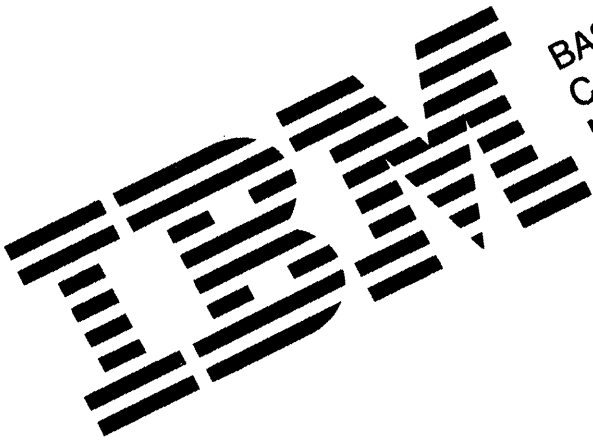
BASIC Compiler/2™
C/2™ Version 1.00 and 1.10
Macro Assembler/2™
Pascal Compiler/2™

Debug

Programming Family

15F0377



BASIC Compiler/2™
C/2™ Version 1.00 and
Macro Assembler/2™
Pascal Compiler/2™

Debug

Programming Family

Preface

This book explains how to use CodeView 1.10 to debug programs written with IBM C/2, Versions 1.00 and 1.10, IBM Macro Assembler/2, IBM Pascal Compiler/2, or IBM BASIC Compiler/2.

You should be familiar with your programming language, personal computer, and operating system. Experienced programmers should use this book along with their specific language reference books.

Related Publications

- *IBM C/2 Compile, Link, and Run*
- *IBM C/2 Language Reference*
- *IBM C/2 Fundamentals*
- *IBM MASM/2 Fundamentals*
- *IBM MASM/2 Assemble, Link, and Run*
- *IBM MASM/2 Language Reference*
- IBM Operating System/2 Version 1.00 (Standard and Extended Editions)
- *Programmer's Guide*
- IBM Operating System/2 Version 1.10
- *Programming Guide*
- The technical reference for your personal computer.
- The technical reference for your operating system.
- *IBM System Application Architecture Common Programming Interface C Reference*
- *iAPX 86, 88 User's Manual*, Copyright 1981, Intel Corp., Santa Clara, CA.
- *iAPX 286 Hardware Reference Manual*, Copyright 1983, Intel Corp., Santa Clara, CA.
- *iAPX 286 Programmer's Reference Manual*, Copyright 1985, Intel Corp., Santa Clara, CA.

Contents

Summary of Changes

The following are the new features of CodeView:

- Support for multithreaded applications - CodeView can debug the individual threads of a multithreaded application in OS/2 mode only.
- Support for dynamically linked libraries - CodeView can debug dynamic link libraries in OS/2 mode only.
- Support for child process - CodeView can debug child processes started by your program in OS/2 mode only.
- 80386 support - CodeView supports debugging of code written specifically for the 80386 processor. You can decode and assemble 386 instructions, as well as view 386 registers. 80386 support is only provided in DOS and the DOS mode of OS/2.
- Expanded memory support - DOS, with expanded memory, can substantially reduce the amount of main memory required to debug a program. Many programs that were previously too large can now be run with CodeView.
- 8087, 80287, or 80387 emulator support - Linking to an emulator library instead of using a math coprocessor, you can take advantage of the 7 command. CodeView displays pseudo-8087, 80287, or 80387 registers the same way a math coprocessor does.
- Overlayed modules - CodeView is fully compatible with programs that use overlays. It can also debug library modules.
- New commands - The *SYMDEB* commands COMPARE, FILL, MOVE, INPUT, and OUTPUT were added to CodeView. The GRAPHIC DISPLAY command allows you to examine nested structures and linked lists of pointers.
- The Help System - Enhanced online.

Chapter 1. Introducing CodeView

CodeView, Version 1.10, is a debugging program that helps you test and debug executable files developed with IBM C/2™, Versions 1.00 and 1.10, IBM Macro Assembler/2™, IBM Pascal Compiler/2™, and IBM BASIC Compiler/2™ programs.

CodeView helps you track down logical errors in programs while the program is running. CodeView displays source code or assembly code, indicates which line is about to be run, dynamically watches the values of variables (local or global), switches screens to display program output, and performs many other related functions.

CodeView is designed for use with a mouse but can also be used with the keyboard. Within CodeView there are two modes of operation, window and sequential. These modes are discussed in Chapter 2.

The intended audience for this book is C, Macro Assembler, Pascal, and BASIC programmers. We have chosen to use the C language in many of our examples; however, where it is required for clarity, the examples are language specific.

Depending on which operating system you have, you will need to use the appropriate CodeView executable file. The following defines the two versions of the CodeView execs available in this package:

CV.EXE Executable file that runs on the following operating systems:
DOS, Versions 3.30 and 4.00
OS/2 Standard Edition, Versions 1.00 and 1.10
OS/2 Extended Edition, Version 1.00

CVP.EXE Executable file that runs on the following operating systems:
OS/2 Standard Edition, Versions 1.00 and 1.10
OS/2 Extended Edition, Version 1.00

Note: In this book, *DOS mode* refers to DOS, Versions 3.30 and 4.00, or the DOS mode of OS/2.

C/2, Versions 1.00 and 1.10, Macro Assembler/2, Pascal Compiler/2, and BASIC Compiler/2 are trademarks of the International Business Machines Corporation.

Restrictions

The following restrictions apply to CodeView, regardless of the language being used. This list briefly describes the types of files not directly supported by CodeView.

Restriction	Explanation
Include files	CodeView cannot debug source code in include files.
Packed files	CodeView symbolic information cannot be put into a packed file.
.COM files	Files with the extension .COM can be debugged in assembly mode only; they can never contain symbolic information.
Memory-resident programs	CodeView can only work with disk-resident .EXE and .COM files. Debugging of memory-resident files is not supported.
Programs that alter the environment	Programs running under CodeView can read the environment, but they cannot permanently change it. Upon exit from CodeView, no changes to the environment are saved.
Program Segment Prefix (PSP)	CodeView automatically preprocesses a program's PSP the same way a C program does; quote marks are removed, and exactly one space is left between the command-line arguments. This is only an obstacle if you are debugging a program written in a language other than C and try to access command-line arguments.

Chapter 2. Getting Started with CodeView

This chapter describes how to start CodeView, run the sample session, and use the CodeView options. It describes the window and sequential modes, introduces the menu selections, and includes dialog commands and arguments.

Getting started with CodeView requires that you prepare a special-format executable file for the program you want to debug; then you can start CodeView.

Copying Files from the Diskettes

C/2 1.1 Users

Do not follow these procedures if you have already installed the program using either SETUP or INSTAID. CodeView was automatically installed during that procedure.

Before starting CodeView, copy the master files included with this book. We recommend you make a working copy of the diskettes using the DISKCOPY command.

1. Using the MKDIR command, create a directory for your new CodeView files. For example:

   ```
   MD C:\CODEVIEW
   ```

2. Using your working copy of the diskettes, copy the files to your target directory. For example:

   ```
   XCOPY A:*.* C:\CODEVIEW /S
   ```

By using XCOPY with the /S option, the CodeView sample session will automatically be copied to a directory named SAMPLE in the directory where you are installing CodeView.

For more information on DISKCOPY, MKDIR, and XCOPY, refer to your respective operating system's user's guide.

If CodeView was installed for you when you installed IBM C/2 1.10, the sample session was placed in a directory named SAMPLE in the directory where your binary files were installed.

If you installed CodeView using the procedures given in the section "Copying Files from the Diskettes" on page 2-1, the sample session was placed in a directory named SAMPLE in the directory where you installed CodeView.

2. Type **SAMPLE** and press Enter.

   The SAMPLE.BAT file starts CodeView with the appropriate options. The batch file specifies LIFE.EXE as the program that CodeView will debug.

3. Follow the online instructions.

4. From the Demonstration Menu, choose one of the 5 options. If you are new to CodeView, we suggest you choose option **T** for an overall demonstration of the program and commands.

Note: Option **P** on the **Demonstration Menu** allows you to print the CODEVIEW.DOC file which includes a comprehensive quick reference to aid you in learning CodeView.

When you complete the session, SAMPLE.BAT returns to the operating system.

Interrupting the Sample Session

If you want to quit before the sample session ends, press Ctrl+C or Ctrl+Break. If you are still in the batch file (SAMPLE.BAT), a prompt appears that asks if you want to end the session. Type **Y** and press Enter. If you are in CodeView, the word BREAK appears, followed by the CodeView prompt (>). Enter **Q** (the QUIT command). This returns you to the operating system.

The *options* are one or more of the choices described in "Choosing Startup Options" on page 2-6. The *executable file* is the name of a file that CodeView loads. The name of this file must have the extension .EXE or .COM.

If you try to load a nonexecutable file, the following error message appears:

```
Not an executable file
```

Programs containing CodeView symbolic information always have the extension .EXE. You can debug files with the extension .COM in assembly mode, but they cannot contain symbolic information. CodeView can debug programs that use overlays.

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

If you specify the executable file as a filename with no extension, CodeView searches for a file with the given base name and the extension .EXE. If the file that 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 starting CodeView. If you omit the executable file, CodeView displays a message showing the correct usage format. Use the CodeView options to replace the default startup mode.

If your program is in C, BASIC, or Pascal, CodeView is now at the beginning of the startup code that precedes your program. In source mode, you can enter a run command (such as TRACE or PROGRAM STEP) to automatically run through the startup code to the beginning of your program. At this point, you are ready to start debugging your program.

Setting the Display Mode

The /B option tells CodeView to start in the black/white mode. Normally, if CodeView detects a monochrome adapter, it displays in two colors. If it detects a color/graphics adapter, multiple colors appear.

Format

```
/B | -B
```

If you use a two-color monitor with a color/graphics adapter, you may want to use color. Monitors that display in 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 a black-and-white display. Most two-color monitors have four color distinctions: background (black), normal text, intense text, and reverse video text.

Specifying Startup Commands

The /C option lets you specify one or more *commands* that CodeView runs automatically at startup. Use these options to call CodeView from a batch or MAKE file. A semicolon separates each command from the previous command.

Format

```
/Ccommands | -Ccommands
```

If one or more of your startup commands have arguments that require spaces between them, enclose the entire option in double quotation marks. Otherwise, CodeView interprets each argument as a separate command-line argument instead of a command argument.

Example

Example 1 loads CodeView with COUNT.EXE as the executable file and COUNT.TXT as the argument. On startup, CodeView performs the C startup code with the command GMAIN. Because CodeView requires no space between the GO command (G) and its argument (*main*), you need not enclose the option in double quotation marks.

```
CV /Gmain COUNT.EXE COUNT.TXT      ;* Example 1
```

Example 2 loads the same file with the same argument, but the command list is more extensive. CodeView starts in assembly mode (S-) with a radix of 16 (N 16). It runs to the function *countwords* (G

The limitations of screen flipping are not present in screen swapping. In screen swapping, CodeView creates a buffer in storage and for the unused screen. At your request, CodeView swaps the screen in the display buffer for the one in the storage buffer.

The buffer size for screen swapping with a monochrome adapter, is 4KB, or 16KB for a graphics adapter. The /43 and /50 options cause larger buffers to be used.

CodeView defaults to screen swapping if you use a combination of the /F option with a monochrome adapter.

The following table shows the default exchange mode (swapping or flipping) and the default display mode (sequential or window) for various setups.

Display Adapter	Default Modes	Alternate Modes
Graphics	/F /W	/S if your program uses video-display pages or graphics /T for sequential mode
Monochrome	/S /W	/T for sequential mode

Turning On Window or Sequential Mode

CodeView operates in *window* or *sequential* mode. (Window mode is the default.) Window mode displays up to four windows, showing different aspects of the debugging session program at the same time. You can use a mouse in window mode but not in sequential mode.

Format

```
/T | -T
/W | -W
```

Sequential mode is useful with redirection commands. Debugging information appears in sequence down the screen. Although window mode is more convenient, CodeView can do any operation in sequential mode that it can do in window mode.

Using the 80386 Debug Registers

When using an 80386 machine with DOS or DOS mode, /R permits CodeView to use the 80386's debug registers if possible when using watchpoints.

Format

```
/R | -R
```

Note: The /R option is valid only for CV.EXE. OS/2 mode does not support all the 80386's features.

Using the 80386's debug registers permits your program to run faster because CodeView does not have to pause between each instruction to check watchpoints.

Using Enhanced Text Modes

If you have an Enhanced Graphics Adapter (EGA) or a PS/2 display, use the /43 option to enable a 43-line by 80-column text mode. If you have a Personal System/2 display, you may use the /50 option to enable a 50-line by 80-column text mode.

The enhanced modes operate like the 25-line by 80-column mode. The advantage is that more file text fits on the display. The use of these options does not effect the mode of the output screen.

Note: The /43 and /50 options are recognized only if the current setting is in 25-line mode.

Format

```
/43 | -43
/50 | -50
```

You cannot use these modes if you have a Color Graphics Adapter (CGA) or a monochrome adapter. If CodeView detects an improper type of display adapter, it ignores the option.

Debugging Dynamic-Link Modules

CVP.EXE debugs dynamic-link modules, but only if it is instructed which libraries to search at run time. A module in a dynamic-link library does not store code or symbolic information in the .EXE file of your program. It stores the code and symbols in the library (.DLL) file, and they do not join with the main program until run time. Therefore, CodeView must search the dynamic-link library for symbolic information. It does not automatically know which libraries to look

Using Expanded Memory

Format

/E

The /E option enables the use of expanded memory. If expanded memory is present, CodeView stores the symbolic information of the program there. This may be as much as 85 percent of the size of the executable file for the program and represents space that would otherwise be in the main memory.

Note: This option enables only expanded memory, not extended memory.

Using Two Video Adapters

Format

/2

The /2 option permits the use of two monitors with CodeView. The program display appears on the default monitor, while the CodeView display appears on the other monitor. (Two monitors and two adapters are required to use this option.)

For example, if you have both a Color Graphics Adapter (CGA) and a monochrome adapter, you might want to set the CGA up as the default adapter. You could then debug a graphics program with the graphics display appearing on the graphics monitor, and the CodeView display appearing on the monochrome monitor.

Note: Dual-monitor debugging is not supported on PS/2s.

The following are the elements of the CodeView display marked on this screen:

1. You can open menus by specifying the appropriate title on the menu bar. Press Alt+highlighted letter, or click on the menu with the mouse.
2. The menu highlight is a reverse-video or colored strip indicating the current selection. Move the highlight up or down to change the current selection using either the arrow keys or the mouse.
3. The display window shows the program being debugged. It can contain source code (as in the example), assembler language instructions, or any specified text file.
4. The register window shows the current status of the registers and flags. This is an optional window that will open or close with one keystroke.
5. The scroll bars are the vertical bars on the right side of the screen. Scroll one line or page using the mouse.
6. The mouse pointer shows the current position of the mouse. It appears only if you have a mouse installed.
7. The cursor is a thin, blinking line that shows where from the keyboard. Use the arrow keys to move the cursor up or down. Use F6 to switch between windows.
8. The dialog window is where you enter dialog commands. These commands have 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.
9. The display/dialog separator is the line that 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.
10. The current location line (the next line of the program that will be run) displays in reverse-video or in a different color. This line is not always visible because you can scroll to earlier or later parts of the program.
11. Previously set breakpoints show in intensified text.
12. 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.
13. The menu bar shows titles of menus and commands that you can activate with the keyboard or the mouse.
14. Dialog windows (not shown) appear in the center of the screen when a menu selection requires a response. The window is a prompt for your response. It disappears when you enter your answer, press Esc, or click either mouse button.

Key	Function
F2	Displays or removes the register window. You can also display the register window with the **Register** selection from the **View** menu.
F3	Changes between source, mixed, and assembly modes. You can also change using the **View** menu.
F4	Switches to the output screen. You can also select **Output** from the **View** menu.
Ctrl + G	Increases the size of the dialog or display window. (Grow)
Ctrl + T	Decreases the size of the dialog or display window. (Tiny)

Running a Program with Keyboard Commands

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

Key	Function
F5	Runs to the next breakpoint or to the end of the program if CodeView finds no breakpoint. 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 runs to that line (or to a previously set breakpoint or the end of the program if CodeView finds either before the temporary breakpoint). In source mode, if the line does not correspond to code (for example, if the line is a data declaration or comment line), CodeView 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	Runs a TRACE command. CodeView runs 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 starts tracing through the call (enters the call and is ready to run the first source line or instruction). This command does not trace into DOS function calls (interrupt 21h).

