missing.

`No code at this line number`

You tried to set a breakpoint on a source line that does not correspond to code.

For instance, the line may be a data declaration or a comment.

`No match of regular expression`

No match was found for the regular expression you specified with the Search command or with the Find selection from the Search menu.

`No previous regular expression`

You selected Previous from the Search menu, but there was no previous match for the last regular expression specified.

`No program to debug`

You have executed to the end of the program you are debugging.

You must restart the program (using the Restart command) before using any command that executes code.

`No source lines at this address`

The address you specified as an argument for the View command (V) does not have any source lines.

For instance, it could be an address in a library routine or an assembly-language module.

`No such file/directory`

A file you specified in a command argument or in response to a prompt does not exist.

For instance, the message appears when you select Load from the File menu, and then enter the name of a nonexistent file.

`No symbolic information`

The program file you specified is not in the CodeView format.

You cannot debug in source mode, but you can use assembly mode.

`Not a text file`

You attempted to load a file using the Load selection from the File menu or using the View command, but the file is not a text file.

The CodeView debugger determines if a file is a text file by checking the first 128 bytes for characters that are not in the ASCII ranges 9 to 13 and 20 to 126.

`Not an executable file`

The file you specified to be debugged when you started the CodeView debugger is not an executable file having the extension **.EXE** or **.COM**.

`Not enough space`

You typed the Shell Escape command (**!**) or selected Shell from the File menu, but there is not enough free memory to execute **COMMAND.COM**.

Since memory is released by code in the FORTRAN start-up routines, this error always occurs if you try to use the Shell Escape command before you have executed any code. Use any of the code-execution commands (Trace, Program Step, or Go) to execute the FORTRAN start-up code, then try the Shell Escape command again. The message also occurs with assembly-language programs that do not specifically release memory.

`Object too big`

You entered a Tracepoint command with a data object (such as an array) that is larger than 128 bytes.

`Operand types incorrect for this operation`

An operand in a FORTRAN expression had a type incompatible with the operation applied to it.

For example, if `P` is declared as `CHARACTER P(10)`, then `? P+5` would produce this error, since a character array cannot be an operand of an arithmetic operator.

`Operator must have a struct/union type`

You used one of the C member-selection operators (`->` or `.`) in an expression that does not reference an element of a structure or union.

`Operator needs lvalue`

You specified an expression that does not evaluate to a memory location in an operation that requires one. (An lvalue is an expression that refers to a memory location.)

For example, `buffer(count)` is correct because it represents a symbol in memory. However, `I .EQV. 10` is invalid because it evaluates to TRUE or FALSE instead of a single memory location.

`Program terminated normally (`*number*`)`

You executed your program to the end. The number displayed in parentheses is the exit code returned to DOS by your program.

You must use the Restart command (or the Start menu selection) to start the program before executing more code.

`Radix must be between 2 and 36 inclusive`

You specified a radix outside the allowable range.

`Register variable out of scope`

You tried to specify a register variable using the period (.) operator and a routine name.

For example, if you are in a third-level routine, you can display the value of a local variable called `local` in a second-level routine called `parent` with the following command:

```
? parent.local
```

However, this command will not work if `local` is declared as a register variable.

`Regular expression too complex`

The regular expression specified is too complex for the CodeView debugger to evaluate.

`Regular expression too long`

The regular expression specified is too long for the CodeView debugger to evaluate.

`Simple variable cannot have argument`

In an expression, you specified an argument to a simple variable.

For example, given the declaration `INTEGER NUM`, the expression `NUM(I)` is not allowed.

`Substring range out of bound`

A character expression exceeds the length specified in the **CHARACTER** statement.

`Syntax error`

You specified an invalid command line for a dialog command.

Check for an invalid command letter. This message also appears if you enter an invalid assembly-language instruction using the Assemble command. The error will be preceded by a caret that points to the first character the CodeView debugger could not interpret.

`Too few array bounds given`

The bounds you specified in an array subscript do not match the array declaration.

For example, given the array declaration `INTEGER IARRAY(3,4)`, the expression `IARRAY(I)` would produce this message.

`Too many array bounds given`

The bounds you specified in an array subscript do not match the array declaration.

For example, given the array declaration `INTEGER IARRAY(3,4)`, the expression `IARRAY(I,3,J)` would produce this message.

`Too many breakpoints`

You tried to specify a 21st breakpoint; the CodeView debugger only permits 20.

`Too many open files`

You do not have enough file handles for the CodeView debugger to operate correctly.

You must specify more files in your **CONFIG.SYS** file. See your DOS user's guide for information on using the **CONFIG.SYS** file.

`Type clash in function argument`

The type of an actual parameter does not match the corresponding formal parameter.

This message also appears when a subroutine that uses alternate returns is called and the values of the return labels in the actual parameter list are not 0.

`Type conversion too complex`

You tried to type cast an element of an expression in a type other than the simple types or with more than one level of indirection.

An example of a complex type would be type casting to a struct or union type. An example of two levels of indirection would be `char **`.

`Unable to open file`

A file you specified in a command argument or in response to a prompt cannot be opened.

For instance, the message appears when you select Load from the File menu, and then enter the name of a file that is corrupted or has its file attributes set so that it cannot be opened.

`Unknown symbol`

You specified an identifier that is not in the CodeView debugger's symbol table.

Check for a misspelling. This message may also occur if you try to use a local variable in an argument when you are not in the routine where the variable is defined. The message also occurs when a subroutine that uses alternate returns is called and the values of the return labels in the actual parameter list are not 0.

`Unrecognized option` *option*
`Valid options: /B /C<command> /D /F /I /M /S /T /W /43 /2`

You entered an invalid option when starting the CodeView debugger.

Try retyping the command line.

`Usage: cv [options] file [arguments]`

You failed to specify an executable file when you started the CodeView debugger.

Try again with the syntax shown in the message.

`Video mode changed without /S option`

The program changed video modes (from or to one of the graphics modes) when screen swapping was not specified.

You must use the **/S** option to specify screen swapping when debugging graphics programs. You can continue debugging when you get

this message, but the output screen of the debugged program may be damaged.

`Warning: packed file`

You started the CodeView debugger with a packed file as the executable file.

You can attempt to debug the program in assembly mode, but the packing routines at the start of the program may make this difficult. You cannot debug in source mode because all symbolic information is stripped from a file when it is packed with the **/EXEPACK** linker option or the **EXEPACK** utility.

`Wrong number of function arguments`

You specified an incorrect number of arguments for a function. The messages listed below indicate potential problems but do not hinder compilation and linking. The **/W** compiler option has no effect on the output of these messages.

B.2 Linker Error Messages

This section lists and describes error messages generated by the Microsoft Overlay Linker, **LINK**.

Fatal errors cause the linker to stop execution. Fatal error messages have the following format:

location : `error L1`*xxx*: *messagetext*

Nonfatal errors indicate problems in the executable file. **LINK** produces the executable file. Nonfatal error messages have the following format:

location : `error L2`*xxx*: *messagetext*

Warnings indicate possible problems in the executable file. **LINK** produces the executable file. Warnings have the following format:

location : `warning L4`*xxx*: *messagetext*

In these messages, *location* is the input file associated with the error, or `LINK` if there is no input file. If the input file is an **.OBJ** or **.LIB** file and has a module name, the module name is enclosed in parentheses, as shown

in the following examples:

```
SLIBC.LIB(_file)
MAIN.OBJ(main.c)
TEXT.OBJ
```

The following error messages may appear when you link object files with the Microsoft Overlay Linker, **LINK**:

Number	Linker Error Message
L1001	*option* : option name ambiguous A unique option name did not appear after the option indicator (/). For example, the command `LINK /N main;` generates this error, since **LINK** cannot tell which of the three options beginning with the letter "N" was intended.
L1002	*option* : unrecognized option name An unrecognized character followed the option indicator (/), as in the following example: `LINK /ABCDEF main;`
L1004	*option* : invalid numeric value An incorrect value appeared for one of the linker options. For example, a character string was given for an option that requires a numeric value.
L1010	*option* : stack size exceeds 65536 bytes The size specified for the stack in the **/STACK** option of the **LINK** command was more than 65,536 bytes.
L1007	*option* : interrupt number exceeds 255 A number greater than 255 was given as a value for the **/OVERLAYINTERRUPT** option.

Number	Linker Error Message
L1008	*option* : segment limit set too high The limit on the number of segments allowed was set to greater than 1024 using the **/SEGMENTS** option.
L1009	*option* : CPARMAXALLOC : illegal value The number specified in the **/CPARMAXALLOC** option was not in the range 1–65,535.
L1020	no object modules specified No object-file names were specified to the linker.
L1021	cannot nest response files A response file occurred within a response file.
L1022	response line too long A line in a response file was longer than 127 characters.
L1023	terminated by user You entered CONTROL-C.
L1024	nested right parentheses The contents of an overlay were typed incorrectly on the command line.
L1025	nested left parentheses The contents of an overlay were typed incorrectly on the command line.
L1026	unmatched right parenthesis A right parenthesis was missing from the contents specification of an overlay on the command line.
L1027	unmatched left parenthesis A left parenthesis was missing from the contents specification of an overlay on the command line.

Number	Linker Error Message
L1043	`relocation table overflow` More than 32,768 long calls, long jumps, or other long pointers appeared in the program. Try replacing long references with short references, where possible, and recreate the object module.
L1045	`too many TYPDEF records` An object module contained more than 255 **TYPDEF** records. These records describe communal variables. This error can appear only with programs produced by the Microsoft FORTRAN Compiler or other compilers that support communal variables. (**TYPDEF** is a DOS term. It is explained in the *Microsoft MS-DOS Programmer's Reference* and in other reference books on DOS.)
L1046	`too many external symbols in one module` An object module specified more than the limit of 1023 external symbols. Break the module into smaller parts.
L1047	`too many group, segment, and class names in one module` The program contained too many group, segment, and class names. Reduce the number of groups, segments, or classes, and recreate the object file.
L1048	`too many segments in one module` An object module had more than 255 segments. Split the module or combine segments
L1049	`too many segments` The program had more than the maximum number of segments. (The **/SEGMENTS** option specifies the maximum legal number; the default is 128.) Relink using the **/SEGMENTS** option with an appropriate number of segments.

Number	Linker Error Message
L1050	`too many groups in one module` **LINK** encountered more than 21 group definitions (**GRPDEF**) in a single module. Reduce the number of group definitions or split the module. (Group definitions are explained in the *Microsoft MS-DOS Programmer's Reference* and in other reference books on DOS.)
L1051	`too many groups` The program defined more than 20 groups, not counting **DGROUP**. Reduce the number of groups.
L1052	`too many libraries` An attempt was made to link with more than 32 libraries. Combine libraries, or use modules that require fewer libraries.
L1053	`symbol table overflow` The program had more than 256K of symbolic information (such as public, external, segment, group, class, and file names). Combine modules or segments and recreate the object files. Eliminate as many public symbols as possible.
L1054	`requested segment limit too high` The linker did not have enough memory to allocate tables describing the number of segments requested. (The default is 128 or the value specified with the **/SEGMENTS** option.) Try linking again using the **/SEGMENTS** option to select a smaller number of segments (for example, use 64 if the default was used previously), or free some memory by eliminating resident programs or shells.

