
MICROSOFT®
The High Performance Software™

# Microsoft® CodeView™

## Window-Oriented Debugger

## 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 Number 410840010-400-R00-0486
Part Number 048-014-036

# Contents

# Figures

# Tables

# Chapter 1

# Introduction to the Microsoft® CodeView™ Debugger

# 1.1 Introduction

The Microsoft® CodeView™ debugger is a debugging program that helps you test executable files developed with the Microsoft C Compiler. This chapter introduces you to CodeView debugger, and summarizes the manual and the conventions used in it.

# 1.2 Overview

The CodeView debugger can display and execute program code, control program 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.

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 (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 Debugging Utility (**SYMDEB**) or the **DEBUG** utility provided with MS-DOS®. The CodeView user interface shares some features with its predecessors, but also incorporates powerful new features such as popup menus, multiple windows, mouse support, and single-keystroke commands.

## 1.3 About This Manual

This manual explains how to use the CodeView debugger to examine programs and locate program errors. It is a companion manual to the *Microsoft C Compiler User's Guide*, the *Microsoft C Compiler Language Reference*, and the *Microsoft C Compiler Run-Time Library Reference*. See those manuals for information about developing C programs.

Although the manual focuses on debugging C programs using source code, the CodeView debugger can also debug assembly-language programs, or it can debug C programs at the assembly-language level. If you are not familiar with assembly-language programming, some aspects of the debugger may be unfamiliar (assembly mode and the register window, in particular).

However, all CodeView features relating to assembly language are optional. You can simply ignore the assembly-mode features and debug your programs in the C source mode. If you wish to learn more about assembly-language programming, you should consider purchasing the Microsoft Macro Assembler and reading its manual along with one of the many tutorial books on assembly-language programming.

The following list tells where to find information on various aspects of CodeView:

| For This Information: | See: |
|---|---|
| Compiling and linking C programs in the special format required by the CodeView debugger, and invoking the debugger with various command-line options | Chapter 2, "Getting Started" |
| Using elements of the CodeView display, including windows, popup menus, and the mouse | Chapter 3, "The CodeView Display" |
| Specifying arguments for dialog commands and using the CodeView operators to create expressions | Chapter 4, "Using Dialog Commands" |
| 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 or procedure 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" |

In addition to the information above, the following information is included in the appendixes:

| For This Information: | See: |
|---|---|
| A summary of CodeView modes, commands, and menus | Appendix A, "Command and Mode Summary" |

| | |
|---|---|
| How to use regular expressions to find variable text strings in a source file | Appendix B, "Regular Expressions" |
| A list of CodeView error messages | Appendix C, "Error Messages" |
| Definitions of terms used in the manual | Glossary |

---

*Important*

There may be additional information about the CodeView debugger in the **README.DOC** file provided on your Microsoft C Compiler distribution disk. This file will describe any additions to the documentation or changes made to the program after the manual was printed.

---

## 1.4 Notational Conventions

The following notational conventions are used throughout this manual:

| Convention | Meaning |
|---|---|
| **Bold** | Bold type indicates text that must be typed as shown. Text that is shown in bold type includes the following: operators, keywords, and standard functions. Examples are shown below:<br>**+=** **_setargv**<br>**if** **sizeof**<br>**main** **int** |
| **BOLD CAPITALS** | Bold capital letters are used for registers, environment variables, and the names of executable files and files provided with the product. Commands typed at the MS-DOS level are also capitalized. These commands include built-in MS-DOS commands such as **SET**, as well as program names such as **CV**. You are not required to use capital letters when you actually enter these commands. |

*Italics*

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 typeface so that they will look more like the programs you create with a text editor or the output of commonly used computer printers.

`User Input`

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
>
```

Ellipsis dots

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;
.
.
.
*pc++;
.
.
.
count = 0;
```

[[Double brackets]]

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*.

Single brackets are used to indicate brackets used by C array declarations and subscript expressions. For instance, `a[10]` is an example of brackets in a C subscript expression.

Vertical bar

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.

Some C constructs require quotation marks. Quotation marks required by the language have the form "" rather than “ ”. For example, a C string used in an example would be shown in the following form:

```
"abc"
```

SMALL CAPITALS

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 2.5.1, "Starting with a Black-and-White Display," for more information). Figure 1.1 shows an example. Screens will be slightly different if you use a color monitor in color mode.

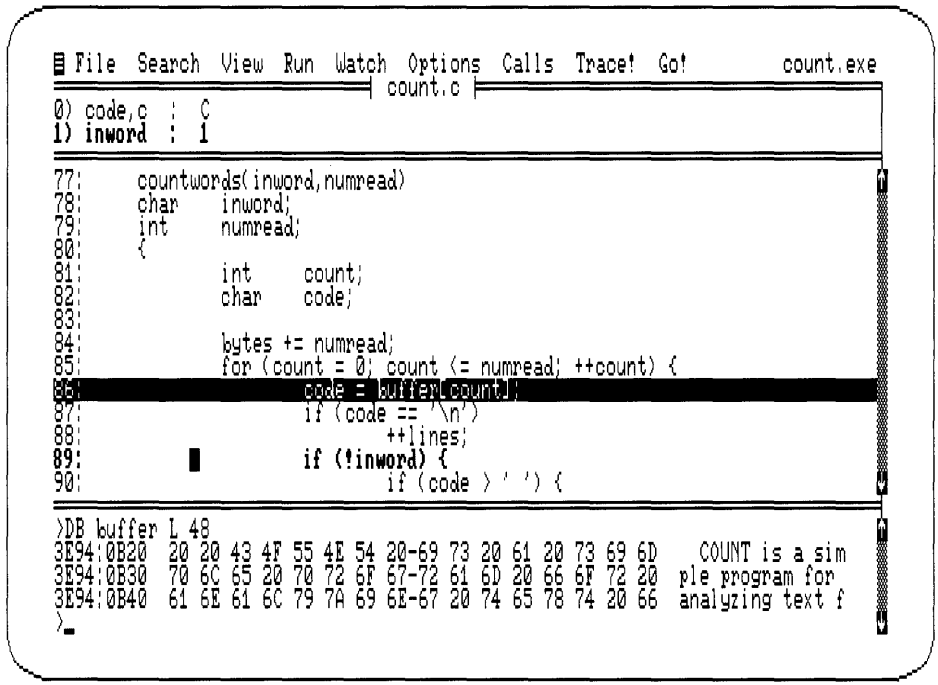


**Figure 1.1 CodeView Screen in Window Mode**

# Chapter 2
# Getting Started

# 2.1 Introduction

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 operation of the debugger.

This chapter describes a disk-based tutorial session that introduces you to the CodeView debugger. The chapter also describes how to compile and link a C program to produce an executable file in the CodeView format, and how to load a program into the CodeView debugger. Using the debugger with assembly-language programs is also discussed briefly.

# 2.2 Starting the Sample Session

The sample session provided on the demonstration disk illustrates many reatures of the CodeView debugger. You may wish to try this session before reading further in the manual.

To start the session, log on to the drive containing the sample disk and type `SAMPLE`. This starts the batch file **SAMPLE.BAT**, which displays explanatory text describing the purpose of the session.

During the session, the batch file automatically loads a program into the debugger. Debugging commands are then redirected from a text file into the debugger. Throughout the session, comments will tell you what is happening and why. Periodically, you will be asked to press a key to continue; this allows you to control the pace of the session.

---

*Note*

If you wish to quit before the sample session is finished, press CONTROL-C or CONTROL-BREAK. If you are still in the batch file, a prompt appears asking if you want to terminate. Enter `Y` for yes. If the CodeView debugger has been started, the work `break` appears, followed by the CodeView prompt (>). Enter `Q` (the Quit command.) This returns you to the MS-DOS operating system.

---

# 2.3 Preparing C Programs for the CodeView Debugger

Executable files must be in a special format in order to be used with the CodeView debugger. The special-format files contain line-number information and a symbol table in addition to executable code. You must use the correct options to put this additional information into the object files during compilation, and then into the executable file during linking.

If you try to debug an executable file that does not contain this additional information, the debugger will not be able to interpret symbols or to correlate code addresses and source line numbers. You can still debug your program in assembly mode, but many of the most powerful features of the CodeView debugger will be disabled.

You must compile your C programs with one of the two compilation-control programs provided with the Microsoft C Compiler: **MSC** or **CL**. You must link your program with the Microsoft Overlay Linker, **LINK**. See the *Microsoft C Compiler User's Guide* for complete instructions on compiling and linking. The special steps required for compiling and linking programs for the CodeView debugger are explained in the sections 2.3.1–2.3.3.

## 2.3.1 Writing C Source Code

Any source code that is legal in C can be compiled to an executable file and debugged with the CodeView debugger. However, some programming practices make debugging more difficult. You should be aware of how these practices affect debugging.

The CodeView debugger will be easier to use if you put each source statement on a separate source line. For example, the following code is legal in C:

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

This code is actually made up of three separate C statements. When they are placed together on the same line, the separate statements cannot be accessed individually during source debugging. You would not be able to put a breakpoint on the statement `++lines;` or execute to the statement `if (code == '\n')`. (A breakpoint is an address that stops program execution each time the address is encountered.) The same code would be easier to debug if it were written in the following form:

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

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

Macros cannot be easily debugged with the CodeView debugger. This is not a problem with most simple macros, but if you have complex macros with potential side effects, you may need to write them first as regular source statements. After they are debugged, you can put them in macros.

The C language also permits you to put code in separate include files and then read the files into your source file using the #**include** directive. However, you will not be able to use the CodeView debugger to debug source code in include files. Therefore, it is better to use include files for macros only. The preferred method of using standard code in C is to write and compile separate library modules, then link the resulting object files with your programs. The CodeView debugger does support this technique.

### 2.3.2 Compiling Source Files for the CodeView Debugger

When you compile a source file for a program you want to debug, you must specify the /**Zi** option in response to the **MSC** prompts, or in an **MSC** or **CL** command line. The /**Zi** option instructs the compiler to include line-number and symbol 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. The completed executable file will be significantly smaller than if you had compiled all options with /**Zi**. 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, you will probably want to specify the /**Od** option. This option 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.

Note that you cannot debug a program until you compile it successfully. The CodeView debugger will not help you correct syntax or compiler errors. You should refer to a source listing and the appropriate reference books to correct your code until you can compile it successfully; then use the debugger to locate logical errors in the program. Finally, you can correct these errors in the source code and recompile.

### ■ Example

```
MSC COUNT /Zi /Od;
```

This example compiles the source file **COUNT.C** to produce an object file called **COUNT.OBJ**. The object file contains line-number information, a symbol table, and unoptimized object code.

## 2.3.3 Linking Object Files for the CodeView Debugger

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

No other option is necessary for the debugger, but you may use other options if your program requires them (see the *Microsoft C Compiler User's Guide*). For example, you could use the **/MAP** or **/PAUSE** option.

---

*Warning*

You should not use the **/EXEPACK** option with the CodeView debugger, since this option strips all symbolic information from the executable file. The **EXEPACK** utility also strips symbolic information. If the debugger detects a packed file on start-up, you will see a warning message. You can still attempt to debug the file in assembly mode, but there will be no symbolic information.

---

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

■ **Examples**

```
LINK /CO COUNT;

CL /Zi /Od COUNT.C
```

The first example links the object file **COUNT.OBJ** to form an executable file, **COUNT.EXE**, containing the symbolic and line-number information required by the CodeView debugger.

The second example uses the **CL** command to compile and link an executable file in one operation. When you use the **CL** command, you do not need to specify the **/CODEVIEW** link option, since the program automatically supplies this option when you specify the **/Zi** compile option.

## 2.4 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 C source 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 C compiler files according to the suggestions in the *Microsoft C Compiler User's Guide*, you could put **CV.EXE** in the **\BIN** directory. |
| **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 C compiler files according to the suggestions in the *Microsoft C Compiler User's Guide*, you could 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. |
| *program*.**C** | Normally source files should be in the current directory. However, if you give 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 `MSC DEMO`, the CodeView debugger will expect the source file to be in the current directory. However, if you compiled with the command line `MSC \C\DEMO`, then the debugger will expect the source file to be in directory `\C`. If the debugger cannot find the source file in the directory specified by 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 MS-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 2.5. 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
```

C 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. Programs that use overlays cannot be debugged with the CodeView debugger.

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**. 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 **SIEVE.EXE**, you could start the debugger with the following command line:

```
CV SIEVE
```

If you give this command line on an IBM Personal Computer, window mode will be selected automatically. The display will look like Figure 2.1, shown on the following page.

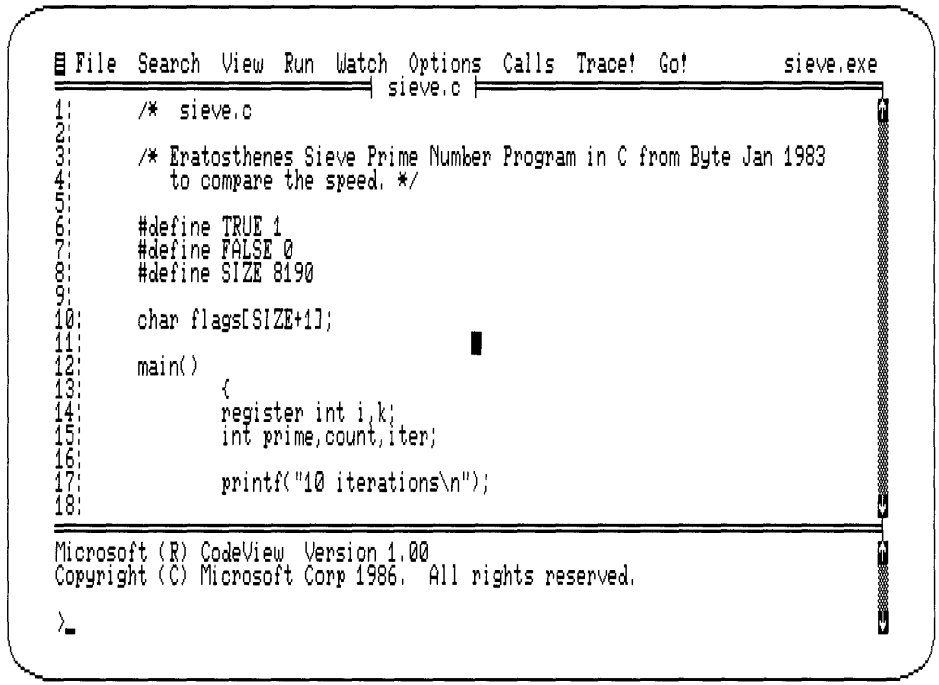


**Figure 2.1 CodeView Start-Up Screen in Window Mode**

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 1.00
Copyright (C) Microsoft Corp 1986. All rights reserved.

>
```

You can use CodeView options, as described in Section 2.5, to override the default start-up mode.