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

   a. Use the arrow keys to move the cursor to the items on the menu. When the item is highlighted, press Enter.
   (or)
   b. Press the key corresponding to the menu-selection highlighted letter. (In most cases, the letter is the first letter of the menu selection.)

   For most menu selections, the choice runs immediately. The items on the **View, Options,** and **Language** menus have small double arrows next to them when the option is on, and no arrows when the option is off. The status of the arrows alternates each time the menu opens.

3. A dialog window opens if the item selected from the menu requires a response. Items that require a response have an ellipsis (...) after them. Type your response in the window and press Enter. (If you do not enter a response and press Enter, the menu is canceled.) If the response is invalid, you receive an error message. Press any key to make the message disappear.

Note: While a menu is open, you can press the arrow left or right keys to move from one menu to an adjacent menu.

Cancelling a window

To cancel a window, you can either press the Esc key, press the Enter key, or click either mouse button when no response has been entered.

Selecting from Menus with the Mouse

The menu bar at the top of the screen has eleven titles: **File**, **View**, **Search**, **Run**, **Watch**, **Options**, **Language**, **Calls**, **Help**, **F8=Trace**, and **F5=Go**. Of these, the first nine are menus. TRACE and GO are commands that run by selecting with the mouse. The steps for opening a menu and making a selection follow.

1. To open a menu, point to the title you want to select.
2. With the mouse pointer on the title, press and hold either button. The selected title is highlighted and a menu box with a list of selections appears below the title.

CodeView uses two mouse buttons. The terms *click right*, *click left*, and *click either* are sometimes used to designate which buttons to use. Use either button when dragging.

Changing the Screen with the Mouse

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

Single line separating display and dialog windows	Drag the separator line up to increase (grow) the size of the dialog window while decreasing (tiny) the size of the display window. Alternately, drag the line down to increase the size of the display window while decreasing the size of the dialog window. Eliminate either window by dragging the line all the way up or down (providing the cursor is not in the window you want to eliminate).
Arrows on the scroll bar	**To scroll up or down one line at a time,** click on one of the two arrows on the scroll bar. **To scroll continuously,** press and hold either button while not moving the mouse. (Continuous scrolling is easier to use when you want to scroll more than a couple of lines.) Scrolling stops upon release of the button.

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: **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 appear in highlighted text. This corresponds with the BREAKPOINT SET and BREAKPOINT CLEAR command. **click right** A temporary breakpoint is set on the line and CodeView runs until reaching it (or until reaching a previously set breakpoint or the end of the program). This corresponds with the GO command when given with a destination breakpoint. If you click on a line that does not correspond to code (for example, if a line is a declaration or comment), CodeView sounds a warning and ignores the command.

sonal Computer AT or IBM Personal System/2, you can use the SysReq key to interrupt a program regardless of how the program uses Ctrl + Break and Ctrl + C.

Selecting Text with the Mouse

You can use the mouse to pick symbol names, commands, and other character strings directly from the display window or the dialog window. The string is then used as input to a dialog command or a menu option's dialog window.

To select a string of characters, point the mouse to one end of the string. Press the left button and hold while moving the pointer left or right to the other end of the string; then release. As the mouse moves, the characters highlight. Notice that the character directly under the mouse pointer is not included as part of the string. You must move the pointer past the last character desired.

The selected text is then used in one of two ways:

1. If you select a menu option requiring an argument, the dialog window opens and the text appears.
2. If you press Ins, the text is copied to the end of the dialog window buffer.

The selected text can be used only once. To use the same string repeatedly, select the string again. If the string is selected and not used immediately, the highlight disappears; however, CodeView remembers the string until you press Shift + Ins or select a menu item that requires the input dialog window.

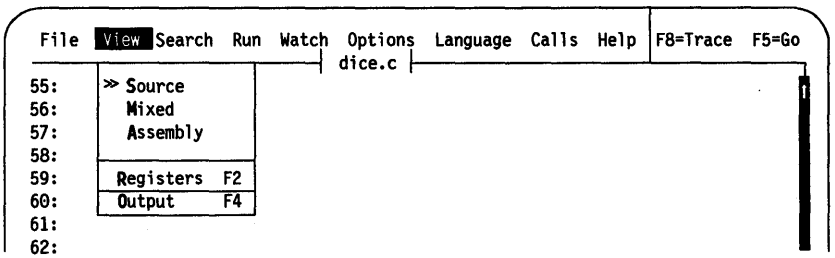
Describing the Menu Selections

This section describes the selections on each of the CodeView menus. Make these selections with the keyboard or the mouse.

command processor. This requires a large amount of free memory since CodeView, the command processor, symbol tables, and the debugged program must remain in memory. If there is not enough memory to run the SHELL command, an error message appears. Even if there is enough memory to run the SHELL, there may not be enough memory left to run large programs from the SHELL.

Exit Ends CodeView and returns to the operating system.

The View Menu



The **View** menu includes selections for changing between source and assembly modes and for switching between the debugging screen and output screen. The corresponding function keys for menu selection are on the right side of the menu where appropriate.

One of the following selections will have small double arrows to the left of the name: **Source, Mixed,** or **Assembly.** These arrows indicate which of the three display modes is in use. The **Registers** selection may or may not have double arrows to the left, depending on whether the register window is displayed.

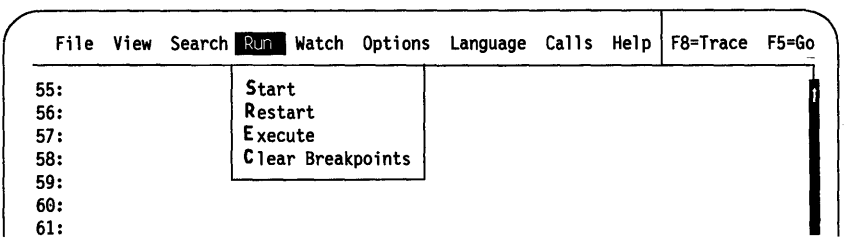
Selection	Action
Source	Changes to source mode (showing source lines only).
Mixed	Changes to mixed mode (showing both unaltered code and source lines). In a C program, the source lines do not show until you run the C startup code to *main*.
Assembly	Changes to assembly mode (showing unaltered code only).
Registers	(F2) Selecting this option *toggles* the register window on and off.

Next Searches for the next match of the current regular expression. This selection is meaningful only after using the SEARCH command to specify the current regular expression. If CodeView searches to the end of the file without finding another match for the expression, it wraps around and starts searching at the beginning of the file.

Previous Searches for the previous match of the current regular expression. This selection is meaningful only after using the SEARCH command to specify the current regular expression. If CodeView searches to the beginning of the file without finding another match for the expression, it wraps around and starts searching from the end of the file.

Label... Searches the executable code for a label. A label can be a function name, subroutine name, or an assembler language label. If CodeView finds the label, the cursor moves to the source line or instruction containing it. CodeView switches to assembly mode, if necessary, to show a label in a library routine or an assembler language module.

The Run Menu



The **Run** menu includes selections for running your program.

Selection	Action
Start	CodeView runs the program from the beginning to the first breakpoint or to the end of the program, if it does not encounter a breakpoint. Any previously set breakpoints or watch statements remain in effect. (This has the same effect as selecting **Restart** and then entering the GO command.)

Watchpoint... Adds a watchpoint statement to the watch window. A dialog window opens, asking for the source-level expression whose value you want to test. A *watchpoint* is a conditional breakpoint that causes running to stop when the expression becomes nonzero (true).

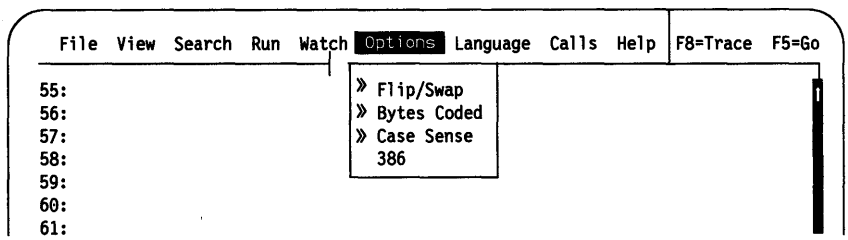
Tracepoint... Adds a tracepoint statement to the watch window. A dialog window opens, asking for the source-level expression or storage range whose value you want to test. A *tracepoint* is a conditional breakpoint that causes a run to stop when the value of a given expression changes. You cannot specify a storage range 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 is displayed. Type the expression followed by a comma and a type specifier. If you do not give a type specifier, CodeView displays the value in a default format.

Delete Watch... (Ctrl+U) 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 to delete, then click either button. If you are using a keyboard, use the arrow up or down keys. Move to the item you want to delete, then press Enter.

Delete All Watch Deletes all statements in the watch window.

The Options Menu



The Options menu lets you set options that control CodeView. 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 reverses each time you select

The following example shows the appearance of the same code with the option on:

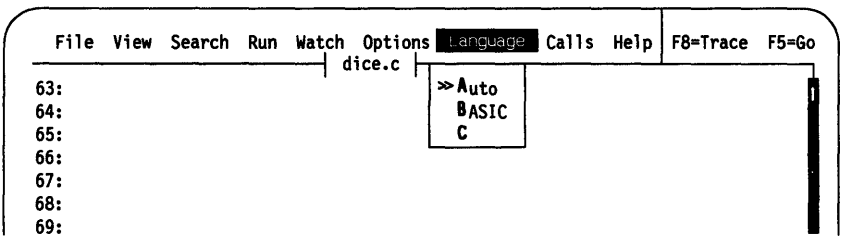
```
27:             name = gets(namebuf);
32AF:003E 8D46DE    LEA   AX,Word Ptr [namebuf]
32AF:0041 50        PUSH  AX
32AF:0042 E89C03    CALL  _gets (03E1)
32AF:0045 83C402    ADD   SP,02
32AF:0048 8946DA    MOV   Word Ptr [name],AX
```

Case Sense When on, CodeView assumes symbol names are case sensitive; when off, symbol names are not case sensitive. (This option is on for C and Macro Assembler programs, and off for BASIC and Pascal. We suggest leaving the option in its default setting.)

386 When on, the register window displays the registers in the wider 386 format. Furthermore, this option enables you to assemble and run instructions that reference 32-bit registers.

Warning: If the **386** option is off, then any data stored in the high-order word of a 32-bit register is lost.

The Language Menu



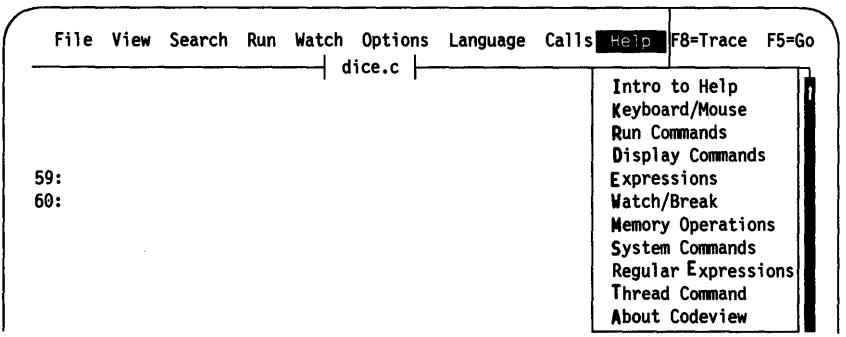
The **Language** menu allows you to either select the expression evaluator or instruct CodeView to select it automatically. As with the **Options** menu, the selection that is on is marked by double arrows. However, unlike the **Options** menu, only one item can be selected at a given time.

Selection	Action
Auto	CodeView selects the expression evaluator. The debugger automatically selects the evaluator each time a new source file is loaded. CodeView examines the extension of the source file, to determine which expression evaluator to select. For example, the **Auto**

selecting a routine from the **Calls** menu does not (by itself) affect the program's running. It does provide a convenient way to view previously called routines.

Note: If you are using CodeView to debug assembly language programs, only the current routine shows in the **Calls** menu.

The Help System



The CodeView online help uses menus that provide quick access to help screens on a variety of subjects. Help is available in window and sequential modes, but the help differs. The following explains the help system in window mode.

Help (on menu bar) If you have a keyboard, press Alt+H to open the **Help** menu. (Mouse users can click on **Help**.) Move the cursor down to highlight the desired help item, then press Enter; or press the highlighted character to choose an item. (Mouse users can click on the desired item.)

F1 or H F1 or H takes you to the **Intro to Help** screen.

The only help available in sequential mode is the dialog version of the HELP command (H). The help available in sequential mode is a listing of the dialog commands and their formats.

Note: CV.HLP (for DOS or the DOS mode of OS/2) and/or CVP.HLP (for OS/2 mode) must be in your current directory or within the directories specified by your PATH command. If CodeView does not find the help file, it issues an error message.

Key	Function
F9	Sets or clears a breakpoint at the current program location. If the current program location has no breakpoint, one is set. If the current location has a breakpoint, it is removed. This is equal to the BREAKPOINT SET (BP) dialog command without an argument.
F10	Runs 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 run to the end and CodeView is ready to run the line or instruction after the call. This is equal to the PROGRAM STEP (P) dialog command.

The CodeView WATCH (W), WATCHPOINT (WP), and TRACEPOINT (TP) commands work in sequential mode, but because there is no watch window, the watch statements do not show. You must use the WATCH LIST command (W) to examine watch statements and watch values with no arguments.

All the CodeView commands that affect program operation (such as TRACE, GO, BREAKPOINT SET) are available in sequential mode. Any debugging operation that you can do in window mode you can also do in sequential mode.

Using Dialog Commands

You can use CodeView dialog commands in sequential mode or from the dialog window in window mode. In sequential mode, they are the primary method of entering commands. In window mode, use dialog commands 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.

location. However, by pressing Enter, CodeView places the new command 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 see the command just entered. If you scroll back to the point where you entered the command, you see the original characters rather than the characters you typed over.

Upon starting CodeView, the buffer contains only the copyright message. As commands are entered during the session, the buffer gradually fills from the bottom to the top. If you do not fill the entire buffer and you press the Home key to go to the top of the buffer, the screen will be blank.

Formatting CodeView Commands and Arguments

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

```
command [argument] ... [;command2]
```

The *command* is a one-, two-, or three-character name, and *arguments* are expressions that represent values or addresses used by the command. Use any combination of uppercase and lowercase letters in commands. Often, the first *argument* is after the *command* with no space.

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. Use a semicolon as a command separator if you want to specify more than one command on a line. (Arguments may also be case sensitive.)

Chapter 3. Preparing Programs for CodeView

This chapter describes how to produce an executable program that can be debugged with CodeView.

Preparing C Programs

You must compile and link with the correct options to use a C program with CodeView. These options direct the compiler and the linker to produce an executable file that contains line-number information and a symbol table, in addition to the executable code. This creates a larger executable file. To minimize a program's size after debugging, recompile and link your final version without the options for using CodeView.

Compile Options

Option	Description
/Zi	For full symbolic debugging information
/Zd	For line number records only
/Od	Turns off optimization

To use CodeView to display source code, you must compile with either /Zi or /Zd. /Zd writes less symbolic information to the object file saving disk space and memory. For example, if you are working on a program made up of five modules but only need to debug one module, you can compile that module with the /Zi option and the other modules with the /Zd option. You can examine global variables and see source lines in modules compiled with the /Zd option, but local variables are unavailable. If you do not use either of the /Z options, CodeView will work only in assembly mode.

The /Od option to turn off optimization is desirable but not required. Optimization rearranges code for greater efficiency and, as a result, the instructions 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.

Example

```
CL /Zi /Od EXAMPLE.C            ;* Example 1


CC /Zi /Od EXAMPLE;             ;* Example 2
LINK /CO EXAMPLE;


CL /Zi /Od /c MOD1.C            ;* Example 3
CL /Zd /Od /c MOD2.C
CL /Zi MOD1 MOD2


masm sub1;                       ;* Example 4
masm sub2;
cl -c -Zi cmain.c
link /Co cmain+sub1+sub2,,,slibc+slibc3;
```

In Example 1, CL is used to compile and link the source file EXAMPLE.C. CL creates an object file, EXAMPLE.OBJ, and then automatically starts the linker with the /CO option. Example 2 shows how to compile and link the source file EXAMPLE.C using the CC program provided with the compiler. Both examples result in an executable file, EXAMPLE.EXE, which has the required line-number information, symbol table, and unoptimized code.