Number	Linker Error Message
L1056	`too many overlays` The program defined more than 63 overlays.
L1057	`data record too large` A **LEDATA** record (in an object module) contained more than 1024 bytes of data. This is a translator error. (**LEDATA** is a DOS term which is explained in the *Microsoft MS-DOS Programmer's Reference* and in other DOS reference books.) Note which translator (compiler or assembler) produced the incorrect object module and the circumstances. Please report this error to Microsoft Corporation using the Software Problem Report form at the back of this manual.
L1070	`segment size exceeds 64K` A single segment contained more than 64K of code or data. Try compiling and linking using the large model.
L1071	`segment _TEXT larger than 65520 bytes` This error is likely to occur only in small-model C programs, but it can occur when any program with a segment named **_TEXT** is linked using the **/DOSSEG** option of the **LINK** command. Small-model C programs must reserve code addresses 0 and 1; this range is increased to 16 for alignment purposes.
L1072	`common area longer than 65536 bytes` The program had more than 64K of communal variables. This error cannot appear with object files generated by the Microsoft Macro Assembler, **MASM**. It occurs only with programs produced by the Microsoft FORTRAN Compiler or other compilers that support communal variables.
L1080	`cannot open list file` The disk or the root directory was full. Delete or move files to make space.

Number	Linker Error Message
L1081	`out of space for run file` The disk on which the **.EXE** file was being written was full. Free more space on the disk and restart the linker.
L1083	`cannot open run file` The disk or the root directory was full. Delete or move files to make space.
L1084	`cannot create temporary file` The disk or root directory was full. Free more space in the directory and restart the linker.
L1085	`cannot open temporary file` The disk or the root directory was full. Delete or move files to make space.
L1086	`scratch file missing` An internal error has occurred. Note the circumstances of the problem and contact Microsoft Corporation using the Software Problem Report form at the back of this manual.
L1087	`unexpected end-of-file on scratch file` The disk with the temporary linker-output file was removed.
L1088	`out of space for list file` The disk on which the listing file was being written was full. Free more space on the disk and restart the linker.
L1089	*filename* `: cannot open response file` **LINK** could not find the specified response file. This usually indicates a typing error.

Number	Linker Error Message
L1090	`cannot reopen list file` The original disk was not replaced at the prompt. Restart the linker.
L1091	`unexpected end-of-file on library` The disk containing the library probably was removed. Replace the disk containing the library and run the linker again.
L1093	`object not found` One of the object files specified in the linker input was not found. Restart the linker and specify the object file.
L1101	`invalid object module` One of the object modules was invalid. If the error persists after recompiling, please contact Microsoft Corporation using the Software Problem Report form at the back of this manual.
L1102	`unexpected end-of-file` An invalid format for a library was encountered.
L1103	`attempt to access data outside segment bounds` A data record in an object module specified data extending beyond the end of a segment. This is a translator error. Note which translator (compiler or assembler) produced the incorrect object module and the circumstances in which it was produced. Please report this error to Microsoft Corporation using the Software Problem Report form at the back of this manual.
L1104	*filename* `: not valid library` The specified file was not a valid library file. This error causes **LINK** to abort.

Number	Linker Error Message
L1113	`unresolved COMDEF; internal error` Note the circumstances of the failure and contact Microsoft Corporation using the Software Problem Report form at the back of this manual.
L1114	`file not suitable for /EXEPACK; relink without` For the linked program, the size of the packed load image plus packing overhead was larger than that of the unpacked load image. Relink without the **/EXEPACK** option.
L2001	`fixup(s) without data` **A FIXUPP** record occurred without a data record immediately preceding it. This is probably a compiler error. (See the *Microsoft MS-DOS Programmer's Reference* for more information on **FIXUPP**.)

L2002	`fixup overflow near` *number* `in frame seg` *segname* `target seg` *segname* `target offset` *number*
	The following conditions can cause this error:
	• A group is larger than 64K.
	• The program contains an intersegment short jump or intersegment short call.
	• The name of a data item in the program conflicts with that of a subroutine in a library included in the link.
	• An **EXTRN** declaration in an assembly-language source file appeared inside the body of a segment, as in the following example:
	`code SEGMENT public 'CODE'`
	`EXTRN main:far`
	`start PROC far`
	`call main`
	`ret`
	`start ENDP`
	`code ENDS`

Number **Linker Error Message**

The following construction is preferred:

```
        EXTRN   main:far
code    SEGMENT public 'CODE'
start   PROC    far
        call    main
        ret
start   ENDP
code    ENDS
```

Revise the source file and recreate the object file. (For information about frame and target segments, refer to the *Microsoft MS-DOS Programmer's Reference.*)

L2003 `intersegment self-relative fixup`

An intersegment self-relative fixup is not allowed.

L2004 `LOBYTE-type fixup overflow`

A **LOBYTE** fixup generated an address overflow. (See the *Microsoft MS-DOS Programmer's Reference* for more information.)

L2005 `fixup type unsupported`

A fixup type occurred that is not supported by the Microsoft linker. This is probably a compiler error.

Note the circumstances of the failure and contact Microsoft Corporation using the Software Problem Report form at the back of this manual.

L2011 '*name*' : NEAR/HUGE conflict

Conflicting **NEAR** and **HUGE** attributes were given for a communal variable. This error can occur only with programs produced by the Microsoft FORTRAN Compiler or other compilers that support communal variables.

L2012 '*name*' : array-element size mismatch

A far communal array was declared with two or more different array-element sizes (for example, an array was declared once as an array of characters and once as an array of real numbers). This error cannot occur with object

Number	Linker Error Message
	files produced by the Microsoft Macro Assembler. It occurs only with the Microsoft FORTRAN Compiler and any other compiler that supports far communal arrays.
L2024	*name* : symbol already defined One of the special overlay symbols required for overlay support was defined by an object.
L2025	'*name*' : symbol defined more than once Remove the extra symbol definition from the object file.

L2029	unresolved externals
	One or more symbols were declared to be external in one or more modules, but they were not publicly defined in any of the modules or libraries. A list of the unresolved external references appears after the message, as shown in the following example:
	`EXIT in file(s):`
	` MAIN.OBJ (main.for)`
	`OPEN in file(s):`
	` MAIN.OBJ (main.for)`
	The name that comes before `in file(s)` is the unresolved external symbol. On the next line is a list of object modules that have made references to this symbol. This message and the list are also written to the map file, if one exists.

L4012	load-high disables EXEPACK The **/HIGH** and **/EXEPACK** options cannot be used at the same time.
L4015	/CODEVIEW disables /DSALLOCATE The **/CODEVIEW** and **/DSALLOCATE** options cannot be used at the same time.
L4016	/CODEVIEW disables /EXEPACK The **/CODEVIEW** and **/EXEPACK** options cannot be used at the same time.

Number	Linker Error Message
L4020	*name* : code-segment size exceeds 65500 Code segments of 65,501–65,536 bytes in length may be unreliable on the Intel 80286 processor.
L4021	no stack segment The program did not contain a stack segment defined with **STACK** combine type. This message should not appear for modules compiled with the Microsoft FORTRAN Compiler, but it could appear for an assembly-language module. Normally, every program should have a stack segment with the combine type specified as **STACK**. You can ignore this message if you have a specific reason for not defining a stack or for defining one without the **STACK** combine type. Linking with versions of the linker earlier than Version 2.40 might cause this message, since these linkers search libraries only once.
L4031	*name* : segment declared in more than one group A segment was declared to be a member of two different groups. Correct the source file and recreate the object files.
L4050	too many public symbols The **/MAP** option was used to request a sorted listing of public symbols in the map file, but there were too many symbols to sort (more than 3072 symbols by default). Relink using **/MAP:***number*. The linker produces an unsorted listing of the public symbols.
L4051	*filename* : cannot find library The linker could not find the specified file. Enter a new file name, a new path specification, or both.
L4053	VM.TMP : illegal file name; ignored **VM.TMP** appeared as an object-file name. Rename the file and rerun the linker.

Number	Linker Error Message
L4054	*filename* : cannot find file The linker could not find the specified file. Enter a new file name, a new path specification, or both.

B.3 LIB Error Messages

Error messages generated by the Microsoft Library Manager, **LIB**, have one of the following formats:

{*filename* | LIB} : fatal error U1*xxx*: *messagetext*
{*filename* | LIB} : warning U4*xxx*: *messagetext*

The message begins with the input-file name (*filename*), if one exists, or with the name of the utility. If possible, **LIB** prints a warning and continues operation. In some cases errors are fatal and **LIB** terminates processing. **LIB** may display the following error messages:

Number	LIB Error Message
U1150	page size too small The page size of an input library was too small, which indicates an invalid input **.LIB** file.
U1151	syntax error : illegal file specification A command operator such as a minus sign (–) was given without a following module name.
U1152	syntax error : option name missing A forward slash (/) was given without an option following it.
U1153	syntax error : option value missing The **/PAGESIZE** option was given without a value following it.

Number	LIB Error Message
U1154	`option unknown` An unknown option was given. Currently, **LIB** only recognizes the **/PAGESIZE** option.
U1155	`syntax error : illegal input` The given command did not follow correct **LIB** syntax as specified in Chapter 5, "Managing Libraries."
U1156	`syntax error` The given command did not follow correct **LIB** syntax as specified in Chapter 5, "Managing Libraries."

U1157	`comma or new line missing`
	A comma or carriage return was expected in the command line but did not appear. This may indicate an inappropriately placed comma, as in the following line:
	`LIB math.lib,-mod1+mod2;`
	The line should have been entered as follows:
	`LIB math.lib -mod1+mod2;`

U1158	`terminator missing` Either the response to the "Output library" prompt or the last line of the response file used to start **LIB** did not end with a carriage return.
U1161	`cannot rename old library` **LIB** could not rename the old library to have a **.BAK** extension because the **.BAK** version already existed with read-only protection. Change the protection on the old **.BAK** version.
U1162	`cannot reopen library` The old library could not be reopened after it was renamed to have a **.BAK** extension.

Number	LIB Error Message
U1163	`error writing to cross-reference file` The disk or root directory was full. Delete or move files to make space.
U1170	`too many symbols` More than 4609 symbols appeared in the library file.
U1171	`insufficient memory` **LIB** did not have enough memory to run. Remove any shells or resident programs and try again, or add more memory.
U1172	`no more virtual memory` Note the circumstances of the failure and notify Microsoft Corporation using the Software Problem Report form at the back of this manual.
U1173	`internal failure` Note the circumstances of the failure and notify Microsoft Corporation using the Software Problem Report form at the back of this manual.
U1174	`mark: not allocated` Note the circumstances of the failure and notify Microsoft Corporation using the Software Problem Report form at the back of this manual.
U1175	`free: not allocated` Note the circumstances of the failure and notify Microsoft Corporation using the Software Problem Report form at the back of this manual.
U1180	`write to extract file failed` The disk or root directory was full. Delete or move files to make space.

Number	LIB Error Message
U1181	`write to library file failed` The disk or root directory was full. Delete or move files to make space.
U1182	*filename* `: cannot create extract file` The disk or root directory was full, or the specified extract file already existed with read-only protection. Make space on the disk or change the protection of the extract file.
U1183	`cannot open response file` The response file was not found.
U1184	`unexpected end-of-file on command input` An end-of-file character was received prematurely in response to a prompt.
U1185	`cannot create new library` The disk or root directory was full, or the library file already existed with read-only protection. Make space on the disk or change the protection of the library file.
U1186	`error writing to new library` The disk or root directory was full. Delete or move files to make space.
U1187	`cannot open VM.TMP` The disk or root directory was full. Delete or move files to make space.
U1188	`cannot write to VM` Note the circumstances of the failure and notify Microsoft Corporation using the Software Problem Report form at the back of this manual.

