SINIX Open Desktop Development System CodeView Debugger

Edition April 1990 (SINIX Open Desktop V1.0)

Order No. U5760-J-Z95-1-7600
Printed in the Federal Republic of Germany
4460 AG 2900.5 (5580)

Published by Bereich
Daten- und Informationstechnik
Postfach 83 09 51, D-8000 München 83

Siemens Aktiengesellschaft

SCO Document Number: 6-26-89-6.0/3.2.0

Contents

Chapter 1

Introduction

Introduction

Welcome to the CodeView® debugger. This is an executable program that helps you debug software written with the C and Macro Assembler languages.

The CodeView debugger is a window-oriented tool that enables you to track down logical errors in programs; it allows you to analyze a program as the program is actually running. The CodeView debugger displays source code or assembly code, indicates which line is about to be executed, dynamically watches the values of variables (local or global), switches screens to display program output, and performs many other related functions. The debugger can be easily learned and used, by assembly and high-level language programmers alike.

To use CodeView, you first create an executable file from compiled object files. (When a program is made into an executable file, it is in the form that can be loaded and executed by the system.) This executable file must be compiled and linked with the correct options so that it contains the line-number information and a symbol table needed by CodeView. You can use the C compiler, or **cc**, which calls the linking program, **ld**, The correct options for compiling and linking for use with CodeView are described in Chapter 2, ''Getting Started.''

About this Manual

This manual explains the use of the CodeView debugger. Commands, display, and interface of the debugger are presented here.

The manual is comprised of the following chapters:

- Chapter 2, ''Getting Started,'' explains how to create a C or assembly program that can be run with the CodeView debugger; it also explains how to start the debugger and select various command-line options.

- Chapter 3, ''The CodeView Display,'' discusses the CodeView display screen and interface, including function keys and keyboard commands.

- Chapter 4, ''Using Dialog Commands,'' presents the general form of CodeView commands.

- Chapter 5, ''CodeView Expressions,'' describes how to build complex expressions for use in commands.

- Chapter 6, ''Executing Code,'' explains the CodeView commands that execute code from within a program.

- Chapter 7, ''Examining Data and Expressions,'' discusses several data-evaluation commands.

- Chapter 8, ''Managing Breakpoints,'' explains how to use breakpoints to suspend execution.

- Chapter 9, ''Managing Watch Statements,'' describes the use of watch statement commands to set, delete, and list watch statements.

- Chapter 10, ''Examining Code,'' discusses several commands that let you examine program code or data related to code.

- Chapter 11, ''Modifying Code or Data,'' explains how to alter code temporarily for testing in the CodeView debugger.

- Chapter 12, ''Using CodeView System-Control Commands,'' discusses commands that control the operation of the CodeView debugger.

Chapter 2

Getting Started

Introduction

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

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

Restrictions

You cannot use the CodeView debugger to debug source code in include files. This restriction applies generally to the use of the CodeView debugger, regardless of the language being used.

Preparing Programs for the CodeView Debugger

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

Note

For the sake of brevity, this section and its three subsections use the term ``compiling'' to refer to the process of producing object modules. However, almost everything said about compiling in this section applies equally well to assembling. Exceptions are noted in the section ``Preparing Assembly Programs'' in this chapter.

Not all compiler and linker versions support CodeView options. Consult the specific language documentation for information about compiler versions. If you try to debug an executable file that was not compiled and linked with CodeView options, or if you use a compiler that does not support these options, then you are only able to use the debugger in assembly mode. This means that the CodeView debugger does not display source code or understand source-level symbols, such as symbols for functions and variables.

The two CodeView basic display modes are source mode, in which the program is displayed as source lines, and assembly mode, in which the program is displayed as assembly-language instructions. These two modes can be combined in mixed mode, in which the program is displayed with both source lines and assembly-language instructions.

Programming Considerations

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

The C and Macro Assembly languages permit you to put code in separate include files, and to read the files into your source file by using an include directive. However, you cannot use the CodeView debugger to debug source code in include files. The preferred method of developing programs is to create separate object modules, and then link the object modules with your program. The CodeView debugger supports the debugging of separate object modules in the same session.

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

The C and Macro Assembly languages support a type of macro expansion. However, the CodeView debugger does not help you debug macros in source mode. You need to expand the macros yourself before debugging them; otherwise, the debugger treats them as simple statements or instructions.

CodeView Compile Options

When you compile a source file for a program you want to debug, you must specify the **-Zi** option on the command line. The **-Zi** option instructs the compiler to include line-number and symbolic information in the object file. You can also use **-g**, which is synonymous with **-Zi**.

If you do not need complete symbolic information in some modules, you can compile those modules with the **-Zd** option instead of **-Zi**. The **-Zd** option writes less symbolic information to the object file, so using this option saves 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 are able to examine global variables and see source lines in modules compiled with the **-Zd** option, but local variables are unavailable.

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

Note

The **-Od** option has no effect when used with the Macro Assembler.

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

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

CodeView Link Options

If you use **ld** separately to link an object file or files for debugging, you should specify the **-g** option. This option instructs the linker to incorporate addresses for symbols and source lines into the executable file.

Note that if you use a driver program that automatically invokes the linker (such as **cc** with C), then the linker is automatically invoked with the **-g** option whenever you specify **-Zi** on the command line.

Although executable files prepared with the **-g** option can be executed from the command line like any other executable files, they are larger because of the extra symbolic information in them. To minimize program size, you may want to use the **strip** command or recompile and link your final version without the **-Zi** option when you finish debugging a program. See the *Programmer's Reference* for information about the **strip** command.

Linking examples are given in the sections below on compiling and linking C and assembly language programs.

Preparing C Programs

In order to use the CodeView debugger with a program written in C, you need to compile it with the C Compiler. Early versions of the compiler do not support the CodeView compile options. Please see the *Development System Release Notes* for more information.

Writing C Source Code

The C language supports the use of include files, through the use of the **#include** directive. However, you cannot debug source code put into include files. Therefore, you should reserve the use of include files for **#define** macros and structure definitions.

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

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

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

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

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

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

Compiling and Linking C Programs

The **-Zi**, **-Zd**, and **-Od** options are all supported by the C Compiler. (For a description of these options, see the section ''CodeView Compile Options.'') The options are accepted by the **cc** driver.

The CodeView debugger supports mixed-language programming. For an example of how to link a C module with modules from other languages, see the section ''Preparing Assembly Programs'' in this chapter.

Examples

```
cc -Zi -Od -o example example.c

cc -c -Zi -Od example.c
cc -g -o example example.o

cc -Zi -Od -c mod1.c
cc -Zd -Od -c mod2.c
cc -Zi mod1.o mod2.o
```

In the first example, **cc** is used to compile and link the source file **example.c** The **cc** command creates an object file in the CodeView format, **example.o**, and then automatically invokes the linker with the **-g** option. The second example demonstrates how to compile and link the source file, **example.c**, by using the **-c** option with **cc**. Since **cc -c** does not invoke the linker, you must enter **cc** a second time to link the object file. These examples result in an executable file, **example**, which has the line-number information, symbol table, and unoptimized code required by the CodeView debugger.

In the third example, the source module **mod1.c** is compiled to produce an object file with full symbolic and line information, while **mod2.c** is compiled to produce an object file with limited information. Then, **cc** is used again to link the resulting object files. (This time, **cc** does not recompile, because the arguments have a **.o** extension.) Typing **-Zi** on the command line causes the linker to be invoked with the **-g** option. The result is an executable file, called **a.out**, in which one of the modules, **mod2.c** is harder to debug. It contains less symbolic information, such as the names of local variables. However, the executable file takes up substantially less space on disk than it would if both modules were compiled with full symbolic information.

Preparing Assembly Programs

In order to use all the features of the CodeView debugger with assembly programs, you need to assemble with Macro Assembler. (The section "Working with Older Versions of the Assembler" in this chapter discusses how to use earlier versions the Macro Assembler with the debugger.)

Writing Assembler Source Code

If you have Version 2.3 or later of the Macro Assembler, then you can use the simplified segment directives. Use of these directives ensures that segments are declared in the correct way for use with the CodeView debugger. (These directives also aid mixed-language programming.) If you do not use these directives, then you need to make sure that the class name for the code segment is **CODE**.

You cannot trace through macros while in source mode. Macros are treated as single instructions unless you are in assembly or mixed mode, so you do not see comments or directives within macros. Therefore, you may want to debug code before putting it into a macro.

The Macro Assembler also supports include files, but you cannot debug code in an include file. You are better off reserving include files for macro and structure definitions.

Because the assembler does not have its own expression evaluator, you have to use the the C-expression, evaluator. C is the closest to assembly language. To make sure that the expression evaluator recognizes your symbols and labels, you should observe the following guidelines when you write assembly modules:

- The assembler has no explicit way to declare real numbers. However, it passes the correct symbolic information for reals and integers if you initialize each real number with a decimal point and each integer without a decimal point. (The default type is integer.) For example, the following statements correctly initialize *REALSUM* as a real number and *COUNTER* as an integer:

```
REALSUM     DD     0.0
COUNTER     DD     0
```

You must initialize real number data in data definitions. If you use *?*, then the assembler considers the variable an integer when it generates symbolic information. The CodeView debugger, in turn, does not properly evaluate the value of the variable.

- Avoid the use of special characters in symbol names.

- Assemble with **-Mx** or **-Ml** to avoid conflicts due to case when you do mixed-language programming. By default, the assembler converts all symbols to uppercase when it generates object code. C, however, does not do this conversion. Therefore, the CodeView debugger does not recognize that *var* in a C program and *var* in an assembly program are the same variable, unless you leave Case Sense off when using the debugger.

Assembling and Linking

The assembler supports the **-Zi** and **-Zd** assemble-time options. The **-Od** option does not apply, and so is not supported.

If you link your assembly program with a module written in C (which is case sensitive), you probably need to assemble with **-Mx** or **-Ml**.

After assembling, link with the **-g** option to produce an executable file in the CodeView format.

Examples

```
masm -Zi example.asm
cc -g example.o

masm -Zi mod1.asm
masm -Zd mod2.asm
cc -g mod1.o mod2.o
```

The first example assembles the source file **example.asm** and produces the object file **example.o** which is in the CodeView format. The linker is then invoked by entering **cc** with the **-g** option and produces an executable file, called **a.out**, containing the symbol table and line-number information required by the debugger.

The second example produces the object file **mod1.o** which contains symbol and line-number information, and the object file **mod2.o** which contains line-number information but no symbol table. The object files are then linked. The result is an executable file, called **a.out**, in which the second module is harder to debug. The second module contains less symbolic information, such as the names of local variables. This executable file, however, is smaller than it would be if both modules were assembled with the **-Zi** option.

Starting the CodeView Debugger

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

File	**Location**
/usr/bin/cv	The CodeView program file is located in the */usr/bin* directory.
/usr/lib/cv.hlp	The CodeView help file is located in the directory */usr/lib*. If the CodeView debugger cannot find the help file, you can still use the debugger, but you see an error message if you use one of the help commands.
program	The executable file for the program that you wish to debug must be in the current directory or in a directory that you specify by including its pathname when you type the CodeView command line. The CodeView debugger displays an error message and does not start unless the executable file is found.
source.ext (extension depends on language)	Normally, source files should be in the current directory. However, if you specify a file specification for the source file during compilation, that specification becomes part of the symbolic information stored in the executable file. For example, if you compiled with the command line argument *demo.ext*, the CodeView debugger expects the source file to be in the current directory. However, if you compiled with the command line argument with the pathname */source/demo.ext*, then the debugger expects the source file to be in directory */source*. If the debugger cannot find the source file in the directory specified in the executable file (usually the current directory), the program prompts you for a

new directory. You can either enter a new directory, or you can press the <Return> key to indicate that you do not want a source file to be used for this module. If no source file is specified, you must debug in assembly mode.

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

cv [*options*] *executablefile* [*arguments*]

The *options* are one or more of the options described in the section "Using CodeView Options" in this chapter. The *executablefile* is the name of an executable file to be loaded by the debugger. If you try to load a nonexecutable file, the following message appears:

```
Not an executable file
```

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

If the file is not in the CodeView format, the debugger starts in assembly mode and displays the following message:

```
No symbolic information
```

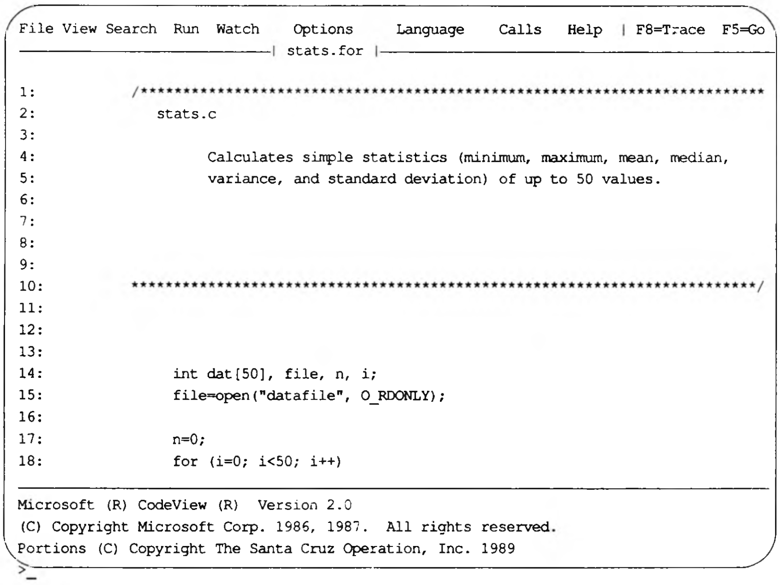
You must specify an executable file when you start the CodeView debugger. If you omit the executable file, the debugger displays a message showing the correct command-line format.

When you give the debugger a valid command line, the executable program and the source file are loaded, the address data are processed, and the CodeView display appears. The initial display is in window mode.

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

```
cv benchmrk
```

If you give this command line, window mode is selected automatically. The display looks like the following screen example:


File View Search Run Watch Options Language Calls Help | F8=Trace F5=Go
stats.for

```
          /**************************************************************************
              stats.c

                    Calculates simple statistics (minimum, maximum, mean, median,
                    variance, and standard deviation) of up to 50 values.




          **************************************************************************/




              int dat[50], file, n, i;
              file=open("datafile", O_RDONLY);

              n=0;
              for (i=0; i<50; i++)

Microsoft (R) CodeView (R)  Version 2.0
(C) Copyright Microsoft Corp. 1986, 1987.  All rights reserved.
Portions (C) Copyright The Santa Cruz Operation, Inc. 1989
>_
```

If sequential mode is selected, the following lines appear:

```
Microsoft (R) CodeView (R)  Version 2.0
(C) Copyright Microsoft Corp. 1986, 1987. All rights reserved.
Portions (C) Copyright The Santa Cruz Operation, Inc. 1989

>
```

You can use CodeView options, as described in the section ''Using Code-View Options'' in this chapter, to override the default start-up mode.

If your program is written in a high-level language, the CodeView debugger is now at the beginning of the start-up code that precedes your program. In source mode, you can enter an execution command (such as Trace or Program Step) to execute automatically through the start-up code to the beginning of your program. At this point, you are ready to start debugging your program, as described in Chapters 4-12.

Using CodeView Options

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

An option is a sequence of characters preceded by a dash (-). Unlike compiler command-line options, CodeView command-line options are not case sensitive.

A file whose name begins with a dash must be renamed before you use it with the CodeView debugger, so that the debugger does not interpret the dash as an option designator. You can use more than one option in a command line, but each option must have its own dash, and spaces must separate each option from other elements of the line. The following list suggests some situations in which you might want to use an option. If more than one condition applies, you can use more than one option (in any order). If none of the conditions applies, you need not use any options.

Condition	Option
You have a two-color monitor, a color graphics adapter, and an IBM or IBM-compatible computer.	**-B**
You want the CodeView debugger to automatically execute a series of commands when it starts up.	**-C***commands*
You wish to debug in sequential mode (for example, with redirection).	**-T**

The CodeView options are described in more detail in the following sections.