In Example 3, the source module MOD1.C is compiled to produce an object file with full symbolic and line information, while MOD2.C is compiled to produce an object file with limited information. CL is again used to link the resulting object files. (CL does not recompile because the arguments do not have .C extensions.) Typing /Zi on the command line causes the linker to start with the /CO option. The result is an executable file in which one of the modules, MOD2.C, is harder to debug; however, the executable file takes up substantially less space on a disk than it would if both modules were compiled with full symbolic information.

Preparing Pascal Programs

To use CodeView with a program written in Pascal, you must compile the program with the IBM Pascal Compiler/2 and link with the IBM Linker/2.

Writing Pascal Source

Pascal Compiler/2 supports the use of include files by providing the $INCLUDE metacommand. You cannot debug source code put into include files. To debug a large program, you can debug in separately compiled source files, rather than using include files. Variables that might be referenced from within CodeView, must be declared PUBLIC. The PUBLIC attribute causes the compiler to pass information about these symbols to the linker for use by CodeView.

Debugging is simpler if you put each source statement on a separate source line. When they are on the same line, you cannot find the separate statements individually. For example, code is easier to debug if you write it in the following form:

```
if i = max then
   begin
      k := k+1;
      i := 0
   end;
```

Compiling and Linking Pascal Programs

To prepare a Pascal program, compile normally; then link your program using the /CO option. CodeView supports *mixed-language programming*. An example of how to link a Pascal module with modules from other languages is in "Preparing Macro Assembler Programs."

Example

```
PAS1 TEST;
PAS2
LINK /CO TEST;
```

The example compiles the source file TEST.PAS to produce an object file, TEST.OBJ, which contains the symbol and required line-number information. Then the linker starts with the /CO option.

- If you access command arguments in the Program Segment Prefix (PSP), CodeView changes the PSP. Tabs, quotation marks, and extra spaces are removed so that one space separates each argument. CodeView retains quotation marks (along with any quoted material) for command arguments given with the L command.

Assembling and Linking Macro Assembler Programs

To prepare an assembler program, assemble normally; then link your program using the /CO option. If you link your assembler program with a module written in C (which is case sensitive), you need to assemble with /MX or /ML. After assembling, link with the /CO option to produce an executable file.

Example

```
MASM EXAMPLE;                    ;* Example 1
LINK /CO EXAMPLE;


CL /Zi /Od /c /AL prog.c         ;* Example 2
BASCOM /Zi sub1;
MASM /MX sub2;
LINK /Co prog+sub1+sub2,,,slibc+slibc3;
```

Example 1 assembles the source file EXAMPLE.ASM, and produces the object file EXAMPLE.OBJ. The linker then starts with the /CO option and produces an executable file containing the symbol table information required.

Example 2 shows how to create mixed-language executable files that CodeView can use. CodeView is able to trace through source files in the same session, regardless of the language.

Note: When using CodeView with programs created with the IBM Macro Assembler/2, the CodeView display starts in assembly mode. (This is the startup default if CodeView does not find line-number information.) Source mode cannot be used for debugging, but you can load the source file into the display window and view it in source mode. Any labels or variables declared PUBLIC can be displayed and referenced by name instead of address. They cannot be used in expressions because type information is not written to the object file.

Chapter 4. Managing Expression Evaluators

This chapter describes CodeView's *expression evaluators* and their use. Expression evaluators allow the use of a specific programming language's syntax when entering expressions and addresses as arguments with CodeView commands. CodeView provides two expression evaluators, one using C syntax and one using BASIC syntax.

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

Each expression evaluator has a different set of operators and rules of precedence. You can change the expression evaluator. If you specify a language other than the one used in the source file, the expression evaluator recognizes your program symbols. (For example, C does not accept Basic type-declaration characters.) If you are debugging an assembly routine called from BASIC, choose the BASIC expression evaluator rather than C, which is the default for assembly programs.

If the **Auto** option is on, CodeView examines the file extension of each new source file that is traced. C, Pascal, and assembly modules cause CodeView to select C as the expression evaluator. BASIC modules cause CodeView to select the BASIC expression evaluator.

C Expression Evaluator

The C evaluator allows you to specify CodeView arguments using C syntax. The following sections discuss the operators, symbols, constants, strings and functions allowed by the evaluator. CodeView uses the most commonly used C operators and additional operators. These operators are listed in the following table in order of precedence.

The *type* operator (used in type casting) can be any of the predefined C types. CodeView limits casts of pointer types to one level of indirection.

When you use a C expression as an argument with a command that takes multiple arguments, the expression must 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.

CodeView cannot evaluate an expression containing **huge** objects.

C Memory Operators

The BY, WO and DW operators are unary operators. They return the result of a direct memory operation. Use BY to access a byte, WO to access a word, and DW to access a doubleword. These operators are of special interest to assembler programmers because they simulate the BYTE PTR, WORD PTR, and DWORD PTR operations.

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

Accessing Bytes (BY)

Access the byte at an address by using the BY operator.

Format

```
BY address
```

The result is a short integer that contains the value of the first byte stored at *address*.

Example

Example 1 returns the first byte at the address of *sum*.

```
>?BY sum            ;* Example 1
101
```

Example 2 returns the byte pointed to by the BP register, with a displacement of 6.

```
>?BY bp+6           ;* Example 2
42
```

Example

Example 1 returns the first doubleword at the address of *sum*.

```
>? DW sum                    ;* Example 1
>132120365
```

Example 2 returns the doubleword pointed to by the SI register.

```
>? DW si,x                   ;* Example 2
>3F880000
```

C Symbols

Format

```
name
```

A symbol is a name that represents a register, a segment address, an offset address, or a full 32-bit address. At the C source level, a symbol (identifier) is a variable name or the name of a function. Symbols follow the naming rules of the C compiler.

Note: 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 DISPLAY SYMBOLS command, CodeView displays some symbol names in the object-code format. This format includes a leading underscore. For example, the function **main** is displayed as **_main.** Only global labels (such as procedure names) show in this format. Labels within library routines sometimes appear with a double underscore (__chkstk). You must use two leading underscores when accessing these labels with CodeView commands.

C Strings

Format

```
"null-terminated-string"
```

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

Example

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

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

C Functions

When debugging C programs using the C expression evaluator, expressions can contain functions that are part of the executable file. For example, if the program contained the function *sorted*, the following example is a valid expression:

```
success=sorted (array1)
```

BASIC Expression Evaluator

The BASIC expression evaluator uses a subset of the BASIC operators. It also supports one important BASIC command, LET, and one operator in addition to the BASIC operators, the colon.

The CodeView BASIC operators are listed in the following table in order of precedence.

Precedence	Operator
1 (Highest)	()
2	. :
3	* /
4	\ MOD
5	+ -
6	= <> > < >= <=

The *routine* must be a high-level language routine and the *variable* must be a local variable within the routine.

When a BASIC expression is used as an argument with a command that takes multiple arguments, the expression must not have any internal spaces. Commands (such as the DISPLAY EXPRESSION command) that take only one argument do permit spaces in expressions.

BASIC Symbols

Format

```
name
```

A symbol is a name that represents a register, a segment address, an offset address, or a full 32-bit address. At the BASIC source level, a symbol is simply a variable name or the name of a routine; you do not necessarily need to know what kind of address it represents. With the BASIC expression evaluator, symbols follow the naming rules of the BASIC compiler. In particular, all the type specifiers used in BASIC ($, %, &, !, and #) are accepted by the BASIC expression evaluator.

Note: Symbols are never case sensitive to BASIC, whether the **Case Sense** option is on or not.

BASIC Constants

Format

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

digits      Integer, default radix
&Odigits    Octal radix
&digits     Alternative octal radix
&Hdigits    Hexadecimal radix
```

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

BASIC Strings

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

By using the ENTER ASCII command, you can enter a string literal at a given address. To use this technique effectively, you need to understand how BASIC handles string variables.

BASIC Intrinsic Functions

When entering an expression, use a limited number of BASIC intrinsic functions. The primary use of these functions is to convert a BASIC variable or value from one type to another for purposes of calculation. The following are the intrinsic functions recognized by the expression evaluator. (See your *BASIC Compiler/2 Language Reference* for a complete description of the functions.)

Argument[1]	Function	Input Type	Output Type
ASC	ASCII value of first character	String	Integer
CDBL	Data-type conversion	Numerical expression	Double
CINT	Conversion, with rounding	Numerical expression	Integer
CSNG	Data-type conversion	Numerical expression	Single
CVI	Data-type conversion	Two-byte string	Integer
CVS	Data-type conversion	Four-byte string	Long
CVD	Data-type conversion	Eight-byte string	Double
FIX	Conversion with truncation	Numerical expression	Integer

To switch expression evaluators using a dialog command, enter the following:

```
USE [language]
```

where language is C, BASIC, or Auto.

The command is not case sensitive. Entered on a line by itself, USE displays the name of the current expression evaluator.

Example

Example 1 switches to the C expression evaluator.

```
>USE C                          ;* Example 1
C
```

Example 2 displays the name of the current expression evaluator, which is BASIC.

```
>USE                            ;* Example 2
BASIC
```

Working with Pascal Programs

CodeView uses the C expression evaluator when debugging Pascal programs. Therefore, expressions are translated into C when CodeView commands are entered.

Translating Pascal Expressions into C

Pascal PUBLIC symbols can be translated with a C construct called *cast*. For example, if the Pascal REAL variable *x* is a PUBLIC symbol, CodeView recognizes that there is a symbolic reference with the name *x* and its location in memory.

Note: Translate integer variables if they are used in expressions with other variables or if you use the expression evaluator to change values.

Record

Assume you have the following declarations:

```
a_rec : RECORD
          field_a : ARRAY [0...5] OF WORD;
          field_b : REAL8;
          field_c : WORD
          END;
```

To get the value in a_rec.field_c, the format for the equivalent C expression is:

```
*(int *)&a_rec + byte offset
```

As field_c cannot be made PUBLIC, you will need to compute the byte offset. The byte offset for field_c is 20 (12 bytes for field_a + 8 bytes for field_b).
So the equivalent C expression for a_rec.field_c is:

```
*(int *)&a_rec + 20
```

```
BYTE PTR [di+6]                 BY di+6
BYTE PTR [si][bp+6]             BY si+bp+6
WORD PTR [bx][si]               WO bx+si
```

3. Taking the address of a variable - Use the ampersand (&) to get the address of a variable with the C expression evaluator.

```
OFFSET var                      &var
```

4. The PTR operator - With CodeView, C-type casts perform the same function as the Macro Assembler PTR operator.

```
BYTE PTR var                    (char) var
WORD PTR var                    (int) var
DWORD PTR var                   (long) var
```

5. Accessing array elements - Accessing arrays declared in assembly code can cause problems because Macro Assembler emits no type information to indicate which variables are arrays. Therefore, CodeView treats an array name like any other variable.

   In C, an array name is equated with the address of the first element. Therefore, if you prefix an array with the address operator (&), the C expression evaluator gives correct results for array operations.

```
string[12]                      (&string)[12]
warray[bx+di]                   (&warray)(bx+di)/2
darray[4]                       (&darray)[1]
```

   In the second and third examples, notice that the indexes used in the assembly source-code expressions differ from the indexes used in the CodeView expressions. This difference is necessary because C arrays are automatically scaled according to the size of elements. In assembly, the program must do the scaling.

Regular Expressions

This section explains all of the special characters that form regular expressions. You do not need to learn the whole system to use CodeView SEARCH commands. The simplest regular expression is a text string. For example, to search for all instances of the symbol count, specify *count* as the string to be found.

essary only for the left bracket; the right bracket is not considered a special character.

Backslashes are also required when searching for the XOR operator (^), the XOR assignment operator (^=), the period in member-selection expressions, or the dollar sign ($) in variable names.

Using the Period

A period in a regular expression matches any single character. This corresponds to the question mark used in specifying DOS filenames.

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.

When you use the period as a global file character, you find the strings you are looking for, but you may also find other strings that you are not interested in. Use brackets to be precise about the strings you want to find.

Using Brackets

Brackets can specify a character or characters you want to locate. 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 special characters in regular expressions have no special meaning when used within brackets. The only special characters within brackets are the dash (-), caret (^), and right bracket (]). These characters have a special meaning only in certain contexts, as explained in the next sections.

Using the Asterisk

Placement of the asterisk (*) after a character causes a repeated sequence of that character. The character to be located in the text being matched may repeat once, numerous times, or not repeat at all.

The regular expression [for **(test*] matches any of the following strings:

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

This is convenient if the text contains some spaces, but you do not know how many. Be careful in this situation; you cannot be sure if the text contains a series of spaces or a tab. Notice that the last example contains no repetitions of the space character.

You might also use the asterisk to search for a symbol when you are not sure of the spelling. For example, use *first*ime* if you are not sure if the symbol is spelled *firsttime* or *firstime*.

One use of the asterisk is to combine it with the period (.*). This combination searches for any group of characters and is similar to the asterisk used in specifying DOS filenames. 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 alphabetic characters or symbols within brackets ([count]). Empty brackets ([]) are also matched since the characters in the brackets are repeated zero times.

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.

The regular expression ^C matches any uppercase C that starts a line. Similarly,) $ matches a right parenthesis at the end of a line, but not a right parenthesis within a line.

Specify a register name to use the current value stored in the register. Registers are not needed in C source debugging, but they are used frequently for assembler language debugging.

Format

```
[@]register
```

When you specify a symbol, CodeView first checks the symbol table to see if there is a symbol with that name. If CodeView does not find a symbol, it checks to see if the symbol is a valid register name. To identify the name with a register regardless of any name in the symbol table, use the sign @ as a prefix to the register name.

CodeView recognizes the register names in the following table:

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

Note: The 32-bit registers are only available if the 386 option is on and the computer is running in 386 mode.

In Example 3, the DUMP BYTES command dumps storage starting at a point 10 bytes beyond the symbol *label.*

```
>DB label+10              ;* Example 3
```

In Example 4, the DUMP BYTES command dumps storage at the address having the segment value stored in ES and the offset address 200.

```
>DB ES:200                ;* Example 4
```

Example 4 uses the UNASSEMBLE command (U) to list the assembler language statements starting 30 instructions before *label* and continuing to *label*.

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

Chapter 5. Executing Code

The TRACE (T), PROGRAM STEP (P), GO (G), EXECUTE (E) and RESTART (L) commands run code within a program. One difference between the commands is the size of the step that each command runs.

Command	Action
TRACE (T)	Runs the current source line in source mode or the current instruction in assembly mode; traces into functions, procedures, and interrupts.
PROGRAM STEP (P)	Runs the current source line in source mode or the current instruction in assembly mode; steps over functions procedures or interrupts.
GO (G)	Runs the current program.
EXECUTE (E)	Runs the current program in slow motion.
RESTART (L)	Restarts the current program.

In window mode, CodeView updates the screen to show any changes when you run a TRACE, PROGRAM STEP, or GO command. The highlight marking the current location moves to the new current instruction in the display window. Values change, if appropriate, in the register and watch windows.

In sequential mode, CodeView displays the current source line or instruction after each TRACE, PROGRAM STEP, or GO command. The format of the display depends on the display mode.

If the display mode is source (S+) in sequential mode, CodeView shows the current source line. If the display mode is assembly (S-), CodeView shows the status of the registers and flags and the new current instruction in the format of the REGISTER command. The mixed display mode (S&) shows the registers, the new source line, and the new instruction.

Use the following format to enter the dialog command.

Format

```
T [count]
```

If you specify the optional *count*, the command runs *count* times before stopping.

Example

These examples show the TRACE command in sequential mode. In window mode, there is no output from the commands, but CodeView updates the display showing changes caused by the command.

Example 1 sets the display mode to source. It uses the SOURCE LINE command to display the current source line. Notice that the current source line calls the function *analyze*. The TRACE command then runs the next four source lines. These lines are the first four lines of the *analyze* function.

The TRACE command operates similarly for all the languages. If you run the TRACE command when the current source line contains a subroutine, procedure, or function call, CodeView runs the first line of the called routine.

```
>S+                ;* C 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 2 sets the display mode to assembly and traces the current instruction. This example and the next one are the same as the examples of the PROGRAM STEP command. The TRACE and PROGRAM STEP commands are different only if the current instruction is a function, procedure, interrupt call, or REP instruction when the command runs.