Number	LIB Error Message
U1189	`cannot read from VM` Note the circumstances of the failure and notify Microsoft Corporation using the Software Problem Report form at the back of this manual.
U1200	*name* `: invalid library header` The input library file had an invalid format. It was either not a library file, or it had been corrupted.
U1203	*name* `: invalid object module near` *location* The module specified by *name* was not a valid object module.
U4150	*modulename* `: module redefinition ignored` A module was specified to be added to a library but a module with the same name was already in the library. Or, a module with the same name was found more than once in the library.
U4151	*symbol*(*modulename*) `: symbol redefinition ignored` The specified symbol was defined in more than one module.
U4152	*filename* `: cannot create listing` The directory or disk was full, or the cross-reference-listing file already existed with read-only protection. Make space on the disk or change the protection of the cross-reference-listing file.
U4153	*number* `: page size too small; ignored` The value specified in the **/PAGESIZE** option was less than 16.
U4155	*modulename* `: module not in library; ignored` The specified module was not found in the input library.

Number	LIB Error Message
U4156	*libraryname* : output-library specification ignored An output library was specified in addition to a new library name. For example, specifying `LIB new.lib+one.obj,new.lst,new.lib` where `new.lib` does not already exist causes this error.
U4157	*filename* : cannot access file **LIB** was unable to open the specified file.
U4158	*libraryname* : invalid library header; file ignored The input library had an incorrect format.
U4159	*filename* : invalid format *hexnumber;* file ignored The signature byte or word *hexnumber* of the given file was not one of the following recognized types: Microsoft library, Intel library, Microsoft object, or Xenix archive.

B.4 MAKE Error Messages

Error messages displayed by the Microsoft Program Maintenance Utility, **MAKE**, have one of the following formats:

{*filename*¦MAKE} : fatal error U1*xxx*: *messagetext*
{*filename*¦MAKE} : warning U4*xxx*: *messagetext*

The message begins with the input file name (*filename*), if one exists, or with the name of the utility. If possible, **MAKE** prints a warning and continues operation. In some cases, errors are fatal and **MAKE** terminates processing. **MAKE** generates the following error messages:

Number	MAKE Error Message
U1001	`macro definition larger than` *number* A single macro was defined to have a value string longer than the number stated, which is the maximum. Try rewriting the **MAKE** description file to split the macro into two or more smaller ones.

U1002	`infinitely recursive macro`
	A circular chain of macros was defined, as in the following example:
	`A=$(B)`
	`B=$(C)`
	`C=$(A)`

U1003	`out of memory` **MAKE** ran out of memory for processing the **MAKE** description file. Try to reduce the size of the **MAKE** description file by reorganizing or splitting it.
U1004	`syntax error : macro name missing` The **MAKE** description file contained a macro definition with no left side (that is, a line beginning with =).
U1005	`syntax error : colon missing` A line that should be an outfile/infile line lacked a colon indicating the separation between outfile and infile. **MAKE** expects any line following a blank line to be an outfile/infile line.
U1006	*targetname* `: macro expansion larger than` *number* A single macro expansion, plus the length of any string to which it may be concatenated, was longer than the number stated. Try rewriting the **MAKE** description file to split the macro into two or more smaller ones.

Number	MAKE Error Message
U1007	`multiple sources` An inference rule was defined more than once.
U1008	*name* `: cannot find file or directory` The file or directory specified by *name* could not be found.
U1009	*command* `: argument list too long` A command line in the **MAKE** description file was longer than 128 bytes, which is the maximum that DOS allows. Rewrite the commands to use shorter argument lists.
U1010	*filename* `: permission denied` The file specified by *filename* was a read-only file.
U1011	*filename* `: not enough memory` Not enough memory was available for **MAKE** to execute a program.
U1012	*filename* `: unknown error` Note the circumstances of the failure and notify Microsoft Corporation using the Software Problem Report form at the back of this manual.
U1013	*command* `: error` *errcode* One of the programs or commands called in the **MAKE** description file returned with a nonzero error code.
U4000	*filename* `: target does not exist` This usually does not indicate an error. It warns the user that the target file does not exist. **MAKE** executes any commands given in the block description, since in many cases the outfile will be created by a later command in the **MAKE** description file.
U4001	`dependent` *filename* `does not exist; target` *filename* `not built` **MAKE** could not continue because a required infile did not exist.

Number	MAKE Error Message
	Make sure that all named files are present and that they are spelled correctly in the **MAKE** description file.
U4013	*command* : error *errcode* (ignored) One of the programs or commands called in the **MAKE** description file returned with a nonzero error code, and **MAKE** was run with the **/I** option. **MAKE** ignores the error and continues.
U4014	usage : make [/n] [/d] [/i] [/s] [name=value ...] file **MAKE** has not been invoked correctly. Try entering the command line again with the syntax shown in the message.

B.5 EXEPACK Error Messages

Error messages in the Microsoft EXE File Compression Utility, **EXEPACK**, have one of the following formats:

{*filename* | EXEPACK} : fatal error U1*xxx*: *messagetext*
{*filename* | EXEPACK} : warning U4*xxx*: *messagetext*

The message begins with the input-file name (*filename*), if one exists, or with the name of the utility.

If possible, **EXEPACK** prints a warning and continues operation. In some cases, errors are fatal and **EXEPACK** terminates processing. Fatal errors have an exit code of 1.

EXEPACK generates the following error messages:

Number	EXEPACK Error Message
U1100	out of space on output file The disk or root directory is full. Delete or move files to make space.

Number	EXEPACK Error Message
U1101	*filename* `: file not found` The file specified by *filename* could not be found.
U1102	*filename* `: permission denied` The file specified by *filename* was a read-only file.
U1103	`cannot pack file onto itself` It is illegal to specify the same file for both input and output.
U1104	`usage : exepack <infile> <outfile>` The **EXEPACK** command line was not specified properly. Try again using the syntax shown.
U1105	`invalid .EXE file; bad header` The given file was not an executable file, or it had an invalid file header.
U1106	`cannot change load-high program` When the minimum allocation value and the maximum allocation value are both 0, the file cannot be compressed.
U1107	`cannot pack already-packed file` The file specified for **EXEPACK** had already been packed using **EXEPACK**.
U1108	`invalid .EXE file; actual length less than reported` The second and third fields in the file header indicated a file size greater than the actual size.
U1109	`out of memory` The **EXEPACK** utility did not have enough memory to operate.

Number	EXEPACK Error Message
U1110	`error reading relocation table` The file could not be compressed because the relocation table could not be found or was invalid.
U1111	`file not suitable for packing` The packed load image of the specified file was larger than the unpacked load image, so the file could not be packed.
U1112	*filename* `: unknown error` An unknown system error occurred while the specified file was being read or written. Try running **EXEPACK** again.
U4100	`omitting debug data from output file` **EXEPACK** strips symbolic debug information from the input file before packing.

You may also encounter DOS error messages if the **EXEPACK** program cannot read from, write to, or create a file.

B.6 EXEMOD Error Messages

Error messages from the Microsoft EXE File Header Utility, **EXEMOD**, have one of the following formats:

{*filename* | EXEMOD} `: fatal error U1`*xxx*`:` *messagetext*
{*filename* | EXEMOD} `: warning U4`*xxx*`:` *messagetext*

The message begins with the input-file name (*filename*), if one exists, or with the name of the utility. If possible, **EXEMOD** prints a warning and continues operation. In some cases, errors are fatal and **EXEMOD** terminates processing. **EXEMOD** generates the following error messages:

Number	EXEMOD Error Message
U1050	`usage : exemod file [-/h] [-/stack n] [-/max n] [-/min n]` The **EXEMOD** command line was not specified properly. Try again using the syntax shown. Note that the option indicator can be either a slash (/) or a hyphen (-). The single brackets ([ ]) in the error message indicate that your choice of the item within them is optional.
U1051	`invalid .EXE file : bad header` The specified input file is not an executable file or has an invalid file header.
U1052	`invalid .EXE file : actual length less than reported` The second and third fields in the input-file header indicate a file size greater than the actual size.
U1053	`cannot change load-high program` When the minimum allocation value and the maximum allocation value are both 0, the file cannot be modified.
U1054	`file not .EXE` **EXEMOD** automatically appends the **.EXE** extension to any file name without an extension; in this case, no file with the given name and an **.EXE** extension could be found.
U1055	*filename* `: cannot find file` The file specified by *filename* could not be found.
U1056	*filename* `: permission denied` The file specified by *filename* was a read-only file.
U4050	`packed file` The given file was a packed file. This is a warning only.

Number	EXEMOD Error Message
U4051	`minimum allocation less than stack; correcting minimum` If the minimum allocation value is not enough to accommodate the stack (either the original stack request or the modified request), the minimum allocation value is adjusted. This is a warning message only; the modification is still performed.
U4052	`minimum allocation greater than maximum; correcting maximum` If the minimum allocation value is greater than the maximum allocation value, the maximum allocation value is adjusted. This is a warning message only; the modification is still performed. **EXEMOD** will still modify the file. The values shown if you ask for a display of DOS header values will be the values after the packed file is expanded.

B.7 SETENV Error Messages

Messages generated by the Microsoft Environment Expansion Utility, **SETENV**, have the following format:

{*filename*¦ SETENV} : fatal error U1*xxx*: *messagetext*

The message begins with the input-file name (*filename*), if one exists, or with the name of the utility. **SETENV** generates the following error messages:

Number	SETENV Error Message
U1080	`usage : setenv <command.com> [envsize]` The command line was not specified properly. This usually indicates that the wrong number of arguments was given. Try again with the syntax shown in the message.

Number	SETENV Error Message
U1081	`unrecognizable COMMAND.COM` The **COMMAND.COM** file was not one of the accepted versions (IBM PC-DOS, Versions 2.0, 2.1, 2.11, 3.0, and 3.1).
U1082	`maximum for Version 3.1 : 992` The user specified a file that was recognized as **COMMAND.COM** for IBM PC-DOS, Version 3.1, and gave an environment size greater than 992 bytes, the maximum allowed for that version.
U1083	`maximum environment size : 65520` The environment size specified was greater than 65,520 bytes, the maximum size allowed.
U1084	`minimum environment size : 160` The environment size specified was less than 160 bytes, the minimum size allowed.
U1085	*filename* `: cannot find file` The specified file was not found, or it was a directory or some other special file.
U1086	*filename* `: permission denied` The specified file was a read-only file.
U1087	*filename* `: unknown error` An unknown system error occurred while the specified file was being read or written. Try running **SETENV** again.

B.8 ERROUT Error Messages

Messages that indicate errors on the command line used to invoke the compiler have one of the following formats:

```
command line error U1xxx: messagetext
execution error U2xxx: messagetext
```

ERROUT generates the following error messages:

Number	ERROUT Error Message
U1251	`no arguments` No arguments were specified to **ERROUT**.
U1252	`bad command line switch` An option other than **/f** was given on the **ERROUT** command line.
U1253	`missing file name` The **/f** option was given on the **ERROUT** command line without a file name.
U1254	`missing command` No *command* was given on the **ERROUT** command line.
U2251	`cannot open file` **ERROUT** could not open the given *stderrfile*.
U2252	`cannot redirect standard error` The *stderrfile* given on the **ERROUT** command line could not be used for standard error output.
U2253	`command failed` The *command* given on the **ERROUT** command line failed.