Starting with a Black-and-White Display

Option

-B

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

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

Example

```
cv -B calc calc.dat
```

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

Specifying Start-Up Commands

Option

-C*commands*

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

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

Note

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

Examples

```
cv -CGmain calc calc.dat
```

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

```
cv "-C;S&;G INTEGRAL;DS ARRAYX L 20" calc calc.dat
```

The example above loads the same file with the same argument as the first example, but the command list is more extensive. The debugger starts in mixed source/assembly mode (**S&**). It executes to the routine **INTEGRAL** (**G INTEGRAL**), and then dumps 20 short real numbers, starting at the address of the variable **ARRAYX** (**DS ARRAYX L 20**). Since several of the commands use spaces, the entire option is enclosed in double quotation marks.

```
cv "-C<input.fil" calc calc.dat
```

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

Enabling Sequential Mode

Options

-T

The CodeView debugger can operate in window mode or in sequential mode. Window mode displays up to four windows, enabling you to see different aspects of the debugging-session program simultaneously.
M003 You can also use a mouse in window mode. Window mode

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

The behavior of each mode is discussed in detail in Chapter 3, ''The CodeView Display.''

Note

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

Examples

```
cv -T sieve
```

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

Working with Older Versions of the Assembler

You can run the CodeView debugger with files developed using prior versions of the Macro Assembler. Since older versions do not write line numbers to object files, some of the CodeView debugger's features are unavailable when you debug programs developed with the older assemblers. The following considerations apply, in addition to the considerations mentioned in the section ''Preparing Assembly Programs'' in this chapter.

The procedure for assembling and debugging executable files by using older versions of the assembler is summarized below.

1. In your source file, declare public any symbols, such as labels and variables, that you want to reference in the debugger. If the file is small, you may want to declare all symbols public.

2. As mentioned earlier, make sure that the code segment has class name *CODE*.

3. Assemble as usual. No special options are required, and all assembly options are allowed.

4. Use **ld**. Refer to the *Development System Release Notes* for information about which version of **ld** to use. Use the **-g** option when linking.

5. Debug in assembly mode (this is the start-up default if the debugger fails to find line-number information). You cannot use source mode for debugging, but you can load the source file into the display window and view it in source mode. Any labels or variables that you declared public in the source file can be displayed and referenced by name instead of by address. However, they cannot be used in expressions because type information is not written to the object file.

Chapter 3

The CodeView Display

Introduction

The CodeView screen display can appear in two different modes—window and sequential. Either mode provides a useful debugging environment, but the window mode is the more powerful and convenient of the two. The CodeView debugger accepts either window commands or dialog commands. Dialog commands are entered as command lines following the CodeView prompt (>) in sequential mode. They are discussed in Chapter 4, ''Using Dialog Commands.''

You probably want to use window mode. In window mode, the pull-down menus and function keys offer fast access to the most common commands. Different aspects of the program and debugging environment can be seen in different windows simultaneously. Window mode is described in the section ''Using Window Mode'' in this chapter.

Sequential mode is sometimes useful when redirecting command input or output. Sequential mode is described in the section ''Using Sequential Mode'' in this chapter.

Using Window Mode

The elements of the CodeView display marked in the figure on the next page include the following:

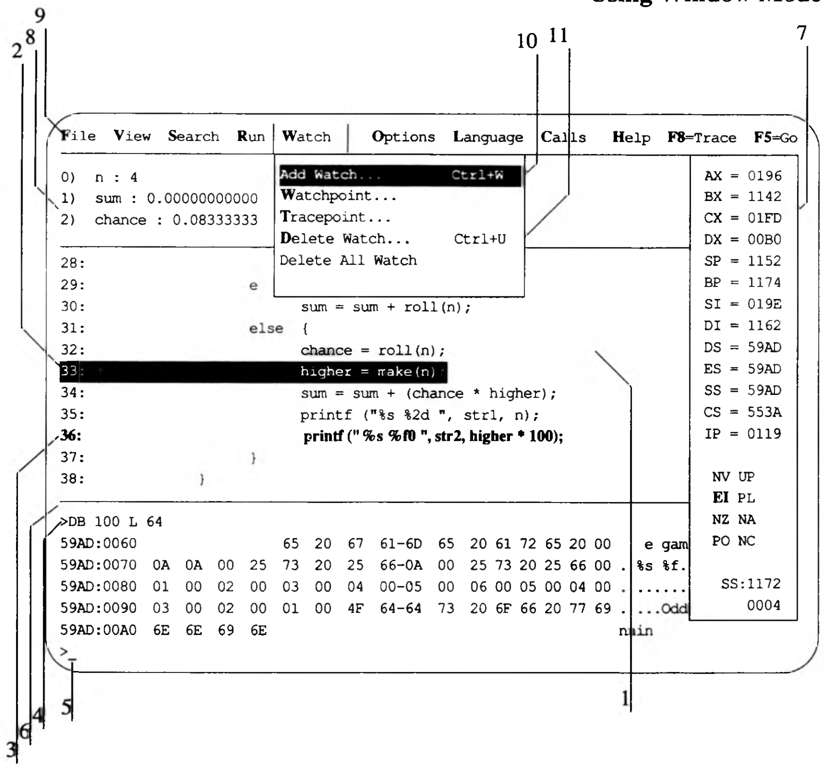
1. The display window shows the program being debugged. It can contain source code (as in the example), assembly-language instructions, or any specified text file.

2. The current location line (the next line the program will execute) is displayed in reverse video or in a different color. This line may not always be visible, because you can scroll to earlier or later parts of the program.

3. Lines containing previously set breakpoints are shown in high-intensity text.

4. The dialog window is where you enter dialog commands. These are the commands with optional arguments that you can enter at the CodeView prompt (>). You can scroll up or down in this window to view previous dialog commands and command output.

5. The cursor is a thin, blinking line that shows the location at which you can enter commands from the keyboard. You can move the cursor up and down, and place it in either the dialog or display window.



```
File  View  Search  Run  Watch  Options  Language  Calls  Help  F8=Trace  F5=Go

0)  n : 4                      Add Watch...        Ctrl+W      AX = 0196
1)  sum : 0.00000000000        Watchpoint...                   BX = 1142
2)  chance : 0.08333333        Tracepoint...                   CX = 01FD
                               Delete Watch...     Ctrl+U      DX = 00B0
                               Delete All Watch                SP = 1152
                 e                                             BP = 1174
                           sum = sum + roll(n);                SI = 019E
                 else      {                                   DI = 1162
                           chance = roll(n);                   DS = 59AD
                           higher = make(n)                    ES = 59AD
                           sum = sum + (chance * higher);      SS = 59AD
                           printf ("%s %2d ", str1, n);        CS = 553A
                           printf (" %s %f0 ", str2, higher * 100);   IP = 0119
                 }
             }                                                 NV UP
                                                               EI PL
>DB 100 L 64                                                   NZ NA
59AD:0060             65 20 67 61-6D 65 20 61 72 65 20 00   e gam  PO NC
59AD:0070 0A 0A 00 25 73 20 25 66-0A 00 25 73 20 25 66 00 .%s %f.
59AD:0080 01 00 02 00 03 00 04 00-05 00 06 00 05 00 04 00 .......  SS:1172
59AD:0090 03 00 02 00 01 00 4F 64-64 73 20 6F 66 20 77 69 ....Odd   0004
59AD:00A0 6E 6E 69 6E                                     nnin
>_
```

6. The display/dialog separator line divides the dialog window from the display window.

7. The register window shows the current status of processor registers and flags. This is an optional window that can be opened or closed with one keystroke. The register window also displays the effective address at the bottom of the window; the effective address shows the actual location of an operand in physical memory. It is useful when debugging in assembly mode.

8. The optional watch window shows the current status of specified variables or expressions. It appears whenever you create watch statements.

9. The menu bar shows titles of menus and commands that you can activate with the keyboard. Trace and Go represent commands; the other titles are all menus.

10. Menus can be opened by specifying the appropriate title on the menu bar. On the sample screen, the Watch menu has been opened.

11. The menu "highlight" is a reverse-video or colored strip indicating the current selection in a menu. You can move the highlight up or down to change the current selection.

12. The scroll bar (not shown) is the vertical bar on the right side of the screen. This bar graphically represents the ratio of read to unread portions as you scroll through the file.

13. Dialog boxes (not shown) appear in the center of the screen when you choose a menu selection that requires a response. The box prompts you for a response and disappears when you enter your answer.

14. Message boxes (not shown) appear in the center of the screen to display errors or other messages.

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

Executing Window Commands

The most common CodeView debugging commands, and all the commands for managing the CodeView display, are available with window commands. Window commands are one-keystroke commands that can be entered with function keys, <CTL> key combinations, <ALT> key combinations, or the direction keys on the numeric keypad. The window commands available from the keyboard are described by category in the following sections.

Moving the Cursor with Keyboard Commands

The following keys move the cursor or scroll text in the display or dialog window.

Key	**Function**
F6	Moves the cursor between the display and dialog windows. If the cursor is in the dialog window when you press **F6**, it moves to its previous position in the display window. If the cursor is in the display window, it moves to its previous position in the dialog window.

<CTL>g	Makes the size of the dialog or display window grow. This works for whichever window the cursor is in. If the cursor is in the display window, then the display/dialog separator line moves down one line. If the cursor is in the dialog window, then the separator line moves up one line.
<CTL>t	Makes the size of the dialog or display window smaller. This works for whichever window the cursor is in. If the cursor is in the display window, then the display/dialog separator line moves up one line. If the cursor is in the dialog window, then the separator line moves down one line.
UP ARROW	Moves the cursor up one line in either the display or dialog window.
DOWN ARROW	Moves the cursor down one line in either the display or dialog window.
<PgUp>	Scrolls up one page. If the cursor is in the display window, the source lines or assembly-language instructions scroll up. If the cursor is in the dialog window, the buffer of commands entered during the session scrolls up. The cursor remains at its current position in the window. The length of a page is the current number of lines in the window.
<PgDn>	Scrolls down one page. If the cursor is in the display window, the source lines or assembly-language instructions scroll down. If the cursor is in the dialog window, the buffer of commands entered during the session scrolls down. The cursor remains at its current position in the window. The length of a page is the current number of lines in the window.
<HOME>	Scrolls to the top of the file or command buffer. If the cursor is in the display window, the text scrolls to the start of the source file or program

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

Changing the Screen

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

Key	**Function**
F1	Displays initial on-line help screen. The help system is discussed in the section ''Using the Help System.'' You can also take advantage of the help system by using the Help menu, as mentioned in the section ''Using Menu Selections'' in this chapter.
F2	Toggles the register window. The window disappears if present, or appears if absent. You can also toggle the register window with the Register selection from the View menu, as described in ''Using Menu Selections.''
F3	Switches between source, mixed, and assembly modes. Source mode shows source code in the display window, whereas assembly mode shows assembly-language instructions. Mixed mode shows both. You can also change modes with the Source, Mixed, and Assembly selections from the View menu, as described in ''Using Menu Selections.''

F4	Switches to the output screen. The output screen shows the output, if any, from your program. Press any key to return to the CodeView screen.

Controlling Program Execution

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

Key	Function
F5	Executes to the next breakpoint or to the end of the program if no breakpoint is encountered. This keyboard command corresponds to the Go dialog command when it is given without a destination breakpoint argument.
F7	Sets a temporary breakpoint on the line with the cursor, and executes to that line (or to a previously set breakpoint or the end of the program if either is encountered before the temporary breakpoint). In source mode, if the line does not correspond to code (for example, data declaration or comment lines), the CodeView debugger sounds a warning and ignores the command. This window command corresponds to the Go dialog command when it is given with a destination breakpoint.
F8	Executes a Trace command. The CodeView debugger executes the next source line in source mode or the next instruction in assembly mode. If the source line or instruction contains a call to a routine or interrupt, the debugger starts tracing through the call (enters the call and is ready to execute the first source line or instruction). This command will not trace into function calls.
F9	Sets or clears a breakpoint on the line with the cursor. If the line does not currently have a breakpoint, one is set on that line. If the line already has a breakpoint, the breakpoint is cleared. If the cursor is in the dialog

window, the CodeView debugger sounds a warning and ignores the command. This window command corresponds to the Breakpoint Set and Breakpoint Clear dialog commands.

F10 Executes the Program Step command.

The CodeView debugger executes the next source line in source mode, or the next instruction in assembly mode. If the source line or instruction contains a call to a routine or interrupt, the debugger steps over the entire call (executes it to the return) and is ready to execute the line or instruction after the call.

Note

You can usually interrupt program execution by pressing either <CTL><BREAK> or <DEL>. These key combinations can be used to exit endless loops or to interrupt loops that are slowed by the Watchpoint or Tracepoint commands (see Chapter 9, ''Managing Watch Statements''). The <CTL><BREAK> or <DEL> keystrokes may not work if your program has a special use for one or both of these key combinations.

Selecting from Menus with the Keyboard

This section discusses how to make selections from menus with the keyboard. The effects of the selections are in the section ''Using Menu Selections.''

The menu bar at the top of the screen has eleven titles: File, View, Search, Run, Watch, Options, Language, Calls, Help, Trace, and Go. The first nine titles are menus, and the last two are commands.

The four steps for opening a menu and making a selection are:

1. To open a menu, press the <ALT> key and the mnemonic (the first letter) of the menu title. This can be accomplished by holding down the <ALT> key and then pressing the letter. For example, press <ALT>s. to open the Search menu. The menu title is highlighted, and a menu box listing the selections pops up below the title.

You can type either an uppercase or lowercase letter to open any of the menus.

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

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

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

   b. Press the key corresponding to the menu-selection mnemonic. The mnemonic is simply a single letter that represents the selection. In color displays, this letter is in red; in black-and-white displays, this letter is in bold. In most cases, but not all, the letter is simply the first letter of the name of the selection. You can type either an uppercase or lowercase letter for the same selection.

3. After a selection is made from the menu, one of three things happens:

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

   b. The items on the View, Options, and Language menus have small double arrows next to them if the option is on, or no arrows if the option is off. Choosing the item toggles the option. The status of the arrows is reversed the next time an option is chosen.

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

4. If the item you select requires a response, a dialog box opens when you select a menu item. Type your response to the prompt in the box and press the <RETURN> key. For example, the Find dialog box asks you to enter a regular expression.

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

At any point during the process of selecting a menu item, you can press the <ESC> key to cancel the menu. While a menu is open, you can press the LEFT ARROW or RIGHT ARROW key to move from one menu to an adjacent menu, or to one of the command titles on the menu bar. Pressing <RETURN> without entering any characters in response to a message box also cancels the menu.

Using Menu Selections

This section describes the selections on each of the CodeView menus. These selections can be made with the keyboard, as described in the section ''Executing Window Commands.''

Note that although the Trace and Go commands appear on the menu bar, they are not menus.

The File Menu

The File menu includes selections for working on the current source or program file. The File menu selections are explained below.

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

Note

If the debugger cannot find the source file when it switches modules, a dialog box appears asking for a file specification for the source file. You can either enter a new file specification if the file is in another directory, or press the <RETURN> key if no source file exists. If you press the <RETURN> key, the module can only be debugged in assembly mode.

Shell	Exits to a shell. This brings up the standard screen, where you can execute operating system commands or executable files. To return to the CodeView debugger, type **exit** at the operating system command prompt. The CodeView screen reappears with the same status it had when you left it.
Exit	Terminates the debugger and returns to the system.

The View Menu

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

Note

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

The use of one mode or another affects Trace and Program Step commands, as explained in Chapter 6, ``Executing Code.''

At all times only one of the following selections has a small double arrow to the left of the name: Source, Mixed, and Assembly. This arrow indicates which of the three display modes is in use. If you select a mode when you are already in that mode, the selection has no effect. The Registers selection may or may not have a double arrow to the left, depending on whether or not the register window is being displayed.

Selection	**Action**
Source	Changes to source mode (showing source lines only).
Mixed	Changes to mixed mode (showing both unassembled machine code and source lines).
Assembly	Changes to assembly mode (showing only unassembled machine code).
Registers	Selecting this option toggles the register window on and off. You can also turn the register on and off by pressing the **F2** key.
Output	Selecting this option displays the output screen. The entire CodeView display temporarily disappears, but come back as soon as you press any key. The Output command can also be selected with the **F4** key.