If your program is written in C, the CodeView debugger is now at the beginning of the C 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 through 11.

## 2.5 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 (–). A file containing a dash in its file name must be renamed before use with CodeView 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, and by checking for PC-DOS, the IBM version of MS-DOS. If you use PC-DOS on certain IBM-compatible computers, the CodeView debugger may think you have an IBM computer and act accordingly.

---

The list on the following page 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, do 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 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) | **/43** |
| 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 C start-up code automatically each time. You could create a batch file called GRAFBUG.BAT consisting of the following line:

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

The CodeView options are described in more detail in sections 2.5.1–2.5.6.

## 2.5.1 Starting with a Black-and-White Display

### ■ Option

**/B | –B**

The **/B** option forces the CodeView debugger to display in two colors even if you have a CGA. The debugger checks upon start-up to see what kind of display adapter is attached to your computer. If the debugger detects a monochrome adapter, 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 COUNT COUNT.TXT
```

The 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.

## 2.5.2 Specifying Start-Up Commands

### ■ Option

*/Ccommands* | *–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 debugger 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 MS-DOS.

---

### ■ Examples

```
CV /CGmain COUNT.EXE COUNT.TXT

CV "/CS-;N16;G countwords;D buffer L 100" COUNT.EXE COUNT.TXT

CV "/C<INPUT.FIL" COUNT.EXE COUNT.TXT
```

The first example loads the CodeView debugger with `COUNT.EXE` as the executable file and `COUNT.TXT` as the argument. Upon start-up, the debugger executes the C 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.

The second example loads the same file with the same argument, but the command list is more extensive. The debugger starts in assembly mode (`S-`) with a radix of 16 (`N 16`). It executes to procedure `countwords` (`G countwords`), then dumps 100 bytes, starting at the variable `buffer` (`D buffer L 100`). Since several of the commands use spaces, the entire option is enclosed in double quotation marks.

The third example loads the same file and argument, 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 MS-DOS.

## 2.5.3 Setting the Screen-Exchange Mode

### ■ Options

**/F | –F**
**/S | –S**

The CodeView debugger allows you to move quickly back and forth from 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 (MAs).

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 only works 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 4K if you have an MA, or 16K if you have a CGA or EGA. The amount of memory used by the CodeView debugger is increased by the size of the buffer.

Table 2.1 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 2.5.4, "Enabling Window or Sequential Mode."

**Table 2.1**

**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 COUNT COUNT.TXT

CV /S GRAFIX
```

The first 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.

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

## 2.5.4 Enabling Window or Sequential Mode

### ■ Options

**/T | -T**
**/W | -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 2.1 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

CV /T SIEVE
```

The first 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.

The second 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.

## 2.5.5 Turning Off the Mouse

### ■ Option

**/M** | **-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 have a version of the Microsoft Mouse prior to 5.02. The latest version of the mouse driver program (**MOUSE.SYS**) is included on the Microsoft C Compiler distribution disk. You should replace your old mouse driver program with this updated version. You will then be able to use the mouse both with the CodeView debugger and the program you are debugging. See your Microsoft Mouse user's guide for information on installing **MOUSE.SYS**. This file will not work with pointing devices from other manufacturers.

## 2.5.6 Using the Enhanced Graphics Adapter's 43-Line Mode

### ■ Option

**/43 | –43**

If you have an EGA, you can use the **/43** option to enable a 43-line-by-80-column text mode. You cannot not use this mode 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 25-line-by-80-column 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 COUNT COUNT.TXT
```

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

# 2.6 Using the CodeView Debugger with the Macro Assembler

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 normal. No special options are required, and all assembly options are allowed.
3. Link with the linker provided with Version 4.0 of the Microsoft C Compiler. 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 C library routines or assembly-language modules called by your C program.

# Chapter 3
# The CodeView Display

# 3.1 Introduction

The CodeView screen display can appear in two different modes: window and sequential. Both modes provide 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, popup 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 3.2.

Sequential mode should be familiar to most programmers. It is similar to the display mode of the CodeView debugger's predecessors, the Microsoft Symbolic Debugging Utility (**SYMDEB**) and the MS-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 3.3.

# 3.2 Using Window Mode

Figure 3.1 shows the CodeView window-mode display with all windows open.

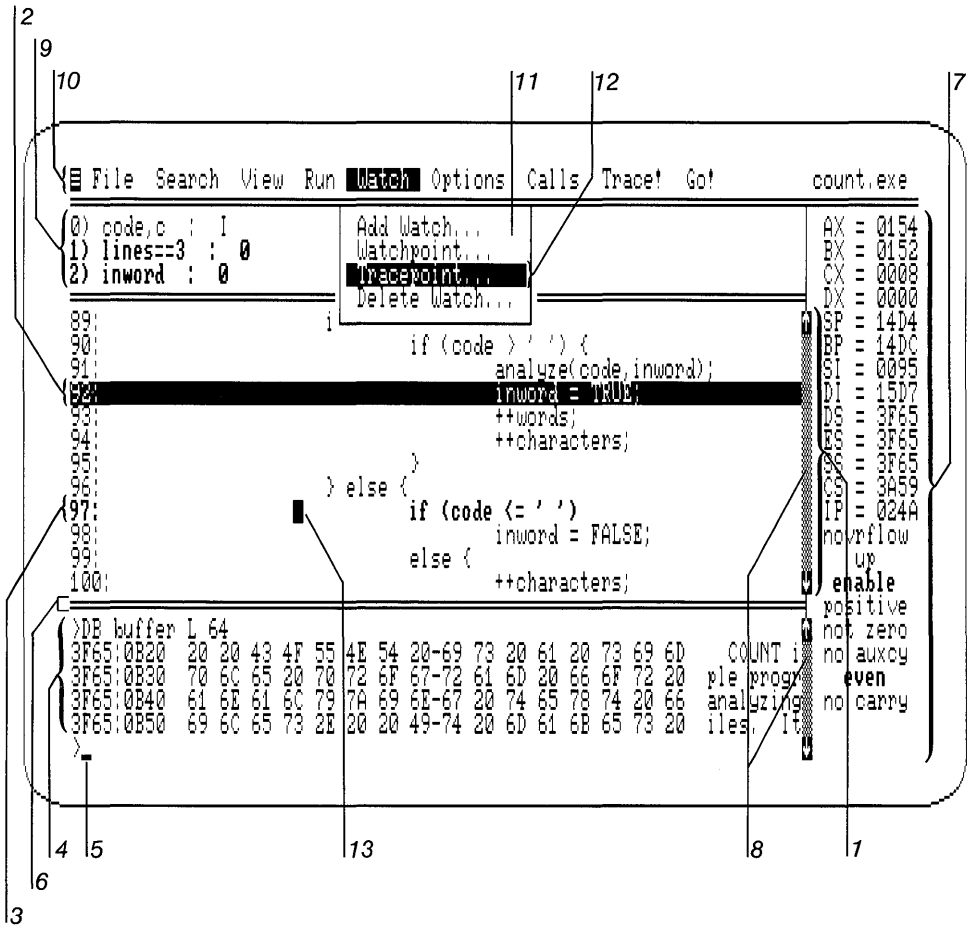


**Figure 3.1 Elements of the CodeView Debugging Screen**

The elements of the CodeView display marked in Figure 3.1 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.
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.

## 3.2.1 Executing Window Commands with the Keyboard

CodeView 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 4, "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 3.2.2. The window commands available from the keyboard are described by category in the following sections. For a table of commands by name, see Appendix A, "Command and Mode Summary."

### 3.2.1.1 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. |

### 3.2.1.2 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 3.2.4. You can also get to the initial help screen by selecting Help from the View menu, as described in Section 3.2.3.3. |
| 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 3.2.3.6. |
| F3 | Switches between source and assembly modes. Source mode shows source code in the display window, while assembly mode shows assembly-language instructions. You can also change modes with the Source and Assembly selections from the View menu, as described in Section 3.2.3.3. |
| 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. You can also change to the output screen with the Output selection from the View menu, as described in Section 3.2.3.3. |

### 3.2.1.3 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 function, procedure, or interrupt call, 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 MS-DOS function calls (interrupt 0x21). |
| 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 function, procedure, or interrupt call, 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.

---

### 3.2.1.4 Selecting from Menus with the Keyboard

The CodeView debugger has seven popup menus. This section discusses how to make selections from menus. The effects of the selections are discussed in Section 3.2.3.

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, and 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 Section 3.2.3.2, "Using the Search Menu," or Chapter 10, "Modifying Code or Data," for an 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.

## 3.2.2 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.

### 3.2.2.1 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 3.2.3 for a description of the menu selections.

### 3.2.2.2 Controlling Program Execution with the Mouse

By clicking on 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 as follows 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 function, procedure, or interrupt call, 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 MS-DOS function calls (interrupt 0x21). |
| 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 function, procedure, or interrupt call, CodeView 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 only different if the current location is the start of a procedure, interrupt, or function 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 3.2.1.3 for more information.

---

### 3.2.2.3 Selecting from Menus with the Mouse

The CodeView debugger has seven popup menus. This section discusses how to make selections from these menus. The effect of each selection is discussed in Section 3.2.3.

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 3.2.3.2, "Using the Search Menu," or Appendix B, "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.

## 3.2.3 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 3.2.1, or with the mouse, as described in Section 3.2.2.

### 3.2.3.1 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 3.2, and the selections are explained below:

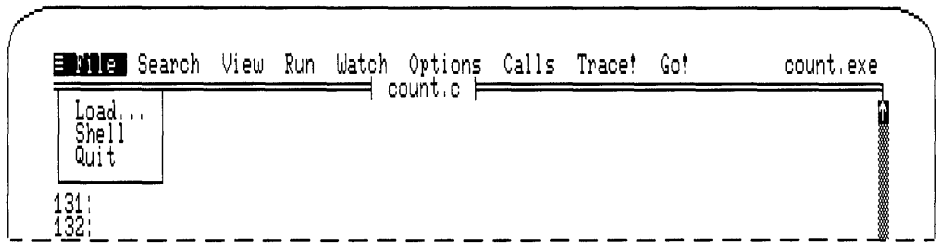


**Figure 3.2 The File Menu**

| 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.<br><br>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 an MS-DOS shell. This brings up the MS-DOS screen, where you can execute MS-DOS commands or executable files. To return to the CodeView debugger, type `exit` at the MS-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 the C 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`.

Quit

Terminates the CodeView debugger and returns to MS-DOS.

### 3.2.3.2 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 show in Figure 3.3 and the selections are explained below:

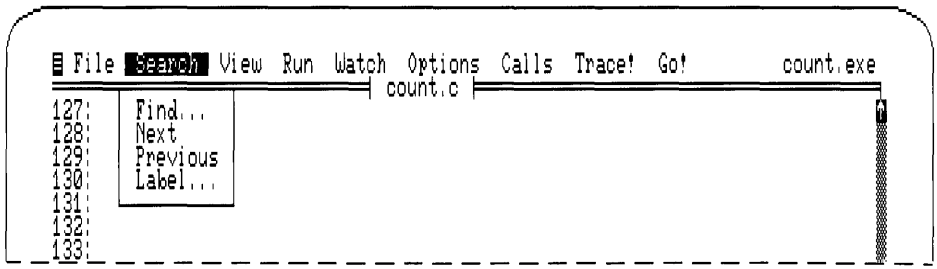


**Figure 3.3 The Search Menu**

| 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. CodeView starts at the current or most recent cursor position in the display window and searches for the expression.<br><br>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.<br><br>Regular expressions are a method of specifying variable text strings. This method comes from the XENIX® and UNIX™ operating systems, and is similar to the MS-DOS method of using wild cards in file names. Regular expressions are explained in detail in Appendix B.<br><br>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, you would use `\\n` to find `\n` or use `buffer\[count]` to find `buffer[count]`. The period in some member-selection expressions and the caret in the XOR operator and the XOR assignment operator must also be preceded by a backslash.

Next — Searches for the next match of the current regular expression. This selection is only meaningful 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 only meaningful 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 a label. A label can be a function name or an assembly-language label. If the label is found, the cursor moves to the source line or instruction containing the label. The debugger will switch to assembly mode, if necessary, to show a label in a library routine or an assembly-language module.

### 3.2.3.3 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 3.4, and the selections are explained below:

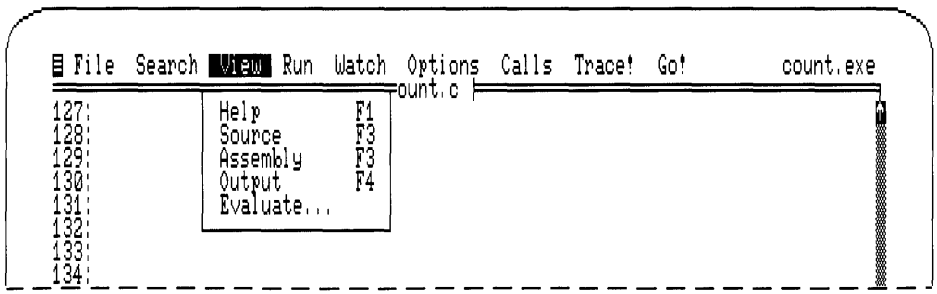


**Figure 3.4 The View Menu**

| Selection | Action |
|---|---|
| Help | Opens the initial help menu. Section 3.2.4 tells how to move through the on-line-help system and return to the debugging screen. |
| Source | Changes from assembly mode (showing assembly-language instructions in the display window) to source mode (showing source lines). If you select this mode when you are already in source mode, the selection will be ignored. |
| Assembly | Changes from source mode (showing source lines in the display window) to assembly mode (showing assembly-language instructions). If you select this mode when you are already in assembly mode, your selection will be ignored. |
| Output | Replaces the CodeView screen with the output screen. The output screen shows the current output of your program. If the program has not taken over the entire screen, previous MS-DOS commands may still be visible on the output screen. To return to the |

CodeView screen, press any key or click a mouse button. The output screen is only for viewing; you cannot change it.