Appendix C
Using Exit Codes

C.1 Exit Codes with MAKE 395
C.2 Exit Codes with DOS Batch Files 395
C.3 Exit Codes for Programs 396
C.3.1 CodeView™ Exit Codes 396
C.3.2 LINK Exit Codes 397
C.3.3 LIB Exit Codes 397
C.3.4 MAKE Exit Codes 397
C.3.5 EXEPACK Exit Codes 397
C.3.6 EXEMOD Exit Codes 397
C.3.7 SETENV Exit Codes 398
C.3.8 ERROUT Exit Codes 398

Most of the utilities return some exit code (sometimes called an "error-level" code) that can be used by DOS batch files or other programs such as **MAKE**. If the program finishes without errors, it returns a code of 0. The code returned varies if the program encounters an error. This appendix discusses several uses for exit codes, and lists the exit codes that can be returned by each utility.

C.1 Exit Codes with MAKE

The Microsoft Program Maintenance Utility (**MAKE**) automatically stops execution if a program executed by one of the commands in the **MAKE** description file encounters an error. The exit code is displayed as part of the error message, unless a minus sign (–) precedes the command line in the **MAKE** file.

For example, assume the **MAKE** description file `TEST` contains the following lines:

```
TEST.OBJ :      TEST.FOR
      FL /c TEST.FOR
```

If the source code in `TEST.FOR` contains a program error (but not if it contains a warning error), you would see the following message the first time you use **MAKE** with the **MAKE** description file `TEST`:

```
make: CL /c TEST.FOR - error 2
```

This error message indicates that the command `CL /c TEST.FOR` in the **MAKE** description file returned exit code 2.

C.2 Exit Codes with DOS Batch Files

If you prefer to use DOS batch files instead of **MAKE** description files, you can test the code returned with the **IF ERRORLEVEL** command. The following sample batch file, called `COMPILE.BAT`, illustrates how to do this:

```
CL /c %1
IF NOT ERRORLEVEL 1 LINK %1;
```

```
IF NOT ERRORLEVEL 1 %1
```

You can execute this sample batch file with the following command:

```
COMPILE TEST.C
```

DOS then executes the first line of the batch file, substituting `TEST.C` for the parameter `%1`, as in the following command line:

```
CL /c TEST.C
```

It returns a code of 0 if the compilation is successful, or a higher code if the compiler encounters an error. In the second line, DOS tests to see if the code returned by the previous line is 1 or higher. If it is not (that is, if the code is 0), DOS executes the following command:

```
LINK TEST;
```

LINK also returns a code, which will be tested by the third line.

C.3 Exit Codes for Programs

An exit code of 0 always indicates execution of the program with no fatal errors. Warning errors also return exit code 0. Some programs can return various codes indicating different kinds of errors, while other programs return only 1 to indicate that an error occurred. The exit codes for each program are listed in Sections B.4.1–B.4.9.

C.3.1 CodeView™ Exit Codes

The Microsoft CodeView debugger does not return exit codes. However, it does display codes returned by programs that are run within the debugger. For example, if you run an executable file named `TEST.EXE` within the debugger and the program encounters an error that returns 1, you will see the following line:

```
Program terminated normally (1)
```

C.3.2 LINK Exit Codes

Code	Meaning
0	No error
1	Any **LINK** fatal error

C.3.3 LIB Exit Codes

Code	Meaning
0	No error
1	Any **LIB** fatal error

C.3.4 MAKE Exit Codes

Code	Meaning
0	No error
1	Any **MAKE** fatal error

If a program called by a command in the **MAKE** description file produces an error, the exit code will be displayed in the **MAKE** error message.

C.3.5 EXEPACK Exit Codes

Code	Meaning
0	No error
1	Any **EXEPACK** fatal error

C.3.6 EXEMOD Exit Codes

Code	Meaning
0	No error

1	Any **EXEMOD** fatal error

C.3.7 SETENV Exit Codes

Code	Meaning
0	No error
1	Any **SETENV** fatal error

C.3.8 ERROUT Exit Codes

Code	Meaning
0	No error
1	Any **ERROUT** fatal error

Index

> (CodeView prompt), 81
> (greater-than sign)
  CodeView prompt, 49, 50
  Redirected Output command, 259
[ ] (brackets)
  notational conventions, 6
  regular expressions, used in, 349
( ) (parentheses)
  FORTRAN, 92
& (ampersand)
  LIB command symbol, 306
* (asterisk)
  Comment command, 261
* (Asterisk)
  FORTRAN, 92
* (asterisk)
  regular expressions, used in, 350
* (asterisk), LIB command symbol, 303, 307, 311
** (asterisks)
  exponentiation operator, FORTRAN, 92
@ (at sign)
  Redraw command, 250
  register prefix, 105
^ (caret)
  Exponentiation operator, BASIC, 97
^ (caret), regular expressions, used in, 350, 351
: (colon), Delay command, 263
: (colon), LINK command, 271
: (colon) operator, 89, 93, 98, 106
, (comma)
  LIB command symbol, 301
  LINK command symbol, 271
– (dash)
  option designator, 19
  option designator,CodeView, 31
  regular expressions, used in, 349
$ (dollar sign), regular expressions, used in, 351
= (equal sign)
  assignment operator, 92
  Redirected Input and Output command, 260
! (exclamation point)
  command indicator, 49
  Shell Escape command, 255, 361
/ (forward slash)
  LINK option character, 279
< (less-than sign), Redirected Input command, 258
- (minus sign)
  FORTRAN, 92
– (minus sign), LIB command symbol, 302, 304, 307, 311
–* (minus sign-asterisk), LIB command symbol, 303
-+ (minus sign-plus sign), LIB command symbol, 302, 304, 311
(number sign), Tab Set command, 257
. (period)
  Current Location command, 211
  operator
    C, 89
    FORTRAN, 93, 362
  regular expressions, used in, 348
+ (plus sign)
  FORTRAN, 92
  LIB command symbol
    appending object files, 307, 310
    combining libraries, 312
    Intel, XENIX files, used with, 299
    specifying library, 304
  LINK command symbol, 271, 274
" (quotation marks)
  notational conventions, 6
  Pause command, 263
; (semicolon)
  LIB command symbol, 300, 301, 306, 313
  LINK command, 271
  LINK command symbol, 273, 274
/ (slash)
  division operator
    FORTRAN, 92
  option designator, 19
  option designator,CodeView, 31
  Search command, 253, 360, 362

_ (underscore), in symbol names, 90, 93
¦ (vertical bar), notational conventions, 6
10-byte reals
  dumping, 157
/2 CodeView option, 41
386 option, 72
/43 CodeView option, 40
7 (8087 command), 164
8087
  command, 164
  coprocessor, 164, 222
  stack, 165
8259 masking, 39
A (Assemble command), 220, 363
Absolute addresses, 106
Accessing bytes, 110
Adapters, using two, 41
Address ranges, arguments, used as, 107
Addresses
  absolute, 183
  arguments, used in, 106, 355, 360
  full, 106, 183
  segment start, 291
Alignment types, 291, 292
Ampersand (&), LIB command symbol, 306
.AND. operator, 92
Archives, XENIX, 299, 312
Arguments
  CodeView
    dialog commands, 81, 83
  dialog commands
    errors, 359, 363
  LINK options, 279
  program, 29
  routine, 75, 212
Arguments, CodeView
  program, 128
Arithmetic operator
  addition (+)
    FORTRAN, 92
  division (/)
    FORTRAN, 92
  multiplication (*)
    FORTRAN, 92
  subtraction (–)
    FORTRAN, 92
Arrays
Arrays *(continued)*
  copying, 236
ASCII characters
  displayed by CodeView, 151
ASCII characters, displayed by CodeView, 152
Assemble command, 219, 363
Assembly address, 220
Assembly mode
  display options, 71
  example, 207
  setting, 203
  using, 42, 360
Assembly programs. *See* Macro Assembler
Assembly rules, 220
Assignment operator
  BASIC, 98
  FORTRAN, 92
Asterisk (*)
  Comment command, 261
  multiplication operator
    FORTRAN, 92
  regular expressions, used in, 350
Asterisk (*), LIB command symbol, 303, 311
Asterisks (**)
  exponentiation operator, FORTRAN, 92
At sign (@)
  Redraw command, 250
  register prefix, 105
/B CodeView option, 6, 33

Backslash (\), Screen Exchange command
  using, 250
BACKSPACE key, 82
BASIC
  assignment operator, 98
  Case sensitivity, 99
  colon operator (:), 98
  concatenation, string, 97
  Constants, 99
  decimal notation, 99
  exponentiation operator (^), 97
  expressions, 97
  Floating-point numbers, 98
  hexidecimal notation, 99
  identifiers, 99

BASIC *(continued)*
intrinsic functions, 101
LET (assignment operator), 98
octal notation, 99
operators, 97
precedence of operators, 97
radix, 99
relational operators
string, 97
string operators, 97
strings, 100
symbols, 99
type conversion, 101
BASIC assignment operator, 98
BASIC compiler
CodeView support, 24
BASIC Constants, 99
BASIC exponentiation operator (^), 97
BASIC expression evaluator, 87
BASIC expressions
symbols, 99, 97
BASIC expressions. *See also* BASIC
BASIC functions
intrinsic, 101
BASIC identifiers, 99
BASIC intrinsic functions, 101
BASIC operators, 97
BASIC operators. *See also* BASIC
BASIC programs
compiling and linking, 25
include files, 24
preparing for CodeView, 24
statements, 24
writing source, 24
BASIC relational operators, 97
BASIC statements, 24
BASIC string operators, 97
BASIC strings, 100
BASIC symbols, 99
Batch files
exit codes, 395
BC (Breakpoint Clear command), 173, 355, 356
BD (Breakpoint Disable command), 174, 355, 356
BE (Breakpoint Enable command), 176, 355, 356
BEGDATA class name, 287
Binary Real, Microsoft format, 98, 103
BL (Breakpoint List command), 177
Black-and-white display, 6
Black-and-white display, CodeView, 33
Blocks of memory
copying, 236
filling, 234
moving, 236
Bold type, notational conventions, 4
BP (Breakpoint Set command), 170, 359, 363
Brackets ([ ])
notational conventions, 6
regular expressions, 349
Breakpoint Clear command
argument requirements, 355, 356
Run menu selection, 68
Breakpoint Clear Command
Run menu selection, 177
Breakpoint Clear command
using, 172
Breakpoint Disable command, 174, 355, 356
Breakpoint Enable command
argument requirements, 355, 356
using, 175
Breakpoint List command, 176
Breakpoint Set command
F9 function key, 54, 78
mouse, executing with, 58
using, 169
Breakpoints
address, 124
conditional, 69, 169
defined, 169
deleting, 172
displaying, 49, 170
Go command, used with, 123
listing, 176
BSS class name, 287
Buffer, CodeView command, 51
Buffer, command
CodeView, 82
BY operator, 110
/C CodeView option, 34