The Search Menu

The Search menu includes selections for searching through text files for text strings and for searching executable code for labels. The Search menu selections are explained below.

Selection	Action
Find...	Searches the current source file or other text file for a specified regular expression. (This selection can also be made without pulling down a menu, simply by pressing <CTL>**f**.) When you make this selection, a dialog box opens, asking you to enter a regular expression. Type the expression you want to search for and press the <RETURN> key. The CodeView debugger starts at the current or most recent cursor position in the display window and searches for the expression. If your entry is found, the cursor moves to the first source line containing the expression. If you are in assembly mode, the debugger automatically switches to source mode when the expression is found. If the entry is not found, a message box opens, telling you the problem and asking you to press a key to continue. Regular expressions are a method of specifying variable text strings. This method is similar to the standard method of using wild cards in file names. You can use the Search selections without understanding regular expressions. Since text strings are the simplest form of regular expressions, you can simply enter a string of characters as the expression you want to find. For example, you could enter **count** if you wanted to search for the word "count." The following characters have a special meaning in regular expressions: backslash (\), asterisk (*), left bracket ([), period (.), dollar sign ($), and caret (^). In order to find strings containing these characters, you must precede the characters with a backslash; this cancels their special meanings.

For example,
with C, you would use ***ptr** to find ***ptr**.

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

Next	Searches for the next match of the current regular expression. This selection is meaningful only after you have used the Search command to specify the current regular expression. If the CodeView debugger searches to the end of the file without finding another match for the expression, it wraps around and starts searching at the beginning of the file.
Previous	Searches for the previous match of the current regular expression. This selection is meaningful only after you have used the Search command to specify the current regular expression. If the debugger searches to the beginning of the file without finding another match for the expression, it wraps around and starts searching at the end of the file.
Label...	Searches the executable code for an assembly-language label. If the label is found, the cursor moves to the instruction containing the label. If you start the search in source mode, the debugger switches to assembly mode to show a label in a library routine or an assembly-language module.

The Run Menu

The Run menu includes selections for running your program. The Run menu selections are explained below.

Selection	**Action**
Start	Starts the program from the beginning and runs it. Any previously set breakpoints or watch statements are still in effect. The CodeView debugger runs your program from the beginning to the first

breakpoint, or to the end of the program if no breakpoint is encountered. This has the same effect as selecting Restart (see the next selection), then entering the Go command.

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

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

Execute
: Executes in slow motion from the current instruction.

  This is the same as the Execute dialog command (**e**). To stop execution, press any key.

Clear Breakpoints
: Clears all breakpoints.

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

Note

Although Start Restart retain breakpoints, along with pass count and arguments (see Chapter 6, "Executing Code,"), any instructions entered with the Assemble command will be overwritten by the original program.

The Watch Menu

The Watch menu includes selections for managing the watch window. Selections on this menu are also available with dialog commands. The Watch menu selections are explained below.

Selection	Action
Add Watch...	Adds a watch-expression statement to the watch window. (This selection can also be made directly, by pressing <CTL>**w**.) A dialog window opens, asking for the source-level expression (which may be simply a variable) whose value you want to see displayed in the watch window. Type the expression and press the <RETURN> key The statement appears in the watch window in normal text. You cannot specify a memory range to be displayed with the Add Watch selection as with the Watch dialog command. You can specify the format in which the value is displayed. Type the expression, followed by a comma and a CodeView format specifier. If you do not give a format specifier, the CodeView debugger displays the value in a default format. See Chapter 8, ''Examining Data and Expressions,'' for more information about format specifiers and the default format. See the section ''Setting Watch-Expression and Watch-Memory Statements'' in Chapter 9 for more information about the Watch command.
Watchpoint...	Adds a watchpoint statement to the window. A dialog window opens, asking for the source-level expression whose value you want to test. The watchpoint statement appears in the watch window in high-intensity text when you enter the expression. A watchpoint is a conditional breakpoint that causes execution to stop when the expression becomes nonzero (true). See the section ''Setting Watchpoints'' in Chapter 9 for more information.
Tracepoint...	Adds a tracepoint statement to the watch window. A dialog window opens, asking for the source-level expression or memory range whose value you want to test. The tracepoint statement appears in the watch window in high-intensity text when you enter the expression. A tracepoint is a conditional breakpoint that causes execution to stop

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

When setting a tracepoint expression, you can specify the format in which the value is displayed. After the expression type a comma and a format specifier. If you do not give a format specifier, the CodeView debugger displays the value in a default format. See Chapter 7, ''Examining Data and Expressions,'' for more information about format specifiers and default. See the section ''Setting Tracepoints'' in Chapter 9 for more information about tracepoints.

Delete Watch... Deletes a statement from the watch window. (This selection can also be made directly, by pressing <CTL>**u**).

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

Delete All Watch

Deletes all statements in the watch window.

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

The Options Menu

The Options menu allows you to set options that affect various aspects of the behavior of the CodeView debugger. The Options menu selections are explained below. Selections on the Options menu have small double arrows to the left of the selection name when the option is on. The status of the option (and the presence of the double arrows) is reversed each time you select the option. By default, the Save Output and Bytes Coded options are on when you start the CodeView debugger. Depending on which language your main program is in, the debugger automatically turns Case Sense on (if your program is in C) or off (if your program is in another language) when you start debugging.

The selections from the Options menu are discussed below.

Selection | **Action**

Save Output — When this option is on, which is the default setting, the output from your debugged program is saved. When it is off, any program output is not saved.

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

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

```
27:         name = gets(namebuf);
   LEA        AX,Word Ptr [namebuf]
   PUSH       AX
   CALL       _gets (03E1)
   ADD        SP,02
   MOV        Word Ptr [name],AX
```

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

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

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

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

The Language Menu

The Language menu allows you either to select the expression evaluator, or to instruct the CodeView debugger to select it for you automatically. The Language menu selections are explained below.

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

The Auto selection causes the debugger to select automatically the expression evaluator each time a new source file is loaded. The debugger examines the extension of the source file in order to determine which expression evaluator to select. The Auto selection uses the C expression evaluator if the current source file does not have a **.bas, .f, .for,** or **.pas** extension.

If you change to a source module with an **.asm** extension, then Auto causes the debugger to select the C expression evaluator, but not all of the C defaults are used; system radix is hexadecimal, case sensitivity is turned off, and the register window is displayed.

When a language expression evaluator is selected, the debugger uses that evaluator, regardless of what kind of program is being debugged.

The Calls Menu

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

The mnemonic for each item in the Calls menu is a number. Type the number displayed immediately to the left of a routine in order to select it. You can also use the UP ARROW or DOWN ARROW key to move to your selection, and then press the <RETURN> key.

The effect of making a selection from the Calls menu is to view a routine. The cursor goes to the line at which the selected routine was last executing. For example, selecting **main** causes CodeView to display **main**, at the point at which **main** made a call to **calc** (the function immediately above it). Note that selecting a routine from the Calls menu does not (by itself) affect program execution. It simply provides a convenient way to view previously called routines.

The Calls menu shows the current routine and the trail of routines from which it was called. The current routine is always at the top. The routine from which the current routine was called is directly below. Other active routines are shown in the reverse order in which they were called. With C programs, the bottom routine should always be **main**. (The only time when **main** will not be the bottom routine is when you are tracing through the standard library's start-up or termination routines.)

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

The Help Menu

The Help menu lists the major topics in the help system. For help, open the Help menu and then select the topic that you want to view.

Each topic may have any number of subtopics. You must go to the major topic first. Information on how to move around within the help system is provided in the next section.

The bottom selection on the Help menu is the About command. When you make this selection, the debugger displays a small box at the center of the screen that gives the name of the product and the version number.

Using the Help System

The CodeView on-line help system uses tree-structured menus to give you quick access to help screens on a variety of subjects. The system uses a combination of menu access and sequentially linked screens, as explained below.

The help file is called *cv.hlp* and is located in the */usr/lib* directory. If this file is not found, the CodeView debugger still operates, but you cannot use the help system. An error message appears if you try to use a help command.

When you request help, either by pressing the **F1** key, by using the **H** dialog command, or by selecting the Help menu, the first help screen appears. You can select **N** for Next and **P** for Previous to page through the screens. The screens are arranged in a circular fashion, so that selecting

Next on the last screen get you to the first screen. Select **C** for Cancel to return to the CodeView screen. Pressing the <PgDn>, <PgUp>, and <ESC> keys achieves the same results as selecting Next (**N**), Previous (**P**), and Cancel (**C**).

You can enter the help system at a particular topic by selecting the topic from the Help menu. Once into the system, use Next (**N**) and Previous (**P**) to page to other screens.

Using Sequential Mode

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

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

Command	Action
F1	Displays a command-syntax summary.
F2	Displays the registers. This is equivalent to the Register (**R**) dialog command.
F3	Toggles between source, mixed, and assembly modes. Pressing this key rotates the mode between source, mixed, and assembly. You can achieve the same effect by using the Set Assembly (**S-**), Set Mixed (**S&**), and Set Source(**S+**) dialog commands.
F4	Switches to the output screen, which shows the output of your program. Press any key to return to the CodeView debugging screen. This is equivalent to the Screen Exchange (\) dialog command.
F5	Executes from the current instruction until a breakpoint or the end of the program is encountered. This is equivalent to the Go dialog command (**G**) with no argument.
F8	Executes the next source line in source mode, or the next instruction in assembly mode.

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

The CodeView Watch (**W**), Watchpoint (**WP**), and Tracepoint (**TP**) commands work in sequential mode, but since there is no watch window, the watch statements are not shown. You must use the Watch List command (**W**) to examine watch statements and watch values. See Chapter 9, "Managing Watch Statements," for information on Watch Statement commands.

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

Chapter 4

Using Dialog Commands

Introduction

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

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

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

Entering Commands and Arguments

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

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

Using Special Keys

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

Key	**Action**
<DEL>	Stops the current output or cancels the current command line. For example, if you are watching a long display from a Dump command, you can press <DEL> to interrupt the output and return to the CodeView prompt. If you make a mistake while entering a command, you can press <DEL> to cancel the command without executing it. A new prompt appears, and you can reenter the command.
<CTL>s	Pauses during output of a command. You can press any key to continue output. For example, if you are watching a long display from a Dump command, you can press <CTL>s when a part of the display appears that you want to examine more closely. Then press any key when you are ready for the output to continue scrolling.
<BKSP>	Deletes the previous character on the command line and moves the cursor back one space. For example, if you make an error while typing a command, you can use the <BKSP> key to delete the characters back to the error-then retype the rest of the command.

Using the Command Buffer

In window mode, the CodeView debugger has a command buffer where the last 2-4 screens of commands and command output are stored. The command buffer is not available in sequential mode.

When the cursor is in the dialog window, you can scroll up or down to view the commands you have entered earlier in the session. The commands for moving the cursor and scrolling through the buffer are explained in Chapter 3, ''The CodeView Display.''

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

If you have scrolled through the dialog buffer to look at previous commands and output, you can still enter new commands. When you type a command, it appears to be overwriting the previous line where the cursor is located, but when you press the <RETURN> key, the new command is entered at the end of the buffer. For example, if you enter a command while the cursor is at the start of the buffer and then scroll to the end of the buffer, you see the command you 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 the originals.

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

Format for CodeView Commands and Arguments

The general format for CodeView commands is shown below:

"*<command>* [*<arguments>*] [;*<command2>*]"

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

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

Examples

```
>DB 100 200    ;* Example 1

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

>U sum; DB     ;* Example 3, multiple commands
```

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

In Example 2, **U** is the first command (for the Unassemble command), and the C language variable **Labell** is a command argument.

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

Chapter 5

CodeView Expressions

Introduction

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 or the Macro Assembler. CodeView has an expression evaluator for C that computes the value of source-level expressions.

Each of the expression evaluators has a different set of operators and rules of precedence. However, the basic syntax for registers, addresses, and line numbers is the same regardless of the language. You can always change the expression evaluator. If you specify a language other than the one used in the source file, then the expression evaluator still recognizes your program symbols, if possible.

If the Auto option is on, then the debugger examines the file extension of each new source file you trace through. Both C and assembly modules cause the debugger to select C as the expression evaluator.

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

Note

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

C Expressions

The C expression evaluator uses a subset of the most commonly used C operators. It also supports the colon operator (**:**), which is described in the section ''Addresses'' in this chapter, and the three memory operators (**BY**, **WO**, and **DW**), which are discussed in the section ''Memory Operators'' in this chapter. The memory operators are primarily useful for debugging assembly source code. The CodeView C-expression operators are listed in Table 5.1 in order of precedence.

Table 5.1

CodeView C-Expression Operators

Precedence	Operators
(Highest)	
1	( ) [ ] –> .
2	! ~ – (*type*) ++ – – * & **sizeof**
3	* / % :
4	+ –
5	< > <= >=
6	== !=
7	&&
8	||
9	= += –= *= /= %=
10	**BY** **WO** **DW**
(Lowest)	

The minus sign with precedence 2 is the *unary minus* indicating the sign of a number; the minus sign with precedence 4 is a *binary minus* indicating subtraction. The asterisk with precedence 2 is the pointer operator; the asterisk with precedence 3 is the multiplication operator. The ampersand with precedence 2 is the address-of operator. The ampersand as a bitwise AND operator is not supported by the CodeView debugger.

See the *C Language Referece* for a description of how C operators can be combined with identifiers and constants to form expressions. With the C-expression evaluator, the period (.) has its normal use as a member selection operator, but it also has an extended use as a specifier of local variables in parent functions. The syntax is shown below:

<function>.<variable>

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

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

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

C Symbols

Syntax

> <*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 is a variable name or the name of a function. Symbols (also called identifiers) follow the naming rules of the C compiler. Note that although 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 input from the Examine Symbols command, the CodeView debugger displays some symbol names in the object-code format produced by the C Compiler. This format includes a leading underscore. For example, the function **main** is displayed as **_main**. Only global labels (such as procedure names) are shown in this format. You do not need to include the underscore when specifying such a symbol in CodeView commands. Labels within library routines are sometimes displayed with a double underscore (**__chkstk**). You must use two leading underscores when accessing these labels with CodeView commands.

C Constants

Syntax

<*digits*>	Default radix
0<*digits*>	Octal radix
0x<*digits*>	Hexadecimal radix
0n<*digits*>	Decimal radix

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

The default radix for the C expression evaluator is decimal. However, you can use the Radix command (**N**) to specify an octal or hexadecimal radix, as explained in ''Radix Command'' in Chapter 12.

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

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

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

Table 5.2 shows how a sample number (63 decimal) would be represented in each radix.

Table 5.2

C Radix Examples

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

C Strings

Syntax

"*<null-terminated-string>*"

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

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*.

Assembly Expressions

The **-Zi** Macro Assembler option provides variable size information for the CodeView debugger. This makes for correct evaluation of expressions derived from assembly code (except with arrays, which are discussed later in this section). If you have an early version of the Macro Assembler, you need to use C type casts to get correct evaluation. See the *Release Notes* for more information about Macro Assembler versions.

When a program assembles or when the Auto switch is on, source files with an **.asm** extension cause CodeView to select the C-expression evaluator. However, the following options are set differently from the C default options:

- System radix is hexadecimal (not decimal).
- Register window is on.
- Case Sense is off.

The C-expression evaluator supports the memory operators described in the section ''Memory Operators'' in this chapter, and generally is the appropriate expression evaluator to debug assembly with, because of its flexibility.

However, you cannot always use the C-expression evaluator to specify an expression exactly as it would appear in assembly code. The list below describes the principal differences between assembler syntax and syntax used with the C-expression evaluator.

Note

The examples below present *expressions*, not CodeView commands. You can see the results of these expressions by using them as operands for the Display Expression command (**?**), described in Chapter 7, ''Examining Data and Expressions.''

In the following list, examples of assembly source code are shown in the left-hand column. Corresponding CodeView expressions (with the C-expression evaluator) are shown in the right-hand column.