Evaluate... Evaluates a C expression. A dialog box opens and asks for the expression to be evaluated. Enter a C expression made up of identifiers and the C operators recognized by the CodeView debugger (see Chapter 4, "Using Dialog Commands"). The value of the expression appears at the CodeView prompt in the dialog window. This selection is similar to the Display Expression dialog command.

You can specify the format in which the value will be displayed. Type the expression, followed by a comma and a **printf** type specifier. If you do not give a type specfier, the debugger displays the value in a default format. See Chapter 6, "Examining Data and Expressions," or Appendix A, "Command and Mode Summary," for more information about type specifiers and the default format.

### 3.2.3.4 Using the Run Menu

The Run menu includes selections for running your program. The Run menu is shown in Figure 3.5, and the selections are explained below:

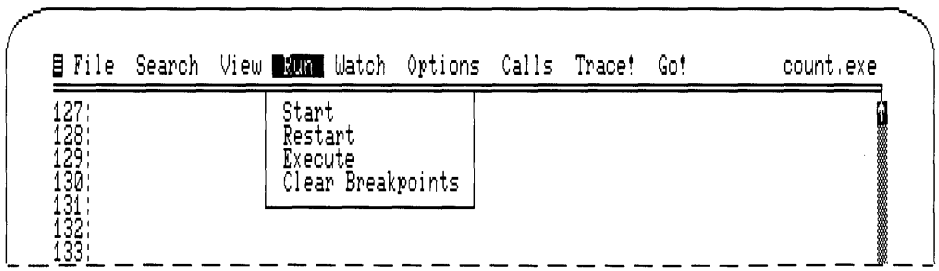


**Figure 3.5 The Run Menu**

| 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. |

### 3.2.3.5 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 3.6, and the selections are explained below:

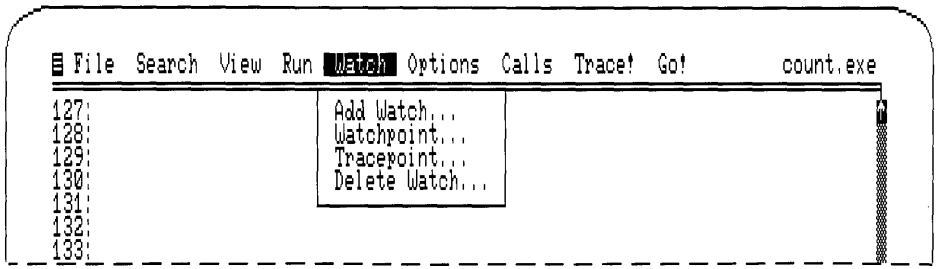


**Figure 3.6 The Watch Menu**

| Selection | Action |
|---|---|
| Add Watch... | Adds a watch-expression statement to the watch window. A dialog window opens, asking for the C expression 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.<br><br>You can specify the format in which the value will be displayed. Type the expression, followed by a comma and a **printf** type specifier. If you do not give a type specifier, the CodeView debugger displays the value in a default format. See Chapter 6, "Examining Data and Expressions," or Appendix A, "Command and Mode Summary," for more information about type specifiers and the default format. See Section 8.2, "Setting Watch-Expression or 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 C 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 8.3, "Setting Watchpoints," for more information. |
| Tracepoint... | Adds a tracepoint statement to the watch window. A dialog window opens, asking for the C 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 type specifier. If you do not give a type specifier, the CodeView debugger displays the value in a default format. See Chapter 6, "Examining Data and Expressions," for more information about type specifiers and the default format. See Section 8.4, "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.

### 3.2.3.6 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 3.7, and the selections are explained below:

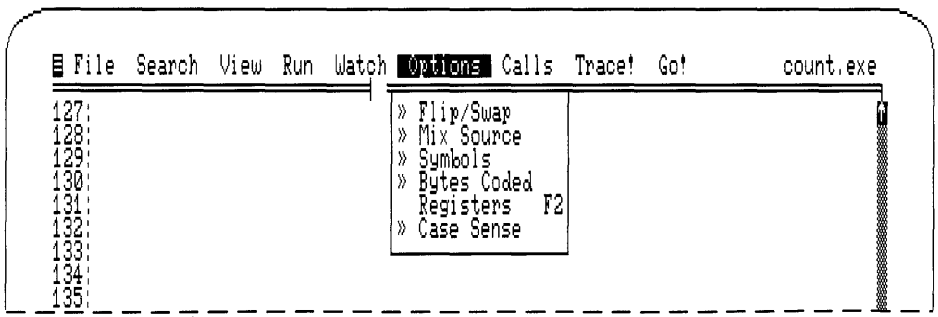


**Figure 3.7 The Options Menu**

Three of the options control the way assembly-language instructions are displayed in assembly mode. The default setting for assembly mode shows source lines, if available, with the corresponding assembly-language

instructions; the bytes for each instruction are shown as well as the instruction mnemonics; and symbols are used to show variables and labels. The default assembly display for one source line is shown below:

```
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
```

Compare this display with the variations shown in the descriptions for each assembly option.

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, all the options except Registers are on when you start the CodeView debugger.

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 to the dialog window. The CodeView debugger will detect that the screen has changed and will redraw the screen, thus destroying the program output.

---

Mix Source

When on (the default), source lines are shown with assembly language instructions; when off, instructions are shown without source lines. This option only affects assembly mode. The sample default display at the beginning of this section shows the appearance of the screen when the option is on. The following display shows the appearance of the same code when the option is off:

```
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
```

Symbols

When on (the default), symbols are used to indicate variables; when off, addresses or offsets from register values are shown for variables. This option only affects assembly mode. The sample default display at the beginning of this section shows the appearance of the screen when the option is on. The following display shows the appearance of the same code when the option is off:

```
27:                 name = gets(namebuf);
32AF:003E 8D46DE        LEA     AX,Word Ptr [BP-22]
32AF:0041 50            PUSH    AX
32AF:0042 E89C03        CALL    03E1
32AF:0045 83C402        ADD     SP,02
32AF:0048 8946DA        MOV     Word Ptr [BP-26],AX
```

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 only affects assembly mode. The sample default display at the beginning of this section shows the appearance of the screen when the option is on. The following display shows the appearance of the same 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
```

Registers

When on, the register window is displayed; when off (the default), the register window is not displayed. You can also turn on the register window by pressing F2, as indicated on the right side of the menu.

Case Sense — When on (the default), 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. The C language is normally case sensitive, so you will probably want to leave this option on. However, you may want to turn it off when debugging assembly-language programs.

### 3.2.3.7 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 3.8.

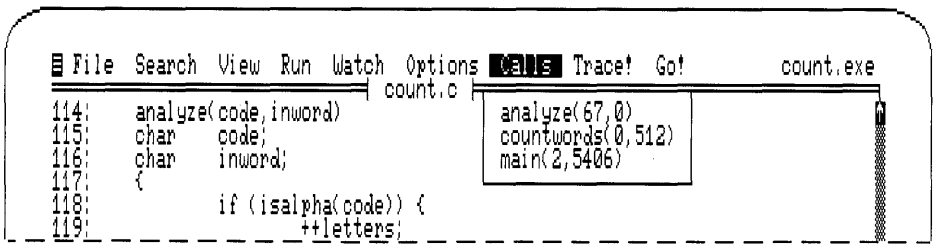


**Figure 3.8 The Calls Menu**

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 keys 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 function and the trail of functions from which it was called. The current function is always at the top. The function from which the current function was called is directly below. Other active functions are shown in the reverse order in which they were called. With C programs, the bottom function will always be **main**.

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

If you want to view code at the point where one of the functions was called, select the function 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 function where the next-level function was called. If you select the current (top-level) function, the cursor moves to the current location in that function.

For example, if the Calls menu shown in Figure 3.8 appears and you select `countwords`, the cursor will move to the line in function **countwords** where function **analyze** is called.

### 3.2.4 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 miminum 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 MS-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.

---

*Note*

For a quick summary of dialog commands, you can use the dialog version of the Help command (**H**). This is the only help available in sequential mode. It is completely different from the on-line-help system available with window commands.

---

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. |

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. For example, if you press W to select Watch from a menu, the first screen of watch information appears. You can press W or PGDN to get to the second watch screen.

## 3.3 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 Debugging Utility (**SYMDEB**) and the MS-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 in window mode. |
| F2 | Displays the registers. This is equivalent to the Register (**R**) dialog command. |
| F3 | Toggles between source and assembly modes. If the current mode is source (displaying source lines as output to the Trace, Program Step, and Go commands), the system switches to assembly mode (displaying assembly-language instructions). If the current mode is assembly, the system switches to source. This is equivalent to using the Set Assembly (**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 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, "Managing Watch Statements," 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 4
# Using Dialog Commands

# 4.1 Introduction

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 and how to specify command arguments.

# 4.2 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.

## 4.2.1 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.

## 4.2.2 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.

## 4.3 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 C expressions that represent values or addresses to be used by the command. Any combination of uppercase and lowercase letters can be used in commands. Since arguments consist of C expressions, they are normally case sensitive. 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.

As a convention for clarity, examples in this manual have uppercase letters for commands and lowercase letters for arguments.

### ■ Examples

```
>DB 100 200      ;* Example 1

>U label1        ;* Example 2

>U label2 ;DB    ;* Example 3
```

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 `label1` is a command argument. Again the Comment command labels the example.

Example 3 consists of three commands, separated by semicolons. The first is the Unassemble command (U) with `label2` as an argument. The second is the Dump Bytes command (DB) with no arguments. The third is the Comment command.

## 4.4 C Expressions

CodeView arguments always consist of C expressions. Expressions can include identifiers (also called symbols), constant numbers, registers, and operators. Expressions evaluate to 8-, 16-, or 32-bit values that can be used as numbers or addresses by CodeView commands.

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

**Table 4.1**

**CodeView Operators**

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

**Table 4.1** *(continued)*

| Precedence | Operators |
|---|---|
| 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 operator that does not come from C. It acts as a *segment:offset* separator, as described in Section 4.4.3, "Registers."

In the CodeView debugger, 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.

---

*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.

---

Sections 4.4.1–4.4.5 tell how to specify arguments using CodeView expressions.

## 4.4.1 Identifiers

### ■ Syntax

*name*

An identifier is a name that represents a register, an absolute value, a segment address, or an offset address. Identifiers (also called symbols) 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` will be 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.

## 4.4.2 Constant Numbers

### ■ 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 input radix.

The default radix for the C-language 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 11.4, "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 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 (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**
**CodeView Radix Examples**

| Input Radix | Octal | Decimal | Hexadecimal |
|---|---|---|---|
| 8 | 77 | On63 | 0x3F |
| 10 | 077 | 63 | 0x3F |
| 16 | 077 | On63 | 3F |

## 4.4.3 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 C source 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 before 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 4.3.

**Table 4.3**
**Registers**

| Type | Names | | | |
|---|---|---|---|---|
| 16-bit general purpose | **AX** | **BX** | **CX** | **DX** |
| 8-bit high registers | **AH** | **BH** | **CH** | **DH** |
| 8-bit low registers | **AL** | **BL** | **CL** | **DL** |

**Table 4.3** *(continued)*

| Type | Names | | | |
|---|---|---|---|---|
| 16-bit segment | **CS** | **DS** | **SS** | **ES** |
| 16-bit pointer | **SP** | **BP** | **IP** | |
| 16-bit index | **SI** | **DI** | | |

## 4.4.4 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 C debugging. Occasionally they may be convenient for referring to addresses outside the program, such as BIOS (basic input/output system) or MS-DOS addresses. A full address is equivalent to a C expression cast as a far pointer to a **char**. For example, the full address `1234:5678` means `(char far *)1234:5678`.

### ■ Examples

```
>DB 100                  ;* Example 1

>DB array[count]         ;* Example 2

>DB label+10             ;* Example 3

>DB 0xB800:0xFF          ;* Example 4
```

In Example 1, 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.

In Example 2, the Dump Bytes command is used to dump memory starting at the address of the variable `array[count]`.

In Example 3, the Dump Bytes command is used to dump memory starting at a point 10 bytes beyond the symbol `label`.

In Example 4, the Dump Bytes command is used to dump memory at the absolute address having the segment value `0xB800` and the offset address `0xFF`.

### 4.4.5 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                ;* Example 1

>DB buffer buffer+20      ;* Example 2

>DB buffer L 20           ;* Example 3

>U label-30 label         ;* Example 4
```

Example 1 dumps a range of memory starting at `buffer`. Since the end of the range is not given, the default size (128 bytes for the Dump Bytes command) is assumed.

Example 2 dumps a range of memory starting at `buffer` and ending at `buffer+20` (the point 20 bytes beyond `buffer`).

Example 3 uses an object range to dump the same range as in Example 2. 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 one byte, since that is the command size.

Example 4 uses the Unassemble command (`U`) to list the assembly-language statements starting 30 instructions before `label` and continuing to `label`.