C compiler
CodeView support, 21, 22
C constants, 90
C expression evaluator, 87
C expressions
symbols, 89, 88
C identifiers, 89

C indirection operator (*), 89
C language
  colon operator (:), 89
  constants, 90
  expressions
    identifiers, 89, 88
  identifiers, 89
  indirection (*), 89
  levels of indirection, 89
  operators, 88
  period operator (.), 89
  precedence of operators, 88
  strings, 91
  symbols, 89
  underscore (_), in symbol names, 90
C language macros, 21
C language statements, 21
C operators, 88
C programs
  compiling and linking, 22
  include files, 21
  macros, 21
  preparing for CodeView, 21
  statements, 21
  writing source, 21
C programs. *See also* C language
C strings, 91
C symbols, 89
Call, defined, 8
Calling conventions, 213
Calls, 118, 121
Calls menu, 74, 213
Calls, stepping over, 121
Calls, tracing into, 118
Canonical frame number. *See* Frame number
Capital letters, notational conventions, 4, 6
Caret (^), regular expressions, used in, 350, 351
Case sensitivity
  BASIC expression evaluator, 99
  C symbols, 89
  CodeView, 83
  CodeView options, 19
  FORTRAN symbols, 93
  Macro Assembler options, 26, 72, 364
Case significance
  LINK, 269, 284
CL driver, 22
Class names
Class names *(continued)*
  BEGDATA, 287
  BSS, 287
  CODE, 287
  linking procedure, used in, 291
  STACK, 287
Class types, 291
Click, defined, 56
/CO linker option, 20
/CO option. *See* LINK options, /CODEVIEW
CODE class name, 287
CodeView
  8259 masking, 39
  absolute addresses, 106
  address expressions, 106
  address ranges, 107
  arguments
    dialog commands, 81, 83
  Arguments
    program, 29
  assembling and linking, 26
  BASIC operators. *See also* BASIC Operators
  breakpoints
    displaying, 49
  case sensitivity, 19, 83, 89, 93
  colon (:) operator, 106
  colon operator (:), 89, 93, 98
  command buffer, 51, 82
  command line, 28, 29
  compatibility, 36, 37
  compiler options
    /Od, 20
    /Zd, 20
    /Zi, 19, 20
  compiling and linking
    BASIC programs, 25
    C programs, 22
    FORTRAN programs, 23
  Constant Numbers. *See* Constant Numbers
  constant numbers. *See* Constant numbers
  CONTROL-BREAK, 117
  CONTROL-C, 117
  cursor, 81
  default
    expression format, 184
  default segment, 106
  defaults

defaults *(continued)*
address-range size, 149
radix, 212, 247, 248
start-up behavior, 29
type
enter command, 227
display. *See* CodeView display
error messages, 355
ESCAPE key, 56
executable file format, 18, 20
executable file, location of, 28
exit code, 125
exit codes, 123
function keys. *See* Function keys
graphics adapters, 36, 40, 41
graphics adapters, using two, 41
indirection levels, 89
instruction, current, 118, 120
languages supporting
BASIC, 24
C, 21
FORTRAN, 23
Macro Assembler, 25
line numbers expressions, 104
linker option
/CODEVIEW, 42
linker options
/CODEVIEW, 20
local variables, 87
memory operators, 109
menu options
Bytes Coded, 204
Mix Source, 204
Registers, 238
Symbols, 204
menus. *See* CodeView menus
menus. *See* Menus, CodeView
mixed-language support, 26
mouse. *See* Mouse
NMI trapping, 39
object ranges, arguments, used as, 108
Operators
BASIC. *See also* BASIC Operators
operators
BY, 110
DW, 111
memory, 109
OFF, 109
SEG, 109
WO, 110
CodeView *(continued)*
optimization, affect of, 20
options
/2, 41
/43, 40
/B, 33
/C, 34
command line, used in, 28
D, 39
/F, 35
/I, 39
/M, 38
/S, 35
summary, 31
/T, 37
/W, 37
Parameters, program, 29
period operator (.), 89, 93
program arguments, 128
radix. *See* Radix
register expressions, 105
regular expressions
searches, use in, 64
Restrictions, 17
sequential mode. *See* Sequential mode
Source-module files, location of, 28
source-module files, location of, 62
Start-up
command line, 28
commands, 34
file configuration, 27
symbolic information, 20
symbols, 89, 93, 99
syntax
summary, 245
SYSTEM-REQUEST key, 118
variables, local, 87
watch window. *See* Watch
window commands, 81
window mode. *See* Window mode
CodeView command line, 29
CodeView defaults
type
dump command, 150
CodeView display
! (exclamation point)
command indicator, 49
assembly mode
setting, 203, 207
CONTROL-D (separator line down), 51

CodeView display *(continued)*
CONTROL-U (separator line up), 51
cursor, 49, 81
dialog box, 50, 55, 61
Display mode, 117
display mode, 209
DOWN ARROW key (cursor down), 51
END key (scroll to bottom), 52
Exclamation point (!)
command indicator, 49
highlight, 49
HOME key (scroll to top), 52
menu bar, 49
message box, 50, 56, 61
mouse pointer, 50
output screen, 250
PGDN (scroll page down), 51
PGUP (scroll page up), 51
register window, 49, 52
scroll bar, 49
separator line, 49
set mode command, 52
UP ARROW key (cursor up), 51
window
commands, 50, 49
CodeView exit codes, 396
CodeView expressions
used as arguments, 87
CodeView expressions. *See also specific languages*
CodeView format prefixes, 135, 355
CodeView format specifiers
defined, 8
summary, 134
CodeView function keys. *See* Function keys
/CODEVIEW linker option, 20, 42
CodeView menus. *See* Menus, CodeView
/CODEVIEW option (LINK), 289
CodeView options
compiler
/Od, 20
/Zd, 20
/Zi, 19, 20
/F, 35
CodeView prompt, (>) greater-than sign, 49
CodeView string expressions. *See* Strings
Colon (:), Delay command, 263
Colon (:), LINK command, 271
Colon (:) operator, 106
Colon operator (:), 89, 93, 98
Color graphics adapter (CGA), 36, 40, 41
.COM extension, debugged files, used for, 29, 42, 361
Combine
types
COMMON, 292
LINK, 292
PRIVATE, 292
PUBLIC, 292
STACK, 292
Comma (,)
LIB command symbol, 301
LINK command symbol, 271
Command buffer, 51
Command line
CodeView, 28
LIB, 300
LINK, 269
COMMAND.COM, Shell command, used with, 63, 254
Commands
Assemble, 363
Breakpoint Clear
argument requirements, 355, 356
Breakpoint Disable, 355, 356
Breakpoint Enable, 355, 356
CodeView. *See* CodeView commands
Radix
limits, 356
setting, 100
Search
regular expressions, use with, 347
Shell Escape
space problem solutions, 361
Tracepoint
data object size limit, 361, 191
View, 360
Watch, 182, 355
Watch Delete, 196
Watch List, 197
Watchpoint, 186, 355
Commands, CodeView
8087, 164
Assemble, 219
Breakpoint Clear
Run menu selection, 67, 68

Breakpoint Clear *(continued)*
using, 172
Breakpoint Disable, 174
Breakpoint Enable, 175
Breakpoint List, 176
Breakpoint Set
F9 function key, 54, 78
mouse, executing with, 58
using, 169
Breakpoints
address, 124
buffer, using, 82
calls, stepping over, 121
calls, tracing through, 118
command buffer, 82
Comment, 261
Current Location, 211
Delay, 263
destination address, used with Go, 123
dialog, 49, 50, 81
Display Expression, 133
Dump
10-Byte Reals, 157
ASCII, 151
Bytes, 151
dump
default size, 149, 150
Dump
Double Words, 154
Integers, 152
Long Reals, 156
Short Reals, 155
Unsigned Integers, 153
Words, 154
Enter
ASCII, 228
Bytes, 227
default size, 226
Double Words, 231
Integers, 229
Long Reals, 233
Short Reals, 232
Unsigned Integers, 229
Words, 230
Enter commands
using, 223
ESCAPE key, 56
Examine Symbols, 143
Execute, 68, 126
Expression, 133
Commands, CodeView *(continued)*
Fill Memory, 234
Go
F5 function key, 53, 78
mouse, executing with, 59
using, 123
Goto
comment line, 123
F5 function key, 53
mouse, executing with, 58
using, 123
Help
66
76
F1 function key, 52
using, 245
window mode, 75, 77
move cursor down, 51
move cursor up, 51
Move Memory, 236
move separator line down, 51, 57
move separator line up, 51, 57
Output, Port, 237
Pause, 263
Port Output, 237
Program Step
F10 function key, 54, 78
mouse, executing with, 59
Program Step, using, 120
Quit, 63, 246
Radix
using, 247, 90, 94
Redirected Input, 258
Redirected Input and Output, 260
Redirected Output, 259
redirection, 34, 257
Redraw, 249
Register
changing register values, 238
displaying registers, 162
F2 function key, 52, 77
mouse, executing with, 59
Options menu selection, 67
Restart
Run menu selection, 68
using, 127
Screen Exchange
F4 function key, 53, 77
using, 250
scroll line down, 57
scroll line up, 57

Commands, CodeView *(continued)*
scroll page down, 51, 57
scroll page up, 51, 57
scroll to bottom, 52, 57
scroll to top, 52, 57
Search
menu selections, 64
using, 251
Set Mode
F3 function key, 52, 77
using, 203
View menu selection, 66, 67
Shell Escape
File menu selection, 63
using, 253
Stack Trace
display contents, 74
using, 212
stepping over calls, 121
T (Trace command), 119
Tab Set, 256
Trace
F8 function key, 53, 78
mouse, executing with, 58
using, 118
Tracepoint
sequential mode, 78, 69
tracing through calls, 118
Unassemble, 205
View, 208
Watch
menu selections, 69
sequential mode, 78
Watch Delete, 70
Watch Delete All, 70
Watch List, 78
Watchpoint
sequential mode, 78
watch menu selection, 69
window, 81
Comment
command, 261
Comment line, 123, 124, 169, 171
Compiler
error messages
correctable, 365
Compiler errors, and CodeView, 20
COMSPEC environment variable, 254
Concatenation, string
BASIC, 97
Conditional breakpoints, 69, 169, 181
CONFIG.SYS file, 363
Conjunction operator
FORTRAN, 92
Consistency checking (LIB), 301, 313
Constant numbers
BASIC, 99
C, 90
Constant Numbers
FORTRAN, 94
Constant numbers, arguments, used as, 357
CONTROL-BREAK, 54, 117, 191
CONTROL-BREAK, and CodeView, 39
CONTROL-C, 54, 81, 117, 191
CONTROL-C, and CodeView, 39
CONTROL-D (separator line down), 51
Controlling
data loading, 288
executable-file loading, 288
LINK, 279
number of segments, 286
stack size, 284
CONTROL-S, 82
CONTROL-U (separator line up), 51
Copying arrays, 236
Correctable error messages, 365
/CP option. *See* LINK options, /CPARMAXALLOC
/CPARMAXALLOC link option, 254
/CPARMAXALLOC option (LINK), 285
Cross-reference listing (LIB), 303, 312
Current Location command, 211
Current location line, 49
Cursor
CodeView, 81, 49
CV.EXE, location of, 27
CV.HLP, location of, 27, 75
/D CodeView option, 39