Assembly Expressions

1. Register indirection.

   The C-expression evaluator does not extend the use of brackets to registers. To refer to the byte, word, or double word pointed to by a register, use the **BY**, **WO**, or **DW** operator.

```
BYTE PTR [bx]                    BY bx
WORD PTR [bp]                    WO bp
DWORD PTR [bp]                   DW bp
```

2. Register indirection with displacement.

   To perform based, indexed, or based-index indirection with a displacement, use the **BY**, **WO**, or **DW** operator along with addition in a complex expression:

```
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 the CodeView debugger, C type casts perform the same function as the 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 is problematic, because the Macro Assembler emits no type information to indicate which variables are arrays. Therefore the CodeView debugger 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 above, 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.

Line Numbers

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

Syntax

.[<*filename*>:]<*linenumber*>

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

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

Examples

```
>V .100
```

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

```
>V .DEMO.C:301
```

The example above uses **V** to display source code starting at line **301** of **demo.c**, respectively.

Registers and Addresses

This section presents alternative ways to refer to objects in memory, including values stored in the processor's registers. Addresses are basic to each of the expression evaluators. A data symbol represents an address in a data segment; a procedure name represents an address in a code segment. All of the syntax in this section can be considered as an extension to the C-expression evaluator.

Registers

Syntax

[@]<*register*>

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

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

The register names known to the CodeView debugger are shown in the following table.

Table 5.3
Registers

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

Addresses

Syntax

[<*segment*>:]<*offset*>

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

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

Examples

```
>DB 100
```

In the example above, the Dump Bytes command (**DB**) is used to dump memory starting at offset address **100**. Since no segment is given, the data segment (the default for Dump commands) is assumed. In C, a variable might be denoted as *array[count]*.

```
>DB label+10
```

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

```
>DB ES:200
```

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

Address Ranges

Syntax

<startaddress> *<endaddress>*
<startaddress> L *<count>*

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

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

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

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

Examples

```
>DB buffer
```

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

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

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

```
>DB buffer L 20
```

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

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

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

Memory Operators

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

All of the operators listed in this section are part of the CodeView 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.

Note

The memory operators discussed in this section are only available with the C-expression evaluator, and have lowest precedence of any C operators.

Accessing Bytes (BY)

You can access the byte at an address by using the **BY** operator. This operator is useful for simulating the **BYTE PTR** operation of the Macro Assembler. It is particularly useful for watching the byte pointed to by a particular register.

Note

The examples that follow in the section "Memory Operators" make use of the Display Expression (?) Command, which is described in "Display Expression Command" in Chapter 7. The **x** format specifier causes the debugger to produce output in hexadecimal.

Syntax

BY *<address>*

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

Examples

```
>? BY sum
101
```

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

```
>? BY bp+6
42
```

This example returns the byte pointed to by the **BP** register, with a displacement of 6.

Accessing Words (WO)

You can access the word at an address by using the **WO** operator. This operator is useful for simulating the **WORD PTR** operation of the assembler. It is particularly useful for watching the word pointed to by a particular register, such as the stack pointer.

Syntax

WO *<address>*

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

Examples

```
>? WO sum
>13120
```

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

```
>? WO sp,x
>2F38
```

This example returns the word pointed to by the stack pointer; the word therefore represents the last word pushed (the "top" of the stack).

Accessing Double Words (DW)

You can access the word at an address by using the **DW** operator. This operator is useful for simulating the **DWORD PTR** operation of the Macro Assembler. It is particularly useful for watching the word pointed to by a particular register.

Syntax

DW *<address>*

The result is a long integer that contains the value of the first four bytes stored at *address*.

Note

> Be careful not to confuse the **DW** operator with the **DW** command. The operator is only useful for building expressions; it occurs within a CodeView command line, but never at the beginning. The second use of **DW** mentioned above, the Dump Words Command, occurs only at the beginning of a CodeView command line. It displays an entire range of memory (in words, not double words) rather than returning a single result.

Examples

```
>? DW sum
>132120365
```

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

```
>? DW si,x
>3F880000
```

This example returns the double word pointed to by the **SI** register.

Chapter 6

Executing Code

Introduction

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

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

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

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

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

Introduction

The commands that execute code are explained in the following sections.

Note

If you are executing a section of code with the Go or Program Step command, you can usually interrupt program execution by pressing <CTL><BREAK> or <DEL>. This can terminate endless loops, or it can interrupt loops that are delayed by the Watchpoint or Tracepoint command (see Chapter 9, ''Managing Watch Statements'').

Trace Command

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

In source mode, if the current source line contains a call, the CodeView debugger executes the first source line of the called routine. In this mode, the CodeView debugger only traces into functions and routines that have source code. For example, if the current line contains a call to an intrinsic function or a standard C library function, the debugger simply executes the function if you are in source mode, since the source code for standard libraries is not available.

If you are in assembly or mixed mode, the debugger traces into the function. In this mode, if the current instruction is **CALL**, **INT** or **REP**, the debugger executes the first instruction of the procedure, interrupt, or repeated string sequence.

Note

When you debug Macro Assembler programs in source mode, the paragraph above still applies. The debugger does not trace into an **INT** or **REP** sequence when you are in source mode.

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

Keyboard

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

Trace Command

Dialog

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

T [<*count*>]

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

Example

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

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

>
```

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

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

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

The example above sets the display mode to assembly and traces the current instruction. This example and the next example are the same as the examples of the Program Step command in the section ''Program Step Command'' in this chapter. The Trace and Program Step commands behave differently only when the current instruction is a **CALL**, **INT**, or **REP** instruction.

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

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

Program Step Command

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

In source mode, if the current source line contains a call, the CodeView debugger executes the entire routine and is ready to execute the line after the call. In assembly mode, if the current instruction is **CALL**, **INT**, or **REP**, the debugger executes the entire procedure, interrupt, or repeated string sequence. Use the Program Step command if you want to execute over routine, function, procedure, and interrupt calls. If you want to trace into calls, you should use the Trace command (**T**) instead. Both commands execute system calls without tracing into them. There is no direct way to trace into system calls.

Keyboard

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

Dialog

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

P [<*count*>]

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

Example

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

```
>S+     ;* FORTRAN/BASIC example
source
>.
9:           CALL INPUT  (DATA,N,INPFMT)
>P 3
10:          CALL BUBBLE  (DATA,N)
11:          CALL STATS  (DATA,N)
12:          END
>
```

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

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

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

The example above sets the display mode to assembly and steps through the current instruction. This example and the next example are the same as the examples of the Trace command in the section "Trace Command" in this chapter. The Trace and Program Step commands behave differently only when the current instruction is a **CALL**, **INT**, or **REP** instruction.

Program Step Command

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

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

Go Command

The Go command starts execution at the current address. There are two variations of the Go command, Go and Goto. The Go variation simply starts execution and continues to the end of the program or until a breakpoint set earlier with the Breakpoint Set (**BP**), Watchpoint (**WP**), or Tracepoint (**TP**) command is encountered. The other variation is a Goto command, in which a destination is given with the command.

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

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

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

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

Keyboard

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

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

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

In the example above, the same number is entered in different radixes, but the **i** format specifier is used to display the result as a decimal integer in all three cases. See Chapter 7, ''Examining Data and Expressions,'' for more information on format specifiers.

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

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

Note

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

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

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

Execute Command

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

Keyboard

To execute code in slow motion with a keyboard command, press <ALT>**r** to open the Run menu, and then press <ALT>**e** to select Execute.

Dialog

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

E

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

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

Important

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

Restart Command

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

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

Keyboard

To restart the program with a keyboard command, press <ALT>**r** to open the Run menu, and then press either <ALT>**r** to select Restart or <ALT>**s** to select Start. The program is restarted. If the Restart selection is chosen, the program is ready to start executing from the beginning (but not actually running). If the Start selection is chosen, the program starts executing from the beginning and continues until a breakpoint or the end of the program is encountered.

Dialog

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

L [<*arguments*>]

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

Note

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

Examples

```
>L
>
```

The example above restarts the current executable file, retaining the same breakpoints, watchpoints, tracepoints, and command line arguments.

```
>L 6
>
```

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

Chapter 7

Examining Data and Expressions

Introduction

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

Command	Action
Display Expression (**?**)	Evaluates and displays the value of symbols or expressions
Examine Symbol (**X?**)	Displays the addresses of symbols
Dump (**D**)	Displays sections of memory containing data (with variations for examining different kinds of data)
Compare Memory (**C**)	Compares two blocks of memory, byte by byte
Search Memory (**S**)	Scans memory for specified byte values
Register (**R**)	Shows the current values of each register and each flag
8087 (**7**)	Shows the current value in the 80387 or 80287 register

Display Expression Command

The Display Expression command displays the value of a CodeView expression.

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

Note

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

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

You can specify the format in which the values of expressions are displayed by the Display Expression command. Type a comma after the expression, followed by a CodeView format specifier. The format specifiers used in the CodeView debugger are a subset of those used by the C **printf** function. They are listed in table 7.1.

Table 7.1

CodeView Format Specifiers

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

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

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

The ''E'' is uppercase if the type is **E** or **G**; lowercase if the type is **e** or **g**.

The **s** string format is used only with the C-expression evaluator; it prints characters up to the first null.

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

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

When calling a FORTRAN subroutine that uses alternate returns, the value of the return labels in the actual parameter list must be 0. For example, the subroutine call **CALL PROCESS (s-1I,*10,J,*20,*30)** must be called from the debugger as **?PROCESS(IARG1,0,IARG2,0,0)**. Using other values as return labels cause the error `Type clash in function argument` or `Unknown symbol.`

Note

Do not use a type specifier when evaluating strings in FORTRAN, 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, with which other languages conflict; if you use **s**, then the debugger simply displays characters at the given address until a null is encountered.

Keyboard

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

Dialog

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

? <*expression*>[,<*format*>]

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

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

If you are debugging code written with the assembler, you use the C-expression evaluator by default. Consult the section "Assembly Expressions" in Chapter 5 for guidelines on how to use the C-expression evaluator with assembly code.

Examples

```
>? amount
500
>? amount,x
1f4
>? amount,o
764
>
```

The example above displays the value stored in the variable *amount*, an integer. This value is first displayed in the system radix (in this case, decimal), then in hexadecimal, and then in octal.

```
>? 92,x
5c
>? 109*(35+2),o
7701
>? 118,c
v
>
```

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

```
>? chance,f
0.083333
>? chance,e
8.333333e-002
>? chance,E
8.333333E-002
```

The example above shows a double-precision real number, **chance**, displayed in three formats. The **f** format always displays six digits of precision. The **e** format uses scientific notation. Note that the **E** format yields essentially the same display as **e** does.

The rest of the examples in this section are specific to particular languages.

C Examples

The following examples assume that a C source file is being debugged, and that it contains the following declarations:

```
char *text = "Here is a string.";
int  amount;
struct {
     char name[20];
     int  id;
     long class;
} student, *pstudent;

int square(int);
```

Assume also that the program has been executed to the point where the above variables have been assigned values, and that the C-expression evaluator is in use.

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

The example above shows how to examine strings. One method is to evaluate the variable that points to the string, and then dump the values at that address (the Dump commands are explained in the section ''Dump Commands'' in this chapter). A more direct method is to use the s type specifier.

```
>? student.id
19643
>? pstudent->id
19643
>
```

The example above illustrates how to display the values of members of a structure. The same syntax applies to unions.

```
>? amount
500
>? ++amount
501
>? amount=600
600
>
```

The example above shows how the Display Expression command can be used with the C-expression evaluator to change the values of variables.

```
>? square(9)
81
>
```

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

Assembly Examples

By default, the C-expression evaluator is used for debugging assembly modules. However, some C expressions are particularly helpful for debugging assembly code. Some typical examples are presented below.

```
>? BY bx
12
>
```

The example above displays the first byte at the location pointed to by **BX**, and is equivalent to the assembly expression **BYTE PTR [bx]**.

```
>? WO bp+8
9359
>
```

The example above displays the first word at the location pointed to by **[bp+8]**.

```
>? DW si+12
12555324
>
```

The example above displays the first double word at the location pointed to by **[si+12]**.

```
>? (char) var
5
>? (int) var
1005
>
```

The last two examples use type casts, which are similar to the 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. You can alter either of these commands to display results in unsigned format simply by using the **u** format specifier.

```
>? (char) var,u

>? (int) var,u
```

Examine Symbols Command

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

Keyboard

The Examine Symbols command cannot be executed with a keyboard command.

Dialog

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

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

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

The syntax combinations are listed in more detail below.

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

X?<module>!*	All symbols in the specified *module*
X?<routine>.<symbol>	The specified *symbol* in the specified *routine* (looks for *routine* first in the current module, and then in other modules from first to last)
X?<routine>.*	All symbols in the specified *routine* (looks for *routine* first in the current module, and then in other modules from first to last)
X?<symbol>	Looks for the specified *symbol* in this order: 1. In the current routine 2. In the current module 3. In other modules, from first to last
X?*	All symbols in the current routine
X*	All module names
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; you *must* use the *module* field. Therefore, the only versions of this command that work with assembly modules are the following:

X?<*module*>!*
X?<*module*>!<*symbol*>

C Examples

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

```
>X*           ;*Example 1
pi
math
/lib/slibc.a(chkstk)
/lib/slibc.a(crt0)
.
.
.
/lib/slibc.a(itoa)
/lib/slibc.a(unlink)
>
```

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

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

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

```
>X?pi!*          ;* Example 3
3D37:19B2 int    _scratch0    3D37:0A10 char     _p[]
3D37:2954 int    _scratch1    3D377:19B4 char    _t[]
3D37:2956 int    _scratch2    3D377:19B0 int     _q
3A79:0010 int    _main()      3A79:0010 int      main()
3D37:19B2 int     scratch0
3D37:0A10 char    p[]
3D37:2954 int     scratch1
3D37:19B4 char    t[]
3D37:2956 int     scratch2
3D37:19B0 int     q
>
```

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

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

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

Variables local to a function are indented under that function.

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

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

Dump Commands

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

Command	Command Name
D	Dump (size is the default type)
DB	Dump Bytes
DA	Dump ASCII
DI	Dump Integers
DU	Dump Unsigned Integers
DW	Dump Words
DD	Dump Double Words
DS	Dump Short Reals
DL	Dump Long Reals
DT	Dump 10-Byte Reals

Keyboard

The Dump commands cannot be executed with keyboard commands.

Dialog

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

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

The *type* is a one-letter specifier that indicates the type of the data to be dumped. The Dump commands expect either a starting *address* or a *range* of memory. If the starting *address* is given, the commands assume a default range (usually determined by the size of the dialog window) starting at *address*. If *range* is given, the commands dump from the start to the end of *range*. The maximum size of *range* is 32K.

If neither *address* nor *range* is given, the commands assume the current dump address as the start of the range and the default size associated with the size of the object as the length of the range. The Dump Real commands have a default range size of one real number. The other Dump commands have a default size determined by the size of the dialog window (if you are in window mode), or a default size of 128 bytes otherwise.

The current dump address is the byte following the last byte specified in the previous Dump command. If no Dump command has been used during the session, the dump address is the start of the data segment (**DS**). For example, if you enter the Dump Words command with no argument as the first command of a session, the CodeView debugger displays the first 64 words (128 bytes) of data declared in the data segment. If you repeat the same command, the debugger displays the next 64 words following the ones dumped by the first command.

Note

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

The following sections discuss the variations of the Dump commands in order of the size of data they display.

Dump

Syntax

D [<*address*> | <*range*>]

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

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

Dump Bytes

Syntax

DB [<address> | <range>]

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

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

Example

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

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

Dump ASCII

Syntax

DA [<*address*> | <*range*>]

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

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

Examples

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

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

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

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

Dump Integers

Syntax

DI [<*address*> | <*range*>]

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

Note

In this manual an integer is considered a two-byte value, since the CodeView debugger assumes that integer size.

Example

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

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

Dump Unsigned Integers

Syntax

DU [<*address*> | <*range*>]

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

Example

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

The example above displays the byte values from **DS:0** to **DS:36** (24 hexadecimal). Compare the unsigned decimal numbers at the end of this dump with the same values shown as signed integers in the section ''Dump Integers'' in this chapter.