The PROGRAM STEP command runs the current source line in source mode or the current instruction in assembly mode. The current source line or instruction is the one that the CS and IP registers pointed to. In window mode, the current instruction appears in reverse video or in a contrasting color.

In source mode, if the current source line contains a function call, CodeView runs the entire function and is ready to run the line after the function call. In assembly mode, if the current instruction is a Call, INT, or REP instruction, CodeView runs the entire procedure or interrupt and is ready to run the next instruction after the procedure or interrupt call.

Use the PROGRAM STEP command to run over function, procedure, and interrupt calls. To trace into any call except operating system calls. To trace into any call except operating system calls, use the TRACE (T) command. There is no direct way to trace into operating system calls.

Keyboard Selection	Mouse Selection
Press F10.	1. Point on **F8 = Trace**. 2. Click the right button.

Use the following format to enter the dialog command.

Format

```
P[count]
```

If you specify the optional *count*, the command runs *count* times before stopping.

Example

The examples show the PROGRAM STEP command in sequential mode. In window mode, there is no output from the commands, but CodeView updates the display to show changes caused by the command.

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

```
>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
>
```

To enter the GOTO command:

Keyboard Selection	Mouse Selection
1. Move the cursor to the source line or instruction you want to run to. 2. Press F7.	1. Point to the source line or instruction you want to run to. 2. Click the right button.
The highlight marking the current location moves to the source line or instruction unless a breakpoint or the end of the program is encountered. If the target line is in another module, use **Open** from the **Files** menu to load the source file for the other module.	

Use the following format to enter the dialog command.

Format

```
G [breakaddress]
```

If you specify the command with no argument, CodeView runs until it finds a breakpoint or the end of the program.

You can use the GOTO form of the command by specifying a *breakaddress*. The *breakaddress* can be a symbol, a line number, or an address in the *segment:offset* format. An offset address without a segment causes CodeView to use the address in the CS register as the default segment. If you give the *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
```

Example

These examples show the GO command in sequential mode. In window mode, there is no output from the commands, but CodeView updates the display to show changes caused by the command.

In Example 1, the display mode is set to source (S+). When you enter the GO command, CodeView starts the program running at the current address and continues until it reaches the start of the subprogram BUBBLE.

The EXECUTE command is similar to the GO command with no arguments, except that it runs in slow motion. The run starts at the current address and continues to the end of the program or until CodeView reaches a breakpoint, tracepoint, or watchpoint. To stop the command from running the program, press any key or click a mouse button.

Keyboard Selection	Mouse Selection
1. Press Alt+R. 2. Press E.	1. Point to **Run**. 2. Press and hold a button. Drag the highlight to **Execute**; release.

Use the following format to enter the dialog command.

Format

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 runs one instruction at a time. The command displays the current status of the registers and the instruction. In mixed mode, it also shows any source line at the instruction. In source mode, the command runs one source line at a time, displaying the lines as it runs them.

Note: The EXECUTE command has the same command letter (E) as the ENTER command. If the command has at least one argument, CodeView interprets the E as ENTER. If there is no argument, CodeView interprets it as EXECUTE.

Chapter 6. Examining Data and Expressions

The following table summarizes the commands and their actions described in this chapter.

Command	Action
DISPLAY EXPRESSION	Displays values of CodeView expressions.
GRAPHIC DISPLAY	While in window mode, nested structures are shown as a null-terminated ASCII string next to any field that contains a character pointer.
DISPLAY SYMBOLS	Displays the names and addresses of symbols and the names of defined modules within a program.
DUMP COMMANDS	Multiple DUMP commands allow contents of storage to be dumped to the screen or output device.
COMPARE MEMORY	Compares two blocks of memory.
SEARCH MEMORY	Scans a specified area of memory for specific byte values.
PORT INPUT	Reads and displays a byte from a specified hardware port.
REGISTER	Displays the contents of the processor's registers and changes the values of the registers.
87	Dumps the contents of the 8087, 80287, or 80387 registers. Use only with a numeric coprocessor chip or an 8087 emulator math library.

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

produces the output 100000. However, the command ?100000,hd produces the output –31072 because only the short *int* part of the value is evaluated.

Notes:

1. In the C language, CodeView does not support the *n* and *p* type specifiers and the F and H prefixes even though the C *printf* function does.
2. The DISPLAY EXPRESSION command does not work for programs assembled with IBM Macro Assembler/2 because Assembler does not write information to the object file about the type size of each variable. Use the DUMP command instead.
3. Do *not* use a type specifier when evaluating strings in BASIC or Pascal. Simply leave off the type specifier, and the expression evaluator displays the string correctly. The *s* type specifier assumes the C language string format, which other languages conflict with; if you use *s*, then CodeView displays characters at the given address until it encounters a null.

The remainder of this section gives examples that are relevant to all languages, then gives examples specific to C, BASIC, and Pascal.

Use the C expression evaluator to debug code written with Macro Assembler.

Example

Example 1 displays the value of the variable *amount*, an integer. The value stored in *amount* first displayed in the system radix (in this case, decimal), then in hexadecimal, then in octal.

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

Example 2 shows how CodeView can be used as a calculator. It converts between radixes, calculates the value of constant expressions, or checks ASCII equivalences.

Example 2 illustrates how to display the values of members of a structure. The same format applies to unions.

```
>? student.id                ;* Example 2
19643
>? pstudent->id
19643
>
```

Example 3 shows how the DISPLAY EXPRESSION command changes the values of variables with the C expression evaluator.

```
>? amount                    ;* Example 3
500
>? ++amount
501
>? amount
501
>? amount=600
600
>? amount
600
>
```

Example 4 shows how functions can be evaluated in expressions. CodeView runs the function square with an argument of 9 and displays the value returned by the function. You can display the values of functions only after you have run into the function *main*.

```
>? square(9)                    ;* Example 4
81
>
```

BASIC Examples

These examples assume that the BASIC source file contains the following statements:

```
amount% = 500
strng$ = "Here is a string"
```

Also, assume that the program has run to these statements and that the BASIC expression evaluator is in use.

Example 1 shows how to examine strings with the BASIC expression evaluator. Do not use the *s* format specifier.

casting. The C expression evaluator assumes INTEGER when working with Pascal variables unless you tell it otherwise.

Example 1 then uses the address-of operator (&) to display, in hexadecimal, the address where the value of hours is stored. Only the offset portion of the address is shown; the data segment is assumed.

```
>? hours          ;* Example 1
14
>? hours,x
000e
>? hours,o
16
>? &hours,x
0068
>
```

Example 2 displays the value of the variable *speed* first in the default decimal format, then in exponential format. Because *speed* is not an INTEGER, you must specify its type as described in "Working with Pascal Programs" in Chapter 4.

Example 2 then uses the address-of operator (&) to display the address where the value of *speed* is stored.

```
>? *(double *)&speed
8.9285717010498
>? *(double *)&speed,e
8.928572e+000
>? &speed
32932:108
>
```

Example 3 shows how to examine STRINGs. The C expression evaluator expects a zero byte to indicate the end of a string, but Pascal does not provide one. Therefore, the best way to examine STRINGs is not to use the DISPLAY EXPRESSION command but to use the DUMP ASCII command and specify the actual length of the STRING variable (here, 23 bytes).

```
>da buffer 1 23        ;* Example 3
80A4:0050   Here is a string.
>
```

Assembly Examples

By default, the C expression evaluator debugs assembly modules; however, some C expressions are particularly helpful for debugging assembly code. The following are typical examples:

Example 1 displays the first byte at the location pointed out by BX and is equivalent to the assembly expression BYTE PTR [bx].

```
>? BY bx                          ;* Example 1
12
>
```

Example 2 displays the first word at the location pointed out by [bp+8].

```
>? WO bp+8                       ;* Example 2
9359
>
```

Example 3 displays the first doubleword at the location pointed out by [si+12].

```
>? DW si+12                       ;* Example 3
12555324
>
```

Examples 4 and 5 use type castes, which are similar to the Macro Assembler PTR operator. The expression (char) *var* displays the byte at the address of *var* in signed format. The expression *(int)* var displays the word at the same address, also in signed format. Alter either of these commands to display results in unsigned format simply by using the *u* format specifier, as shown in Example 6.

```
>? (char) var                     ;* Example 4
5
>? (int) var                      ;* Example 5
1005
>

>? (char) var,u                   ;* Example 6
>? (int) var,u
```

Nested structures, such as *c*, are displayed as {...}. The dialog window displays a null-terminated ASCII string next to any field that contains a character pointer.

To expand a nested structure or view the data addressed by a pointer, move the cursor to the appropriate field and press Enter, or with a mouse, point to the appropriate field and click the left button. If the field does not contain a nested structure or pointer, CodeView beeps to indicate that you cannot select that field; otherwise, the new structure or the data addressed by the pointer is displayed in the dialog window.

When viewing the data addressed by pointers, the type of the pointer determines what format is used to display the new data. Notice that whatever data is being pointed to will be displayed, even if the pointer does not currently address meaningful data.

To return to the previous structure or pointer, press the Backspace key or click the mouse button on the right.

To quit the GRAPHIC DISPLAY command, press Esc or click the left mouse button while the mouse pointer is outside of the dialog window.

While the dialog window is displayed on the screen, you cannot run other CodeView commands.

Note: To use the GRAPHIC DISPLAY command, CodeView must be in window mode. (In sequential mode, the value of each field is displayed as if the DISPLAY EXPRESSION command were used. You cannot trace through nested structure the way you can in window mode.)

X?*routine.symbol*	The specified *symbol* in the specified *routine*. CodeView looks for *routine* first in the current module, then in other modules from first to last.
X?*routine.**	All symbols in the specified *routine*. CodeView looks for *routine* first in the current module, then in other modules from first to last.
X?*symbol*	Where CodeView looks for the specified *symbol*: 1. In the current function 2. In the current module 3. In other modules, from first to last.
X?*	All symbols in the current function.
X*	All module names.
X	All symbolic names in the program, including all modules and all symbols.

Note: When you debug an assembly module, you cannot use the *routine* field; use the *module* field. Therefore, the only versions of this command that work with assembly modules are:

```
X?module!*
X?module!symbol
```

C Examples

In the following examples, assume the program you are examining is PI.EXE and that it consists of two modules: PI.C and MATH.C. The PI.C module is the driver module. It consists of the *main* function only. The MATH.C module has several functions. Assume that the current function is *div* within the MATH.C module.

Example 1 lists the two user-created modules of the program calls as well as the library modules used in the program.

```
>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 5 shows one specific variable *s* within the *arctan* function.

```
>\fXX?math!arctan.s          ;* Example 5
3A79:00FA int     arctan()
     [BP+0004] int      s
>
```

BASIC Examples

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

Example 6 lists the two modules of the program, including PROG.OBJ, which is the main module. It lists the BASIC library files called by the program.

```
>X*                          ;* Example 6
PROG.OBJ
SORT.OBJ
BRUN303.LIB(ftmdata)
BRUN303.LIB(crt0)
BRUN303.LIB(crt0dat)
⋮
BRUN303.LIB(doexec)
BRUN303.LIB(execmsg)
```

Example 7 lists the symbols in the current routine and is the main program. Although the main program has no label and does not show in a stack trace, it is still an independent routine and has its own local variables. BASIC does not put local variables to the stack unless they are subprogram parameters.

```
>X?*                         ;* Example 7
     5825:17BE integer         A%[array]
     5825:1780 single          HOURS!
     5925:1784 integer         I%
```

```
>X?SORTS!*     ;* Example 11
5B37:0072 <no type>       J      5B37:0070 <no type>       K
5B37:006C <no type>       TEMP   54D1:0003 <no type>       SWAP
54D1:00DC <no type>       SORTS  54D1:003F <no type>       SORT
```

command 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, CodeView displays the first 64 words (128 bytes) of data declared in the data segment. If you repeat the same command, CodeView displays the next 64 words after 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) displays a number that contains one of the following character sequences:

#NaN
#INF
#IND

NaN (not a number) shows data that CodeView cannot evaluate as a real number. INF (infinity) shows data that CodeView can evaluate to infinity. IND (indefinite) shows data that CodeView can evaluate to an indefinite number.

The next sections discuss variations of the DUMP commands. The order corresponds to the length of data that each command displays.

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. Use the following format to enter the dialog command.

Format

```
DB [address | range]
```

Each line displays the address of the first byte in the line, followed by up to 16 hexadecimal byte values. Corresponding ASCII values follow the byte values. Spaces separate the hexadecimal values, except the eighth and ninth values, which a dash separates. CodeView displays ASCII values without separation. ASCII values that CodeView cannot display (lower than 32 or higher than 126) appear as periods. CodeView displays no more than 16 hexadecimal values in a line. The command displays values and characters until the end of the *range* or, if you gave no *range*, until it displays the first 128 bytes.

Example

The example displays the byte values from DS:0 to DS:36 (DS:0x24). CodeView assumes the data segment if no segment is given. ASCII characters appear on the right.

```
>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 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. Spaces separate the values. The command displays values until the end of the *range* or until the first 64 integers are displayed. Use the following format to enter the dialog command.

Format

```
DI [address | range]
```

Note: In the C language, the size of an integer is system dependent. CodeView assumes an integer has a 2-byte value.

Example

This 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.

```
>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 DUMP WORDS command (DW) 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. Spaces separate the hexadecimal values. The command displays values until the end of the *range* or until the first 64 words appear. Use the following format to enter the dialog command.

Format

```
DW [address | range]
```

Example

This example displays the word values from DS:0 to DS:36 (DS:0x24). No more than eight values per line appear.

```
>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 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. Use the following format to enter the dialog command.

Format

```
DS [address | range]
```

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, it shows the hexadecimal values of the bytes in the number, followed by the decimal value of the number. Spaces separate the hexadecimal values.

The decimal value has the form:

```
[-]digit.decimaldigitsE[+|-]exponent
```

If the number is negative, it has a minus sign; positive numbers have a plus sign. A decimal point follows the first digit of the number. Six decimal places appear after the decimal point. After the decimal digits comes the letter E, which marks the start of a 3-digit signed *exponent*.

The command displays at least one value. If it specifies a *range*, all values in the range appear.

Example

This example displays the short-real, floating-point number at the address of the variable *_spi*. Only one value appears per line.

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

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. Use the following format to enter the dialog command.

Format

```
DT [address | range]
```

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 appear, followed by the decimal value of the number. Spaces separate the hexadecimal values.

The decimal value has the form:

```
[–]digit.decimaldigitsE[+|–]exponent
```

If the number is negative, it has a minus sign; positive numbers have a plus sign. A decimal point follows the first digit of the number. Sixteen decimal places appear after the decimal point. After the decimal digits comes the letter E, which marks the start of a 3-digit, signed *exponent*.

The command displays at least one value. If it specifies a *range*, all values in the range appear.

Example

This example displays the 10-byte, real, floating-point number at the address of the variable *_tpi*. Only one number appears per line.

```
>DT _tpi
5E68:0300  DE 87 68 21 A2 DA 0F C9 00 40
3.1415926535897930E+000 >
```

The SEARCH MEMORY command (do not confuse this command with the SEARCH command), scans a specified area of memory for specific byte values. It is primarily of interest to programmers using assembly mode and users who want to test for the presence of specific values within a range of data.

Keyboard Selection	Mouse Selection
Dialog command only.	Dialog command only.

Use the following format to enter the dialog command.

Format

```
S range list
```

CodeView searches the specified *range* of memory locations for the byte values specified in the *list*. If CodeView finds bytes with the specified values, it displays the addresses of each occurrence of bytes in the list.

The *list* can have any number of bytes. A space or comma separates each byte value, unless the list is an ASCII string. If the list contains more than one byte, the SEARCH MEMORY command looks for a series of bytes that precisely matches the order and value of bytes in *list*. If found, the beginning address of each series is displayed.

Example

Example 1 displays the address of each memory location containing the string *error*. The command searches the first 1500 bytes at the address specified by *buffer*. The string was found at the three addresses CodeView displayed.

```
>S buffer L 1500 "error"       ;* Example 1
2BBA:0404
2BBA:05E3
2BBA:0604
>
```

I
PORT INPUT

The PORT INPUT command reads and displays a byte from a specified hardware port. It is primarily of interest to assembly language programmers writing hardware-specific programs.

Keyboard Selection	Mouse Selection
Dialog command only.	Dialog command only.

Use the following format to enter the dialog command.

Format

```
I port
```

The PORT INPUT command reads and displays the byte from the specified *port* that is any 16-bit address. The PORT INPUT command is often used in conjunction with the PORT OUTPUT command.

Example

The example reads input port number 2F8 and displays the result, E8.