D (Dump command), 150
/D option (MAKE), 10, 324
DA (Dump ASCII command), 151
Dash (–)
option designator, 19
option designator,CodeView, 31
regular expressions, 349
Data segments
loading, 288

DB (Dump Bytes command), 151
DD (Dump Double Words command), 154
DEBUG, 10, 47, 76
Debugging, preparing for
  LINK (/CODEVIEW option), 289
Decimal notation
  BASIC, 99
  FORTRAN, 94
Default
  assembly-mode format, 71
  libraries
    ignoring, 277, 284
  responses
    LIB, 306
    LINK, 273
Defaults, CodeView
  address-range size, 149
  expression format, 184
  IBM Personal Computer, used with, 31
  radix, 212, 247, 248
  segment, 106
  start-up behavior, 29
  type
    dump command, 150
    enter command, 227, 184, 194
Delay command, 263
Description file, 4, 318
Destination address, Go command, used with, 123
DGROUP
  allocating memory below, 288
  segment order, 287
DI (Dump Integers command), 152
Dialog
  box, 50, 55, 61
  window, 49
Disjunction, inclusive, 92
Display
  mode, 207
Display Expression command, 133
Display mode, 117, 209
Dividing by zero, 358
DL (Dump Long Reals command), 156
/DO option. *See* LINK options, /DOSSEG
/DO option. *See* LINK options, /DSALLOCATE
Dollar sign ($), regular expressions, used in, 351
DOS
  program header, 5, 335
/DOSSEG option (LINK), 287
Double Words, 111
DOWN ARROW key (cursor down), 51
Drag, defined, 56
Drivers
  CL, 22
  FL, 23
DS (Dump Short Reals command), 155
DS register
  described, 288
/DSALLOCATE option (LINK), 288
DT (Dump 10-Byte Reals command), 157
DU (Dump Unsigned Integers command), 153
Dump address, 149
Dump commands
  10-Byte Reals, 157
  ASCII, 151
  Bytes, 151
  default size, 150
  Double Words, 154
  Integers, 152
  Long Reals, 156
  Short Reals, 155
  Unsigned Integers, 153
  using, 149
  Words, 154
DW (Dump Words command), 154
DW operator, 111

E
  Enter command, 226
  Execute command, 127
\ (backslash), Screen Exchange command
  using, 250
/E CodeView option, 40
/E option. *See* LINK options, /EXEPACK
EA (Enter ASCII command), 228
EB (Enter Bytes command), 227
Echo, redirection, used with, 259
ED (Enter Double Words command), 231
EI (Enter Integers command), 229
EL (Enter Long Reals command), 233
Ellipses, notational conventions, 5

END key (exit help), 76
END key (scroll to bottom), 52
Enhanced graphics adapter (EGA), 36, 40
Enter commands
  ASCII, 228
  Bytes, 227
  default size, 226
  Double Words, 231
  Integers, 229
  Long Reals, 233
  Short Reals, 232
  Unsigned Integers, 229
  using, 223
  Words, 230
Environment
  table
    enlarging, 8, 338
  variables
    LIB, 275, 276
.EQ. operator, 92
Equal sign (=)
  assignment operator
    FORTRAN, 92
  Redirected Input and Output command, 260
Equal-to operator
  FORTRAN, 92
Equivalence operator
  FORTRAN, 92
.EQV. operator, 92
Error
  internal debugger, 356, 358
  logical, 20
  syntax, 20
Error messages
  compiler
    correctable, 365
  ERROUT, 391
  EXEMOD, 387
  EXEPACK, 385
  LIB, 377
  LINK, 365
  MAKE, 382
  run-time
    redirecting, 9, 339
  SETENV, 389
Error messages, CodeView, 355
Errorlevel code
  program run from CodeView, 123
Errorlevel codes. *See* Exit codes
ERROUT
  described, 9, 339
  error messages, 391
  exit codes, 398
ES (Enter Short Reals command), 232
ESCAPE key, 56
EU (Enter Unsigned Integers command), 229
EW (Enter Words command), 230
Examine Symbols command, 143
Exclamation point (!)
  command indicator, 49
  Shell Escape command, 255, 361
.EXE Extension, debugged files, used for, 29
.EXE extension, debugged files, used for, 29, 42, 361
Executable
  files
    changing headers, 5, 335
    compressing, 3, 333
    extensions, 270
    loading, 288
    naming, default, 270
    naming with LINK, 270
    packing, 282
    specifying with LINK, prompts, 272
    specifying with LINK, response file, 274
  image, 290, 291
Executable file
  CodeView format, 18, 20
  CodeView start-up, required for, 29
  command line, used in, 29, 357, 361
Executable file, location of, 28
Execute command, 68, 126
EXEMOD
  described, 5, 335
  error messages, 387
  exit codes, 397
  /H option, 5, 335
  /MAX option, 6, 336
  maximum allocation, changing, 285
  /MIN option, 6, 336
  /STACK option, 5, 335
EXEPACK
  command line, 4, 334
  described, 3, 333
  error messages, 385
  exit codes, 397

EXEPACK *(continued)*
symbolic debug information, stripping, 4, 334
/EXEPACK link option, 365
/EXEPACK option (LINK), 282
Exit code
CodeView, 125
Exit codes
CodeView, 123, 396
DOS, 395
error level
0, 1, 395
program run from CodeView, 123
using, 395
Exit, DOS command, 63, 254
Expanded memory, 40
Exponentiation operator
BASIC, 97
exponentiation operator
FORTRAN, 92
Expression evaluation
Display Expression command, 133, 359
Expression evaluation, CodeView
expression requirement, 87
Expressions
arguments
error in, 358, 359
BASIC, 97
BASIC. *See also* BASIC
C, 88
C. *See also* C language
FORTRAN, 92
FORTRAN. *See* FORTRAN
Macro Assembler, 103
Macro Assembler. *See also* Macro Assembler
regular
searches, use in, 64, 251
specifying, 348, 349, 350, 351, 347, 360, 362
Expressions. *See specific languages*
Extensions
default, LINK, 269
executable files, 270
libraries, 269, 299, 300, 309
map files, 269, 271, 283
object files, 269, 270
Extensions, file, 87
/F CodeView option, 35
F (Fill Memory command), 234
/F option (FL), 284
F1 key (Help), 52, 76, 77, 246
F10 key (Program Step), 54, 61, 78, 121
F2 key (Register), 52, 77, 162, 238
F3 key (Set source/assembly), 77, 204
F3 key (Set source/mixed/assembly), 52
F4 key (Screen Exchange), 53, 77
F5 key (Go), 53, 61, 78, 124
F6 key (switch cursor), 50, 124
F7 key (Goto), 53, 124
F8 key (Trace), 53, 61, 78, 119
F9 key (Breakpoint Clear), 173
F9 key (Breakpoint Set), 54, 78, 175
Far-return mnemonic (RETF), 220
-* (minus sign-asterisk), LIB command symbol, 311
File
handles, 363
menu
Load, 62, 208, 360, 364
Quit, 63, 246
Shell, 63, 254, 361
File extensions, 87
File-name conventions, LINK, 269
Files
executable
naming with LINK, 270
map
creating, 283
frame numbers, 291
/MAP option (LINK), 283
object, 270
Fill Memory command, 234
Fixups, 293
FL driver, 23
FL options
/F, 284
/FPa, 276
/FPc
default libraries, overriding, 276
/FPc87
default libraries, overriding, 276
/Zd, 283
/Zi, 290
Flag bits
changing with the mouse, 59
values
changing, 238
displaying, 162, 355

Flag mnemonics, 239, 355
Flipping, screen, 35
Floating-point numbers
  Internal storage format, 98, 103
Format specifiers
  summary, 134
FORTRAN
  Assignment operator, 92
  colon operator (:), 93
  constants, 94
  decimal notation, 94
  Equivalence operator, 92
  exit codes, 398
  exponentiation (**), 92
  expressions, 92
  hexidecimal notation, 94
  identifiers, 93
  include files, 23
  intrinsic functions, 95
  octal notation, 94
  operators, 92
  Period operator (.), 93
  precedence of operators, 92
  radix, 94
  strings, 95
  symbols, 93
  type conversion, 95
  underscore (_), in symbol names, 93
FORTRAN compiler
  CodeView support, 23
FORTRAN constants, 94
FORTRAN expression evaluator, 87
FORTRAN expressions
  symbols, 93, 92
FORTRAN functions
  intrinsic, 95
FORTRAN identifiers, 93
FORTRAN instrinsic functions, 95
FORTRAN operators
  logical
    .AND., 92
    .EQV., 92
    .NEQV., 92
    .NOT., 92
    .OR., 92
  relational
    .EQ., 92
    .GE., 92
    .GT., 92
    .LE., 92
    .LT., 92
  relational *(continued)*
    .NE., 92
FORTRAN programs
  compiling and linking, 23
  preparing for CodeView, 23
  writing source, 23
FORTRAN strings, 95
FORTRAN symbols, 93
Forward slash (/)
  LINK option character, 279
Frame number, 291
Function calls, stepping over, 121
Function calls, tracing into, 118
Function keys
  CONTROL-D (separator line down), 51
  CONTROL-U (separator line up), 51
  down arrow (cursor down), 51
  End (exit help), 76
  End (scroll to bottom), 52
  F1 (Help), 52, 76, 77, 246
  F10 (Program Step), 54, 61, 78, 121
  F2 (Register), 52, 77, 162, 238
  F3 (Set source/assembly), 77, 204
  F3 (Set source/mixed/assembly), 52
  F4 (Screen Exchange), 53, 77
  F5 (Go), 53, 61, 78, 124
  F6 (switch cursor), 50
  F7 (Goto), 53
  F8 (Trace), 53, 61, 78, 119
  F9 (Breakpoint Clear), 173
  F9 (Breakpoint Set), 54, 78, 175
  Home (scroll to top), 52
  Home (top of help), 76
  PgDn (next help), 76
  PgDn (scroll page down), 51, 210
  PgUp (previous help), 76
  PgUp (scroll page up), 51
  up arrow (cursor up), 51
Functions
  calls, 75
  calls to, 213
  examining, 143
  intrinsic
    BASIC, 101
    FORTRAN, 95
  viewing, 74
Functions keys
  F6 (switch cursor), 124
  F7 (Goto), 124
"C, 90

"Decimal, 90
"Hexidecimal, 90
"Octal, 90

G (Go command), 124
.GE. operator, 92
Global symbols. *See* Public symbols
Go command
  F5 function key, 53, 78
  mouse, executing with, 59
  using, 123
Goto command
  comment line, 123
  F5 function key, 53
  mouse, executing with, 58
  using, 123
Graphics adapters, using two, 41
Graphics programs, debugging, 41, 259
Greater-than operator
  FORTRAN, 92
Greater-than sign (>)
  CodeView prompt, 49, 50
  Redirected Output command, 259
Greater-than-or-equal-to operator
  FORTRAN, 92
Groups
  DGROUP, 287
  linking procedures, used in, 293
.GT. operator, 92

H (Help command), 246
/H option
  EXEMOD, 5, 335
Hardware ports
  output to, 237
/HE option. *See* LINK options, /HELP
Help command
  F1 function key, 52
  sequential mode, 76
  Shell command, used with, 245
  using, 245
  view menu selection, 66
  window mode, 75, 76, 77
/HELP option
  LINK, 280
Hexidecimal notation
  BASIC, 99
  FORTRAN, 94
/HI option. *See* LINK options, /HIGH
/HIGH option (LINK), 288
Highlight, 49
HOME key (scroll to top), 52
HOME key (top of help), 76
Hyphen (-)
  FORTRAN, 92
/I CodeView option, 39
/I option
  LINK. *See also* LINK options, /INFORMATION
  MAKE, 10, 324