Dump Words

Syntax

DW [<*address*> | <*range*>]

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

Example

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

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

Dump Double Words

Syntax

DD [<*address*]> | <*range*>]

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

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

Example

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

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

Dump Short Reals

Syntax

DS [<*address*> | <*range*>]

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

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

The decimal value has the following form:

[-]<*digit*>.<*digits*>**E**{+ | -<*exponent*>

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

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

Example

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

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

Dump Long Reals

Syntax

DL [<*address*> | <*range*>]

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

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

The decimal value has the following form:

[-]<*digit*>.<*digits*>E{+ | -}<*exponent*>

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

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

Example

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

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

Dump 10-Byte Reals

Syntax

DT [<*address*> | <*range*>]

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

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

The decimal value has the following form:

[-]<*digit*>.<*digits*>E{+ | -}<*exponent*>

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

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

Example

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

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

Compare Memory Command

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

Keyboard

The Compare Memory command cannot be executed with a keyboard command.

Dialog

To compare two blocks of memory, enter a command line with the following syntax:

C *<range1start>* *<range1end>* *<range2start>* *<address>*

The bytes in the memory locations specified by *range* are compared with the corresponding bytes in the memory locations beginning at *address*. If one or more pairs of corresponding bytes do not match, each pair of mismatched bytes is displayed.

Examples

```
>C 100 01FF 300    ;* hexadecimal radix assumed
39BB:0102  0A  00  39BB:0302
39BB:0108  0A  01  39BB:0308
>
```

The first example (in which hexadecimal is assumed to be the default radix) compares the block of memory from 100 to 1FF with the block of memory from 300 to 3FF. It indicates that the third and ninth bytes differ in the two areas of memory.

```
>C arr1[0] L 100 arr2[0]    ;* C notation used.
>
```

The example compares the 100 bytes starting at the address of **arr1[0]**, with the 100 bytes starting at address of **arr2[0]**. The CodeView debugger produces no output in response, so this indicates that the first 100 bytes of each array are identical.

Note

You can enter the Compare Memory command using any radix you like; however, any output is still in hexadecimal format.

Search Memory Command

The Search Memory command (not to be confused with the Search command discussed in Section 11.6) scans a specified area of memory, looking for specific byte values. It is primarily of interest to programmers using assembly mode, and to users who want to test for the presence of specific values within a range of data.

Keyboard

The Search Memory command cannot be executed with a keyboard command.

Dialog

To search a block of memory, enter the Search Memory command with the following syntax:

S *<range>* *<list>*

The debugger searches the specified *range* of memory locations for the byte values specified in the *list*. If bytes with the specified values are found, then the debugger displays the addresses of each occurrence of bytes in the list.

The *list* can have any number of bytes. Each byte value must be separated by a space or comma, unless the list is an ASCII string. If the list contains more than one byte, then the Search Memory command looks for a series of bytes that precisely match the order and value of bytes in *list*. If found, then the beginning address of each such series is displayed.

Examples

```
>S buffer L 1500 "error"
2BBA:0404
2BBA:05E3
2BBA:0604
>
```

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

```
>S DS:100 200 0A  ;* hexadecimal radix assumed
3CBA:0132
3CBA:01C2
>
```

The second example displays the address of each memory location that contains the byte value **0A** in the range DS:0100 to DS:0200 (hexadecimal). The value was found at two addresses.

Register Command

The Register command has two functions. It displays the contents of the central processing unit (CPU) registers. It can also change the values of the registers. The display features of the Register command are explained here. The modification features of the command are explained in Chapter 11, ''Modifying Code or Data.''

The flag register display colors are significant; if a flag bit is set, the two-letter code for that condition is displayed as BRIGHT (for monochromatic monitors) or RED (for color monitors). If the flag is clear, the two-letter code for that cleared flag is displayed as NORMAL_INTENSITY (for monochromatic monitors) or CYAN (for color monitors).

Keyboard

To display the registers using a keyboard command in window mode, press the **F2** key. The register window appears on the right side of the screen. If the register window is already on the screen, the same command removes it.

In sequential mode, the **F2** key displays the current status of the registers. (This produces the same effect as entering the Register dialog command with no argument.)

Dialog

To display the registers in the dialog window (or sequentially in sequential mode), enter a command line with the following syntax:

R

The current values of all registers and flags are displayed. The instruction at the address pointed to by the current **CS** and **IP** register values is also shown. (The Register command can also be given with arguments, but only when used to modify registers, as explained in Chapter 11, ''Modifying Code or Data.'')

If the display mode is source (S+) or mixed (**S&**) (see ''Set Mode Command'' in Chapter 10 for more information), the current source line is also displayed by the Register command. If an operand of the instruction contains memory expressions or immediate data, the CodeView debugger evaluates operands and show the value to the right of the instruction. This value is referred to as the ''effective address,'' and is also displayed at the bottom of the register window. If the **CS** and **IP** registers are currently at a breakpoint location, the register display indicates the breakpoint number.

In sequential mode, the Trace (**T**), Program Step (**P**), and Go (**G**) commands show registers in the same format as the Register command.

Examples

```
>S&
mixed
>R
AX=0005  BX=299E  CX=0000  DX=0000  SP=3800  BP=380E  SI=0070  DI=40D1
DS=5067  ES=5067  SS=5067  CS=4684  IP=014F  NV UP EI PL NZ NA PO NC
35:                 VARIAN = (N*SUMXSQ-SUMX* *2)/(N-1)
4684:014F 8B5E06       MOV     BX,Word Ptr [BP+06]      ;BR1  SS:3814=299E
>
```

The example above 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 has been set to mixed (**S&**), the current source line is also shown. The example is from a FORTRAN program, but applies equally well to BASIC and C programs.

```
>S-
assembly
>R
AX=0005  BX=299E  CX=0000  DX=0000  SP=3800  BP=380E  SI=0070  DI=40D1
DS=5067  ES=5067  SS=5067  CS=4684  IP=014F  NV UP EI PL NZ NA PO NC
4684:014F 8B5E06       MOV     BX,Word Ptr [BP+06]      ;BR1  SS:3814=299E
>
```

In the example above, the display mode is set to assembly (**S-**), so no source line is shown. Note the breakpoint number at the right of the last line, indicating that the current address is at Breakpoint 1.

8087 Command

The 8087 command dumps the contents of the 8087 registers. If you do not have an 8087, 80287, or 80387 coprocessor chip on your system, then this command dumps the contents of the pseudoregisters created by the operating system's floating point emulator.

Note

This section does not attempt to explain how the registers of the Intel 8087, 80287, and 80387 processors are organized or how they work. In order to interpret the command output, you must learn about the chip from an Intel reference manual or other book on the subject. Since emulator routines mimic the behavior of the 8087 coprocessor, these references apply to emulator routines as well as to the chips themselves.

Keyboard

The 8087 command cannot be executed with a keyboard command.

Dialog

To display the status of the math co-processor chip (or floating-point emulator routines) with a dialog command, enter a command line with the following syntax:

7

The current status of the chip is displayed when you enter the command. In window mode, the output is to the dialog window.

The following example shows a display for this command.

8087 Example

```
>7
Control 037F  (Projective closure, Round nearest, 64-bit precision)
                    iem=0 pm=1 um=1 om=1 zm=1 dm=1 im=1
Status  6004  cond=1000 top=4 pe=0 ue=0 oe=0 ze=1 de=0 ie=0
Tag     A1FF  instruction=59380  operand=59360  opcode=D9EE
Stack         Exp  Mantissa            Value
ST(3) special 7FFF 8000000000000000  = + Infinity
ST(2) special 7FFF 0101010101010101  = + Not a Number
ST(1) valid   4000 C90FDAA22168C235  = +3.141592265110390E+000
ST(0) zero    0000 0000000000000000  = +0.000000000000000E+000
>
```

In the example above, the first line of the dump shows the current closure method, rounding method, and the precision. The number **037F** is the hexadecimal value in the control register. The rest of the line interprets the bits of the number. The closure method can be either projective (as in the example) or affine. The rounding method can be either rounding to the nearest even number (as in the example), rounding down, rounding up, or using the chop method of rounding (truncating toward zero). The precision may be 64 bits (as in the example), 53 bits, or 24 bits.

The second line of the display indicates whether each exception mask bit is set or cleared. The masks are interrupt-enable mask (**iem**), precision mask (**pm**), underflow mask (**um**), overflow mask (**om**), zero-divide mask (**zm**), denormalized-operand mask (**dm**), and invalid-operation mask (**im**).

The third line of the display shows the hexadecimal value of the status register (**6004** in the example), and then interprets the bits of the register. The condition code (**cond**) in the example is the binary number **1000**. The top of the stack (**top**) is register 4 (shown in decimal). The other bits shown are precision exception (**pe**), underflow exception (**ue**), overflow exception (**oe**), zero-divide exception (**ze**), denormalized-operand exception (**de**), and invalid-operation exception (**ie**).

The fourth line of the display first shows the hexadecimal value of the tag register (**A1FF** in the example). It then gives the hexadecimal values of the instruction (**59380**), the operand (**59360**), and the operation code, or opcode, (**D9EE**).

The fifth line is a heading for the subsequent lines, which contain the contents of each 8087, 80287, or 80387 stack register. The registers in the example contain four types of numbers that may be held in these registers. Starting from the bottom, register 0 contains zero. Register 1 contains a valid real number. Its exponent (in hexadecimal) is **4000** and its mantissa is **C90FDAA22168C235**. The number is shown in scientific notation in the rightmost column. Register 2 contains a value that cannot be interpreted as a number, and register 3 contains infinity.

Chapter 8

Managing Breakpoints

Introduction

The CodeView debugger enables you to control program execution by setting breakpoints. A breakpoint is an address that stops program execution each time the address is encountered. By setting breakpoints at key addresses in your program, you can "freeze" program execution and examine the status of memory or expressions at that point.

The commands listed below control breakpoints:

Command	**Action**
Breakpoint Set (**BP**)	Sets a breakpoint and, optionally, a pass count and break commands
Breakpoint Clear (**BC**)	Clears one or more breakpoints
Breakpoint Disable (**BD**)	Disables one or more breakpoints
Breakpoint Enable (**BE**)	Enables one or more breakpoints
Breakpoint List (**BL**)	Lists all breakpoints

In addition to these commands, the Watchpoint (**WP**) and Tracepoint (**TP**) commands can be used to set conditional breakpoints (see Chapter 10, "Examining Code," for information on these two commands).

Breakpoint Set Command

The Breakpoint Set command (**BP**) creates a breakpoint at a specified address. Any time a breakpoint is encountered during program execution, the program halts and waits for a new command.

The CodeView debugger allows up to 20 breakpoints (0 through 19). Each new breakpoint is assigned to the next available number. Breakpoints remain in memory until you delete them or until you quit the debugger. They are not canceled when you restart the program. Because breakpoints are not automatically canceled, you are able to set up a complicated series of breakpoints, then execute through the program several times without resetting.

If you try to set a breakpoint at a comment line or other source line that does not correspond to code, the CodeView debugger displays the following message:

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

Keyboard

To set a breakpoint with a keyboard command in window mode, move the cursor to the source line or instruction where you want to set a breakpoint. You may have to press the **F6** key to move the cursor to the display window. When the cursor is on the appropriate source line, press the **F9** key. The line is displayed in high-intensity text, and remains so until you remove or disable the breakpoint.

In sequential mode, the **F9** key can be used to set a breakpoint at the current location. You must use the dialog version of the command to set a breakpoint at any other location.

Dialog

To set a breakpoint using a dialog command, enter a command line with the following syntax:

BP [<*address*> [<*passcount*>] [<*commands*>]]

If no *address* is given, a breakpoint is created on the current source line in source mode, or on the current instruction in assembly mode. You can specify the *address* in the *segment:offset* format or as a source line, a routine name, or a label. If you give an offset address, the code segment is assumed.

The dialog version of the command is more powerful than the mouse or keyboard version in that it allows you to give a *passcount* and a string of *commands*. The *passcount* specifies the first time the breakpoint is to be taken. For example, if the pass count is 5, the breakpoint is ignored the first four times it is encountered, and taken the fifth time. Thereafter, the breakpoint is always taken.

The *commands* are a list of dialog commands enclosed in quotation marks (" ") and separated by semicolons (;). For example, if you specify the commands as ?**code;T" "**, the CodeView debugger automatically displays the value of the variable *code* and then execute the Trace command each time the breakpoint is encountered. The Trace and Display Expression commands are described in Chapter 6, ''Executing Code,'' and Chapter 7, ''Examining Data and Expressions,'' respectively.

In window mode, a breakpoint entered with a dialog command has exactly the same effect as one created with a window command. The source line or instruction corresponding to the breakpoint location is shown in high-intensity text.

In sequential mode, information about the current instruction is displayed each time you execute to a breakpoint. The register values, the current instruction, and the source line may be shown, depending on the display mode. See Chapter 10, ''Examining Code,'' for more information about display modes.

When a breakpoint address is shown in the assembly-language format, the breakpoint number is shown as a comment to the right of the instruction. This comment appears even if the breakpoint is disabled (but not if it is deleted).

8

Examples

```
>BP .19 10
>
```

The example above creates a breakpoint at line 19 of the current source file (or if there is no executable statement at line 19, at the first executable statement after line 19). The breakpoint is passed over nine times before being taken on the 10th pass.

```
>BP STATS 10 "?COUNTER = COUNTER + 1;G"
>
```

The example above creates a breakpoint at the address of the routine **STATS**. The breakpoint is passed over nine times before being taken on the 10th pass. Each time execution stops for the breakpoint, the quoted commands are executed. The Display Expression command increments *COUNTER*, then the Go command restarts execution. If *COUNTER* is set to 0 when the breakpoint is set, this has the effect of counting the number of times the breakpoint is taken.

```
>S-   ;* FORTRAN example - uses FORTRAN hexadecimal notation
assembly
>BP #0a94
>G
AX=0006 BX=304A CX=000B DX=465D SP=3050 BP=3050 SI=00BB DI=40D1
DS=5064 ES=5064 SS=5064 CS=46A2 IP=0A94 NV UP EI PL NZ NA PE NC
46A2:0A94 7205      JB     __chkstk+13 (0A9B)     ;BR1
>
```

The example above first sets the mode to assembly, and then creates a breakpoint at the hexadecimal (offset) address **#0A94** in the default (**CS**) segment. (The same address would be specified as **0x0A94** with the C-expression evaluator, and as **&H0A9** with the BASIC-expression evaluator.) The Go command (**G**) is then used to execute to the breakpoint. Note that in the output to the Go command, the breakpoint number is shown as an assembly-language comment (**;BR1**) to the right of the current instruction. The Go command displays this output only in sequential mode; in window mode no assembly-language information appears.

Breakpoint Clear Command

The Breakpoint Clear command (**BC**) permanently removes one or more previously set breakpoints.

Keyboard

To clear a single breakpoint with a keyboard command, move the cursor to the breakpoint line or instruction you want to clear. Breakpoint lines are shown in high-intensity text. Press the **F9** key. The line is shown in normal text to indicate that the breakpoint has been removed.

To remove all breakpoints using a keyboard command, press <ALT>**r** to open the Run menu, and then press <ALT>**c** to select Clear Breakpoints.

Dialog

To clear breakpoints using a dialog command, enter a command line with the following syntax:

BC <*list*>
BC *

If *list* is specified, the command removes the breakpoints named in the list. The *list* can be any combination of integer values from 0 to 19. You can use the Breakpoint List command (**BL**) if you need to see the numbers for each existing breakpoint. If an asterisk (*) is given as the argument, all breakpoints are removed.

8

Examples

```
>BC 0 4 8
>
```

The example above removes breakpoints 0, 4, and 8.

```
>BC *
>
```

The example above removes all breakpoints.

Breakpoint Disable Command

The Breakpoint Disable command (**BD**) temporarily disables one or more existing breakpoints. The breakpoints are not deleted. They can be restored at any time using the Breakpoint Enable command (**BE**).

When a breakpoint is disabled in window mode, it is shown in the display window with normal text; when enabled, it is shown in high-intensity text.