```
>I 2F8       ;* hexadecimal radix assumed
E8
>
```

In sequential mode, the TRACE (T), PROGRAM STEP (P), and GO (G) commands show registers in the same format as the REGISTER command.

Example
(Examples 1 and 2 are both C examples but apply equally well to BASIC.)

Example 1 displays all register and flag values, as well as the instruction at the address pointed to by the CS and IP registers. Because the mode is mixed, the current source line also appears.

The storage operand [*_letters*] is evaluated 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]*) increments the value.

```
>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 2 shows the display mode set to assembly (S-) with no source line. The breakpoint number is at the right of the last line. It shows the current address is breakpoint 2.

```
>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
>
```

Example

In this example, the first line 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 is an interpretation of the bits of the number. The closure method can be projective or affine. The rounding method rounds to the nearest even number, rounding down, rounding up, or the chop method (cutting off toward zero). The precision is 64 bits, 53 bits, or 24 bits.

```
>7
cControl 037F  (Projective closure, Round nearest, 64-bit
precision)    iem=0 pm=1 um=1 om=1 zm=1 dm=1 im=1

cStatus 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  exponent           Value
cST(3) special 7FFF 8000000000000000 = + Infinity
cST(2) special 7FFF 0101010101010101 = + Not a Number
cST(1) valid   4000 C90FDAA22168C235 = + 3.141592265110390E+000
cST(0) zero    0000 0000000000000000 = + 0.000000000000000E+000
>
```

The second line shows if 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)
- Invalid-operation mask (im).

The third line shows the hexadecimal value of the status register (6004 in the example) and an interpretation of 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)

Chapter 7. Managing Breakpoints

You control how a program runs by setting breakpoints. A *breakpoint* is an address where a program stops each time CodeView finds the address. By setting breakpoints at key addresses in the program, you can *freeze* the program and examine the status of memory or expressions at that point. You can also use the WATCHPOINT (WP) and TRACEPOINT (TP) commands to set conditional breakpoints. The following table lists the BREAKPOINT commands that are discussed in this chapter.

Command	Action
BREAKPOINT SET (BP)	Creates a breakpoint at a specified address.
BREAKPOINT CLEAR (BC)	Temporarily removes one or more breakpoints.
BREAKPOINT DISABLE (BD)	Temporarily removes one or more breakpoints.
BREAKPOINT ENABLE (BE)	Restores one or more disabled breakpoints.
BREAKPOINT LIST (BL)	Displays information about current breakpoints.

The dialog version of the command is more powerful than the mouse or keyboard versions because it lets you give a *pass count* and a string of *commands*. The *pass count* specifies the first time that CodeView is to take the breakpoint.

The *commands* are a list of dialog commands enclosed in quotes and separated by semicolons. For example, if you specify the commands as **"? code;G"**, CodeView automatically displays the value of the variable *code* and then runs the GO command each time it encounters a breakpoint.

In window mode, a breakpoint entered with a dialog command is displayed the same way as one created with a window command. CodeView displays the source line or instruction corresponding to the breakpoint location in highlighted text.

In sequential mode, CodeView displays information about the current instruction each time a program encounters a breakpoint. CodeView can show the register values, the current instruction, and the source line depending on the display mode.

In assembly or mixed mode, CodeView also displays a breakpoint as a comment to the right of the instruction. Disabling the breakpoint does not remove the comment; however, deleting the breakpoint does.

Example

Example 1 creates a breakpoint at line 19 of the current source file.

```
>BP .19                          ;* Example 1
>
```

Example 2 creates a breakpoint at the address of the function display.

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

CodeView passes over the breakpoint nine times before taking it on the tenth pass. Each time the program stops for the breakpoint, CodeView runs the quoted commands. The DISPLAY EXPRESSION command increases the counter, then the GO command restarts the program. If you set the counter to 0 when you set the breakpoint, this

The BREAKPOINT CLEAR command permanently removes one or more previously set breakpoints.

Keyboard Selection	Mouse Selection
To remove a single breakpoint: 1. Move the cursor to the breakpoint line or instruction you want to clear. 2. Press F9.	To remove a single breakpoint: 1. Point to the breakpoint line or instruction you want to clear. 2. Click the left button.
To remove all breakpoints: 1. Press Alt+R. 2. Press C.	To remove all breakpoints: 1. Point to **Run**. 2. Press and hold a button. Drag to **Clear Breakpoints**; release.

Use the following format to enter the dialog command.

Format

```
BC [list]
BC *
```

If you specify a *list*, the command removes the breakpoints named in the list. The *list* can be any combination of integer values from 0 through 19. If you specify an asterisk as the argument, CodeView removes all breakpoints.

Example

Example 1 removes breakpoints 0, 4, and 8.

```
>BC 0 4 8          ;* Example 1
>
```

Example 2 removes all breakpoints.

```
>BC *              ;* Example 2
>
```

After using the BREAKPOINT DISABLE command to turn off breakpoints, you can restore them with the BREAKPOINT ENABLE command.

Keyboard Selection	Mouse Selection
1. Move the cursor to the source line or instruction of the breakpoint. 2. Press F9.	1. Point to the source line or instruction of the breakpoint. 2. Click the left button.
This is the same as creating a new breakpoint at that location.	

Use the following format to enter the dialog command.

Format

```
BE [list]
BE *
```

If you specify a *list*, the command restores the breakpoints named in the list. The *list* can be any combination of integer values from 0 through 19. Use the BREAKPOINT LIST command if you need to see the numbers for each existing breakpoint. If you specify an asterisk, CodeView restores all breakpoints.

Example

Example 1 restores breakpoints 0, 4, and 8.

```
>BE 0 4 8   ;* Example 1
>
```

Example 2 restores all breakpoints.

```
>BE *       ;* Example 2
>
```

This shows that you probably set the breakpoint in assembly mode because in source mode it is difficult to set a breakpoint anywhere except on a source line.

Chapter 8. Managing Watch Statements

Watch statement commands let you set, delete, and list watch statements. *Watch statements* are specifications that describe expressions or areas of storage to watch. Some watch statements also specify conditional breakpoints that may or may not be honored, depending on the value of the expression or memory area. The following table lists the watch statement commands discussed in this chapter.

Command	Action
WATCH (W?)	Adds a variable or memory location to the watch list.
WATCHPOINT (WP)	Adds a watchpoint to the watch list.
TRACEPOINT (TP)	Adds a tracepoint to the watch list.
WATCH DELETE (Y)	Removes one or more items from the watch list.
WATCH LIST (W)	Displays information about current watch items.

Important

The format for each CodeView command is always the same, regardless of the expression evaluator. However, the method for specifying an *argument* may vary with the language. Each example in this chapter repeats with C, BASIC, and Pascal arguments. The sample screens feature BASIC. At the end of this chapter are sample screens that incorporate the examples in C, Pascal, and Macro Assembler (with the exception of WATCH DELETE and WATCH LIST).

Watch statements are like breakpoints; they remain in memory until you remove them or leave CodeView. In window mode, you can enter watch statement commands either in the dialog window or with menu selections. Current statements appear between the menu bar and the source window.

Use the WATCH command 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 show in the watch window. The watch window updates to show new values each time the value of the watch statement changes during program execution.

In sequential mode, 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 is displayed. Type the expression followed by a comma and a format specifier. If you do not give a format specifier, CodeView displays the value in a default format.

Note: If your program directly searches for absolute addresses, you might get unexpected results with the DISPLAY EXPRESSION and DUMP commands. However, the WATCH command should show the correct values. This problem sometimes arises when CodeView and your program try to use the same memory location.

Keyboard Selection	Mouse Selection
1. Press Alt+W then A. or 2. Press Ctrl+W. 3. Type the expression; press Enter.	1. Point to **Watch**. 2. Press and hold a button. Drag to **Add Watch**; release. 3. Type the expression; press Enter or a button.

The dialog version of the command must be used when specifying a range of memory for CodeView to watch. Use the following format to enter the command.

Format

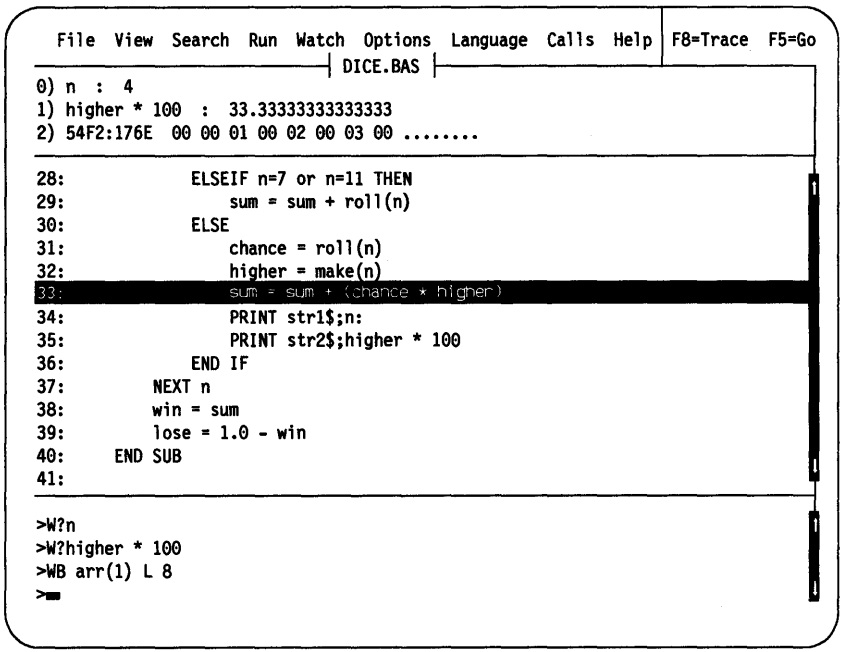
```
W? expression[,format] ;Watch expression
W[type] range          ;Watch memory
```

An *expression* used with the WATCH command can be a simple variable or a complex expression using several variables and operators.

same function, you enter them differently. (C uses a different format for indexing arrays.)

```
WB arr(1) L 8                    ;* BASIC Example 3
WB arr[0] L 8                    ;* C Example 4
```

These commands, entered while debugging a BASIC program, produce the watch window in the following figure. (Corresponding C, Pascal, and Macro Assembler examples are included at the end of the chapter.)



Keyboard Selection	Mouse Selection
1. Press Alt+W. 2. Press W. 3. Type the expression; press Enter.	1. Point to **Watch**. 2. Press a button and hold. Drag to **Watchpoint**; release. 3. Type the expression; press Enter or a button.

Use the following format to enter the dialog command.

Format

```
WP? expression[,format]
```

The *expression* can be any valid CodeView expression. You can enter *format* as a *printf* type specifier, but there is seldom reason to do so since the expression value is normally either 1 or 0.

Example

The following dialog commands display a watch statement (watchpoint) in the watch window:

Example 1 instructs CodeView to break running when the variable *higher* is greater that the variable *chance.* (Note that BASIC and C use the same format.) After setting this watchpoint, use the GO command to run until the condition becomes true.

```
WP? higher > chance          ;* BASIC/C Example 1
```

Examples 2 and 3 instruct CodeView to break running when the variable *n* is equal to 7 or 11. Example 4 instructs CodeView to break when the variable *lines* is equal to 11.

```
WP? n=7 or n=11              ;* BASIC Example 2
WP? n==7 || n==11            ;* C Example 3
WP? (* (short *)&lines)==11  ;* Pascal Example 4
```

Note: BASIC and C each display a numerical result in response to a Boolean expression (0 being equivalent to false, nonzero to true).

These commands, entered while debugging a BASIC program, produce the watch window in the following figure. (Corresponding C,

Use the TRACEPOINT command to set a conditional breakpoint called a tracepoint. A tracepoint breaks program execution when there is a change in the 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 watch and test for change. The statement remains in memory until you delete it or quit CodeView.

Note: In window mode, tracepoint statements and their values appear in highlighted text. In sequential mode, there is no watch window, so the values of tracepoint statements can only display with the WATCH LIST command.

An expression used with the TRACEPOINT command must evaluate an *lvalue*. The expression must refer to an area of memory not more than 128 bytes in size.

For example, *i* = = 10 (similar to I = 10 in BASIC) is 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 calculates the sum of the value of two memory locations. The expresssion buffer is invalid if the buffer is declared as an array of 64 bytes and gives the TRACEPOINT command with the expression buffer checking all 64 bytes of the array. The same command given with the C expression buffer [32], or BUFFER (33) in BASIC, means that it checks only one byte (the 33rd).

Note: The following is relevant only for C programs:

Register variables are not considered *lvalues*. Therefore, if *i* declares as register *int I*, the command TP? is invalid. However, you can still check for changes in the value of *i*. Use the DISPLAY SYMBOLS command to find which register contains the value of *i*. Then find the value of *i*. Finally, set up a watchpoint to test the value. For example, use the following sequence of commands:

Specifier	Size
None	The 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

If you specified no type size, the default type is the last type used by a DUMP, ENTER, WATCH storage, or TRACEPOINT memory command. If you have used none of these commands, the default type is byte.

Example

The following dialog commands display two watch statements (tracepoints) in the watch window.

Example 1 instructs CodeView to stop whenever the variable *sum* changes.

```
TP? sum                         ;* Example 1
```

The following example instructs CodeView to suspend program execution whenever the first byte at the address *n* changes; the address of this byte and its contents display. The value of *n* may change because of a change in the second byte at the address *n*. This change (by itself) has no effect on this tracepoint.

```
TPB n                           ;* Example 2
```

Examples 3 and 4 instruct CodeView to stop whenever any of the first 8 bytes, starting with the address of the first element or *arr*, change in value.

```
TPB arr(1) L 8                  ;* BASIC Example 3
TPB arr[0] L 8                  ;* C Example 4
```

Example 5 displays three watch statements and instructs CodeView to stop whenever the value of the variable *primes* changes.

The WATCH DELETE command lets you 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. For example, if three watch statements are in the window and you delete statement 1, CodeView redraws the window with one less line. Statement 0 remains unchanged, but statement 2 becomes statement 1. If there is only one statement, the window disappears.

To delete a single watch statement:

Keyboard Selection	Mouse Selection
1. Press Alt+W then D or 2. Press Ctrl+U 3. Use the cursor keys to move to the statement(s) you want to delete; press Enter.	1. Point to **Watch**. 2. Press a button and hold. Drag to **Delete Watch**; release. Point to the statement you want to delete; press a button.

To delete all watch statements:

Keyboard Selection	Mouse Selection
1. Press Alt+W. 2. Press L.	1. Point to **Watch**. 2. Press a button and hold. Drag to **Delete All Watch**; release.

Use the following format to enter the dialog command.

Format

```
Y number|*
```

When you set a watch statement, it assigns a number automatically, starting with 0. In window mode, the number appears to the left of the watch statement in the watch window. In sequential mode, use the

The WATCH LIST command lists all previously set watchpoints and tracepoints with their assigned numbers and current values.

This command is the only way to examine current watch statements in sequential mode. The command has little use in window mode because watch statements are already visible in the watch window.

Keyboard Selection	Mouse Selection
Dialog command only.	Dialog command only.

Use the following format to enter the dialog command.

Format

```
W
```

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 does not take an argument. The WATCH command requires at least one argument.