## 4.4.6 Line Numbers

### ■ 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              ;* Example 1

>V .sample.c:10      ;* Example 2
```

Example 1 uses the View command (**V**) to display code starting at the address that corresponds to source line `100`. Since no file is indicated, the current source file is assumed.

Example 2 uses the View command to display source code starting at the address that corresponds to source line `10` of the source file `sample.c`.

## 4.4.7 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 must be 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`.

# Chapter 5
# Executing Code

# 5.1 Introduction

The Trace (**T**), Program Step (**P**), Go (**G**), and Execute (**E**) commands are used to 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 functions, procedures, or interrupts |
| Program Step (**P**) | Executes the current source line in source mode, or the current instruction in assembly mode; steps over functions, 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 9, "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 6, "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 5.2–5.5.

*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 commands (see Chapter 8, "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.

## 5.2 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 function call, the CodeView debugger executes the first source line of the function. In assembly mode, if the current instruction contains a procedure or interrupt, the debugger executes the first instruction of the procedure or interrupt.

Use the Trace command if you want to trace into functions, procedures, or interrupts. If you want to execute calls without tracing into them, you should use the Program Step command (**P**) instead. Both commands execute MS-DOS function calls (interrupt 0x21) without tracing into them. There is no direct way to trace into MS-DOS function calls.

In source mode, the CodeView debugger will only trace into functions that have source code. For example, if the current line contains a call to the C library function **printf**, the debugger will execute the function if you are in source mode, since the source code for library functions 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, since breakpoints cannot be set in ROM, the Program Step command (**P**) will not work. Using it in ROM 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.

## ■ Examples

The following examples all show 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+        ;* Example 1
source
>.
73:                          analyze(code,inword);
```

```
>T 4
90:    char  code;
92:          {
94:              ++letters;
95:              if (strchr("AEIOUaeiou",code) || (strchr("yY",code) && !inword))
>
```

Example 1 sets the display mode to source, then uses the Source Line command to display the current source line. (See Chapter 9, "Examining Code," for a further explanation of the Set Source and Source Line commands.) Note that the current source line calls the function `analyze`. The Trace command is then used to execute the next four source lines. These lines will be the first four lines of the function `analyze`.

```
>S-        ;* Example 2
assembly
>T
AX=0001  BX=0001  CX=0000  DX=0000  SP=1900  BP=1908  SI=04BA  DI=1946
DS=3BB1  ES=3BB1  SS=3BB1  CS=36C0  IP=0237  NV UP EI PL NZ NA PO NC
36C0:0237 50              PUSH      AX
>
```

Example 2 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 5.3. The Trace and Program Step commands are only different if, when the command is executed, the current instruction is a function, procedure, or interrupt call.

```
>S&        ;* Example 3
mixed
>T
AX=0043  BX=0043  CX=025C  DX=0000  SP=1900  BP=1904  SI=04BA  DI=1952
DS=5BE4  ES=5BE4  SS=5BE4  CS=56F3  IP=026E  NV UP EI PL NZ NA PO NC
92:               ++letters;
56F3:026E FF067201      INC      Word Ptr [_letters (0172)]     DS:0172=0000
>
```

Example 3 sets the display mode to mixed and traces the current instruction.

## 5.3 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 function call, the CodeView debugger executes the entire function and is ready to execute the line after the function call. In assembly mode, if the current instruction contains a procedure or interrupt, the debugger executes the entire procedure or interrupt and is ready to execute the next instruction after the procedure or interrupt call.

Use the Program Step command if you want to execute over function, procedure, and interrupt calls. If you want to trace into calls, you should use the Trace command (**T**) instead. Both commands execute MS-DOS function calls (interrupt 0x21) without tracing into them. There is no direct way to trace into MS-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.

### ■ Examples

The examples show 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+      ;* Example 1
source
>.
73:                              analyze(code,inword);
>P 4
74:                              inword = TRUE;
75:                              ++words;
76:                              ++characters;
78:                         }
```

Example 1 sets the display mode to source, then uses the Source Line command to display the current source line. (See Chapter 9, "Examining Code," for a further explanation of the Set Source and Source Line commands.) Note that the current source line calls the function `analyze`. The Program Step command is then used to execute the next four source lines. The first program step executes the entire function `analyze`, and the next three steps execute the lines immediately after the function.

```
>S-      ;* Example 2
assembly
>P
AX=0001  BX=0001  CX=0000  DX=0000  SP=1900  BP=1908  SI=04BA  DI=1946
DS=3BB1  ES=3BB1  SS=3BB1  CS=36C0  IP=0237  NV UP EI PL NZ NA PO NC
36C0:0237 50             PUSH      AX
>
```

Example 2 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 5.2. The Trace and Program Step commands are only different if, when the command is executed, the current instruction is a function, procedure, or interrupt call.

```
>S&      ;* Example 3
mixed
>P
AX=0043  BX=0043  CX=025C  DX=0000  SP=1900  BP=1904  SI=04BA  DI=1952
DS=5BE4  ES=5BE4  SS=5BE4  CS=56F3  IP=026E  NV UP EI PL NZ NA PO NC
92:                ++letters;
56F3:026E FF067201       INC       Word Ptr [_letters (0172)]      DS:0172=0000
>
```

Example 3 sets the display mode to mixed and steps through the current instruction.

# 5.4 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 (**BP**), 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 to which you wish to go; 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 execute the Go command with no destination using a keyboard command, 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                ;* Example 1

Program terminated normally (0)
>
```

Example 1 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+             ;* Example 2
source
>G analyze
22:     int   argc;
>
```

In Example 2, the display mode is first set to source (**S+**). See Chapter 9, "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 function `analyze`.

```
>S&         ;* Example 3
mixed
>G .38
AX=13F3  BX=13EA  CX=0019  DX=0000  SP=130E  BP=133A  SI=04BA  DI=1344
DS=5DA8  ES=5DA8  SS=5DA8  CS=58B4  IP=004B  NV UP EI PL NZ NA PO NC
38:             if ((stream = fopen(name,"rb")) == NULL) return (1);
58B4:004B B86E00          MOV       AX,006E
>
```

Example 3 passes execution control to the program at the current address and executes to the address of source line 38. If the address with the breakpoint is never encountered (for example, if the program has less than 38 lines, or if the breakpoint is on a program branch that is never taken), the CodeView debugger executes to the end of the program.

```
>S-         ;* Example 4
assembly
>G 0x2A8
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
>
```

Example 4 executes to address CS:0x2A8. Since no segment address is given, the **CS** register is assumed.

## 5.5 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.

## 5.6 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. 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 the 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      ;* Example 1
>

>L 6    ;* Example 2
>
```

Example 1 restarts the current executable file, retaining any breakpoints, watchpoints, tracepoints, and arguments. Example 2 restarts the current executable file, but with 6 as the new program argument.

# Chapter 6
# Examining Data and Expressions

# 6.1 Introduction

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 |
| Register (**R**) | Shows the current values of each register and each flag |
| 8087 (**7**) | Shows the current values in the 8087 or 80287 register |

# 6.2 Display Expression Command

The Display Expression command displays the value of CodeView expressions.

A CodeView expression can be any valid C expression consisting of numbers, symbols, addresses, type casts, or operators. The CodeView debugger supports a subset of the C operators, as explained in Chapter 4, "Using Dialog Commands." The simplest form of expression is an identifier representing a variable or label.

In addition to its primary purpose of displaying values, the Display Expression command can also set values as a side effect. For example, if you give the expression `++i` with the Display Expression command, the value of `i` will be incremented and displayed.

You can specify the format in which the values of expressions are displayed. Type a comma after the expression, followed by a **printf** type specifier. The type 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
## Types for printf

| Character | Output Format | Sample Expression | Sample Output |
|---|---|---|---|
| **d** | Signed decimal integer | `?40000,d` | `-25536` |
| **i** | Signed decimal integer | `?40000,i` | `-25536` |
| **u** | Unsigned decimal integer | `?40000,u` | `40000` |
| **o** | Unsigned octal integer | `?40000,o` | `116100` |
| **x** \| **X**[1] | Hexadecimal integer | `?40000,x` | `9c40` |
| **f** | Signed value in floating-point decimal format with six decimal places | `?(float)3/2,f` | `1.500000` |
| **e** \| **E**[2] | Signed value in scientific-notation format with up to six decimal places (trailing zeros and decimal point are truncated) | `?(float)3/2,f` | `1.500000e+000` |
| **g** \| **G**[2] | Signed value with floating-point decimal format (**f** or **F**) or scientific-notation format (**g** or **G**), whichever is more compact | `?(float)3/2,f` | `1.5` |
| **c** | Single character | `?65,c` | `A` |
| **s** | Characters printed up to the first null character | `?"String"` | `String` |

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

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

If no type specifier is given, real numbers of type **float** or **double** are displayed as if the type specifier had been given as **g**. If the numbers are signed, values of all other types are displayed as if the type specifier had been given as **d**. If the numbers are unsigned, values of all other types are displayed as if the type specifier had been given as **U**. Pointers are displayed as if the type specifier had been given as **u**.

The prefix **h** can be used with the integer type specifiers (**d**, **o**, **u**, **x**, and **X**) to specify a **short int**. The prefix **l** can be used with the same types to specify a **long int**. For example, the command `?100000,ld` produces the output `100000`. However, the command `?100000,hd` evaluates only the **short int** part of the value, producing the output `-31072`.

---

*Note*

The **n** and **p** type specifiers and the **F** and **H** prefixes are not supported by the CodeView debugger even though they are supported by the C **printf** function. See the **printf** function in the *Microsoft C Compiler Run-Time Library Reference* for more information.

---

## ■ Mouse

To display the value of an expression with the mouse, point to View on the menu bar, press a mouse button and drag the highlight down to the Evaluate selection, then release the button. A dialog box appears, asking for the expression you want to evaluate. Type the expression and press the ENTER key (or a mouse button). You can add a comma and a type specifier if you want to specify the output format. The value of the expression will appear in the dialog window.

## ■ Keyboard

To display the value of an expression using a keyboard command, press ALT-V to open the View menu, then press ALT-E to select Evaluate. A dialog box appears, asking for the expression you want to evaluate. Type the expression and press the ENTER key. You can add a comma and a type specifier if you want to specify the output format. The value of the expression will appear in the dialog window.

## ■ 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 **printf** type specifier.

## ■ Examples

The examples assume that the 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.

```
>? 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 6.4). 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.

## 6.3 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*!]] [[*function.*]] [[*symbol*]] [[*]]

The syntax combinations are listed in more detail below:

| Syntax | Display |
|---|---|
| **X?***module*!*function.symbol* | The specified *symbol* in the specified *function* in the specified *module* |
| **X?***module*!*function.** | All symbols in the specified *function* in the specified *module* |
| **X?***module*!*symbol* | The specified *symbol* in the specified *module*; symbols within functions (automatic variables and statics within functions) are not found |
| **X?***module*!* | All symbols in the specified *module* |
| **X?***function.symbol* | The specified *symbol* in the specified *function*; looks for *function* first in the current module, then in other modules from first to last |
| **X?***function.** | All symbols in the specified *function*; looks for *function* first in the current module, then in other modules from first to last |
| **X?***symbol* | Looks for the specified *symbol* in this order:<br>1. In the current function<br>2. In the current module<br>3. In other modules, from first to last |
| **X?*** | All symbols in the current function |
| **X*** | All module names |

## ■ Examples

In 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.

## 6.4 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*

Occasionally one of 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 evaluate to infinity. IND (indefinite) indicates that the data evaluate to an indefinite number.

---

Sections 6.4.1–6.4.10 discuss the variations of the Dump commands in order of the size of data they display.

## 6.4.1 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.

## 6.4.2 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 example displays the byte values from `DS:0` to `DS:36` (`DS:0x24`). The data segment is assumed if no segment is given. ASCII characters are shown on the right.

## 6.4.3 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          ;*Example 1
3D7C:0000  Some letters and numbers:
>

>DA 0 36       ;*Example 2
3D7C:0000  Some letters and numbers:.....
>
```

Example 1 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. In Example 2, an ending address is given, so the characters from `DS:0` to `DS:36` (`DS:0x24`) are shown. Unprintable characters are shown as dots.

## 6.4.4 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 integers have been displayed, whichever comes first.

*Note*

In the C language, the size of an integer is system dependent. In this manual an integer is a 2-byte value, since that is the size of integers produced with the Microsoft MS-DOS C Compiler.

### ■ 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 example displays the byte values from DS:0 to DS:36 (DS:0x24). Compare the signed decimal numbers at the end of this dump with the same values shown as unsigned integers in Section 6.4.5.

## 6.4.5 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 example displays the byte values from DS:0 to DS:36 (DS:0x24). Compare the unsigned decimal numbers at the end of this dump with the same values shown as signed integers in Section 6.4.4.

## 6.4.6 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 example displays the word values from DS:0 to DS:36 (DS:0x24). No more than eight values per line are displayed.

## 6.4.7 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 example displays the double-word values from `DS:0` to `DS:36` (`DS:0x24`). No more than four double-word values per line are displayed.

## 6.4.8 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.decimaldigits***E**+|−*mantissa*

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 *mantissa*.

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

### ■ Example

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

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

## 6.4.9 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.decimaldigits***E**+|−*mantissa*

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 *mantissa*.

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

### ■ Example

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

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

## 6.4.10 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.decimaldigits***E**+|–*mantissa*

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 *mantissa*.

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

### ■ Example

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

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

## 6.5 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 10, "Modifying Code or Data."

### ■ Mouse

To display the registers with the mouse, point to Options 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 10, "Modifying Code or Data.")

If the display mode is source (**S+**) or mixed (**S&**) (see Chapter 9, "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&       ;* Example 1
mixed
>R
AX=0043  BX=0043  CX=025C  DX=0000  SP=18F4  BP=18F8  SI=04BA  DI=1946
DS=5BF2  ES=5BF2  SS=5BF2  CS=5701  IP=026E  NV UP EI PL NZ NA PO NC
92:                    ++letters;

5701:026E FF067201      INC      Word Ptr [_letters (0172)]     DS:0172=0000
>
```

Example 1 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 memory operand `[_letters]` in the instruction is evaluated on the right side of the screen as `DS:0172=0000`. This means that the variable `letters` (at offset 0x0172 of the data segment) currently has a value of 0. The next instruction (`INC Word Ptr [_letters]`) will increment the value.

```
>S-        ;* Example 2
source
>R
AX=024F  BX=0001  CX=0000  DX=0000  SP=1900  BP=1908  SI=04BA  DI=1946
DS=5BF2  ES=5BF2  SS=5BF2  CS=5701  IP=021F  NV UP EI PL NZ NA PO NC
5701:021F 807EFC20        CMP       Byte Ptr [code],20        ;BR2
>
```

In Example 2, 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 2.

## 6.6 8087 Command

The 8087 command dumps the contents of the 8087 registers. This command is only useful if you have an 8087 or 80287 coprocessor chip on your system.

---

*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.

---

### ■ 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 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, 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 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.

# Chapter 7
# Managing Breakpoints

# 7.1 Introduction

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 8, "Managing Watch Statements." The Breakpoint commands are discussed in sections 7.2–7.6.

# 7.2 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 6.3 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* either in the *segment*:*offset* format, or as a source line, a function 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.

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 6, "Examining Data and Expressions," and Chapter 5, "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 9, "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                          ;*Example 1
>

>BP display 10 "?++counter;G"  ;*Example 2
>

>S-                              ;*Example 3
>BP 0x205
>G
AX=011D  BX=0183  CX=0000  DX=0000  SP=12FE  BP=1306  SI=04BA  DI=1344
DS=5E61  ES=5E61  SS=5E61  CS=596D  IP=0205  NV UP EI PL NZ NA PO NC
596D:0205 E97800         JMP       _countwords+9a (0280)      ;BR1
>
```

Example 1 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).