IBM PC
  CodeView compatibility with, 9, 36, 37
  recognizing, 31
Identifiers
  BASIC, 99
  C, 89
  FORTRAN, 93
Identifiers, arguments, used as, 364
IEEE format, 98, 103
Ignoring
  case (LINK), 284
  default libraries (LINK), 277, 284
Immediate operand, 221
Include files
  in assembly programs, 26
  in BASIC programs, 24
  in C programs, 21
  in FORTRAN programs, 23
Include files, and CodeView, 17
Inclusive disjunction operator
  FORTRAN, 92
IND (indefinite), 150
IND (indefinite), 150
Indentation, 256
Indirect register instructions, 222
Indirection levels, CodeView, 89
INF (infinity), 150
INF (infinity), 150
Inference rules, 15, 329
Infinity, 150
/INFORMATION option (LINK), 281
Initializing data, 234
Instruction, current, 118, 120
Instruction-name synonyms, 222
Integers, dumping, 152
Interrupt
  (DOS functions), 118

Intrinsic functions
  BASIC, 101
  FORTRAN, 95
Italics, notational conventions, 5

K (Stack Trace command), 214
Key names, notational conventions, 6

L (Restart command), 128, 360, 362
Labels
  defined, 8
  finding, 65, 252
.LE. operator, 92
Less-than operator
  FORTRAN, 92
Less-than sign (<), Redirected Input command, 258
Less-than-or-equal-to operator
  FORTRAN, 92
LET (assignment operator), BASIC, 98
Levels of indirection, CodeView, 89
/LI option. *See* LINK options, /LINENUMBERS
LIB
  addition commands, 308
  backup library file, 309
  change methods, 309
  combining libraries, 302, 310, 312
  consistency checking, 301, 313
  default responses, 306
  deletion commands, 308
  error messages, 377
  exit codes, 397
  extending lines, 306, 307
  extraction commands, 308
  library index, 309
  library modules
    adding, 302, 310
    deleting, 302, 311
    extracting, 303, 311
    extracting and deleting, 303, 311
    replacing, 302, 311
  listing files, 303, 309, 312
  operations, order of, 308
  options
    page size, specifying, 301, 313
    /PAGESIZE, 301, 313
  running
    command line, 300
  running *(continued)*
    prompts, 305
    response file, 307
  specifying commands, 301
  specifying output library, 304
  terminating, 308
  variable, 275, 276
LIB command symbols
  asterisk (*), 303, 307, 311
  minus sign (–), 302, 304, 307, 311
  minus sign-asterisk (–*), 303
  minus sign-asterisk (–*), 311
  minus sign-plus sign (-+), 302, 304, 311
  plus sign (+)
    appending object files, 307, 310
    combining libraries, 312
    specifying library, 304
    using, 301
Libraries
  backup, 309
  changing with LIB, 299, 309, 310
  combining, 302, 310, 312
  creating, 299, 309
  deleting object modules, 302, 311
  extensions, 269, 299, 300, 309
  extracting and deleting modules, 303, 311
  Intel, 299, 312
  LIB input, 300
  LIB output, 304
  listing (LIB), 303, 309, 312
  mixed-language programming, 27
  object modules
    including, 310
    replacing, 311
  object modules, including, 302
  object modules, replacing, 302
  search path, 275, 276
  specifying
    LIB command line, 300, 304
    LINK command line, 271
    LINK prompts, 272
    LINK response file, 274
  standard places, 276
Library manager. *See* LIB
Line number option
  LINK, 283
Line numbers, arguments, used as, 104
/LINENUMBERS option (LINK), 283
LINK

LINK *(continued)*
alignment types, 291
default
command line, 271
responses, 273
error messages
listed, 365
exit codes, 397
file-name conventions, 269
groups, 293
operation, 290
options
/CODEVIEW, 42
ignoring default libraries, 284
/INFORMATION (/I), 281
running
LINK command line, 269
prompts, 272
response file, 274
/STACK option, 5, 335
temporary output file, 277, 280
terminating, 278
with CodeView
C example, 22
FORTRAN example, 23
Macro Assembler example, 26
LINK options
abbreviations, 279
case sensitivity, 284
/CODEVIEW (/CO), 289
compatibility, preserving, 289
/CPARMAXALLOC (/CP), 285
data loading, 288
debugging, 289
displaying with /HELP (/HE), 280
/DOSSEG (/DO), 287
/DSALLOCATE (/DS), 288
executable-file loading, 288
/EXEPACK (/E), 282
/HELP (/HE), 280
/HIGH (/HI), 288
ignoring default libraries
277
line numbers, displaying, 283
/LINENUMBERS (/LI), 283
LINK prompts, responding to, 279
map file, 271, 283
/MAP (/M), 271, 283
/NODEFAULTLIBRARYSEARCH
(/NOD)
described, 284
/NODEFAULTLIBRARYSEARCH
(/NOD) *(continued)*
object files, used with, 277
/NOGROUPASSOCIATION
(/NOG), 289
/NOIGNORECASE (/NOI), 284
numerical arguments, 279
ordering segments, 287
overlay interrupt, setting, 287, 295
/OVERLAYINTERRUPT (/O), 287,
295
packing executable files, 282
paragraph space, allocating, 285
/PAUSE (/P), 280
pausing, 280
process information, displaying, 281
segments, 286
/SEGMENTS (/SE), 286
specifying on LINK command line,
277
stack size, setting, 284
/STACK (/ST), 284
Linker utility. *See* LINK
Listing
LINK options, 280
Listing files
LIB, 303, 309, 312
Load, menu selection, 128
Local variables, 87
Logical
error, 20
FORTRAN operator
.AND., 92
.EQV., 92
.NEQV., 92
.NOT., 92
.OR., 92
Long reals
dumping, 156
entering with CodeView, 233
Loops
tracepoints, used with, 195
watchpoints, used with, 191
.LT. operator, 92
Lvalue, 192
/M CodeView option, 38

M (Move Memory command), 236
/M option. *See* LINK options, /MAP
Macro Assembler

Macro Assembler *(continued)*
class type, 26
CodeView support, 25
expressions, 103
floating-point numbers, 103
include files, 26
macros, 26
operators, 103
real numbers, 103
segment directives, 26
writing source, 26, 42
Macro Assembler expressions, 103
Macro Assembler operators, 103
Macro Assembler programs
assembling and linking, 26
preparing for CodeView, 25
Macro definitions, MAKE, 11, 325
Macros
in C programs, 21
Macro Assembler, 26
MAKE
described, 3, 317
description file, 4, 318
error messages, 382
example, 8, 322
exit codes, 395, 397
inference rules, 15, 329
infile, 6, 320
macro
definitions, 11, 325
names, special, 14, 328
messages, 10, 324
options
/D, 10, 324
/I, 10, 324
/N, 10, 324
/S, 10, 324
using, 10, 324
outfile, 6, 320
running, 9, 323
Map file
/MAP (/M) option, (LINK), 271
Map files
creating, 283
extensions, 269, 271, 283
frame numbers, obtaining, 291
/MAP option, (LINK), 283
naming with LINK, 271
/MAP option (LINK), 271, 283
/MAX option (EXEMOD), 6, 336
Memory
Memory *(continued)*
copying blocks of, 236
filling blocks of, 234
moving blocks of, 236
Memory Operators, 109
Memory release, 254, 361
Menu bar, 49
Menus, CodeView
Calls
Stack Trace command, used with, 213
using, 74
defined, 49
File
Load, 62, 208, 364
Quit, 63, 246
Shell, 63, 254
keyboard selection from, 54
mouse selection from, 60
Options
386, 72
Bytes Coded, 72, 204
Case Sense, 72, 364
Flip/Swap, 71
Mix Source, 204
Registers, 162, 238
Symbols, 204
Run
Clear Breakpoints, 68, 172
Execute, 68, 126
Restart, 68, 128, 362
Start, 67, 128, 362
Search
Find, 64, 252
Label, 65, 252
Next, 65, 252
Previous, 65, 252
View
Assembly, 67, 203
Help, 66, 76, 245
Mixed, 67
Registers, 67
Source, 66, 203
Watch
Add Watch, 69, 183
Delete All, 70
Delete Watch, 70, 196
Tracepoint
Watch menu selection, 69, 193
Watchpoint, 69, 188
Message box, 50, 56, 61

Microsoft Binary Real, 98, 103
Microsoft LIB. *See* LIB
Microsoft LINK. *See* LINK
/MIN option (EXEMOD), 6, 336
Minimum allocation value, controlling, 6, 336
Minus sign (-)
  FORTRAN, 92
Minus sign (–), LIB command symbol, 311
Minus sign-asterisk (–*), LIB command symbol, 303
Minus sign-asterisk (–*), LIB command symbol, 311
Minus sign-plus sign (-+), LIB command symbol
  302 " "311"
Mixed mode, 203
Mixed-language programming
  CodeView, 26
Modules, examination, 143
Monochrome adapter (MA), 36, 40, 41
Mouse
  compatibility, 9
  driver, 39
  ignore option, 38
  pointer, 50, 56
  selecting with, 56
Move Memory command, 236
MSC, 22
/N option (MAKE), 10, 324

N (Radix command), 248, 356
Naming
  executable files
    default, 270
    LINK, 270
  map files, 271
NAN (not a number), 150
.NE. operator, 92
Negation operator
  FORTRAN, 92
.NEQV. operator, 92
NMI trapping, 39
/NOD option. *See* LINK
/NODEFAULTLIBRARYSEARCH option (LINK)
  284 " "277"
NOG option. *See* LINK options, /NOGROUPASSOCIATION
/NOGROUPASSOCIATION option (LINK), 289
/NOI option. *See* LINK options, /NOIGNORECASE
/NOIGNORECASE option (LINK), 284
Nonequivalence operator
  FORTRAN, 92
Nonproportional typeface, notational conventions, 5
.NOT. operator, 92
Notational conventions
  bold type, 4
  brackets, 6
  capital letter, 4
  ellipses, 5
  italics, 5
  listed, 5
  nonproportional typeface, 5
  quotation marks, 6
  sample screens, 6
  vertical bar, 6
Not-equal-to operator
  FORTRAN, 92
NUL, 303
Number sign (#), Tab Set command, 257
Numbers
  arguments, used as, 357
  floating point, 155, 156, 157
/O option. *See* LINK options, /OVERLAYINTERRUPT

Object files
  extensions, 269, 270
  naming
    default, 270
  object modules, difference from, 299
  specifying
    LINK command line, 270
    LINK prompts, 272
    LINK response file, 274
Object modules
  defined, 299
  library
    deleting from, 302, 311
    extracting and deleting from, 303, 311
    including in, 302, 310
  listing (LIB), 303, 312