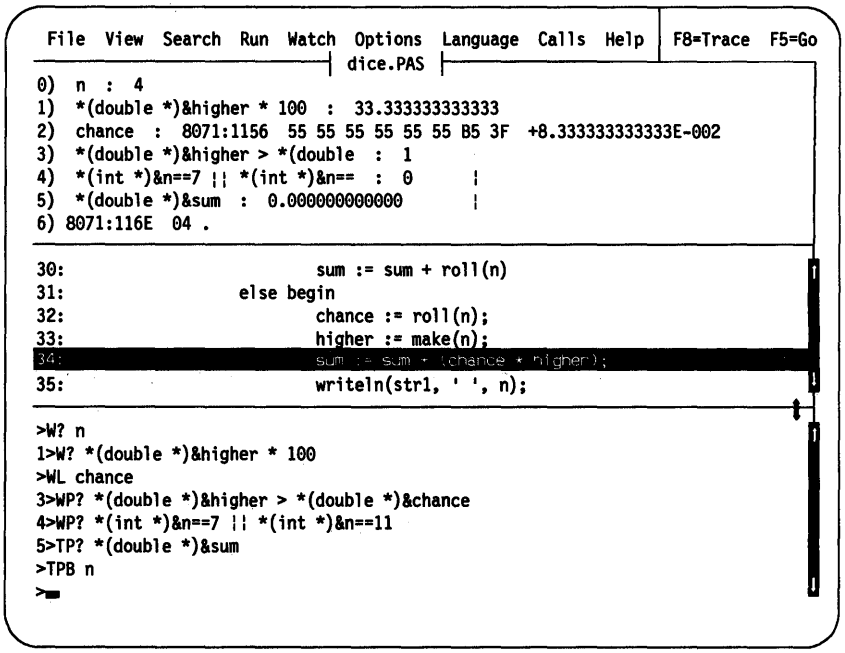
Example

The following example shows the use of the WATCH LIST command.

```
>W
0) code,c  :  I
1) (float)letters/words  :  4.777778
2) 3F65:0B20 20 20 43 4F 55 4E 54 COUNT
3) lines==11 :  0
>
```

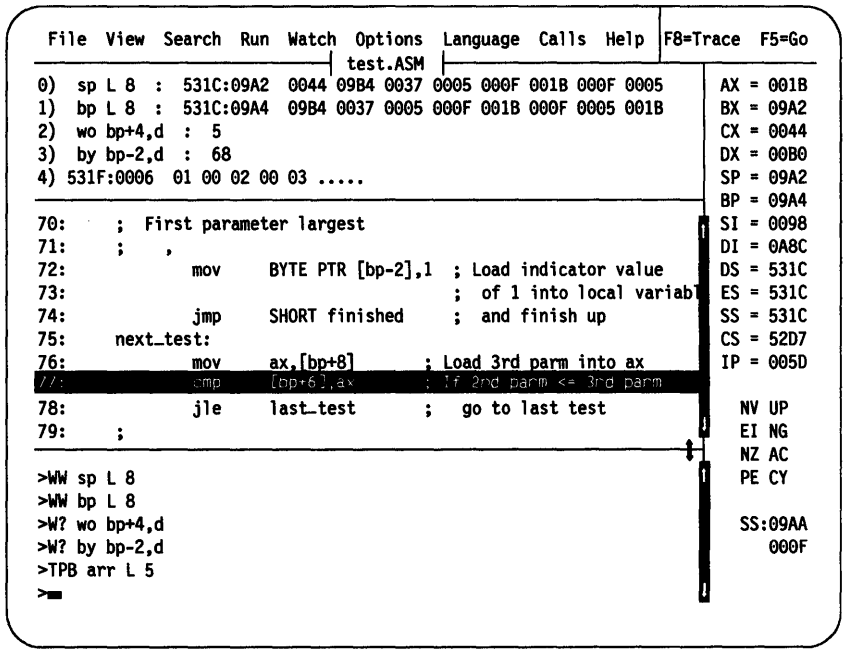
Pascal Examples

The examples shown previously in a BASIC screen are entered in a Pascal debugging session as follows:



- Items 0 through 2 in the watch window are watch statements. They display values but never cause running to break.
- Items 3 and 4 are watchpoints; they cause running to break whenever they evaluate to true (nonzero).
- Item 3 breaks running whenever *higher* is greater than *chance.*
- Item 4 breaks running whenever *n* is equal to 7 or 11.
- The last two items are tracepoints. They cause running to break whenever any bytes change within a specified area of memory.
- Item 5 breaks running whenever the value of *sum* changes.
- Item 6 breaks running whenever there is a change in the first byte at the address of *n*.

The previous examples produce the following screen when entered in a CodeView debugging session.



Chapter 9. Examining Code

Several CodeView commands let you examine program code or data related to code. The following table lists the commands that will be discussed in this chapter.

Command	Action
SET MODE (S)	Sets format for code displays.
UNASSEMBLE (U)	Displays assembly instructions.
VIEW (V)	Displays source lines
CURRENT LOCATION (.)	Displays the current location line.
STACK TRACE (K)	Displays routines or procedures.

Use the following format to enter the dialog command.

Format

```
S[+ | - | &]
```

S+ If you specify the plus sign, CodeView selects source mode and displays the word source.

S- If you specify the minus sign, CodeView selects assembly mode and displays the word assembly. The S- command always turns off the mixed source option.

S& If you specify the ampersand, CodeView selects mixed mode and displays the word mixed. In addition, the S& command turns on the mixed source option.

S If you specify no argument, CodeView displays the current mode (source, assembly, or mixed).

The UNASSEMBLE command in sequential mode is an exception; 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, CodeView displays the name of the new display mode: source, assembly, or mixed.

Example

This example shows how to change the source mode to source, assembly, and mixed. In window mode, the commands change the format of the display window. In the 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 display mode carries them out.

```
>S+
source
>S-
assembly
>S&
mixed
>
```

the dialog window. If you specify an *address*, the instructions in the display window begin at the specified address. If you specify a *range*, CodeView uses only the starting address. If you specify no argument, CodeView scrolls down and displays the next screen of assembler language instructions.

Notes:

1. Occasionally, code similar to the following is displayed:

```
FF30     ???   Byte Ptr  [BX + SI]
```

2. If you attempt to unassemble data, then CodeView may display meaningless instructions.

Example

This sequential mode example displays eight lines of mixed source and unassembled machine code. The output is the same if the mode is source.

```
>S&                     ;* Example 1
mixed
>U 0x11
50CE:0011 8BEC          MOV       BP,SP
50CE:0013 B80000        MOV       AX,0000
50CE:0016 E8F413        CALL      __chkstk (140D)
50CE:0019 57            PUSH      DI
50CE:001A 56            PUSH      SI
49:             if (argc == 1)    /* No command-line argument */
50CE:001B 837E0401      CMP       Word Ptr [argc],+01
50CE:001F 7403          JZ        _main+14 (0024)
```

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

```
>S-                           ;* Example 2
assembly
>U 0x11
50CE:0011 8BEC          MOV       BP,SP
50CE:0013 B80000        MOV       AX,0000
50CE:0016 E8F413        CALL      __chkstk (140D)
50CE:0019 57            PUSH      DI
50CE:001A 56            PUSH      SI
50CE:001B 837E0401      CMP       Word Ptr [argc],+01
50CE:001F 7403          JZ        _main+14 (0024)
50CE:0021 E90900        JMP       _main+1d (002D)
>
```

In sequential mode, the current display mode (source, assembly, or mixed) does not affect the VIEW command; source lines appear regardless of the mode.

In window mode, if you enter the VIEW command while the display mode is assembly, CodeView automatically switches back to source mode. If you specify a *linenumber* or an *expression*, CodeView redraws the display window so that the source line corresponding to the given *address* appears at the top of the source window. If you specify a *filename* with a *linenumber*, CodeView loads the specified file.

If you enter the VIEW command with no arguments, the display scrolls down one line short of a page; that is, the source line that was at the bottom of the window is now at the top.

Note: Entering the VIEW command with no argument is similar to pressing the PgDn key. The difference is that the PAGE DOWN command scrolls down one more line, and the cursor must be in the display window.

Example

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

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

Example 2 loads the source file MATH.C and displays eight source lines starting at line 30.

CURRENT LOCATION

The CURRENT LOCATION command displays the source line or assembler language instruction corresponding to the current program location.

Keyboard Selection	Mouse Selection
Dialog command only.	Dialog command only.

Use the following format to enter the dialog command.

Format

.

In sequential mode, the command displays the current source line. This line appears regardless of whether the current debugging mode is source or assembly. If a program has no symbolic information, CodeView ignores the command.

In window mode, this command puts the current program location, marked with reverse video or a contrasting color, in the center of the display window. The CURRENT LOCATION command does not affect the display, source, or assembly mode. This command is useful if you have scrolled through the source code or assembler language instructions so that the current location line is no longer visible.

For example, if you are debugging a program in window mode and have run the program near the start of the program but scroll the display to a point near the end, the command returns the display to the current program location.

Example

This example shows how to display the current source line in sequential mode. The same command in window mode does not produce any output, but it can change the text shown in the display window.

```
>.
    count = 1
>
```

Keyboard Selection	Mouse Selection
1. Press Alt+C. (The menu shows the current routine at the top, and other routines below it are in reverse order. The values of any routine arguments are shown in parentheses following the routine.) 2. To view code at the point of the called routine, press the down arrow to move to the routine below the one you want to view; press Enter.	1. Point to **Calls**. 2. Press a button. (The current routine is at the top and other routines below it are in reverse order. The values of any routine arguments are shown in parentheses following the routine.) 3. To view code at the point of the called routine, hold a button down; drag to the routine below the one you want to view; release.
The cursor moves to the calling source line or the calling instruction. The cursor indicates the calling location in the selected routine where the next-level routine was called. If you select the current (top-level) routine, the cursor moves to the current location in that routine.	

Use the following format to enter the dialog command.

Format

K

The output from the STACK TRACE dialog command lists the functions in the reverse order that CodeView called them. The arguments for each function appear in parentheses. Finally, the line number from which CodeView called the function appears.

Enter the line number as an argument to the VIEW or UNASSEMBLE command to view code at the point where CodeView called the function.

In window mode, the output from the STACK TRACE dialog command appears in the dialog window. You might need the dialog version instead of the menu version because the **Calls** menu can be cut off 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.

MAKE# are both functions returning a double-precision floating-point number. A function that returned a short integer would have a % type tag. CALC does not have a type tag because it is a subprogram and, therefore, does not return a value.

Chapter 10. Changing Code or Data

CodeView provides the following commands for changing code or data in memory.

Command	Action
Assemble	Modifies code.
Enter	Modifies memory, usually data.
Register	Modifies registers and flags.
Fill Memory	Fills a block of memory.
Move Memory	Copies one block of memory to another.
Port Output	Outputs a byte to a hardware port.

Changes to code are temporary. You can use them for testing in CodeView, but you cannot save them or permanently change the program. To make permanent changes, you must change the source code and recompile.

8086-family instruction mnemonic form. Enter instructions in uppercase, lowercase, or mixed case.

To assemble a new instruction, type the desired mnemonic and press Enter. CodeView assembles the instruction into storage and displays the next available address. Continue entering new instructions until you assemble all the instructions you want. To end assembly and return to the CodeView prompt, press Enter.

If an instruction contains a syntax error, CodeView displays the message:

```
^ Syntax error
```

CodeView then displays the current assembly address again and waits for a correct instruction. The caret mark (^) in the message points to the first character that CodeView cannot interpret.

The following rules govern entry of instruction mnemonics:

- The far-return mnemonic is RETF.
- String mnemonics must explicitly state the string size. For example, use MOVSW to move word strings and MOVSB to move byte strings.
- CodeView automatically assembles SHORT, NEAR, or FAR jumps and calls depending on byte displacement to the destination address. You can cancel these with the NEAR or FAR prefix, as shown in the following examples:

  ```
  JMP      0x502
  JMP NEAR 0x505
  JMP FAR  0x50A
  ```

  You can abbreviate the NEAR prefix as NE, but you cannot abbreviate the FAR prefix.
- CodeView cannot tell whether some operands refer to a word memory location or to a byte memory location. In these cases, you must state the data type explicitly with the prefix WORD PTR or BYTE PTR. Acceptable abbreviations are WO and BY, as shown in the following examples:

Example

This example changes the instruction at address 0x40 so that it moves data into the CX register instead of the BX register. The UNASSEMBLE command shows the instruction before and after the assembly. (With BASIC, use &H40.)

You can change a portion of code for testing, as in the example, but you cannot save the changed program unless you change the source code and recompile.

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

value. You can then replace the value, skip to the next value, return to a previous value, or exit from 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 go past the end of the display, CodeView 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, CodeView 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 from the command at any time.

The following sections describe the ENTER commands in order by the size of data they store. For data types that CodeView normally dumps and enters in hexadecimal (bytes, ASCII, words, and doublewords), the examples specify a radix of 16.

Keyboard Selection	Mouse Selection
Dialog command only.	Dialog command only.

The ENTER BYTES command enters one or more byte values to memory. Use the following format to enter the dialog command.

Format

```
EB address [list]
```

The *address* is the location in memory where CodeView enters values. The *list* contains the values for the bytes that CodeView substitutes for the bytes in memory. CodeView replaces the byte at the specified address with the first byte of the *list*. It replaces the bytes at all subsequent addresses until it uses the values in the *list*.

If you do not specify a *list*, CodeView asks for a new value at *address* by displaying the address, its current value, and a trailing period. You can then replace the value or skip to the next value, go to the preceding byte, or end the command and return to the command prompt.

Example

Example 1 first sets the radix to 16 so that you can enter numbers in hexadecimal. It then replaces the 3 bytes at DS:100, DS:101, and DS:102 with *01*, *2B*, and *E5*, respectively. DS is the default segment address for all ENTER commands.

```
>N16                        ;* Example 1
>EB 100 01 2B E5
```

Example 2 displays the current value on the line following the command and waits for you to enter a value.

```
>EB 100                     ;* Example 2
2344:0100  F3.e_
```

The underscore in Example 2 represents the cursor. CodeView waits for you to enter the first byte value. You can change the value *F3* to the new value *5E* by typing 5E.

Keyboard Selection	Mouse Selection
Dialog command only.	Dialog command only.

The ENTER ASCII command is like the ENTER BYTES command.

Use the following format to enter the dialog command.

Format

```
EA address [list]
```

The *address* is the location in memory where CodeView enters values. The *list* contains the values for the bytes that CodeView substitutes for the bytes in memory. CodeView replaces the byte at the specified address with the first byte of the *list*. It replaces the bytes at all subsequent addresses until it uses all the values in the *list*.

Example

The following example enters the string "Cannot open file into storage". CodeView starts at the symbolic address *message*. You can use the ENTER BYTES command to do the same thing.

```
>EA message "Cannot open file into storage"
```

You can also enter non-string values:

```
>EA message  1 2 3 4 5
```

Keyboard Selection	Mouse Selection
Dialog command only.	Dialog command only.

The ENTER UNSIGNED INTEGERS command enters into memory a word value in the unsigned integers format. Use the following format to enter the dialog command.

Format

```
EU address [list]
```

The *address* is the location in memory where CodeView enters values. The *value* is an unsigned integer that CodeView enters into memory. An unsigned integer can be any decimal integer between 0 and 65,535.

Enter the optional *list* as a list of expressions separated by spaces. Enter and evaluate the expressions in the current radix. If *list* is not given, CodeView prompts for new values. You must enter these values in decimal.

Example

The following example replaces the three unsigned integers at DS:256, DS:258, and DS:260 with 10, 20, and 30. (These addresses correspond to the hexadecimal addresses DS:0100, DS:0102, and DS:0104

```
>EU 256 10 20 30
>
```

This example replaces the integer at DS:256 (DS:0100) with 10.

```
>EU 256
3DA5:0100  130F.10
>
```

Keyboard Selection	Mouse Selection
Dialog command only.	Dialog command only.

The ENTER DOUBLEWORDS command enters a doubleword value into memory. Use the following format to enter the dialog command.

Format

```
ED address [list]
```

The *address* is the location in memory where CodeView enters values. You must type doublewords as two words separated by a colon. If you do not type the colon and enter only one word, only the offset portion of the address changes.

Enter the optional *list* as a list of expressions separated by spaces. Enter and evaluate the expressions in the current radix. If *list* is not given, CodeView prompts for new values. You must enter these values in hexadecimal.

Example

The following example replaces the doubleword at DS:256 (DS:0100 hexadecimal) with the decimal address 8700:12008 (hexadecimal address 21FC:2EE8).

```
>ED 256 8700:12008
>
```

The next example replaces the offset portion of the doubleword DS:256 (DS:0100 hexadecimal) with 2EE9 hexadecimal. Since the segment portion of the address is left out, the existing segment (21FC hexadecimal) does not change.

```
>ED 256
3DA5:0100  21FC:2EE8.2EE9
>
```

Keyboard Selection	Mouse Selection
Dialog command only.	Dialog command only.

The ENTER LONG REALS command enters a long-real value into memory. Use the following format to enter the dialog command.

Format

```
EL address [list]
```

The *address* is the location in memory where CodeView enters values. Enter the optional *list* as a list of real numbers separated by spaces. You must enter these numbers in decimal, regardless of the current radix. If *list* is not given, CodeView prompts for new values. Enter these values in decimal. Enter long-real numbers either in floating-point format or in scientific-notation format.