Note

All disabled breakpoints are automatically enabled whenever you restart the program being debugged. The program can be restarted with the Start or Restart selection from the Run menu, or with the Restart dialog command (**L**). See Chapter 6, "Executing Code."

Keyboard

The Breakpoint Disable command cannot be executed with a keyboard command.

Dialog

To disable breakpoints with a dialog command, enter a command line with the following syntax:

BD <*list*>
BD *

If *list* is specified, the command disables the breakpoints named in the list. The *list* can be any combination of integer values from 0 to 19. Use the Breakpoint List command (**BL**) if you need to see the numbers for each existing breakpoint. If an asterisk (*) is given as the argument, all breakpoints are disabled.

The window commands for setting and clearing breakpoints can also be used to enable or clear disabled breakpoints.

Examples

```
>BD 0 4 8
>
```

The example above disables breakpoints 0, 4, and 8.

```
>BD *
>
```

The example above disables all breakpoints.

Breakpoint Enable Command

The Breakpoint Enable command (**BE**) enables breakpoints that have been temporarily disabled with the Breakpoint Disable command.

Keyboard

To enable a disabled breakpoint using a keyboard command, move the cursor to the source line or instruction of the breakpoint, and then press the **F9** key. The line is displayed in high-intensity text, and remains so until you remove or disable the breakpoint. This is the same as creating a new breakpoint at that location.

Dialog

To enable breakpoints using a dialog command, enter a command line with the following syntax:

> **BE** <*list*>
> **BE** *

If *list* is specified, the command enables the breakpoints named in the list. The *list* can be any combination of integer values from 0 to 19. Use the Breakpoint List command (**BL**) if you need to see the numbers for each existing breakpoint. If an asterisk (*) is given as the argument, all breakpoints are enabled. The CodeView debugger ignores all or part of the command if you try to enable a breakpoint that is not disabled.

Examples

```
>BE 0 4 8
>
```

The example above enables breakpoints 0, 4, and 8.

```
>BE*
>
```

The example above enables all disabled breakpoints.

Breakpoint List Command

The Breakpoint List command (**BL**) lists current information about all breakpoints.

Keyboard

The Breakpoint List command cannot be executed with a keyboard command.

Dialog

To list breakpoints with a dialog command, enter a command line with the following syntax:

BL

The command displays the breakpoint number, the enabled status (**e** for "enabled", **d** for "disabled"), the address, the routine, and the line number. If the breakpoint does not fall on a line number, an offset is shown from the nearest previous line number. The pass count and break commands are shown if they have been set. If no breakpoints are currently defined, nothing is displayed.

Example

```
>BL
0 e 56C4:0105  _ARCTAN:10
1 d 56C4:011E  _ARCTAN:19          (pass = 10) "T;T"
2 e 56C4:00FD  _ARCTAN:9+6
>
```

In the example above, breakpoint 0 is enabled at address **56C4:0105**. This address is in routine **ARCTAN** and is at line **10** of the current source file. No pass count or break commands have been set.

Breakpoint 1 is currently disabled, as indicated by the **d** after the breakpoint number. It also has a pass count of 10, meaning that the breakpoint is not taken until the 10th time it is encountered. The command string at the end of the line indicates that each time the breakpoint is taken, the Trace command is automatically executed twice.

The line number for breakpoint 2 has an offset. The address is six bytes beyond the address for line 9 in the current source file. Therefore, the breakpoint was probably set in assembly mode, since it would be difficult to set a breakpoint anywhere except on a source line in source mode.

Chapter 9

Managing Watch Statements

Introduction

Watch Statement commands are among the CodeView debugger's most powerful features. They enable you to set, delete, and list watch statements. Watch statements describe expressions or areas of memory to watch. Some watch statements specify conditional breakpoints, which depend upon the value of the expression or memory area. The Watch Statement commands are summarized below:

Command	**Action**
Watch (**W**)	Sets an expression or range of memory to be watched
Watchpoint (**WP**)	Sets a conditional breakpoint that is taken when the expression becomes nonzero (true)
Tracepoint (**TP**)	Sets a conditional breakpoint that is taken when a given expression or range of memory changes
Watch Delete (**Y**)	Deletes one or more watch statements
Watch List (**W**)	Lists current watch statements

Watch statements, like breakpoints, remain in memory until you specifically remove them or quit the CodeView debugger. They are not canceled when you restart the program being debugged. Therefore, you can set a complicated series of watch statements once, and then execute through the program several times without resetting.

In window mode, Watch Statement commands can be entered either in the dialog window or with menu selections. Current watch statements are shown in a watch window that appears between the menu bar and the source window.

In sequential mode, the Watch, Tracepoint, and Watchpoint commands can be used, but since there is no watch window, you cannot see the watch statements and their values. You must use the Watch List command to examine the current watch statements.

Note

In order to set a watch statement containing a local variable, you must be in the function where the variable is defined. If the current line is not in the function, the CodeView debugger displays the message `UNKNOWN SYMBOL`. When you exit from a function containing a local variable referenced in a watch statement, the value of the statement is displayed as `UNKNOWN SYMBOL`. When you reenter the function, the local variable again has a value. With the C expression evaluators, you can avoid this limitation by using the period operator to specify both the function and the variable. For example, enter **main.x** instead of just **x**.

Setting Watch-Expression and Watch-Memory Statements

The Watch command is used to set a watch statement that specifies an expression (watch-expression statement) or a range of addresses in memory (watch-memory statement). The value or values specified by this watch statement are shown in the watch window. The watch window is updated to show new values each time the value of the watch statement changes during program execution. Since the watch window does not exist in sequential mode, you must use the Watch List command to examine the values of watch statements.

When setting a watch expression, you can specify the format in which the value is displayed. Type the expression followed by a comma and a format specifier. If you do not give a format specifier, the CodeView debugger displays the value in a default format. See "Display Expression Command" in Chapter 7 for more information about type specifiers and the default format.

Keyboard

To set a watch-expression statement with a keyboard command, press <ALT>**w** to open the Watch menu, and then type **A** (uppercase or lowercase) to select Add Watch. You can also select the Add Watch command directly by pressing <CTL>**w**. A dialog box appears, asking for the expression to be watched. Type the expression and press the <RETURN> key.

You cannot use the keyboard version of the command to specify a range of memory to be watched, as you can with the dialog version.

Dialog

To set a watch-expression statement or watch-memory statement with a dialog command, enter a command line with the following syntax:

W? *<expression>*[,*<format>*]	Watch expression
W[*<type>*] *<range>*	Watch memory

Setting Watch-Expression and Watch-Memory Statements

An *expression* used with the Watch command can be either a simple variable or a complex expression using several variables and operators. The expression should be no longer than the width of the watch window. The characters permitted for *format* correspond to format arguments used in a C **printf** function call. See ''Display Expression Command'' in Chapter 7 for more information on format arguments.

When watching a memory location, *type* is a one-letter size specifier from the following list:

Specifier	**Size**
None	Default type
B	Byte
A	ASCII
I	Integer (signed decimal word)
U	Unsigned (unsigned decimal word)
W	Word
D	Double word
S	Short real
L	Long real
T	10-byte real

If no type size is specified, the default type used is the last type used by a Dump, Enter, Watch Memory, or Tracepoint Memory command. If none of these commands has been used during the session, the default type is byte.

The data is displayed in a format similar to that used by the Dump commands (see ''Display Expression Command'' in Chapter 7 for more information on format arguments). The *range* can be any length, but only one line of data is displayed in the watch window. If you do not specify an ending address for the range, the default range is one object.

Examples

The following three examples display watch statements in the watch window.

```
W? n
```

The example above displays the current value of the variable *n*.

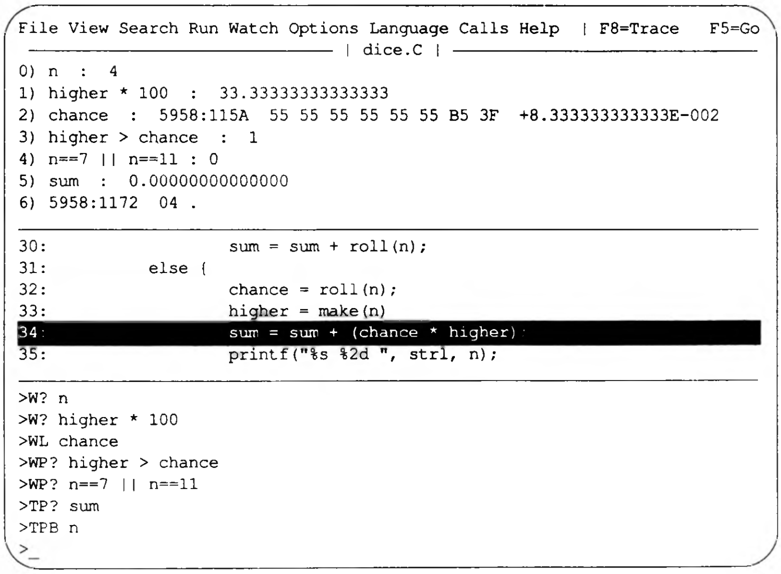
```
W? higher * 100
```

The example above displays the value of the expression **higher * 100**.

```
WL chance
```

The example above displays the double-precision floating-point **chance**, first showing exactly how it is stored in memory. (The command **W? chance** would display the value of **chance** but not any actual bytes of memory.)

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



```
File View Search Run Watch Options Language Calls Help  | F8=Trace   F5=Go
------------------------------------- | dice.C | ------------------------------
0) n  :  4
1) higher * 100  :  33.33333333333333
2) chance  :  5958:115A  55 55 55 55 55 55 B5 3F  +8.333333333333E-002
3) higher > chance  :  1
4) n==7 || n==11 : 0
5) sum  :  0.00000000000000
6) 5958:1172  04 .

                      sum = sum + roll(n);
              else {
                      chance = roll(n);
                      higher = make(n)
                      sum = sum + (chance * higher)
                      printf("%s %2d ", strl, n);

>W? n
>W? higher * 100
>WL chance
>WP? higher > chance
>WP? n==7 || n==11
>TP? sum
>TPB n
>_
```

The first three items in the watch window are simple watch statements. They display values but never cause execution to break.

The next two items are watchpoints; they cause execution to break whenever they evaluate to true (nonzero). The fourth item breaks execution whenever *higher* is greater than *chance*, and the fifth item breaks execution whenever *n* is equal to 7 or 11. Setting watchpoints is described in detail later in this chapter.

The last two items are tracepoints, which cause execution to break whenever any bytes change within a specified area of memory. The sixth item breaks execution whenever the value of *sum* changes; the seventh item breaks execution whenever there is a change in the first byte at the address of *n*. Setting tracepoints is described in detail later in this chapter.

Setting Watchpoints

The Watchpoint command is used to set a conditional breakpoint called a watchpoint. A watchpoint breaks program execution when the expression described by its watch statement becomes true. You can think of watchpoints as ``break when'' points, since the break occurs when the specified expression becomes true (nonzero).

A watch statement created by the Watchpoint command describes the expression that is watched and compared to 0. The statement remains in memory until you delete it or quit the CodeView debugger. Any valid CodeView expression can be used as the watchpoint expression as long as the expression is not wider than the watch window.

In window mode, watchpoint statements and their values are displayed in high-intensity text in the watch window. In sequential mode, there is no watch window, so the values of watchpoint statements can only be displayed with the Watch List command (see the section ``Listing Watchpoints and Tracepoints'' for more information).

Although watchpoints can be any valid CodeView expression, the command works best with expressions that use the relational operators (such as < and > for C. Relational expressions always evaluate to false (zero) or true (nonzero). Care must be taken with other kinds of expressions when used as watchpoints, because the watchpoints breaks execution whenever they do not equal precisely zero. For example, your program might use a loop variable **I**, which ranges from 1 to 100. If you entered **I** as a watchpoint, then it would always suspend program execution, since **I** is never equal to 0. However, the relational expression **I>90** (or **I.GT.90**) would not suspend program execution until **I** exceeded 90.

Keyboard

To execute the Watchpoint command with a keyboard command, press <ALT>**w** to open the Watch menu, and then press <ALT>**w** to select Watchpoint. A dialog box appears, asking for the expression to be watched. Type the expression and press the <RETURN> key.

Setting Watchpoints

Dialog

To set a watchpoint using a dialog command, enter a command line with the following syntax:

WP? *<expression>*[,*<format>*]

The *expression* can be any valid CodeView expression (usually a relational expression). You can enter a format specifier, but there is little reason to do so, since the expression value is normally either 1 or 0.

Examples

The following dialog commands display two watch statements (watchpoints) in the watch window:

```
WP? higher > chance       ;* C example
```

The examples above instruct the CodeView debugger to break execution when the variable *higher* is greater than the variable *chance*. After setting this watchpoint, you could use the Go command to execute until the condition becomes true.

```
WP? n==7 || n==11           ;* C example
```

The example above instructs the CodeView debugger to break execution when the variable *n* is equal to 7 or 11.

Note

C displays a numerical result in response to a Boolean expression (0 being equivalent to false, nonzero to true).

Note

Setting watchpoints significantly slows execution of the program being debugged. The CodeView debugger checks if the expression is true each time a source line is executed in source mode, or each time an instruction is executed in assembly mode. Be careful when setting watchpoints near large or nested loops. A loop that executes almost instantly when run normally can take many minutes if executed from within the debugger with several watchpoints set.

Tracepoints do not slow CodeView execution as much as watchpoints, so you should use tracepoints when possible. For example, although you can set a watchpoint on a Boolean variable (*WP? moving*), a on the same variable (*TP? moving*) has essentially the same effect and does not slow execution as much.

If you enter a seemingly endless loop, press <DEL> to exit. You soon learn the size of loop you can safely execute when watchpoints are set.

Setting Tracepoints

The Tracepoint command is used to set a conditional breakpoint called a tracepoint. A tracepoint breaks program execution when the value of a specified expression or range of memory changes.

The watch statement created by the Tracepoint command describes the expression or memory range to be watched and tested for change. The statement remains in memory until you delete it or quit the CodeView debugger.

In window mode, tracepoint statements and their values are shown in high-intensity text in the watch window. In sequential mode, there is no watch window, so the values of tracepoint statements can only be displayed with the Watch List command (see the section ''Listing Watchpoints and Tracepoints'' in this chapter for more information).

An expression used with the Tracepoint command must evaluate to an ''lvalue.'' In other words, the expression must refer to an area of memory rather than a constant. Furthermore, the area of memory must be not more than 128 bytes in size. For example, **i==10** would be invalid because it is either 1 (true) or 0 (false) rather than a value stored in memory. The expression **sym1+sym2** is invalid because it is the calculated sum of the value of two memory locations. The expression **buffer** would be invalid if **buffer** is an array of 130 bytes, but valid if the array is 120 bytes. Note that if **buffer** is declared as an array of 64 bytes, then the Tracepoint command given with the expression **buffer** checks all 64 bytes of the array. The same command given with the C expression **buffer[32]**, means that only one byte (the 33rd) is checked.

Note

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

```
>X? i
3A79:0264 int           div()
            SI       int            i
>?i
10
>WP? @SI!=10
>
```

When setting a tracepoint expression, you can specify the format in which the value is displayed. Type the expression followed by a comma and a type specifier. If you do not give a type specifier, the CodeView debugger displays the value in a default format. See "Display Expression Command" in Chapter 7 for more information about type specifiers and the default format.

Keyboard

To set a tracepoint-expression statement with a keyboard command, press <ALT>**w** to open the Watch menu, and then press <ALT>**t** to select Trace point. A dialog box appears, asking for the expression to be watched. Type the expression and press the <RETURN> key.

You cannot use the keyboard version of the command to specify a range of memory to be watched, as you can with the dialog version.

Setting Tracepoints

Dialog

To set a tracepoint with a dialog command, enter a command line with one of the following forms of syntax:

TP? *<expression>*,[*<format>*]	@Tracepoint expression
TP[*<type>*] *<range>*	@Tracepoint memory

An *expression* used with the Tracepoint command can be either a simple variable or a complex expression using several variables and operators. The expression should not be longer than the width of the watch window. You can specify *format* using a C **printf** type specifier if you do not want the value to be displayed in the default format (decimal for integers or floating point for real numbers). See ''Display Expression Command'' in Chapter 7 for more information on format arguments.