Example 2 creates a breakpoint at the address of the function `display`. The breakpoint is passed over nine 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 address is passed.

Example 3, shown in sequential mode, first sets the mode to assembly, then creates a breakpoint at the offset address `0xFA2` in the default (**CS**) segment. 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.

## 7.3 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          ;*Example 1
>

>BC *              ;*Example 2
>
```

Example 1 removes breakpoints 0, 4, and 8. Example 2 removes all breakpoints.

## 7.4 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 selections from the Run menu, or with the Restart dialog command (**L**). See Chapter 11, "Using System-Control Commands."

---

### ■ 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          ;*Example 1
>

>BD *              ;*Example 2
>
```

Example 1 disables breakpoints 0, 4, and 8. Example 2 disables all breakpoints.

## 7.5 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          ;*Example 1
>

>BE*               ;*Example 2
>
```

Example 1 enables breakpoints 0, 4, and 8. Example 2 enables all disabled breakpoints.

## 7.6 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 function, 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 example, breakpoint 0 is enabled at address `56C4:0105`. This address is in function `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 0x6 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 Introduction

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.

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. You can avoid this limitation by using the period operator to specify both the function and the variable. For example, enter `main.argc` instead of just `argc`.

## 8.2 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 6, "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 **printf** type specifier. See Chapter 6, "Examining Data and Expressions," or Appendix A, "Command and Mode Summary," 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 6, "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 dialog commands display three watch statements in the watch window:

```
W? code,c                          ;* Example 1
W? (float)letters/words,f          ;* Example 2
WB buffer L 7                      ;* Example 3
```

These commands produce the watch window in Figure 8.1.

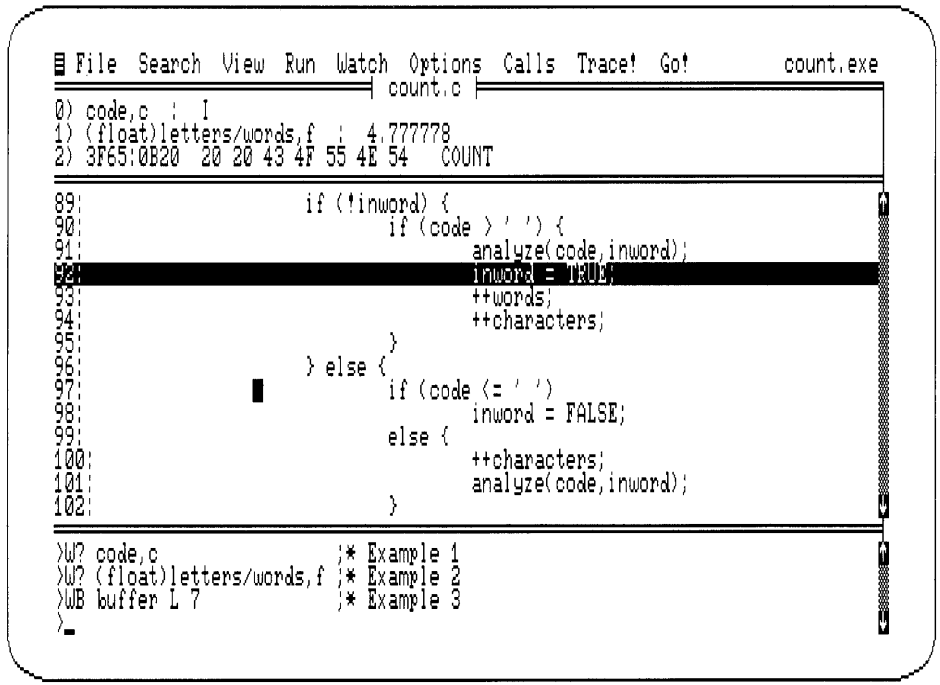


**Figure 8.1 Watch-Command Statements in the Watch Window**

Example 1 displays the current value of the variable `code`. The `c` type specifier indicates that the value is to be displayed as a character.

Example 2 displays the value of the expression `(float) letters/words`. The type cast is necessary to specify real-number division rather than integer division. The expression will be displayed in the default floating-point format.

Example 3 displays the first seven values at the address of the variable `buffer`. The characters to the right of the byte values in the watch window represent the corresponding ASCII characters (the word `COUNT` in the example). Unprintable values (less than 33 or greater than 126) are represented by dots. ASCII values are only shown if the specified *type* is **B** (byte) or **A** (ASCII).

## 8.3 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 8.6 for more information).

Although a watchpoint expression can be any valid CodeView expression, the command works best with relational expressions that use the operators ==, <=, >=, !=, <, and >. Relational expressions always evaluate to 0 (false) or 1 (true). Other expressions usually evaluate to larger numbers, thus the break is always taken.

*Note*

An example of the problems involved with using nonrelational expressions occurs if you accidentally type the simple 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 expression `count=6` resets the value of `count` to 6 every time you try to execute code. The effect is that the break is always taken, so code is always executed one line at a time.

### ■ 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 **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 in the watch window:

```
WP? lines==11              ;* Example 1
WP? buffer[count]<65       ;* Example 2
```

These commands produce the watch window in Figure 8.2.

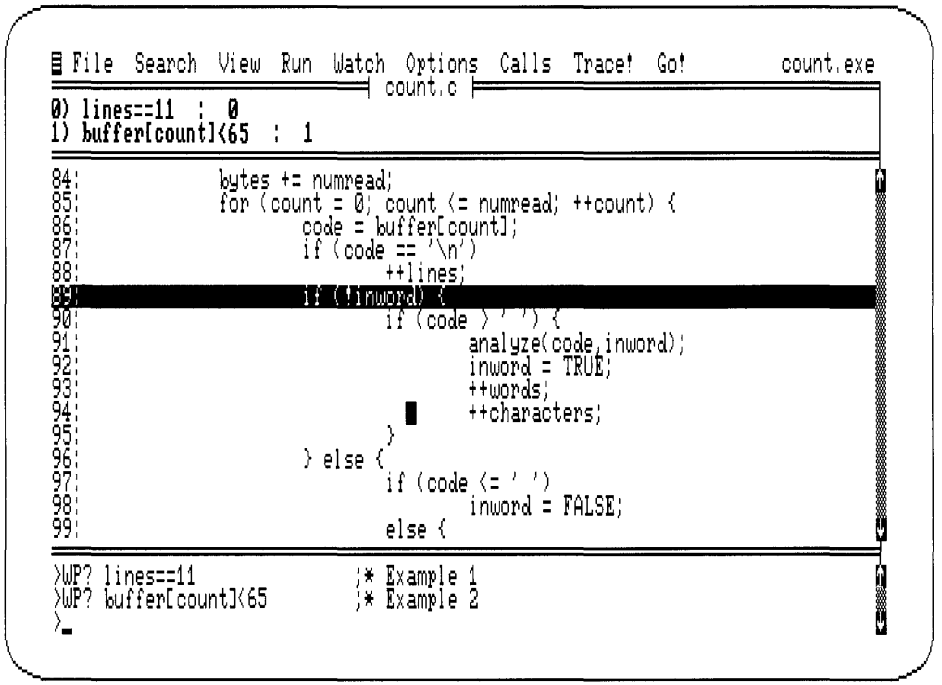


**Figure 8.2 Watchpoint-Command Statements in the Watch Window**

Example 1 instructs the CodeView debugger to break when the variable `lines` is equal to 11. After setting this watchpoint, you could use the Go command to execute until the condition becomes true.

Example 2 instructs the CodeView debugger to break when the expression `buffer[count]` becomes less than 65 (ASCII A). If `buffer` is an array of ASCII characters and `count` is a loop counter that points to the current position in the array, then the watchpoint has the effect of breaking for digits, most punctuation marks, and other characters less than 65.

---

*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.

---

## 8.4 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 8.6, "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 a memory location of not more than 128 bytes. For example, `i==10` 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 `char buffer[63]`, the Tracepoint command given with the expression `buffer` checks all 64 bytes of the array. The same command given with the expression `buffer[32]` means that only one byte (the 33rd) will be checked.

---

*Note*

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 6, "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 **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 6.2, "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 6, "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 in the watch window:

```
TP? prime       ;* Example 1
TPB flags[0] L 16  ;* Example 2
```

These commands produce the watch window in Figure 8.3 on the following page.

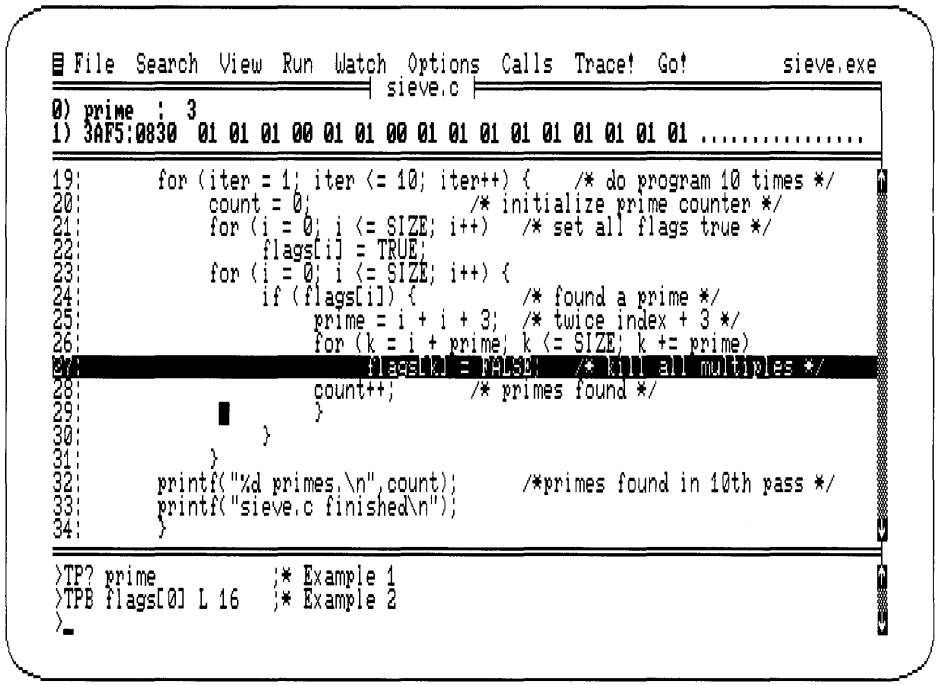


**Figure 8.3 Tracepoints in the Watch Window**

Example 1 instructs the CodeView debugger to stop whenever the value of the variable `prime` changes.

Example 2 instructs the debugger to stop whenever there is a change in the value of any of the 16 bytes starting at the address of the variable `flags`.

*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.

## 8.5 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.

## ■ 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.

## ■ 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.

# 8.6 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
>
```

# Chapter 9
# Examining Code

# 9.1 Introduction

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 functions or procedures |

# 9.2 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, 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, 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 3.2.3.6, "Using the Options Menu," for more information.

### ■ Keyboard

To change the display mode with a keyboard command, press the F3 key. If the current mode is source, the new mode will be assembly. If the current mode is assembly, the new mode will be source. This command works in either window or sequential mode. In sequential mode, the word `source` 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 3.2.3.6, "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 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.

## 9.3 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.

## ■ 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 9.2, "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 9.2, "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 9.2, "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&      ;* Example 1
mixed
>U 0x11
4E21:0011 8BEC            MOV       BP,SP
4E21:0013 B82800          MOV       AX,0028
4E21:0016 E8060C          CALL      __chkstk (0C1F)
4E21:0019 57              PUSH      DI
4E21:001A 56              PUSH      SI
29:           char  inword = FALSE;
4E21:001B C646DC00        MOV       Byte Ptr [inword],00
31:           if (argc > 1) name = argv[1];
4E21:001F 837E0401        CMP       Word Ptr [argc],01
4E21:0023 7F03            JG        _main+18 (0028)
>S-   ;* Example 2
assembly
>U 0x11
4E21:0011 8BEC            MOV       BP,SP
4E21:0013 B82800          MOV       AX,0028
4E21:0016 E8060C          CALL      __chkstk (0C1F)
4E21:0019 57              PUSH      DI
4E21:001A 56              PUSH      SI
4E21:001B C646DC00        MOV       Byte Ptr [inword],00
4E21:001F 837E0401        CMP       Word Ptr [argc],01
4E21:0023 7F03            JG        _main+18 (0028)
>
```

Example 1 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. Example 2 sets the mode to assembly and repeats the same command.

# 9.4 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 mouse, point to File on the menu bar, press a mouse button and drag the highlight to the Load selection, then release the mouse. 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 pressing 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 mouse 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 an address for the View command is 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 key with the mouse. The difference is that pressing the PGDN key enables you to scroll down one more line.

## ■ Examples

```
>V countwords    ;* Example 1
58:      char  inword;
59:      int   numread;
60:            {
61:            int   count;
62:
63:            bytes += numread;
64:            for (count = 0; count <= numread; ++count) {
65:                 char code;
>
```

Example 1 (shown in sequential mode) displays eight source lines, beginning at the function `countwords`.

```
>V .math.c:30    ;* Example 2
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.

# 9.5 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

```
>.
for (i = 0; i <= SIZE; i++);
>
```

The 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.

## 9.6 Stack Trace Command

The Stack Trace command allows you to display functions that have been called during program execution. The first line of the display shows the name of the current function. The succeeding lines (if any) list any other functions that were called to reach the current address.

For each function, the values of any arguments are shown in parentheses after the function name. Values are shown in the current radix (the default is decimal).

The term "stack trace" is used because, as each function is called, its address and arguments are stored (pushed) on the program stack. Therefore, tracing through the stack shows the currently active functions. For C programs, the **main** function will always be at the bottom of the stack.