Example

The following example replaces the four numbers at DS:256, DS:264, DS:272, and DS:280 with the real numbers 23.479, 0.25, −1650.0, and 235.0. (These addresses correspond to the hexadecimal addresses DS:0100, DS:0108, DS:0110, and DS:0118.)

```
>EL 256 23.479 1/4 −1.65E+4 235
>
```

The next example replaces the number at the symbolic address PI with 3.141593.

```
>EL PI
3DA5:0064  42 79 74 65 DC 0F 49 40  5.012391E+001
                                    3.141593
>
```

F
FILL MEMORY

The FILL MEMORY command provides an efficient way of filling a large or small block of memory with any values. It is primarily of interest to assembly programmers because the command enters values directly into memory. However, you may find it useful for initializing large data areas such as an array or structure. Enter the arguments using any radix.

Keyboard Selection	Mouse Selection
Dialog command only.	Dialog command only.

Use the following format to enter the dialog command.

Format

```
F range list
```

The FILL MEMORY command fills the memory in the specified range with the byte values specified in *list*.

The *list* can be a series of byte values (separated by blanks or commas) or an ASCII string enclosed in quotation marks. The values in the list repeat until it fills the whole range. (If you specify only one value, it fills the entire range with that value.) If the list has more values than the number of bytes in the range, the command ignores any extra values.

Note: Hexadecimal radix is assumed for the following examples.

Example

Example 1 fills 255 (100 hexadecimal) bytes of memory starting at DS:0100 with the value 0. You might use this command to reinitialize the program's data without having to restart the program.

```
>F 100 L 100 0              ;* Example 1
>
```

Example 2 fills the 100 (64 hexadecimal) bytes of memory starting at *table* with the following hexadecimal byte values: 42, 79, 74. These three values repeat until they fill all 100 bytes.

The MOVE MEMORY command enables you to copy all the values in one block of memory directly to another block of memory of the same size. This command interests assembly programmers and those who need to do large data transfers efficiently.

Keyboard Selection	Mouse Selection
Dialog command only.	Dialog command only.

Use the following format to enter the dialog command.

Format

```
M range address
```

The values in the block of memory specified by *range* are copied to a block of the same size beginning at *address*. All data in *range* is copied completely over to the destination block, even if the two blocks overlap. If they do overlap, some of the original data in *range* is altered.

To prevent loss of data, the MOVE MEMORY command copies data starting at the source block's lowest address, whenever the source is at a higher address than the destination. If the source is at a lower address, then the MOVE MEMORY command copies data beginning at the source block's highest address.

Example

```
>M arr1[0] L arsize arr2[0]      ;* C Example
>
```

In the example, the block of memory beginning with the first element of *arr1*, and *arsize* bytes long, is copied directly to a block of the same size beginning at the address of the first element of *arr2*.

The REGISTER command has two functions. It displays the contents of the processor's registers and can change the values of those registers. You can find a description of the modification features of the command in this section. For a description of the display features of the REGISTER command, see "Examining Data and Expressions" in Chapter 6.

To display the registers:

Keyboard Selection	Mouse Selection
Press Alt+V then press R or Press F2.	1. Point to **View** menu. 2. Press and hold a button. Drag to **Registers**, then release.

To change register values:

Keyboard Selection	Mouse Selection
No keyboard equivalent.	The only register that can be changed with the mouse is the flag's register. The register's individual bits (called flags) can be set or cleared. 1. Point to the flag to change. 2. Click either button. The highlighting of the mnemonic that represents the flag value changes.

Use the following format to enter the dialog command.

Format

```
R[registername[[=]expression]]
```

F changes a flag value as the *registername*. The REGISTER command displays the current value of each flag as a 2-letter name. You can see the two possible values for each flag in the chart below:

Flag Name	When Set	When Clear
Overflow	OV	NV
Direction	DN	UP
Interrupt	EI	DI
Sign	NG	PL
Zero	ZR	NZ
Auxiliary carry	AC	NA
Parity	PE	PO
Carry	CY	NC

After CodeView displays the flag values, it displays a dash. Enter a new value after the dash to change any flag. You can enter flag values in any order. Press Enter to record the new values. You do not change the value of any flag for which you do not type a new value. To not change any values, press Enter.

CodeView displays an error message if you enter an incorrect flag name. CodeView changes the value of any flag you specified up to the occurrence of the error. CodeView does not change the value of any flag you specify after the occurrence of the error.

Example

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

```
>R IP 0x100                    ;* Example 1
>
```

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

Chapter 11. Using System Control Commands

The sections in this chapter describe the system-control commands. The following table lists the commands and actions associated with them.

Command	Action
Help	Displays online help.
Quit	Returns to DOS.
Radix	Changes radix.
Redraw	Redraws the screen.
Screen Exchange	Switches to output screen
Search	Searches for regular expression.
Shell Escape	Starts new operating system shell.
Tab Set	Sets tab size.
Option	Views or sets CodeView options.
Redirection and related commands	Control redirection of CodeView input or output.

The QUIT command ends CodeView and returns to DOS.

Keyboard Selection	Mouse Selection
1. Press Alt+F. 2. Press X.	1. Point to **File**. 2. Press a button and hold. Drag to **Exit**; release.

Use the following format to enter the dialog command.

Format

```
Q
```

When you enter the command, the DOS (or OS/2) screen with the cursor at the DOS (or OS/2) prompt replaces the CodeView screen.

Use the following format to enter the dialog command.

Format

```
N[radixnumber]
```

The *radixnumber* can be 8 (octal), 10 (decimal), or 16 (hexadecimal). The default radix when you start CodeView is 10 (decimal) unless you write the program in assembler. The default in Macro Assembler is 16 (hexadecimal). If you give the RADIX command with no argument, CodeView displays the current radix.

Example

```
>N10
>N
10
>? prime
107
>

>N8                    ;* C example
>? prime
0153
>

>N8                    ;* BASIC example
>? prime
&153
>
```

The examples show how 107 decimal, stored in the variable prime, is displayed with different radixes. The examples are in different languages; there is no logical connection between the radix and the language.

```
>N8
>? 34,i
28
>N10
>? 28,i
28
>N16
>? 1C,i
28
>
```

The REDRAW command works in window mode by redrawing 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.

Keyboard Selection	Mouse Selection
Dialog command only.	Dialog command only.

Use the following format to enter the dialog command.

Format

```
@
```

The SEARCH command searches for a regular expression in a source file. The target expression is either an argument in a dialog command or in a dialog box. Once you find the expression, you can search for the next or the prior occurrence of the expression.

Regular expressions are a method of specifying variable text patterns. You can use a pattern to search for text strings that match the pattern. This method is similar to using global characters in filenames.

You can use the SEARCH command without understanding regular expressions. Text strings are the simplest form of regular expressions. You can enter a string of characters as the expression to find.

Note: When you search for the next occurrence of a regular expression, CodeView searches to the end of the file, then wraps and begins 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, CodeView is wrapping around and finding the same expression each time.

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 meaning.

For example, use /x*y to find x*y or use /buffer\[count] to find buffer[count]. The periods in the rational operators must also be preceded by a backslash.

The **Case Sense** selection from the **Options** menu has no effect on searches for regular expressions. For more information on regular expressions, see "Regular Expressions" in Chapter 4.

Use the following format to enter the dialog command.

Format

```
/[regularexpression]
```

If you enter a *regularexpression*, CodeView searches the source file for the next line containing the expression. If you do not enter an argument, CodeView searches for the next occurrence of the last regular expression specified.

In window mode, CodeView starts searching at the current cursor position and puts the cursor at the next line containing the regular expression. In sequential mode, CodeView 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 CodeView cannot find the expression. In assembly mode, CodeView automatically switches to source mode when it finds the expression.

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.

Example

The following example shows how to find the word *count* in the source file.

```
>/count
```

Keyboard Selection	Mouse Selection
1. Press Alt+F. 2. Press D.	1. Point to **File**. 2. Press a button and hold. Drag to **DOS Shell**; release.

Use the following format to enter the dialog command.

Format

```
![!][command]
```

To go to DOS (or OS/2) and run several programs or commands, type the command with no arguments. CodeView runs a new copy of the command processor and the operating system screen appears. You can run programs or operating system internal commands. To return to CodeView, type **exit**. The debugging screen appears with the same status as before.

To run a program or internal command from inside CodeView, enter the SHELL ESCAPE command (!) followed by the name of the command or program you want to run. The output screen appears and CodeView runs the command or program. When the output from the command or program finishes, 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.

If you specify two exclamation points before the command, CodeView will not prompt you to press a key to continue. Once your command finishes, you are immediately returned to the debugging screen.

Example

In Example 1, CodeView saves the current debugging context and runs a copy of the command processor. The DOS (or OS/2) screen appears and you can enter any number of commands. To return to CodeView, type **exit**.

```
>!                    ;* Example 1
```

The TAB SET command sets the width in spaces that CodeView 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 you wrote your source code with an editor that indents with spaces rather than with tab characters.

Keyboard Selection	Mouse Selection
Dialog command only.	Dialog command only.

Use the following format to enter the dialog command.

Format

```
#number
```

The *number* is the new number of characters for each tab character. In window mode, CodeView redraws the screen with the new tab width when you enter the command. In sequential mode, any output of source lines reflects the new tab size.

Example

In this example, the SOURCE LINE command (.) shows the source line with the default tab width of eight spaces. The TAB SET command sets the tab width to four spaces. The SOURCE LINE command then shows the same line.

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

extended registers when switching between DOS mode and OS/2 mode.

Use + to turn the specified option on. Use – to turn the specified option off. To check the current status of an option, do not specify either a + or –. Entering O by itself (without specifiying an option or + or –) shows the status of all the options.

Example

In Example 1, the **Case Sense** option is turned off. Until case sensitivity is turned back on, *buffer*, *BUFFER*, and *Buffer* can all be used when referring to the variable *buffer*.

```
>OC-                        ;* Example 1
Case Sense Off
>
```

In Example 2, CodeView displays the current settings of all the options.

```
>O                          ;* Example 2
Flip/Swap On
Bytes Coded On
Case Sense Off
386 Off
>
```

The REDIRECT INPUT command causes CodeView to read all subsequent command input from a device, such as another terminal, or a file. The sample session provided with CodeView is an example of commands redirected from a file.

Keyboard Selection	Mouse Selection
Dialog command only.	Dialog command only.

Use the following format to enter the dialog command.

Format

```
< device name
```

Example

Example 1 redirects input from the device COM1 probably a remote terminal.

```
><COM1                ;* Example 1
```

Example 2 redirects command input from file INFILE.TXT to CodeView. 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, CodeView runs the commands to the end of the text file. If you want the user to continue editing after the session, the last command in the file should be < CON. The command input is returned to the CodeView screen. One way to create such a file is to redirect commands from CodeView to a file and then edit the file to add comments and eliminate the output.

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

REDIRECT OUTPUT

In Example 2, CodeView redirects output to the file OUTFILE.TXT. Use this to get a permanent record of a CodeView session. The optional *T* echoes the session to the CodeView screen as well as to the file. After redirecting some commands to a file, CodeView returns output to the terminal with the command >CON.

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

To redirect more commands to the same file later in the session, use two greater-than symbols, as in Example 3, to add the output to the existing file.

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

Commands Used with Redirection

To redirect commands to or from a file, use the following commands. Although they are always available, these commands have little practical use during a normal debugging session.

Command	Action
COMMENT (*)	Displays comment.
DELAY (:)	Delays running of commands from a redirected file.
PAUSE (")	Interrupts running of commands from a redirected file until you press a key.

When you read the file into CodeView with the REDIRECT INPUT command, you see the comment and the output from the command as shown here:

```
><INPUT.TXT                                    ;* Example 2
>* Dump first 20 bytes of screen buffer
>D 0xB800:0000 L 20
B800:0000  54 07 4D 07 50 07 3D 07-43 07 3A 07 5C 07 43 07  T.M.P.=.C.:.\.C.
B800:0010  32 07 5C 07            -                        2.\.
:
><CON
>
```

Pause

The PAUSE command interrupts the running of commands from a redirected file. CodeView then waits for you to press a key. The redirected commands run as soon as a key is pressed.

Keyboard Selection	Mouse Selection
Dialog command only.	Dialog command only.

Use the following format to enter the dialog command.

Format

```
"
```

Example

This example is a text file that is redirected into CodeView. A COMMENT command prompts you to press a key. The PAUSE command halts the running until you respond.

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

The output looks like this when the text is redirected into CodeView:

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

The next CodeView prompt does not appear until you press a key.

Chapter 12. Additional OS/2 Mode Debugging Features

This chapter discusses the new OS/2 mode features of CodeView.

Debugging (OS/2 Mode Versus DOS Mode)

CVP.EXE operates differently than CV.EXE. The following lists the differences between the two:

- The OUTPUT SCREEN command (\) is different.
- CVP.EXE uses additional command-line options in debugging dynamic-link modules and child processes.
- CVP.EXE has new commands that are not present in CV.EXE to deal with the the multithread and multiprocess capabilities of OS/2. Also, some of the commands for tracing and running in CV.EXE work differently in CVP.EXE. The following sections describe each of these differences.

Note: In the sections that follow, the words *symbolic information* refer to information that CodeView uses to interpret global and local program symbols. Without this information, the debugger cannot provide full debugging capability.

Using CodeView's VIEW OUTPUT Command

In DOS mode, when you switch from the CodeView display to the output window (using the \ command or F4), you remain at the output window until you press a key. In OS/2 mode, CodeView returns to the CodeView display screen after a 3-second delay. You can change the delay seconds by specifying a number following the \ command.

Example

```
\60
```

Another way to view the output is to return to the Presentation Manager screen and select the screen group labeled CVP app. This is the screen group owned by the application being debugged. After viewing the output window, switch back to the CVP.EXE screen group. Use Alt+Esc to toggle between groups.

Choose one of the legal values for the *selector*:

Symbol	Function
(blank)	If you omit a value for the thread selector, the current thread is assumed. If, however, you omit both the thread selector and the thread action, CodeView responds by displaying the status of all threads.
*	Selects all threads.
n	Selects the indicated thread. The value of *n* must be a number corresponding to an existing thread. To determine corresponding numbers for all threads, enter the tilde (~) by itself.
.	Currently selected thread.
#	Last thread that was running.

Choose one of the legal values for *action*:

Symbol	Function
(blank)	The status of the selected thread or threads is displayed.
BP	A breakpoint is set for the specified thread. Because multiple threads can exist within the same program, it is possible to write the program so that a given line of code can be run by any number of different threads. However, by using this version of the BREAKPOINT SET command, you can specify that a breakpoint is taken only when a particular thread runs the specified line of code. The letters BP are followed by the normal syntax for the BREAKPOINT SET command. You can include the optional passcount field.
E	The specified thread is run in slow motion. The thread selected for this command becomes the current thread, while all other threads are temporarily frozen. ~*E is valid only in source mode. This runs the current thread in slow motion but allows all other threads (except frozen ones) to run normally.
F	The specified thread or threads are frozen. A frozen thread will not run, even when you issue a GO command, unless that thread is the current thread. UNFREEZE (U) reverses this condition.

Example 4 unfreezes all threads. Any threads that were frozen before will not run when you give the GO command. If no threads are frozen, this command has no effect.

```
004> ~*T                ;* Example 4
```

The existence of multiple threads affect other CodeView commands. The following table discusses each of these commands:

Command	Behavior in Multithread Programs
.	The CURRENT LINE command is equivalent to the ~S command.
E	The EXECUTE command runs the current thread in slow motion. To carry out this command, CodeView recalls the last line of code that ran within the current thread. (With a TRACE, EXECUTE, or BREAKPOINT command, you can run up to a specific line of thread. CodeView begins running at the next line.)
BP	BREAKPOINT SET is equivalent to the ~*BP command; the breakpoint applies to all threads.
G	GO is equivalent to the ~*G command. Control passes to the operating system, which runs all threads in the program except the frozen ones.
P	PROGRAM STEP runs a step into the current thread.
K	STACK TRACE displays the stack of the current thread.
T	TRACE runs a trace in the current thread.

Note: In general, CodeView commands apply to all threads unless the nature of the command makes it appropriate to deal with only one thread at a time. For example, since each thread has its own stack, the STACK TRACE command does not apply to all threads. In the latter case, the command applies to the current thread only. The CURRENT LINE command (.) is the sole exception to this rule. It applies to the last thread run.

```
cvp myprog /O /L mylib /COC- /C;T
```

then */O /L mylib /COC-* is passed to new CodeView sessions, but */C;T* is not.