In the memory-tracepoint form, *range* must be a valid address range and *type* must be a one-letter memory-size specifier. If you specify only the start of the range, the CodeView debugger displays one object as the default.

Although no more than one line of data is displayed in the watch window, the range to be checked for change can be any size up to 128 bytes. The data is displayed in the format used by the Dump commands (see ''Display Expression Command,'' in Chapter 7 for more information on format arguments). The valid memory-size specifiers are listed below:

Specifier	**Size**
None	Default type
B	Byte
A	ASCII
I	Integer (signed decimal word)
U	Unsigned (unsigned decimal word)
W	Word
D	Double word
S	Short real

L	Long real
T	10-byte real

The default type used if no type size is specified is the last type used by a Dump, Enter, Watch Memory, or Tracepoint Memory command. If none of these commands has been used during the session, the default type is byte.

Examples

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

```
TP? sum
```

The example above instructs the CodeView debugger to suspend program execution whenever the value of the variable *sum* changes.

```
TPB n
```

The example above instructs the CodeView debugger to suspend program execution whenever the first byte at the address of *n* changes; the address of this byte and its contents are displayed. The value of *n* may change because of a change in the *second* byte at the address of *n*; but that change (by itself) would have no effect on this tracepoint.

Note

Setting tracepoints significantly slows execution of the program being debugged. The CodeView debugger has to check to see if the expression or memory range has changed each time a source line is executed in source mode or each time an instruction is executed in assembly mode. However, tracepoints do not slow execution as much as do watchpoints.

Be careful when setting tracepoints near large or nested loops. A loop that executes almost instantly when run from the operating system can take many minutes if executed from within the debugger with several tracepoints set. If you enter a seemingly endless loop, press <DEL> to exit. Often you can tell how far you went in the loop by the value of the tracepoint when you exited.

Deleting Watch Statements

The Watch Delete command enables you to delete watch statements that were set previously with the Watch, Watchpoint, or Tracepoint command.

When you delete a watch statement in window mode, the statement disappears and the watch window closes around it. For example, if there are three watch statements in the window and you delete statement 1, the window is redrawn with one less line. Statement 0 remains unchanged, but statement 2 becomes statement 1. If there is only one statement, the window disappears.

Keyboard

To execute the Delete Watch command with a keyboard command, press <ALT>**w** to open the Watch menu, and then type **D** (uppercase or lowercase) to select Delete Watch. You can also select the Delete Watch command directly by pressing <CTL>**u**. A dialog box appears, containing all the watch statements. Use the UP and DOWN arrow keys to move the cursor to the statement you want to delete, and then press the <RETURN> key. The dialog box disappears, and the watch window is redrawn without the watch statement.

You can also delete all the statements in the watch window at once, simply by selecting the Delete All selection. Do this by pressing **L** (uppppercase or lowercase) after the Watch menu is open.

Dialog

To delete watch statements with a dialog command, enter a command line with the following syntax:

> **Y** <*number*>

When you set a watch statement, it is automatically assigned a number (starting with 0). In window mode, the number appears to the left of the watch statement in the watch window. In sequential mode, you can use the Watch List (**W**) command to view the numbers of current watch statements.

You can delete existing watch statements by specifying the *number* of the statement you want to delete with the Delete Watch command. (The **Y** is a mnemonic for "yank.")

Deleting Watch Statements

You can use the asterisk (*) to represent all watch statements.

Examples

```
>Y 2
>
```

The command above deletes watch statement 2.

```
>Y *
>
```

The command above deletes all watch statements and closes the watch window.

Listing Watchpoints and Tracepoints

The Watch List command lists all previously set watchpoints and with their assigned numbers and their current values.

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

Keyboard

The Watch List command cannot be executed with a keyboard command.

Dialog

To list watch statements with a dialog command, enter a command line with the following syntax:

W

The display is the same as the display that appears in the watch window in window mode.

Example

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

Note

The command letter for the Watch List command is the same as the command letter for the memory version of the Watch command when no memory size is given. The difference between the commands is that the Watch List command never takes an argument. The Watch command always requires at least one argument.

Assembly Examples

By default, assembly source modules are debugged with the C-expression evaluator. Therefore, refer to the C examples for appropriate syntax for entering watch expressions.

In addition, however, certain C expressions tend to be more useful for debugging assembly modules. The following examples show some typical cases used with watch and tracepoint commands.

Examples

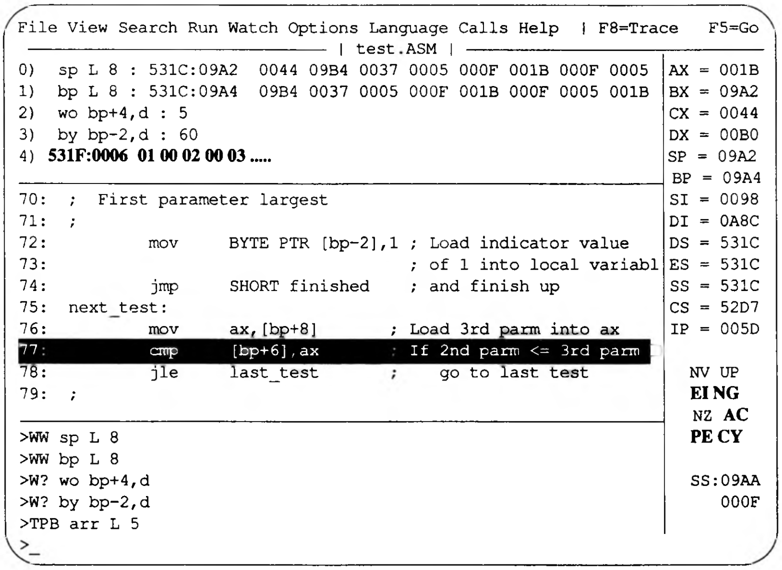
```
>WW sp L 8
>WW bp L 8
>W? wo bp+4,d
>W? by bp-2,d
>TPW arr L 5
>
```

The first two examples watch a range of memory. The watch command **WW sp L 8** is particularly useful because it causes the debugger to watch the stack dynamically; the debugger continually displays the first eight words on the top of the stack as items are pushed and popped. The expression **WW bp L 8** is similar; it causes the debugger to watch the first eight words in memory pointed to by **BP** (the framepointer).

The third example, **W? wo bp+4,d**, is useful if you are using the stack to pass parameters. In this case, the position on the stack four bytes above **BP** holds one of three integer parameters. The **WO** operator returns the same value as the assembler expression **WORD PTR [bp+4]**; the result is displayed in decimal.

You must use the expression **bp+4** in order to watch this parameter; you cannot specify a parameter by name. The assembler does not emit symbolic information for parameters. The fourth command, **W? by bp-2,d**, is similar to the third, but instead of watching a parameter, this command watches a local variable. The operator **BY** returns the same value as the assembler expression **BYTE PTR [bp-2]**.

The final example sets a tracepoint on a range of memory, which corresponds to the first five words of the array *arr*. Range arguments for tracepoint and watch expressions are particularly useful for large data structures, such as arrays. The five examples above produce the following screen, when entered in a CodeView debugging session:



```
File View Search Run Watch Options Language Calls Help  | F8=Trace    F5=Go
 ────────────────────────────── | test.ASM | ──────────────────────────────
0)  sp L 8 : 531C:09A2  0044 09B4 0037 0005 000F 001B 000F 0005 │AX = 001B
1)  bp L 8 : 531C:09A4  09B4 0037 0005 000F 001B 000F 0005 001B │BX = 09A2
2)  wo bp+4,d : 5                                               │CX = 0044
3)  by bp-2,d : 60                                              │DX = 00B0
4) 531F:0006  01 00 02 00 03 .....                              │SP = 09A2
────────────────────────────────────────────────────────────────│BP = 09A4
    ;  First parameter largest                                  │SI = 0098
    ;                                                           │DI = 0A8C
            mov     BYTE PTR [bp-2],1 ; Load indicator value    │DS = 531C
                                      ; of 1 into local variabl│ES = 531C
            jmp     SHORT finished    ; and finish up           │SS = 531C
    next_test:                                                  │CS = 52D7
            mov     ax,[bp+8]        ; Load 3rd parm into ax    │IP = 005D
            cmp     [bp+6],ax        ; If 2nd parm <= 3rd parm  │
            jle     last_test        ;    go to last test       │  NV UP
    ;                                                           │  EI NG
────────────────────────────────────────────────────────────────│  NZ AC
>WW sp L 8                                                      │  PE CY
>WW bp L 8                                                      │
>W? wo bp+4,d                                                   │  SS:09AA
>W? by bp-2,d                                                   │     000F
>TPB arr L 5                                                    │
>_
```

Chapter 10

Examining Code

Introduction

Several CodeView commands allow you to examine program code or data related to code. The following commands are discussed in this chapter:

Command	**Action**
Set Mode (**S**)	Sets format for code displays
Unassemble (**U**)	Displays assembly instructions
View (**V**)	Displays source lines
Current Location (**.**)	Displays the current location line
Stack Trace (**K**)	Displays routines or procedures

Set Mode Command

The Set Mode command sets the mode in which code is displayed. The two basic display modes are source mode, in which the program is displayed as source lines, and assembly mode, in which the program is displayed as assembly-language instructions. These two modes can be combined in mixed mode, in which the program is displayed with both source lines and assembly-language instructions.

In sequential mode, there are three display modes: source, assembly, and mixed. These modes affect the output of commands that display code (Register, Trace, Program Step, Go, Execute, and Unassemble).

In window mode, these same display modes are available, but affect what kind of code appears in the display window.

Source and mixed modes are only available if the executable file contains symbols in the CodeView format. Programs that do not contain symbolic information are displayed in assembly mode.

Keyboard

To change the display mode with a keyboard command, press the **F3** key. This rotates the mode to the next setting; you may need to press **F3** twice to get the desired mode. This command works in either window or sequential mode. In sequential mode, the word *source*, *mixed*, or *assembly* is displayed to indicate the new mode.

Dialog

To set the display mode from the dialog window, enter a command line with the following syntax:

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

If the plus sign is specified (S+), source mode is selected, and the word **source** is displayed.

If the minus sign is specified (S-), assembly mode is selected, and the word **assembly** is displayed. In window mode, the display includes any assembly options, except the Mixed Source option, previously toggled on from the Options menu. The Mixed Source option is always turned off by the S- command.

If the ampersand is specified (**S&**), mixed mode is selected, and the word **mixed** is displayed. In window mode, the display includes any assembly options previously toggled on from the Options menu. In addition, the Mixed Source option is turned on by the **S&** command.

If no argument is specified (**S**), the current mode (*source*, *assembly*, or *mixed*) is displayed.

The Unassemble command in sequential mode is an exception in that it displays mixed source and assembly with both the source (**S+**) and mixed (**S&**) modes. When you enter the dialog version of the Set Mode command, the CodeView debugger outputs the name of the new display mode: *source*, *assembly*, or *mixed*.

Examples

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

The examples above show the source mode being changed to *source*, *assembly*, and *mixed*. In window mode, the commands change the format of the display window. In sequential mode, the commands change the output from the commands that display code (Register, Trace, Program Step, Go, Execute, and Unassemble). See the sections on individual commands for examples of how they are affected by the display mode.

Unassemble Command

The Unassemble command displays the assembly-language instructions of the program being debugged. It is most useful in sequential mode, where it is the only method of examining a sequence of assembly-language instructions. In window mode it can be used to display a specific portion of assembly-language code in the display window.

Note

Occasionally, code similar to the following is displayed:

```
FE30   ???   Byte Ptr  [BX + SI]
```

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

Keyboard

The Unassemble command has no direct keyboard equivalent, but you can view unassembled code at any time by changing the mode to assembly or mixed (see the section "Set Mode Command" in this chapter for more information).

Dialog

To display unassembled code using a dialog command, enter a command line with the following syntax:

U [<*address*> | <*range*>]

The effect of the command varies depending on whether you are in sequential or window mode.

In sequential mode, if you do not specify *address* or *range*, the disassembled code begins at the current unassemble address and shows the next eight lines of instructions. The unassemble address is the address of the instruction after the last instruction displayed by the previous Unassemble command. If the Unassemble command has not been used during the session, the unassemble address is the current instruction.

If you specify an *address*, the disassembly starts at that address and shows the next eight lines of instructions. If you specify a *range*, the instructions within the range are displayed.

The sequential mode format of the display depends on the current display mode (see ''Set Mode Command'' for more information). If the mode is source (S+) or mixed (**S&**), the CodeView debugger displays source lines mixed with unassembled instructions. One source line is shown for each corresponding group of assembly-language instructions. If the display mode is assembly, only assembly-language instructions are shown.

In window mode, the Unassemble command changes the mode of the display window to assembly. The display format reflects any options previously set from the Options menu. There is no output to the dialog window. If *address* is given, the instructions in the display window begin at the specified address. If *range* is given, only the starting address is used. If no argument is given, the debugger scrolls down and displays the next screen of assembly-language instructions.

Note

The 80286 protected-mode mnemonics (also available with the 80386) cannot be displayed with the Unassemble command.

Examples

```
>S&
mixed
>U 0x11
49D0:0011 35068E      XOR  AX,__sqrtjmptab+8cd4  (8E06)
49D0:0014 189A230     SBB  Byte Ptr [BP+SI+0023],BL
49D0:0018 FC          CLD
49D0:0019 49          DEC  CX
49D0:001A CD351ED418  INT  35 ;FSTP  DWord Ptr [__fpinit+ee (18D4)]
49D0:001F CD3D        INT  3D ;FWAIT
7:     A = 0.0
49D0:0021 CD35EE      INT  35 ;FLDZ
```

The sequential mode example above sets the mode to mixed and unassembles eight lines of machine code, plus whatever source lines are encountered within those lines. The display would be the same if the mode were source.

The example demonstrates sequential mode.

```
>S-
assembly
>U 0x11
49D0:0011 35068E     XOR  AX,__sqrtjmptab+8cd4 (8E06)
49D0:0014 189A2300   SBB  Byte Ptr [BP+SI+0023],BL
49D0:0018 FC         CLD
49D0:0019 49         DEC  CX
49D0:001A CD351ED418 INT  35 ;FSTP  DWord Ptr [__fpinit+ee (18D4)]
49D0:001F CD3D       INT  3D ;FWAIT
49D0:0021 CD35EE     INT  35 ;FLDZ
>
```

The sequential mode example above sets the mode to assembly and repeats the same command.

View Command

The View command displays the lines of a text file (usually a source module or include file). It is most useful in sequential mode, where it is the only method of examining a sequence of source lines. In window mode, the View command can be used to page through the source file or to load a new source file.

Keyboard

To load a new source file with a keyboard command, press <ALT>**f** to open the File menu, then press L to select Load. A dialog box appears, asking for the name of the file you wish to load. Type the name of the file, and press the <RETURN> key. The new file appears in the display window.

The paging capabilities of the View command have no direct keyboard equivalent, but you can move about in the source file by first putting the cursor in the display window with the **F6** key, then pressing the <PgUp>, <PgDn>, <HOME>, <END>, UP ARROW, and DOWN ARROW keys. See ''Controlling Program Execution with Keyboard Commands'' in Chapter 3 for more information.

Dialog

To display source lines using a dialog command, enter a command line with the following syntax:

V [<*expression*>]

Since addresses for the View command are often specified as a line number (with an optional source file), a more specific syntax for the command would be as follows:

V [.[<*filename*>:]<*linenumber*>]

The effect of the command varies, depending on whether you are in sequential or window mode.

In sequential mode, the View command displays eight source lines. The starting source line is one of the following:

- The current source line if no argument is given.
- The specified *linenumber*. If *filename* is given, the specified file is loaded, and the *linenumber* refers to lines in it.
- The address that *expression* evaluates to. For example, *expression* could be a procedure name or an address in the *segment:offset* format. The code segment is assumed if no segment is given.

In sequential mode, the View command is not affected by the current display mode (source, assembly, or mixed); source lines are displayed regardless of the mode.