Object modules *(continued)*
object files, difference from, 299
Object ranges, arguments, used as, 108
Octal notation
BASIC, 99
FORTRAN, 94
/Od compiler option, 20
OFF operator, 109
Offset, 109
Operands, machine instruction
displayed by CodeView, 163
Operators
addition (+)
FORTRAN, 92
BASIC, 97
BASIC. *See also* BASIC
C, 88
C. *See also* C language
division (/)
FORTRAN, 92
FORTRAN, 92
FORTRAN. *See* FORTRAN
Macro Assembler, 103
Macro Assembler. *See also* Macro Assembler
multiplication (*)
FORTRAN, 92
subtraction (-)
FORTRAN, 92
Operators. *See specific languages*
Optimization, and CodeView, 20
Optional fields, conventions for, 6
Options
CodeView
/S, 364
CodeView menu
Bytes Coded, 204
Mix Source, 204
Registers, 238
Symbols, 204
LINK
/CODEVIEW, 20
/CPARMAXALLOC, 254
linker
/EXEPACK, 365
menu
386, 72
Bytes Coded, 72
Case Sense, 72, 364
Flip/Swap, 71
Registers, 162
Options, CodeView
/2, 41
/43, 40
/B, 33
/C, 34
command line, used in, 28
D, 39
/F, 35
/I, 39
/M, 38
/S, 35
summary, 31
/T, 37
/W, 37
Options, LINK
/CODEVIEW, 42
Options, LINK. *See* LINK options
"Break when" point, 355
.OR. operator, 92
Out/dependent file descriptions, 4, 318
Output, Port (command), 237
Output screen
CodeView, 250
Output screen, CodeView, 35
/OVERLAYINTERRUPT option (LINK), 287, 295
Overlays
interrupt number, setting, 287, 295
overlay manager prompts, 295
restrictions, 295
search path, 295
specifying (LINK), 294

P (Program Step command), 121
Packed files, and CodeView, 17
Packing executable files, LINK, 282
Page size
library, 301, 313
/PAGESIZE option (LIB), 301, 313
Paragraph space, 285
Parameters, program, 29
Parentheses ( )
FORTRAN, 92
Pass count, 170, 177
PATH command
MAKE, used with, 16, 330
Setting up CodeView, 27
Pause command, 263
/PAUSE (/P) option (LINK), 280
Period (.)

Period (.) *(continued)*
Current Location command, 211
operator
FORTRAN, 93, 362
regular expressions, used in, 348
Period operator (.)
C, 89
PGDN (next help), 76
PGDN (scroll page down), 51, 210
PGUP (previous help), 76
PGUP (scroll page up), 51
Plus sign (+)
FORTRAN, 92
LIB command symbol
appending object files, 310
combining libraries, 302, 312
Intel, XENIX files, used with, 299
specifying library, 304
using, 302
LINK command symbol, 271
Plus sign (+)
LINK command symbol, 274
Point, defined, 56
Pointer, mouse, 50, 56
Port Output command, 237
Precedence of operators
BASIC, 97
Precedence, of operators
C, 88
Precedence of operators
FORTRAN, 92
Prefixes, format specifiers, used with, 135, 355
printf type specifiers, 184, 187, 192
Procedure calls, stepping over, 121
Procedure calls, tracing into, 118
Procedures
calls to, 213, 143, 213
Program arguments, CodeView, 128
Program header, inspection of, 5, 335
Program maintainer. *See* MAKE
Program Step command
F10 function key, 54, 78
mouse, executing with, 59
Program Step command, using, 120
Prompt (>), CodeView, 81
Prompt, CodeView, (>) greater-than sign, 49, 50
Protected-mode (80286) mnemonics, 205, 207, 219
Public names. *See* Public symbols
Public symbols, 42
Public symbols, listing
LIB, 303, 309, 312
LINK, 283

Q (Quit command), CodeView, 247
Quit command, 63
Quit command, CodeView, 246
Quotation marks (")
notational conventions, 6
Pause command, used as, 263

R (Register command), 162, 239, 355, 356
Radix
command
limits, 356
setting, 100
using, 247, 90, 94
current, 75, 90, 94, 99, 212
Radix command, 90, 94
Ranges, arguments, used as, 107
README.DOC file, 4
Redirecting error messages, 9, 339
Redirection
commands, 257
Redirected Input and Output command, 260
Redirected Input command, 258
Redirected Output command, 259
start-up commands, used in, 34
Redraw command, 249
References
long, 294
near segment-relative, 294
near self-relative, 294
resolving, 284, 293
short, 293
unresolved, 293
unresolved. *See* Unresolved references
Register
argument, used as, 105
command
changing register values, 238
displaying registers, 162
F2 function key, 52, 77
mouse, executing with, 59
View menu selection, 67

Register *(continued)*
variables, 89, 192, 362
Register prefix (@), 105
Register window, 49
Registers
DS
described, 288
Regular expressions
searches, use in, 64, 251
specifying, 347, 64, 347, 360, 362
Relational
FORTRAN operators
.EQ., 92
.GE., 92
.GT., 92
.LE., 92
.LT., 92
.NE., 92
Relational expressions, 187
Relational operators
string
BASIC, 97
Relocation information, 290
Response files
LIB, 307
LINK, 274
Restart command
Run menu selection, 68
using, 127, 360, 362
Restrictions
CodeView, 17
Return codes. *See* Exit codes
ROM (read-only memory), 118
Routines
arguments, value of, 212
calls to, 213
defined, 8, 213
Run menu
Clear Breakpoints, 68, 172
Execute, 68, 126
Restart, 68, 128, 362
Start, 67, 128, 362
Run time
error messages
redirecting, 9, 339
libraries, 299
Running
LIB
command line, 300
prompts, 305
response file, 307
Running *(continued)*
LINK
LINK command line, 269
prompts, 272
/S CodeView option, 35, 364
/S option (MAKE), 10, 324

S (Set Mode command), 204
Screen
buffer, 183
Exchange command
F4 function key, 77
movement commands, 51
notational conventions, 6
two, using, 41
Screen, CodeView
Exchange command
using, 250
Screen exchange
method, 35
Screen Exchange command
F4 function key, 53
Scroll bar, defined, 49
Search
command
menu selections, 64
regular expressions, use with, 347
using, 251
menu
Find, 64, 252
Label, 65, 252
Next, 65, 252
Previous, 65, 252
Search paths
libraries, 275, 276
overlays, 295
SEG operator, 109
Segments
alignment types, 291, 292
class names, 291
class types, 291
combine types, 292
combining, 292
number allowed, 286
order, 287, 291
/SEGMENTS option (LINK), 286
Semicolon (;)
LIB command symbol, 300, 301, 306, 313
LINK command symbol, 271, 273, 274

Separator line, 49
Sequential mode
  redirection, use with, 260
  starting, 37
  using, 76
Sequential mode, CodeView
  required when, 47
Set Block, DOS function call (# 4A), 254
Set Mode command
  F3 function key, 52, 77
  using, 203
  View menu selection, 66, 67
SETENV
  error messages, 389
  exit codes, 398
  utility, 8, 338
Shell Escape command
  File menu selection, 63
  space problem solutions, 361
  using, 253
Short reals
  dumping, 155
  entering with CodeView, 232
Slash (/)
  division operator
    FORTRAN, 92
  option designator, 19
  option designator,CodeView, 31
  Search Command, 253, 360, 362
Small capitals, notational conventions, 6
Source
  file, line-number arguments, used with, 104
  mode, 203, 360
Source-module files, location of, 28, 62
Special macro names, MAKE, 14, 328
/ST option. *See* LINK options, /STACK
Stack
  size, controlling, 5, 335
Stack, 8087, 165
STACK class name, 287
/STACK option
  (EXEMOD), 5, 335
  (LINK), 5, 284, 335
Stack size
  setting, 284
Stack Trace command
  display contents, 74
Stack Trace command *(continued)*
  using, 212
Standard places
  libraries, 276
Start-up
  code, 30, 63, 254
  command line, 357, 361
  file configuration
    CodeView, 27
Start-up routine, 285
Stopping
  library manager (LIB), 301, 308
  linker (LINK), 278
String concatenation
  BASIC, 97
String mnemonics, 220
String operators
  BASIC, 97
Strings, arguments, used as, 358
Strings, as arguments
  BASIC, 100
  C, 91
  FORTRAN, 95
Subprogram calls, stepping over, 121
Subprogram calls, tracing into, 118
Subprograms
  calls to, 213
Swapping disks
  during linking, 280
Swapping, screen, 35
Symbols
  arguments, used in, 364
  BASIC, 99
  C, 89
  defined, 8
  examining, 143
  FORTRAN, 93
  underscore (_), in names, 90, 93
SYMDEB, 10, 47, 76
Syntax
  error, 20
Syntax, CodeView
  summary, 245
Syntax conventions. *See* Notational conventions
SYSTEM-REQUEST key, 54, 118
/T CodeView option, 37

T (Trace command), 119
Tab Set command, 256

Text files, identifying, 360
Text strings
  finding, 64, 251, 347
TOOLS.INI file, 16, 330
TP (Tracepoint command), 193, 355, 361
Trace command
  F8 function key, 53, 78
  mouse, executing with, 58, 118
Tracepoint
  command
    data object size limit, 361
    sequential mode, 78
    Watch menu selection, 69, 355
Tracepoint command, 191
Tracepoint, defined, 191
Two-color graphics display, CodeView, 33
Type casting, 138, 356
Type specifiers, 184, 187, 192

U (Unassemble command), 206
Unassemble command, 205
Underscore (_), in symbol names, 90, 93
Unsigned integers, dumping, 153
UP ARROW key (cursor up), 51
Uppercase letters, notational conventions, 4
Utilities
  ERROUT. *See* ERROUT
  EXEMOD. *See* EXEMOD
  EXEPACK. *See* EXEPACK
  library manager. *See* LIB
  linker. *See* LINK

V (View command), 209, 360
Variables, local, 87, 182
Variables, local, and CodeView, 20
Vertical bar (¦), notational conventions, 6
Video modes, 364
Video-display pages, 35
View
  command, 208, 360
  menu
    Assembly, 67, 203
    Help, 66, 76, 245
    Mixed, 67
  menu *(continued)*
    Registers, 67
    Source, 66, 203
VM.TMP file, 277, 280
/W CodeView option, 37

W (Watch command), 184, 355
W (Watch List command), 198
WAIT instruction, 222
Watch
  menu
    Add Watch, 69
    Delete All, 70
    Delete Watch, 70
    Tracepoint, 69
    Watchpoint, 69
  statements
    defined, 49
  window, 49
Watch command
  menu selections, 69
  sequential mode, 78, 182, 355
Watch Delete All command, 70
Watch Delete command, 70, 196
Watch expression statement, 183
Watch List command, 78, 197
Watch memory statement, 183
Watch menu
  Add Watch, 183
  Delete Watch, 196
  Tracepoint, 193
  Watchpoint, 188
Watch Statement commands, 181
Watch statements
  defined, 49
  deletion, 196
  listing, 197, 181
Watch window, 182
Watchpoint
  command
    sequential mode, 78
    watch menu selection, 69, 355
  defined, 355
Watchpoint command, 186
Watchpoint, defined, 186
Window
  commands, 50
  mode
    starting, 37
Window commands, 81

Window mode, CodeView
  required when, 47
WO operator, 110
Words (units of memory), 110
WP (Watchpoint command), 188, 355

X (Examine Symbols command), 143

Y (Watch Delete command), 197
/Zd compiler option, 20
/Zd option (FL), 283

Zero, dividing by, 358
/Zi compiler option, 19, 20
/Zi option (FL), 290