- When you exit from the new CodeView session, you return to the original CodeView session (not the operating system).

Use the following dialog command to select a new current process or to check the status of a process.

Format

```
| [selector[action]]
```

Choose one of the legal values for *selector*:

Symbol	Function
(blank)	If you omit a value for the process selector, the current process is assumed. If, however, you omit both the selector and the action, CodeView responds by displaying the statuses of all the processes in the debugging session.
n	Selects the specified process. The value of n must be the process ID of one of the processes you are debugging. To determine the IDs of valid processes, enter the vertical bar (|) by itself.

Choose one of the legal values for *action*:

Command	Function
(blank)	The status of the selected process or processes is displayed.
S	The specified process is selected as the current process, and you are switched to the CodeView session for that process.

You can also switch between CodeView sessions using the Task Manager.

pass commands to CodeView to continue running the program or even to quit the debug session.

To debug a pop-up routine, disable the calls to VioPopUp and VioEndPopUp and then debug the code. When the VioPopUp and VioEndPopUp calls are being used, you can skip around the pop-up routine by setting a breakpoint after the VioEndPopUp and then issuing the GO command.

Programs that use VioSavRedrawWait - If your program depends upon VioSavRedrawWait for notification of screen-group switches so it can save or restore the screen, OS/2 will wait for that thread to complete its work and call VioSavRedrawWait again before continuing with the task switch. If CodeView halts this thread in the middle of its execution, then OS/2 will not receive its notification to continue.

The /2 option can be useful when debugging programs that use VioSavRedrawWait. If you use two displays, CodeView does not have to change screen groups to display both itself and your program's output. Therefore, as long as you do not use Ctrl-Esc or Alt-Esc to explicitly change screen groups, you can avoid this problem.

Appendix A. Quick Reference

This appendix summarizes the modes, options, and commands of CodeView.

Starting CodeView

The syntax for starting CodeView in DOS mode is:

CV [options] executablefile [arguments]

The syntax for starting CodeView in OS/2 mode is:

CVP [options] executablefile [arguments]

The *executablefile* is the name of the program you want to debug. The *arguments* are any arguments that you want to pass to this program.

To start CodeView, use the following startup options when running in OS/2 mode, DOS, or DOS mode:

Option	result
/B	Starts with a black-and-white display and a Color Graphics Adapter.
/C *commands*	Runs commands on startup.
/M	Disables the mouse.
/T	Starts in sequential mode.
/W	Starts in window mode.
/43	Starts in 43-row mode with an Enhanced Graphics Adapter or PS/2 display.
/50	Starts in 50-row mode with a Video Graphics Adapter or PS/2 display.
/2	Uses two video adapters.

If You Want To:	Do This:
Dump a portion of memory	Enter D (for DUMP) and the address where you want the dump to start. For example, enter: D DS:100
Examine a variable or expression	Enter ? and the variable or expression. For example, to view the sum of *sym1* and *sym2* divided by the constant *3*, enter: ? (sym1 + sym2)/3
View source code	Enter V followed by a period and the number of the first source line. For example, to view source code starting at line 36, enter: V .36

Though the next table concentrates on methods of entering commands from the keyboard, it also includes some versions of the mouse commands.

If You Want To:	Do This:
View **Help**	Press F1. Use the mouse, menu selection letters, or the Tab and Enter keys to move through the system.
Move through the source code	Place the cursor in the window containing the source code. Pressing F6 allows you to move to the correct window. Press PgUp, PgDn, Home, End, Cursor Up or Cursor Down to move through the source code.
Run code one step at a time	Press F8 (for TRACE) or F10 (for PROGRAM STEP). If you use a mouse, click TRACE on the menu bar. Use the right mouse button for the TRACE command or the left mouse button for the PROGRAM STEP command.
Run the program	Press F5 (for GO). If the program encounters a breakpoint, running stops. To use a mouse, you can click either button on GO in the menu bar.
Run to a specified line of code	Move the cursor to the line that you want to run. Pressing F6 switches the cursor to the correct window. Press F7. The reverse video line marking the current location moves to the specified line. To use the mouse, move the cursor to the specified line and click the right button.

WINDOW COMMAND SUMMARY

The following table shows the window commands that you can run with either the mouse or the function keys:

Action	Function Keys	Mouse
Open **Help** menu	Press F1	**Help** menu
Open Register window	Press F2	**View** menu
Toggle Source, Assembly, Mixed	Press F3	**View** menu
Switch to output screen	Press F4	**View** menu
GO	Press F5	Click on F5 = Go
Switch cursor window	Press F6	No function
Go to cursor line	Press F7 at location	Click right on source line
Trace through functions	Press F8	Click left on **F8 = Trace**
Set breakpoint at cursor	Press F9 at location	Click left on source code
Step over functions	Press F10	Click right on **F8 = Trace**
Change flag	No function	Click on flag
Make window larger (Grow)	Press Ctrl+G	Drag bar up or down
Make window smaller (Tiny)	Press Ctrl+T	Drag bar up or down
Move up a line in window	Move cursor off top	Click left on up arrow
Page up in window	Press PgUp	Click above elevator
Move to top of window	Press Home	Move elevator to top

Command	Format
REGISTER	R [*reg*] [[=]*expression*]
8087 DUMP	7
PORT INPUT	I *port*
SEARCH MEMORY	S *range list*
COMPARE MEMORY	C *range address*

Breakpoints

Command	Format
BREAKPOINT SET	BP [*address*] [*count*] ["*command*"]
BREAKPOINT CLEAR	BC [*list* | *]
BREAKPOINT DISABLE	BD [*list* | *]
BREAKPOINT ENABLE	BE [*list* | *]
BREAKPOINT LIST	BL

System Control

Command	Format
HELP	H
OPTION	O[*F*|*B*|*C*|*3* *[+|-]*]
QUIT	Q
RADIX	N[*radix*]
REDRAW	@
SCREEN EXCHANGE	\ [*seconds*]
SEARCH	/[*regularexpression*]
TAB SET	#[*number*]

Redirection

Command	Format
REDIRECT INPUT	<device
REDIRECT OUTPUT	[*T*]>[>] device
REDIRECT BOTH	= device
PAUSE	"
DELAY	:
COMMENT	*[*comment*]

Appendix B. Error Messages

CodeView displays an error message whenever it detects a command it cannot run. Except for startup errors, most errors stop the CodeView command in which the error occurred, but do not stop CodeView.

Already have base register	You supplied more than one base register for an operand.
Already have index register	You supplied more than one index register for an operand.
Argument list or environment too large	You issued a SHELL ESCAPE command but there was not enough space for either the number of parameters or the size of the environment you tried to pass.
Bad address	You specified the address in an invalid form. For example, you might 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 of 0 through 19.
Bad emulator info	If this message occurs, note the circumstances of the error and report it to an IBM Authorized Dealer.
Bad flag	You specified an invalid flag mnemonic with the REGISTER dialog command (R). Use one of the mnemonics that appears when you enter the command RF.

Breakpoint # or "*" expected	You entered the BREAKPOINT CLEAR (BC), BREAKPOINT DISABLE (BD), or BREAKPOINT ENABLE (BE) commands with no argument. These commands require that you specify the number of the breakpoint at which CodeView is to act or that you specify an asterisk, indicating that CodeView is to act on all breakpoints.
Byte register is illegal	The instruction you are assembling cannot accept a byte register. For example, PUSH AL is illegal.
Cannot execute *filename*	CodeView could not execute the program specified in your SHELL ESCAPE command.
Cannot find *filename* **Please enter new program spec:**	The DOS overlay manager could not find the file that contains the overlay it needs to load. Type the name of the overlay file (including the path, if necessary) and press Enter.
Cannot load overlay: too many open files	The DOS overlay manager was unable to open the overlay file it needed. Try increasing the number in the FILES command in your CONFIG.SYS file before debugging your program again. (See your IBM Disk Operating System manual for more information on the FILES command.)
Cannot use struct or union as scalar	You cannot use a structure or union variable as a scalar value in a C expression. The address-of operator must precede structure or union variables, and a field specifier must follow them.
Can't find *filename*	CodeView cannot find the executable file you specified when you started. You probably misspelled the filename, or the file is in a different directory.

EMM memory not found	You specified the /E option when starting CodeView, but you do not have expanded memory or the expanded memory device driver is not installed. Install the device driver or start CodeView without the /E option.
EMM software error	The expanded memory device driver has returned an unexpected error code.
Enter directory for *filename* (cr for none)?	CodeView cannot find the source file for your program. Enter the name of the directory where the source file is located, including the final backslash (for example: C:\PROJECT\SOURCE\). To continue without using the source file, press Enter without entering a directory name.
Exec format error	CodeView was unable to start a secondary copy of itself. Under OS/2, Version 1.00, you can run only one copy of CodeView at a time so no secondary copies can be started. Under OS/2, Version 1.10, this message means that the call to DosStartSession to start the new copy of CodeView failed.
Expression: internal error	If this message occurs, note the circumstances of the error and report it to an IBM Authorized Dealer.
Expression not a memory address	You specified an expression that does not evaluate to an lvalue. For example, TP? 'A' is invalid.
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. CodeView evaluates the valid arguments and ignores the rest. Often in this situation, CodeView does not evaluate the arguments in the order that you intended.

Incorrect DOS version	An incompatibility problem occurred while the DOS overlay manager was performing a task.
Insufficient EMM memory	There was not enough expanded memory available to hold all the symbolic information for the program you want to debug. Try starting CodeView without the /E option.
Internal debugger error	If this message occurs, note the circumstances of the error and report it to an IBM Authorized Dealer.
Invalid argument	One of the arguments you specified is not a valid CodeView expression.
Invalid executable file format - please relink	The executable file was not linked with the version of the linker released with this version of CodeView. Relink with the more current version of the linker.
Invalid option	The option specified cannot be used with the CodeView OPTION command.
Invalid process ID	You issued a PROCESS command for an invalid process. For a list of the valid processes at any given time, enter | at the CodeView command prompt.
Invalid thread ID	You issued a THREAD command for an invalid thread. For a list of the valid threads at any given time, enter ~* at the CodeView command prompt.
I/O error	If this message occurs, note the circumstances of the error and report it to an IBM Authorized Dealer.
Library module not loaded	CodeView cannot access one of the dynamic link libraries (DLLs) that it requires. Either the DLL was never loaded, or it was prematurely terminated.
Missing '"'	You specified a string as an argument to a dialog command, but you did not supply a closing double quotation mark.

No previous regular expression	You selected **Previous** from the **Search** menu, but there was no previous match for the last regular expression specified.
No source lines at this address	The address you specified as an argument for the VIEW command (V) does not have any source lines. It might 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, this message appears when you select **Open** 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. CodeView determines if a file is a text file by checking the first 128 bytes for characters that are not in the ASCII range of 9 through 13 and 20 through 126.
Not an executable file	The file you specified for debugging when you started CodeView is not an executable file having the extension .EXE or .COM.

Operator must have a struct/union type	You used the one of the C member selection operators (-> or .) in an expression that does not refer to 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.
Outdated EMM software (3.0 required)	To use expanded memory with CodeView, you must have Version 3.00 or higher of the expanded memory device driver. If you do not have Version 3.00, start CodeView without the /E option.
Out of memory	CodeView cannot allocate enough memory to perform your request.
Overlay Manager stack overflow	A stack overflow occurred while the DOS overlay manager was performing a task.
Overlay not found	The DOS overlay manager was unable to find the overlay it needed.
Overlay not resident	CodeView was unable to access one of its overlays.
PID *n* segmentation violation	A child process you chose not to debug (process n) has caused a general protection fault.
PID *n* zombied	A child process you chose not to debug (process n) has executed an INT 3 instruction. The process will not be able to continue.
Please insert diskette containing *filename* into drive *x*: and strike any key when ready	The DOS overlay manager requires an overlay file that is not in drive *x*. Replace the diskette in drive *x* with the diskette that contains the file indicated, and press a key to continue.
Please restore original diskette. Strike any key when ready.	The DOS overlay manager has finished reading the overlay file. Reinsert the diskette that you removed when you supplied the overlay file, and press a key to continue.

Subscript not an integer	In BASIC, array subscripts must be integer values.
Subscript out of bound	The array subscript value you specified indicates an element that does not exist for the array. For example, given the definition DIM X%(12), the expression X%(13) is illegal.
Symbol not defined	While assembling code, you specified a symbol that is not defined. If you are attempting to use an IBM Macro Assembler or IBM Pascal Compiler/2 program symbol, make sure that the symbol was declared as a PUBLIC symbol in your program.
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 follows a caret that points to the first character that CodeView cannot interpret.
Thread terminated normally (*number*)	The current thread ran to the end. The number displayed in parenthesis is the exit code that the thread returned.
Too few array bounds given	The bounds you specified at an array subscript do not match the array declaration. For example, given the array declaration DIM IARRAY%(3, 4), the expression IARRAY(I%) would produce this message.
Too many array bounds given	The bounds you specified in an array subscript do not match the array declaration. For example, given the array declaration DIM IARRAY%(3, 4), the expression IARRAY%(I%, J%, K%J) would produce this message.
Too many breakpoints	You tried to specify a 21st breakpoint. Codeview permits only 20 breakpoints.

Usage: cv [options] file [arguments]	You failed to specify an executable file when you started CodeView. Try again with the syntax shown in the message.
Value is out of range	The specified value is too large for its expected use. For example, you cannot move a double word to a byte register.
Variable ambiguous, use type specifier	The symbol name you are using could refer to more than one program symbol. For example, you entered ? B when both B% and B$ exist in your program. Add one of the BASIC type specifiers (%, &, !, #, or $) to the end of the symbol name to indicate which symbol you mean to use.
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 you are debugging graphics programs. You can continue debugging when you get this message, but the output screen of the debugged program might be damaged.
VioGetBuf failing	CodeView was denied access to the logical screen buffer by the operating system.
VioGetPhysBuf failing	CodeView was denied access to a physical screen buffer by the operating system.
Warning: packed file	You started CodeView 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 might make this difficult. You cannot debug in source mode because EXEPACK strips all symbolic information from a file when it packs the file. This occurs with the /EXEPACK linker option.
Wrong type of register	The instruction you are assembling requires a type of register different from the one you supplied.

Index

E

P

T

Y

Z

Numerics

Special Characters

Continued from inside front cover.

SUCH WARRANTIES ARE IN LIEU OF ALL OTHER WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Some states do not allow the exclusion of implied warranties, so the above exclusion may not apply to you.

LIMITATION OF REMEDIES

IBM's entire liability and your exclusive remedy shall be as follows:

1) IBM will provide the warranty described in IBM's Statement of Limited Warranty. If IBM does not replace defective media or, if applicable, make the Program operate as warranted or replace the Program with a functionally equivalent Program, all as warranted, you may terminate your license and your money will be refunded upon the return of all of your copies of the Program.
2) For any claim arising out of IBM's limited warranty, or for any other claim whatsoever related to the subject matter of this Agreement, IBM's liability for actual damages, regardless of the form of action, shall be limited to the greater of $5,000 or the money paid to IBM, its Authorized Dealer or its approved supplier for the license for the Program that caused the damages or that is the subject matter of, or is directly related to, the cause of action. This limitation will not apply to claims for personal injury or damages to real or tangible personal property caused by IBM's negligence.
3) In no event will IBM be liable for any lost profits, lost savings, or any incidental damages or other consequential damages, even if IBM, its Authorized Dealer or its approved supplier has been advised of the possibility of such damages, or for any claim by you based on a third party claim.

Some states do not allow the limitation or exclusion of incidental or consequential damages so the above limitation or exclusion may not apply to you.

GENERAL

You may terminate your license at any time by destroying all your copies of the Program or as otherwise described in this Agreement.

IBM may terminate your license if you fail to comply with the terms and conditions of this Agreement. Upon such termination, you agree to destroy all your copies of the Program.

Any attempt to sublicense, rent, lease or assign, or, except as expressly provided herein, to transfer any copy of the Program is void.

You agree that you are responsible for payment of any taxes, including personal property taxes, resulting from this Agreement.

No action, regardless of form, arising out of this Agreement may be brought by either party more than two years after the cause of action has arisen except for breach of the provisions in the Section entitled "License" in which event four years shall apply.

This Agreement will be construed under the Uniform Commercial Code of the State of New York.