The Stack Trace command also enables you to find and view the source lines where individual functions were called.

---

*Note*

This discussion refers to functions, since that is the terminology for C programs. 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 only work if procedures were called with the calling convention used by Microsoft languages. This calling convention is explained in Chapter 10, "Interfaces with Other Languages," of the *Microsoft C Compiler User's Guide.*

---

### ■ 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 function at the top and other functions below it in the reverse order in which they were called; for example, the first function called will be at the bottom. The values of any function arguments will be shown in parentheses following the functions.

If you want to view code at the point where one of the functions was called, hold the mouse button down and drag the highlight to the function 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 function where the next-level function was called. If you select the current (top-level) function, the cursor moves to the current location in that function.

### ■ Keyboard

To view a stack trace with a keyboard command, press ALT-C to open the Calls menu. The menu will show the current function at the top, and other functions below it in the reverse order in which they were called; for example, the first function called will be at the bottom. The values of any function arguments will be shown in parentheses following the functions.

If you want to view code at the point where one of the functions was called, press the DOWN ARROW key to move the highlight to the function below the one you want to view, then press the ENTER key. The cursor will move to the calling source line (in assembly mode) or the calling instruction (in assembly or mixed mode). In other words, the cursor will indicate the calling location in the selected function where the next-level function was called. If you select the current (top-level) function, the cursor moves to the current location in that function.

### ■ 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 functions in the reverse order in which they were called. The arguments to each function

are shown in parentheses. Finally, the line number from which the function 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 function 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 functions or function arguments. The dialog display wraps around if necessary, so that you can see all functions and all arguments.

## ■ Example

```
>K
analyze(67,0), line 94
countwords(0,512), line 73
main(2,5098), line 42
>
```

In the example, the first line of output indicates that the current function 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 function is `line 94`.

The second line indicates that `analyze` was called by `countwords`, and that its arguments have the values `0` and `512`. Function `analyze` was called from `line 73` of function `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(0x2,0x13ea)` in hexadecimal or `main(02,011752)` in octal.

# Chapter 10
# Modifying Code or Data

# 10.1 Introduction

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 |

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.

# 10.2 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.

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,0x21
MOV     AX,[0x21]
```

The first statement moves `0x21` into AX. The second statement moves the data at offset `0x21` into AX.

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   0x100
LOOPE   0x100
JA      0x200
JNBE    0x200
```

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.

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

```
>U 0x40 L 1
39B0:0040 89C3              MOV         BX,AX
>A 0x40
39B0:0040 MOV       CX,AX
39B0:0042
>U 0x40 L 1
39B0:0040 89C1              MOV         CX,AX
>
```

The example modifies the instruction at address `0x40` so that it moves data into the `CX` register instead of the `BX` register. 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.

## 10.3 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 the size of the Enter command. 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 next 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 10.3.1–10.3.10 discuss the Enter commands in order of the size of data they accept.

## ■ Examples

```
>EW place 16 32 ;* Example 1
```

Example 1 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 0x10 and 0x20.

```
>EW place                ;* Example 2

3DA5:0B20  00F3._
```

Example 2 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 (0x10) 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                    ;* Example 2

3DA5:0B20  00F3.10_
```

You could then skip to the next value by pressing the SPACEBAR. The CodeView debugger responds by displaying the value of the next value, as shown below:

```
>EW place                    ;* Example 2

3DA5:0B20  00F3.10  4F20._
```

You could then type another hexadecimal value, such as 30:

```
>EW place                    ;* Example 2

3DA5:0B20  00F3.10  4F20.30_
```

Press the SPACEBAR to move to the next value:

```
>EW place                    ;* Example 2

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                    ;* Example 2

3DA5:0B20  00F3.10  4F20.30  3DC1.
3DA5:0B22  0030._
```

Type the correct value, 20:

```
>EW place                    ;* Example 2

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                    ;* Example 2

3DA5:0B20  00F3.10  4F20.30  3DC1.
3DA5:0B22  0030.20
>_
```

## 10.3.1 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 byte. 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.

---

## 10.3.2 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 10.3.3, "Enter ASCII Command."

### ■ Examples

```
>EB 0x100 10 20 30    ;* Example 1
>
```

If the current radix is 10, Example 1 replaces the three bytes at DS:0x100, DS:0x101, and DS:0x102 with the decimal values 10, 20, and 30.

```
>EB 0x100                ;* Example 2

3DA5:0100  130F.A
>
```

Example 2 replaces the byte at DS:0x100 with 10 (0xA).

## 10.3.3 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 10.3.2, "Enter Bytes Command." 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 `Can\'t open file` is entered starting at the symbolic address `message`.

You can also use the Enter Bytes command to enter a string expression, or you can enter nonstring values using the Enter ASCII command, as described in Section 10.3.2, "Enter Bytes Command."

## 10.3.4 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 –32768 and 32767.

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 0x100 -10 10 -20        ;* Example 1
>
```

If the current radix is 10, Example 1 replaces the three integers at DS:0x100, DS:0x102, and DS:0x104 with the decimal values `-10`, `10`, and `-20`.

```
>EI 0x100                   ;* Example 2

3DA5:0100  130F.-10
>
```

Example 2 replaces the integer at DS:0x100 with `-10`.

## 10.3.5 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, a signed integer can be any decimal integer between 0 and 65535.

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 0x100 10 20 30    ;* Example 1
>
```

If the current radix is 10, Example 1 replaces the three unsigned integers at DS:0x100, DS:0x102, and DS:0x104 with the decimal values 10, 20, and 30.

```
>EU 0x100                ;* Example 2

3DA5:0100  130F.10
>
```

Example 2 replaces the integer at DS:0x100 with 10.

## 10.3.6 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 0x100 10 20 30    ;* Example 1
>
```

If the current radix is 10, Example 1 replaces the three words at DS:0x100, DS:0x102, and DS:0x104 with the hexadecimal values A, 14, and 1E.

```
>EW 0x100                    ;* Example 2

3DA5:0100  130F.A
>
```

Example 2 replaces the integer at DS:0x100 with 10 (0xA).

## 10.3.7 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 0x100 8700:12008 ;* Example 1
>
```

If the current radix is 10, Example 1 replaces the double words at DS:0x100 with the hexadecimal address 21FC:2EE8 (8700:12008).

```
>ED 0x100                    ;* Example 2

3DA5:00C8  21FC:2EE8.2EE9
>
```

Example 2 replaces the offset portion of the double word at DS:0x100 with 0x2EE9. Since the segment portion of the address is not provided, the existing segment (0x21FC) is unchanged.

## 10.3.8 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 0x100 23.479 1/4 -1.65E+4 235    ;* Example 1
>
```

Example 1 replaces the four numbers at DS:0x100, DS:0x104, DS:0x108, and DS:0x10C with the real numbers 23.479, 0.25, -1650.0, and 235.0.

```
>ES pi                                      ;* Example 2
3DA5:0064  42 79 74 65   7.215589E+022  3.141593
>
```

Example 2 replaces the number at the symbolic address `pi` with `3.141593`.

## 10.3.9 Enter Long Reals Command

**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 0x100 23.479 1/4 -1.65E+4 235     ;* Example 1
>
```

Example 1 replaces the four numbers at DS:0x100, DS:0x108, DS:0x110, and DS:0x118 with the real numbers `23.479`, `0.25`, `-1650.0`, and `235.0`.

```
>EL pi                                          ;* Example 2
3DA5:0064  42 79 74 65 DC OF 49 40   5.012391E+001  3.141593
>
```

Example 2 replaces the number at the symbolic address `pi` with `3.141593`.

## 10.3.10 Enter 10-Byte Reals Command

### ■ Syntax

**EL** *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 0x100 23.479 1/4 -1.65E+4 235     ;* Example 1
>
```

Example 1 replaces the four numbers at DS:0x100, DS:0x10A, DS:0x114, and DS:0x118 with the real numbers `23.479`, `0.25`, `-1650.0`, and `235.0`.

```
>ET pi                                          ;* Example 2
3DA5:0064  42 79 74 65 DC OF 49 40 7F BD  -3.292601E-193  3.141593
>
```

Example 2 replaces the number at the symbolic address `pi` with `3.141593`.

## 10.4 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 6, "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 |

**Table 10.1** *(continued)*

| Flag Name | Set Dialog | Set Window | Clear Dialog | Clear Window |
|---|---|---|---|---|
| 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 0x100 ;* Example 1
>
```

Example 1 changes the IP register to the value 256 (0x100).

```
>R AX              ;* Example 2
AX 0E00
:_
```

Example 2 displays the current value of the AX register and prompts for a new value (the underscore represents the CodeView cursor). You can now type any 16-bit value after the colon. For example, if the current radix is 10, you can enter 256 to change the AX value to 256 (0x100):

```
>R AX
AX 0E00
:256
>_
```

Examples 3 and 4 below show the command line and prompting methods of changing flag values:

```
>R F UP EI PL          ;* Example 3
>R F                   ;* Example 4
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 (Example 4), 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 Introduction

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 MS-DOS |
| Radix (**N**) | Changes radix |
| Redraw (@) | Redraws screen |
| Screen Exchange (\) | Switches to output screen |
| Search (/) | Searches for regular expression |
| Shell Escape (!) | Starts new MS-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.

# 11.2 Help Command

The CodeView debugger has two help commands: one command accesses a complete on-line-help system. This command can only be used in window mode. The other command provides a syntax summary of dialog commands only. It can be used in either window or 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. The syntax-summary screen cannot be reached with the mouse.

### ■ Keyboard

To enter the complete on-line-help system with a keyboard command, press the F1 key. If you are in window mode, the initial help screen appears. See Chapter 3, "The CodeView Display," for details on using the on-line-help system. If you are in sequential mode, the syntax-summary screen appears.

### ■ Dialog

The on-line-help system cannot be reached with a dialog command. To view the syntax summary, enter a command line with the following syntax:

**H**

The screen displays all CodeView dialog commands with the syntax for each. This screen is the only help available in sequential mode. It may also be useful as a quick summary in window mode.

## 11.3 Quit Command

The Quit command terminates the CodeView debugger and returns control to MS-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 MS-DOS screen, with the cursor at the MS-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 MS-DOS screen, with the cursor at the MS-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 MS-DOS screen, with the cursor at the MS-DOS prompt.

## 11.4 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. Type 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). If you give the Radix command with no argument, the debugger displays the current radix.

## ■ Examples

```
>N8                ;* Example 1
>? prime
0153
>N10
>? prime
107
>N16
>? prime
0x6b
>

>N8                ;* Example 2
>? 34,i
28
>N10
>? 28,i
28
>N16
>? 1C,i
28
>
```

In Example 1, the value of a variable is displayed in each radix. In Example 2, the same number is entered in different radixes, but the `i` type specifier is used to display the result as a decimal integer in all three cases. See Chapter 6, "Examining Data and Expressions," for more information on type specifiers.

## 11.5 Redraw Command

The Redraw command can only be used 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:

@

## 11.6 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

To switch to the output screen with the mouse, point to View on the menu bar, press a mouse button and drag the highlight down to the Output selection, then release the button. The output screen appears. Press any key or a mouse button when you are ready to return to the debugging screen.

### ■ Keyboard

To switch to the output screen with a keyboard command, press the F4 key. The output screen appears. Press any key when you are ready to return to the debugging screen. This command works in either window or sequential mode.

### ■ Dialog

To execute the Execute 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.

## 11.7 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 as an argument to a dialog command or in a dialog box. 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 comes from the XENIX and UNIX operating systems, and is similar to the MS-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 `\\n` to find `\n` or use `buffer\[count]` to find `buffer[count]`. The period in some member-selection expressions and the caret in the XOR operator and the XOR assignment operator must also be preceded by a backslash.

---

*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 function 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 function 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.

## 11.8 Shell Escape Command

The Shell Escape command allows you to exit the CodeView debugger to an MS-DOS shell. You can execute MS-DOS commands or programs from within the debugger, or you can exit from the debugger to MS-DOS while retaining your current debugging context.

The Shell Excape 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 C Compiler do this automatically if the C 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 MS-DOS function call 0x4A (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 an MS-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 MS-DOS screen will appear. You can execute any MS-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 an MS-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 MS-DOS screen will appear. You can execute any MS-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 an MS-DOS shell using a dialog command, enter a command line with the following syntax:

! [[*command*]]

If you want to exit to MS-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 MS-DOS screen appears. You can run programs or MS-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 MS-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

```
>!                ;* Example 1

>!DIR a:*.c       ;* Example 2

>!CHKDSK a:       ;* Example 3
```

In Example 1, the CodeView debugger saves the current debugging context and executes a copy of **COMMAND.COM**. The MS-DOS screen appears and you can enter any number of commands. To return to the debugger, enter `exit`.

In Example 2, the MS-DOS internal command `DIR` is executed with the argument `a:*.c`. The directory listing will be followed by a prompt telling you to press any key to return to the CodeView debugging screen.

In Example 3, the MS-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 an MS-DOS external program.

# 11.9 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:                 for (j = q; j >= 0; j--)
>#4
>.
32:         for (j = q; j >= 0; j--)
>
```

In this 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.

## 11.10 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 only relevant when used with redirected files. The redirection commands and related commands are discussed in sections 11.10.1–11.10.4.3.

### ■ 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 11.10.1–11.10.4.3.

### 11.10.1 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 provided with the debugger is an example of commands being redirected from a file.

### ■ Examples

```
><COM1          ;* Example 1

><INFILE.TXT    ;* Example 2
```

Example 1 redirects commands from the device (probably a remote terminal) designated as COM1 to the CodeView terminal.

Example 2 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. If you want the user to be able to continue editing after the session, the last command in the file should be < CON so that command input will be returned to the CodeView console or terminal. One way to create such a file is to redirect commands from the CodeView debugger to a file (see Section 11.10.3) and then edit the file to eliminate the output and to add comments.

## 11.10.2 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           ;* Example 1

>T>OUTFILE.TXT   ;* Example 2
.
.
.
>>CON
.
.
.
>T>>OUTFILE.TXT ;* Example 3
```

Example 1 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.

In Example 2, 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. If, later in the session, you want to redirect more commands to the same file, use the double greater-than symbol, as in Example 3, to append the output to the existing file.

## 11.10.3 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 only practical 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 only act as a sequential (nonwindow) device.

### ■ Example

```
>=COM1
```

In this 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.

## 11.10.4 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 |

### 11.10.4.1 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        ;* Example 1
>* Dump first 20 bytes of screen buffer
>D 0xB800:0 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 Example 1, 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.

Example 2 below 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. For example, you might use an editing program to create the following text file called INPUT.TXT:

```
* Dump first 20 bytes of screen buffer
D 0xB800:0 L 20
.
.
.
< 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. The output is shown in Example 2 below:

```
><INPUT.TXT          ;* Example 2
>* Dump first 20 bytes of screen buffer
>D 0xB800:0 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
>
```

### 11.10.4.2 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 this 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.

### 11.10.4.3 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 this 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.

The output will look like the following when the text is redirected into the debugger:

```
>* Press any key to continue
>"
```

The next CodeView prompt will not appear until the user presses a key.

# Appendix A
# Command and Mode Summary

# A.1 Introduction

This appendix summarizes the CodeView debugger's modes, options, and commands.

# A.2 Modes

Many CodeView commands and options deal with switching between modes. The debugger has four types of modes: screen modes, debugging modes, display modes, and exchange modes. The modes for each of these types are shown in Table A.1.

**Table A.1**
**CodeView Modes**

| Type | Mode | Purpose | Command or Option |
|---|---|---|---|
| Screen | Debugging | Debugs program | Default |
| | Output | Views program output | F4<br>Screen Exchange (\)<br>Output from View menu |
| | Help | Views command and option summary | F1<br>Help from View menu |
| Debugging | Window | Views program in windows | Default for IBM<br>**/W** option |
| | Sequential | Views program sequentially | Default for non-IBM<br>**/T** option |
| Display | Source | Displays source lines | Default start-up for C programs<br>F3<br>**S+**<br>Source from View menu |

**Table A.1** *(continued)*

| Type | Mode | Purpose | Command or Option |
|---|---|---|---|
| | Assembly | Displays assembly instructions | Default start-up for programs with no symbolic information<br>F3<br>**S–**<br>Assembly from View menu |
| | Mixed | Displays source and assembly | **S&**<br>Mixed from Options menu |
| Exchange | Flipping | Flips between video pages | Default for IBM<br>**/F** option |
| | Swapping | Swaps between buffers | **/S** option |

# A.3 Options

The following start-up options are available with the CodeView debugger:

| Option | Effect |
|---|---|
| **/B** | Starts in black and white with CGA |
| **/C***commands* | Executes commands on start-up |
| **/F** | Starts with screen flipping (exchanges screens by flipping video pages) |
| **/M** | Disables the mouse |
| **/T** | Starts in sequential mode |
| **/S** | Starts with screen swapping (exchanges screens by changing buffers) |
| **/W** | Starts in window mode |
| **/43** | Starts in EGA 43-row mode |

# A.4 Window Commands

Table A.2 shows the keyboard and mouse versions of each window command.

**Table A.2**
**Window Commands**

| Action | Keyboard | Mouse |
|---|---|---|
| Go to help screen | F1 | Help from View menu |
| Open register window | F2 | Registers from Options menu |
| Toggle source/assembly | F3 | Source/Assembly from View menu |
| Switch to output screen | F4 | Output from View menu |
| Go | F5 | Click either on Go! |
| Switch to display/dialog | F6 | None |
| Execute to here | F7 at location | Click right at location |
| Trace through procedure | F8 | Click left on Trace! |
| Set breakpoint here | F9 at location | Click left at location |
| Step over procedure | F10 | Click right on Trace! |
| Change flag | None | Click either on flag |
| Move separator line up | CONTROL-U | Drag line up |
| Move separator line down | CONTROL-D | Drag line down |
| Scroll up line in window | None | Click left on up arrow |
| Scroll up page in window | PGUP | Click right on up arrow |
| Scroll to top of window | HOME | Click both on up arrow |
| Scroll down line in window | None | Click left on down arrow |
| Scroll down page in window | PGDN | Click right on down arrow |
| Scroll to bottom of window | END | Click both on down arrow |
| Move cursor | UP ARROW/ DOWN ARROW | None |

Table A.3 lists and explains the selections on the CodeView menus.

## Table A.3
## Menu Selections

| Menu | Selection | Action |
|---|---|---|
| File | Load... | Loads new text or source file |
| | Shell | Starts new DOS shell |
| | Quit | Quits the CodeView debugger |
| Search | Find... | Finds first regular expression |
| | Next | Finds next regular expression |
| | Previous | Finds previous regular expression |
| | Label... | Finds function or code label |
| View | Help | Opens on-line-help screen |
| | Source | Displays source lines |
| | Assembly | Displays assembly |
| | Output | Switches to output screen |
| | Evaluate... | Evaluates expression |
| Run | Start | Restarts current program and runs |
| | Restart | Restarts current program |
| | Execute | Executes in slow motion |
| | Clear Breakpoints | Clears all breakpoints |
| Watch | Add Watch... | Sets watch expression |
| | Watchpoint... | Sets watchpoint |
| | Tracepoint... | Sets tracepoint |
| | Delete Watch... | Deletes watch statement |
| Options | Flip/Swap | Toggles screen exchange |
| | Mix Source | Toggles mixed source and instructions |
| | Symbols | Toggles symbolic reference to variables |
| | Bytes Coded | Toggles byte display for instructions |
| | Registers | Toggles register window |
| | Case Sense | Toggles case sensitivity of symbols |
| Calls | *function* | Goes to *function* call line |

# A.5 Dialog Commands

The CodeView dialog commands and the syntax for each are listed alphabetically in this section. Many of the commands require a type to indicate the size of the data accepted by the command. The types used by the Dump, Enter, Watch Memory, and Tracepoint commands are listed below in order of their data size:

| Type | Description |
|---|---|
| No type | The default type (the last one used with a Dump, Enter, Watch Memory, or Tracepoint Memory command, or byte if none of these commands has been used) |
| **A** (ASCII) | ASCII (8-bit) characters |
| **B** (Byte) | Byte (8-bit) hexadecimal values |
| **I** (Integer) | Integer (16-bit) decimal values; equivalent to C **int** (signed) on MS-DOS systems |
| **U** (Unsigned) | Unsigned (8-bit) decimal values; equivalent to C **unsigned** |
| **W** (Word) | Word (16-bit) hexadecimal values |
| **D** (Double Word) | Double-word (32-bit) hexadecimal values |
| **S** (Short Real) | Short-real (32-bit) values; equivalent to C **float** |
| **L** (Long Real) | Long-real (64-bit) values; equivalent to C **double** |
| **T** (10-Byte Real) | 10-byte-real (80-bit) values |

The CodeView dialog commands and the syntax for each are shown in Table A.4.

## Table A.4
## Dialog Commands

| Name | Syntax | Description |
|---|---|---|
| Assemble | **A** [[*address*]] | Assembles mnemonics starting at *address* |
| Breakpoint Clear | **BC** [[*list*]] | Clears breakpoints in *list* |
| Breakpoint Disable | **BD** [[*list*]] | Disables breakpoints in *list* |
| Breakpoint Enable | **BE** [[*list*]] | Enables breakpoints in *list* |
| Breakpoint List | **BL** | Lists breakpoints with status of each |
| Breakpoint Set | **BP** [[*address* [[*count*]] [[" *cmds*"]]] | Sets breakpoint at *address*; *count* is pass count; *cmds* are commands to be executed at each break |
| Comment | * | Displays explanatory text |
| Delay | : | Delays execution of redirected commands (may be repeated for longer delays) |
| Display Expression | ? *expression*[[,*format*]] | Displays *expression* in *format* |
| Dump | **D**[[*type*]] [[*range*]] | Dumps memory range in *type* format |
| Enter | **E**[[*type*]] *address* [[*list*]] | Enters memory value in *type* format |
| Examine Symbols | **X?**[[*mod!*]][[*func.*]][[*sym*]][[*]] | Displays specified symbols |
| Execute | E | Executes in slow motion |
| Go | **G** [[*address*]] | Executes to *address* or to end |
| Help | **H** | Displays dialog commands and syntax |
| Pause | " | Interrupts execution of redirected commands and waits for keystroke |

**Table A.4** *(continued)*

| Name | Syntax | Description |
|---|---|---|
| Program Step | **P** [[*count*]] | Executes source lines or instructions, stepping over function, procedure, and interrupt calls; repeats *count* times |
| Quit | **Q** | Exits and returns to MS-DOS |
| Radix | **N**[[*radix*]] | Sets input radix |
| Redirection | [[**T**]]>[[>]]*device*<br><*device*<br>=*device* | Redirects input or output to or from *device* |
| Redraw | @ | Redraws the screen |
| Register | **R**[[*register* [[=]]*expression*]]] | Displays registers and flags, or sets new registers and flags |
| Load | **L** [[*arguments*]] | Restarts program |
| Screen Exchange | \ | Exchanges the CodeView and output screens |
| Search | /[[*regularexpression*]] | Searches for a regular expression |
| Set Mode | **S**[[+\|-\|&]] | Sets display mode to source, assembly, or mixed |
| Shell Escape | ![[*command*]] | Escapes to a new MS-DOS shell |
| Current Location | . | Displays the current source line |
| Stack Trace | **K** | Displays active functions on the stack |
| Tab Set | # *number* | Sets *number* of spaces for each tab character |
| Trace | **T** [[*count*]] | Executes source lines or instructions, tracing into function, procedure, or interrupt calls; repeats *count* times |

**Table A.4** *(continued)*

| Name | Syntax | Description |
|---|---|---|
| Tracepoint | **TP?** *expression*[,*format*]<br>**TP**[*type*] [*range*] | Breaks when *expression* or *range* changes; displays in watch window |
| Unassemble | **U** [*range*] | Displays unassembled instructions |
| View | **V** [*address*] | Displays source lines |
| Watch | **W?** *expression*[,*format*]<br>**W**[*type*] [*range*] | Displays *expression* or *range* in watch window |
| Watch Delete | **Y** *number* | Deletes (yanks) watch statements |
| Watch List | **W** | Lists tracepoints and watchpoints |
| Watchpoint | **WP?** *expression*[,*format*] | Breaks when *expression* is true; displays in watch window |
| 8087 | **7** | Displays 8087 registers |

# A.6 Type Specifiers

Several commands allow you to specify the format in which expression values are displayed. The following is the syntax for commands that can have formatted output:

| | |
|---|---|
| **?** *expression*[,*format*] | Expression command |
| **W?** *expression*[,*format*] | Watch command |
| **WP?** *expression*[,*format*] | Watchpoint command |
| **TP?** *expression*[,*format*] | Tracepoint command |

The *format* in these commands can be a **printf** type specifier from among those listed in Table A.5.

**Table A.5**

**Type Specifiers**

| Character | Argument Type | Output Format |
|---|---|---|
| **d** | Integer | Signed decimal integer |
| **i** | Integer | Signed decimal integer |
| **u** | Integer | Unsigned decimal integer |
| **o** | Integer | Unsigned octal integer |
| **x** ¦ **X** | Integer | Hexadecimal integer |
| **f** | Floating point | Signed value in floating-point decimal format with six decimal places |
| **e** ¦ **E** | Floating point | Signed value in scientific-notation format with up to six decimal places (trailing zeros or decimal point truncated) |
| **g** ¦ **G** | Floating point | Signed value with floating-point decimal or scientific notation, whichever is more compact |
| **c** | Character | Single character |
| **s** | String | Characters printed up to the first null character |

The prefix **h** can be used with the integer type specifiers (**d**, **o**, **u**, **x**, and **X**) to specify a **short int**. The prefix **l** can be used with the same types to specify a **long int**.

# Appendix B
# Regular Expressions

# B.1 Introduction

Regular expressions are used to search for variable text strings. Special characters can be used within regular expressions to specify groups of characters to be searched for.

Regular expressions come from the XENIX and UNIX operating systems, where they can be used in search-and-replace commands. Since the CodeView debugger never needs to replace text, its use of regular expressions is a subset of the XENIX and UNIX regular-expression syntax.

This appendix explains all the special characters you can use to form regular expressions, but you do not need to learn the whole system to use CodeView Search commands. The simplest form of regular expression is simply a text string. For example, if you wanted to search for all instances of the symbol `count`, you could 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.

# B.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 ([ ]) | Match 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 minus sign |

## B.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 `\*name` matches `*name`. The backslash is necessary because the indirection operator (*) is a special character in regular expressions. Similarly, you could use `\\n` to search for the newline character (\n), or use `buffer\[count]` to find `buffer[count]`. Note that the backslash is only necessary for the left bracket; the right bracket is not considered a special character.

Backslashes may also be required to search for the XOR operator (^), XOR assignment operator (^=), periods in member-selection expressions, or the dollar sign ($) in variable names.

## B.4 Using the Period

A period in a regular expression matches any single character. This corresponds to the question mark (?) used in specifying MS-DOS file names.

For example, you could use the regular expression `ato.` to search for any of the functions `atof`, `atoi`, or `atol`. 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.

## B.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`. Similarly, `\\[ntvbrfa'"\0x]` matches any escape sequence.

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.

### B.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 only has this special meaning 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 [+/*-].

### B.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.

### B.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 [@%].

## B.6 Using the Asterisk

The asterisk (*) is used following a character, to match a repeated sequence of that character. The character in the text being matched may be repeated once, zero times, or numerous times.

For example, the regular expression for *(test will match any of the following strings:

```
for (test
for        (test
for(test
```

This is convenient if the text you are searching 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.) Note that the last example contains zero repetitions of the space character.

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 MS-DOS file names. For example, the expression `(.*)` matches `(test)`, `(response=='Y')`, `(x=0;x<=20;x++)`, 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 integer constants within brackets (`[1353]` or `[3]`), but does not match symbols within brackets (`[count]`). Empty brackets (`[]`) are also matched, since the characters in the brackets are repeated zero times.

## B.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 `^int  *` matches **int** declarations that start lines, but not indented **int** declarations. (Note that there are two spaces, so that at least one space is required for a match.) 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, `^{$` matches any line consisting of only a left curly bracket in the left margin, and `^$` matches blank lines.

# Appendix C
# 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:

`Bad address`

You specified the address in an invalid form. For example, 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 0–19.

`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 specified an invalid type specifier following an expression. Expressions used with the Display Expression, Watch, Watchpoint, and Tracepoint commands can have **printf** type specifiers set off from the expression by a comma. The valid type specifiers are **d**, **i**, **u**, **o**, **x**, **X**, **f**, **e**, **E**, **g**, **G**, **c**, and **s**. Some type specifiers can be preceded by the prefix **h** or **l**. See Chapter 6, "Examining Data and Expressions," for more information about type specifiers.

`Bad radix (use 8, 10, or 16)`

The CodeView debugger uses 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 type cast`

The valid types for type casting are the C types **void**, **char**, **int**, **short**, **long**, **signed**, **unsigned**, **float**, and **double**. The types **unsigned**, **signed**, **long**, and **short** can be combined with other types (**unsigned char**, for example) as listed in the *Microsoft C Compiler Language Reference*.

`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.

`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.

`Constant too big`

The CodeView debugger cannot accept a constant number larger than 4,294,967,295 (0xFFFFFFFF).

`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. Simplify 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 will not evaluate the arguments the way you intended.

`Floating point 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.

`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.

`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 example, 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 example, 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 example, 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 9–13 and 20–126.

`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 C 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 C 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. You can watch data objects larger than 128 bytes using the Tracepoint Memory command.

`Operand types incorrect for this operation`

An operand in a C expression had a type incompatible with the operation applied to it. For example, if `p` is declared as `char *`, then `? p*p` would produce this error, since a pointer cannot be multiplied by a pointer.

`Operator must have a struct/union type`

You used the one of the member-selection operators (`->` or `.`) in an expression that does not reference an element of a structure or a union.

`Operator needs lvalue`

You specified an expression that does not evaluate to an lvalue in an operation that requires an lvalue. For example, `? 3=100` is invalid. See the *Microsoft C Compiler Language Reference* for more information on lvalues.

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

You executed your program to the end. The number displayed in parentheses is the exit code returned to MS-DOS by your program. You must use the Restart command (or the Start menu selection) to start the program before executing more code.

`Register variable out of scope`

You tried to specify a register variable using the period (.) operator and a function name. For example, if you are in a third-level function, you can display the value of a local variable called `local` in a second-level function 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.

`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 many breakpoints`

You tried to specify a 21st breakpoint. The CodeView debugger permits only 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 MS-DOS user's guide for information on using the **CONFIG.SYS** file.

`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 example, 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. A symbol name spelled with letters of the wrong case will not be recognized unless the Case Sense selection on the Options menu has been turned off. This message may also occur if you try to use a local variable in an argument when you are not in the function where the variable is defined.

`Unrecognized option` *option*
`Valid options: /B /C<command> /F /S /T /W`

You entered an invalid option when starting the CodeView debugger. Retype 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.

# Glossary

**Address**

A C expression that evaluates to an address in memory. Addresses can be given in the *segment*:*offset* format. If the *segment* is not given, the default segment is assumed. The default segment is **CS** for commands related to code and **DS** for commands related to data.

**Address range**

A range of memory bounded by two addresses. The range can be specified in the normal format by giving the starting and ending addresses (inclusive), or it can be specified in the object-range format by specifying the starting address followed first by the letter "L" (uppercase or lowercase) and then by the number of objects in the range (0x100 L 10, for example, specifies the range from 0x100 to 0x109, inclusive).

**Assembly mode**

The mode in which the CodeView debugger displays assembly-language-instruction mnemonics to represent the code being executed.

**Basic input/output system (BIOS)**

The code built into system memory that provides the lowest level of functionality in a computer system. You can trace into the BIOS with the CodeView debugger, using assembly mode.

**Breakpoint**

A specified address where program execution will be halted. The CodeView debugger stops whenever it reaches an address where a breakpoint has been set. *See* "Watchpoint" and "Tracepoint" for a description of conditional breakpoints.

**Click**

To press and quickly release one of the buttons on the mouse while pointing the mouse at an object on the screen.

**Color graphics adapter (CGA)**

A video adapter capable of displaying text characters or graphics pixels. Color can also be displayed with the appropriate display monitor.

### Cursor

The thin blinking line that represents the current location where the CodeView debugger is ready to accept commands. The cursor need not be in the dialog window to enter dialog commands.

### Dialog commands

Commands entered in the dialog window in window mode, or any command in sequential mode. Dialog commands consist of one- or two-character commands that can usually be followed by arguments.

### Dialog window

The window at the bottom of the CodeView screen where dialog commands can be entered and previously entered dialog commands can be reviewed.

### Dialog box

A box that appears when you select a menu item that requires a response. The box asks you to enter some text. After you type your response and press the ENTER key (or a mouse button), the box disappears.

### Display window

The window above the dialog window where source code is displayed in source mode or assembly-language instructions are displayed in assembly mode.

### Drag

To point the mouse at an object on the screen and press a mouse button, then while holding the button down, move the mouse. The object will move in the direction of the mouse movement. When the item is where you want it, release the button. The object will stay at that point.

### Dump

To display the contents of memory at a specified memory location. In the CodeView debugger, the size of the object to be displayed is specified with a type character from the following list: **A** (ASCII), **B** (Byte), **I** (Integer), **U** (Unsigned Integer), **W** (Word), **D** (Double Word), **S** (Short Real), **L** (Long Real), or **T** (10-Byte Real).

### Enhanced graphics adapter (EGA)

A video adapter capable of displaying in all the modes of the color graphics adapter (CGA) plus many additional modes. The CodeView **/43** option displays in the EGA's 43-line text mode.

### Flags

A register that controls the operation of many machine-level instructions. In other registers, the contents of the register are considered as a whole, while in the flags register only the individual bits have meaning. In the CodeView debugger, the current values of the most commonly used bits of the flags register are shown at the bottom of the register window. *See also* "Registers."

### Flipping

A screen-exchange method that uses the video pages of the CGA or EGA to store both the debugging and output screens. Video pages are areas of memory reserved for screen storage. When you request the other screen, the two video pages are exchanged. This method is faster than swapping, the other screen-exchange method, but it does not work with the MA or with programs that do graphics or use the video pages. *See also* "Screen exchange" and "Swapping."

### Function call

A call to a a subroutine that performs a specific action. In C (source mode), subroutines are called functions. In assembly language (assembly mode), subroutines are called procedures.

### Global symbol

A symbol that is available throughout the entire program. In the CodeView debugger, function names are always global symbols. *See also* "Local symbol."

### Highlight

The reverse-video line that marks the current selection on a menu. The highlight moves as you move the mouse or press the UP ARROW or DOWN ARROW key.

### Identifier

A name that identifies a register or a location in memory. The terms "identifier" and "symbol" are used synonymously in CodeView documentation.

### Interrupt call

A machine-level procedure that can be called to execute a BIOS, MS-DOS, or other function. You can trace into BIOS interrupts with the CodeView debugger, but not into the MS-DOS interrupt (0x21).

### Label

A symbol (identifier) representing an address in the code segment (**CS**) register. Labels in C programs can be either function names or labels for **goto** statements.

### Local symbol

A symbol that only has value within a particular function. A function argument or a variable declared as **auto** or **static** within a function can be a local symbol. *See also* "Global symbol."

### Lvalue

An expression that refers to a memory location. For example, `sym` or `buffer[count]` could be lvalues since they represent symbols in memory. However, `i==10` is not an lvalue because it evaluates to either 1 (true) or 0 (false) rather than a value stored in memory. Similarly, `sym1+sym2` could not be an lvalue because it refers to the sum of two variables rather than a single memory location. See the *Microsoft C Compiler Language Reference* for more information on lvalue expressions.

### Menu bar

The bar at the top of the CodeView display containing menu titles and the titles Trace! and Go!.

### Message box

A box that pops up and displays a message. In window mode, message boxes are used to display error messages. You can press any key or a mouse button to make the box disappear.

### Monochrome adapter (MA)

A video adapter capable of displaying only in black and white. Most monochrome adapters display text only; individual graphics pixels cannot be displayed. The CodeView debugger recognizes monochrome adapters and automatically selects swapping as the screen-exchange mode.

### Mouse

A small pointing device designed to fit comfortably under your hand. By moving it about on a flat surface, you can move the mouse pointer in the corresponding direction on the screen.

### Object range

*See* "Address range."

### Output screen

The screen where program output is shown. The Screen Exchange command (\), Output from the View menu, and the F4 key can be used to switch to this screen. The output screen is the same as it would be if you ran the debugged program outside of the CodeView debugger.

### Pointer

The reverse-video square that moves to indicate the current position of the mouse. The mouse pointer only appears if a mouse is installed. To select an item with the mouse, move the mouse until the pointer rests on the item.

### Popup menu

A menu that pops up when you point the mouse cursor to the menu title and press a mouse button. In the CodeView debugger, popup menus also pop up when you press the ALT key and the first letter of the menu title at the same time. You can make a selection from the menu by dragging the highlight up or down with the mouse, by pressing the UP ARROW or DOWN ARROW key to move the highlight, or by pressing the ALT key and the first letter of the selection title at the same time.

### printf

A function in the C standard library that prints formatted output according to instructions supplied with a type-specifier argument. The CodeView debugger uses a subset of the **printf** type specifiers to format expression values.

### Procedure call

A call to a subroutine that performs a specific action. In assembly language (assembly mode), subroutines are called procedures. In C (source mode), subroutines are called functions.

### Program Step

To trace 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 instruction after the call. *See also* "Trace."

### Radix

The number system in which numbers are specified. In the CodeView debugger, numbers can be entered in three radixes: 8 (octal), 10 (decimal), or 16 (hexadecimal). The default radix is 10.

### Redirection

To specify the device from which a program will receive input or to which it will send output. Normally program input comes from the keyboard, and program output goes to the screen. Redirection involves specifying a device (or file) other than the default device. In the MS-DOS operating system, input is redirected with the less-than symbol (<) and output is redirected with the greater-than symbol (>). The same symbols are used in the CodeView debugger to redirect input or output of the debugging session. In addition, the equal sign (=) can be used to redirect both input and output.

### Registers

The places in memory where byte- or word-sized data can be stored during machine-level processing. The registers used with the 8086 family of processors are: **AX**, **BX**, **CX**, **DX**, **SP**, **BP**, **SI**, **DI**, **DS**, **ES**, **SS**, **CS**, **IP**, and the flags register. *See also* "Flags."

### Register window

The optional window in which the central processing unit (CPU) registers and the bits of the flag register are displayed.

### Regular expressions

A system of specifying text patterns that match variable text strings. The CodeView debugger supports a subset of the regular-expression characters used in the XENIX and UNIX operating systems. Regular expressions can be used to find strings in source files.

### Screen exchange

The method by which both the output screen and the debugging screen are kept in memory so that both can be updated simultaneously and

either viewed at the user's convenience. The two screen-exchange modes are flipping and swapping. *See also* "Flipping" and "Swapping."

**Sequential mode**

The mode in which all CodeView output is sequential and no windows are available. Input and output scroll down the screen and the old output scrolls off the top of the screen when the screen is full. You cannot examine previous commands after they scroll off the top. This mode is required with computers that are not IBM compatible. The mouse and most window commands are not supported in sequential mode. Any debugging operation that can be done in window mode can also be done in sequential mode.

**Shell escape**

A method of leaving the CodeView debugger without losing the current debugging context. You can "escape to a shell," do various MS-DOS tasks, and then return to the debugger. The debugging screen will be the same as when you left it. The CodeView debugger creates the shell by saving all current operations to memory and invoking a second copy of **COMMAND.COM**.

**Source mode**

The mode in which the CodeView debugger displays C source code to represent the code being executed.

**Stack trace**

A symbolic representation of the functions that have been executed to reach the current instruction address. As a function is executed, the function address and any function arguments are pushed on the stack (the area of memory starting at the address of the **SS** register). Therefore, a trace of the stack always shows the currently active functions and the values of their arguments.

**Start-up code**

The code that the C compiler places at the beginning of every program to control execution of the program code. When the CodeView debugger is loaded, the first source line executed runs the entire start-up code. If you switch to assembly mode before executing any code, you can trace through the start-up code.

### Swapping

A screen-exchange method that uses buffers to store the debugging and output screens. When you request the other screen, the two buffers are exchanged. This method is slower than flipping, the other screen-exchange method, but it works with any adapter and any type of program. *See also* "Flipping" and "Screen Exchange."

### Symbol

A name that identifies a location in memory. The terms "symbol" and "identifier" are used synonymously in CodeView documentation.

### Toggle

A function key or menu selection that turns a feature off if it is on, or on if it is off. When used as a verb, toggle means to reverse the status of a feature. For example, the F3 key is a toggle that switches between source and assembly modes. You can press the F3 key to toggle between the two modes.

### Trace

To trace 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 first source line or instruction of the call is executed. The CodeView debugger is ready to execute the next instruction inside the call. *See also* "Program Step."

### Tracepoint

A variable breakpoint that is taken when a specified value changes. The value to be tested can be either the value of a CodeView expression, or any of the values in a range of memory. Tracepoints can slow program execution significantly, since the CodeView debugger has to check after executing each source line in source mode or after each instruction in assembly mode to see if the value has changed. *See also* "Breakpoint."

### Type casting

To specify a type specifier in parentheses preceding an expression to indicate the type of the expression's value. For example, if `x` and `y` are integer values with the values 5 and 2 respectively, the expression `x/y` indicates integer division and the expression has the value 2. The expression `(float)x/y` indicates real-number division and has the value 2.5.

### Watch window

The window where watch statements and their values are displayed. The three kinds of watch statements are watch expressions, watchpoints, and tracepoints.

### Watchpoint

A variable breakpoint that is taken when a specified expression becomes nonzero (true). Watchpoints can slow program execution significantly, since the CodeView debugger has to check after executing each source line in source mode or after each instruction in assembly mode to see if the value is true. *See also* "Breakpoint."

### Window commands

Commands that work only in the CodeView debugger's window mode. Window commands consist of function keys, mouse selection, CONTROL and ALT key combinations, and selections from popup menus.

### Window mode

The mode in which the CodeView debugger displays separate windows, which can change independently. The debugger has mouse support and a wide variety of window commands in window mode.

# CodeView Index

MICROSOFT®