In window mode, if you enter the View command while the display mode is assembly, the CodeView debugger automatically switches back to source mode. If you give *linenumber* or *expression*, the display window are redrawn 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*, the specified file is loaded.

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 at the top.

Note

The View command with no argument is similar to pressing the <PgDn> key. The difference is that pressing the <PgDn> key enables you to scroll down one more line.

Examples

```
>V .math.c:30    ;* Example 1, C source code
30:                 register int j;
31:
32:                 for (j = q; j >= 0; j--)
33:                         if (t[j] + p[j] > 9) {
34:                                 p[j] += t[j] - 10;
35:                                 p[j-1] += 1;
36:                         } else
37:                                 p[j] += t[j];
>
```

Example 1 loads the source file *math.c* and displays eight source lines starting at line **30**.

Current Location Command

The Current Location command displays the source line or assembly-language instruction corresponding to the current program location.

Keyboard

The Current Location command cannot be executed with a keyboard command.

Dialog

To display the current location line using a dialog command, enter a command line with the following syntax (a period only):

In sequential mode, the command displays the current source line. The line is displayed regardless of whether the current debugging mode is source or assembly. If the program being debugged has no symbolic information, the command is ignored.

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

For example, if you are in window mode and have executed the program being debugged to somewhere near the start of the program, but you have scrolled the display to a point near the end, the Current Location command returns the display to the current program location.

Example

```
>.
MINDAT = 1.0E6
>
```

The example above illustrates how to display the current source line in sequential mode. The same command in window mode would not produce any output, but it could change the text that is shown in the display window.

Stack Trace Command

The Stack Trace command allows you to display routines that have been called during program execution (see note below). The first line of the display shows the name of the current routine. The succeeding lines (if any) list any other routines that were called to reach the current address. The dialog version of the Stack Trace command also displays the source lines where each routine was called.

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

The term ''stack trace'' is used because, as each routine is called, its address and arguments are stored on (pushed onto) the program stack. Therefore, tracing through the stack shows the currently active routines. With C programs, the **main** routine is always near the bottom of the stack. Only routines called by the main program are displayed.

Note

> This discussion uses the term ''routines,'' which is a general term for functions, subroutines, procedures, subprograms, and function procedures. Each of which uses the stack to transfer control to an independent program unit. In assembly mode, the term ''procedure'' may be more accurate.
>
> If you are using the CodeView debugger to debug assembly-language programs, the Stack Trace command works only if you call procedures with the calling convention appropriate to the procedure's language.

Keyboard

To view a stack trace with a keyboard command, press <ALT>**c** to open the Calls menu. The menu shows the current routine at the top, and other routines below it in the reverse order in which they were called; for example, the first routine called is at the bottom. The values of any routine arguments are shown in parentheses following the routine.

If you want to view one of the routines that was previously called, select the routine by moving the cursor with the arrow keys and then pressing <RETURN> , or by typing the number or letter to the left of the routine. The effect of selecting a routine in the Calls menu is to cause the debugger to display that routine. The cursor is on the last statement that was executed in the routine.

Dialog

To display a stack trace with a dialog command, enter a command line with the following syntax:

K

The output from the Stack Trace dialog command lists the routines in the reverse order in which they were called. The arguments to each routine are shown in parentheses. Finally, the line number from which the routine was called is shown.

You can enter the line number as an argument to the View or Unassemble command if you want to view code at the point where the routine was called.

In window mode, the output from the Stack Trace dialog command appears in the dialog window.

C Example

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

The example above shows the routines on the stack in the reverse order in which they were called. Since **analyze** is on the top, it has been called most recently; in other words, it is the current routine.

Each routine is shown with the arguments it was passed, along with the last source line that it had been executing. Note that **main** is shown with the command line arguments *argc* (which is equal to 2) and *argv* (which is a pointer equal to 5098 decimal). Since the language is C, **main** is always on the bottom of the stack.

Chapter 11

Modifying Code or Data

Introduction

The CodeView debugger provides the following commands for modifying code or data in memory:

Command	**Action**
Assemble (**A**)	Modifies code
Enter (**E**)	Modifies memory, usually data
Register (**R**)	Modifies registers and flags
Fill Memory (**F**)	Fills a block of memory
Move Memory (**M**)	Copies one block of memory to another

These commands change code temporarily. You can use the alterations for testing in the CodeView debugger, but you cannot save them or permanently change the program. To make permanent changes, you must modify the source code and recompile.

Assemble Command

The Assemble command assembles 8086-family (8086, 8087, 8088, 80186, 80287, and 80286 unprotected) instruction mnemonics and places the resulting instruction code into memory at a specified address. The only 8086-family mnemonics that cannot be assembled are 80286 protected-mode mnemonics. In addition, the debugger also assembles 80386 instructions.

Note

The effects of the Assemble command are temporary. Any instructions that you assemble are lost as soon as you exit the program.

The instructions you assemble are also lost when you restart the program with the Start or Restart command, because the original code is reloaded on top of memory you may have altered.

To test the results of an Assemble command, you may need to manipulate the **IP** register (and possibly the **CS** register) to the starting address of the instructions you have assembled. If you do this, you must use the Current Line command (.) to reset the debugger's internal variables so that it traces properly.

Keyboard

The Assemble command cannot be executed with a keyboard command.

Dialog

To assemble code using a dialog command, enter a command line with the following syntax:

A [<*address*>]

If *address* is specified, the assembly starts at that address; otherwise the current assembly address is assumed.

The assembly address is normally the current address (the address pointed to by the **CS** and **IP** registers). However, when you use the Assemble command, the assembly address is set to the address immediately following the last assembled instruction. When you enter any command that executes code (Trace, Program Step, Go, or Execute), the assembly address is reset to the current address.

When you type the Assemble command, the assembly address is displayed. The CodeView debugger then waits for you to enter a new instruction in the standard 8086-family instruction-mnemonic form. You can enter instructions in uppercase, lowercase, or both.

To assemble a new instruction, type the desired mnemonic and press the <RETURN> key. The CodeView debugger assembles the instruction into memory and displays the next available address. Continue entering new instructions until you have assembled all the instructions you want. To conclude assembly and return to the CodeView prompt, press the <RETURN> key only.

If an instruction you enter contains a syntax error, the debugger displays the message `^ Syntax error`, redisplays the current assembly address, and waits for you to enter a correct instruction. The caret symbol in the message points to the first character the CodeView debugger could not interpret.

The following eight principles govern entry of instruction mnemonics:

1. The far-return mnemonic is **RETF**.

2. String mnemonics must explicitly state the string size. For example, **MOVSW** must be used to move word strings, and **MOVSB** must be used to move byte strings.

3. The CodeView debugger automatically assembles short, near, or far jumps and calls, depending on byte displacement to the destination address. These may be overridden with the **NEAR** or **FAR** prefix, as shown in the following examples:

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

   The **NEAR** prefix can be abbreviated to **NE**, but the **FAR** prefix cannot be abbreviated. The examples above use the C notation for hexadecimal numbers.

4. The CodeView debugger cannot determine whether some operands refer to a word memory location or to a byte memory location. In

these cases, the data type must be explicitly stated with the prefix **WORD PTR** or **BYTE PTR**. Acceptable abbreviations are **WO** and **BY**. Examples are shown below:

```
MOV     WORD PTR [BP],1
MOV     BYTE PTR [SI-1],symbol
MOV     WO PTR [BP],1
MOV     BY PTR [SI-1],symbol
```

5. The CodeView debugger cannot determine whether an operand refers to a memory location or to an immediate operand. The debugger uses the convention that operands enclosed in square brackets refer to memory. Two examples are shown below:

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

   The first statement moves **21** hexadecimal into **AX**. The second statement moves the data at offset **0x21** hexadecimal into **AX**.

6. The CodeView debugger supports all forms of indirect register instructions, as shown in the following examples:

   ```
   ADD     BX,[BP+2].[SI-1]
   POP     [BP+DI]
   PUSH    [SI]
   ```

7. All instruction-name synonyms are supported. If you assemble instructions and then examine them with the Unassemble command (U), the CodeView debugger may show synonymous instructions, rather than the ones you assembled, as shown in the following examples:

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

8. Do not assemble and execute 8087 or 80287 instructions if your system is not equipped with one of these math coprocessor chips. If you try to execute the **WAIT** instruction without the appropriate chip, for example, your system will crash.

Example

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

The Unassemble command (U) is used to show the instruction before and after the assembly.

You can modify a portion of code for testing, as in the example, but you cannot save the modified program. You must modify your source code and recompile.

Enter Commands

The CodeView debugger has several commands for entering data to memory. You can use these commands to modify either code or data, though code can usually be modified more easily with the Assemble command (**A**). The Enter commands are listed below:

Command	**Command Name**
E	Enter (size is the default type)
EB	Enter Bytes
EA	Enter ASCII
EI	Enter Integers
EU	Enter Unsigned Integers
EW	Enter Words
ED	Enter Double Words
ES	Enter Short Reals
EL	Enter Long Reals
ET	Enter 10-Byte Reals

Keyboard

The Enter commands cannot be executed with keyboard commands.

Dialog

To enter data (or code) to memory with a dialog command, enter a command line with the following syntax:

E [<*type*>] <*address*> [<*list*>]

The *type* is a one-letter specifier that indicates the type of the data to be entered. The *address* indicates where the data is entered. If no segment is given in the address, the data segment (**DS**) is assumed.

The *list* can consist of one or more expressions that evaluate to data of the size specified by *type* (the expressions in the list are separated by spaces). This data is entered to memory at *address*. If one of the values in the list is invalid, an error message is displayed. The values preceding the error are entered; values at and following the error are not entered.

The expressions in the list are evaluated in the current radix, regardless of the size and type of data being entered. For example, if the radix is 10 and you give the value 10 in a list with the Enter Words command, the decimal value 10 is entered even though word values are normally entered in hexadecimal. This means that the Enter Words, Enter Integers, and Enter Unsigned Integers commands are identical when used with the list method, since two-byte data are being entered for each command.

If *list* is not given, the CodeView debugger prompts for values to be entered to memory. Values entered in response to prompts are accepted in hexadecimal for the Enter Bytes, Enter ASCII, Enter Words, and Enter Double Words commands. The Enter Integers command accepts signed decimal integers, while the Enter Unsigned Integers command accepts unsigned decimal integers. The Enter Short Reals, Enter Long Reals, and Enter 10-Byte Reals commands accept decimal floating-point values.

With the prompting method of data entry, the CodeView debugger prompts for a new value at *address* by displaying the address and its current value. As explained below, you can then replace the value, skip to the next value, return to a previous value, or exit the command.

- To replace the value, type the new value after the current value.
- To skip to the next value, press the <SPACE> bar . Once you have skipped to the next value, you can change its value or skip to the following value. If you pass the end of the display, the CodeView debugger displays a new address to start a new display line.
- To return to the preceding value, type a backslash (\). When you return to the preceding value, the debugger starts a new display line with the address and value.
- To stop entering values and return to the CodeView prompt, press the <RETURN> key. You can exit the command at any time.

Examples

```
>EW PLACE 16 32
```

The example above shows how to enter two word-sized values at the address **PLACE**.

```
>EW PLACE

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

The example above illustrates the prompting method of entering data. When you supply the address where you want to enter data but supply no data to be entered there, the CodeView debugger displays the current value of the address and waits for you to enter a new value. The underscore in this example and the examples below represents the CodeView cursor. You change the value **F3** to the new value 16 (10 hexadecimal) by typing **10** (without pressing the <RETURN> key yet). The value must be typed in hexadecimal for the Enter Words command, as shown below:

```
>EW PLACE

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

You can then skip to the next value by pressing the <SPACE> key. The CodeView debugger responds by displaying the next value, as shown below:

```
>EW PLACE

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

You can then type another hexadecimal value, such as **30**:

```
>EW PLACE

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

To move to the next value, press the <SPACE> key.

```
>EW PLACE

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

Assume that you realize that the last value entered, **30**, is incorrect. You really wanted to enter **20**. You could return to the previous value by typing a backslash. The CodeView debugger starts a new line, starting with the previous value. Note that the backslash is not echoed on the screen:

```
>EW PLACE

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

Type the correct value, **20**:

```
>EW PLACE

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

If this is the last value you want to enter, press the <RETURN> key to stop. The CodeView prompt reappears, as shown below:

```
>EW PLACE

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

Enter Command

Syntax

E *<address>* [*<list>*]

The Enter command enters one or more values into memory at the specified *address*. The data are entered in the format of the default type, which is the last type specified with a Dump, Enter, Watch Memory, or Tracepoint Memory command. If none of these commands has been entered during the session, the default type is bytes.

Use this command with caution when entering values in the list format; values are truncated if you enter a word-sized value when the default type is actually bytes. If you are not sure of the current default type, specify the size in the command.

Note

The Execute command and the Enter command have the same command letter (**E**). The difference is that the Execute command never takes an argument; the Enter command always requires at least one argument.

Enter Bytes Command

Syntax

EB *<address>* [*<list>*]

The Enter Bytes command enters one or more byte values into memory at *address*. The optional *list* can be entered as a list of expressions separated by spaces. The expressions are evaluated and entered in the current radix. If *list* is not given, the CodeView debugger prompts for new values, which must be entered in hexadecimal.

The Enter Bytes command can also be used to enter strings, as described in the section "Enter ASCII Command" in this chapter.

Examples

```
>EB 256 10 20 30
>
```

If the current radix is 10, the above example replaces the three bytes at DS:256, DS:257, and DS:258 with the decimal values **10**, **20**, and **30**. (These three bytes correspond to the hexadecimal addresses DS:0100, DS:0101, and DS:0102.)

```
>EB 256

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

The example above replaces the byte at DS:256 (DS:0100 hexadecimal) with 10 (0A hexadecimal).

Enter ASCII Command

Syntax

EA *<address>* [*<list>*]

The Enter ASCII command works in the same way as the Enter Bytes command (**EB**) described in the section "Enter Bytes Command" in this chapter. The *list* version of this command can be used to enter a string expression.

Example

```
>EA message "File cannot be found"
>
```

In the example above, the string `File cannot be found` is entered starting at the symbolic address **message**. (Note that the double quotation marks are CodeView string delimiters.)

You can also use the Enter Bytes command to enter a string expression, or you can enter nonstring values using the Enter ASCII command.

Enter Integers Command

Syntax

EI *<address>* [*<list>*]

The Enter Integers command enters one or more word values into memory at *address* using the signed-integers format. With the CodeView debugger, a signed integer can be any decimal integer between -32,768 and 32,767.

The optional *list* can be entered as a list of expressions separated by spaces. The expressions are entered and evaluated in the current radix. If *list* is not given, the CodeView debugger prompts for new values, which must be entered in decimal.

Examples

```
>EI 256 -10 10 -20
>
```

If the current radix is 10, the example above replaces the three integers at DS:256, DS:258, and DS:260 with the decimal values **-10**, **10**, and **-20**. (The three addresses correspond to the three hexadecimal addresses DS:0100, DS:0102, and DS:0104.)

```
>EI 256

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

The example above replaces the integer at DS:256 (hexadecimal address DS:0100) with **-10**.

Enter Unsigned Integers Command

Syntax

EU <*address*> [<*list*>]

The Enter Unsigned Integers command enters one or more word values into memory at *address* using the unsigned-integers format. With the CodeView debugger, an unsigned integer can be any decimal integer between 0 and 65,535. The optional *list* can be entered as a list of expressions separated by spaces. The expressions are entered and evaluated in the current radix. If *list* is not given, the CodeView debugger prompts for new values, which must be entered in decimal.

Examples

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

If the current radix is 10, the example above replaces the three unsigned integers at DS:256, DS:258, and DS:260 with the decimal values **10**, **20**, and **30**. (These addresses correspond to the hexadecimal addresses DS:0100, DS:0102, and DS:0104.)

```
>EU 256

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

The example above replaces the integer at DS:256 (DS:0100 hexadecimal) with **10**.

Enter Words Command

Syntax

EW *<address>* [*<list>*]

The Enter Words command enters one or more word values into memory at *address*.

The optional *list* can be entered as a list of expressions separated by spaces. The expressions are entered and evaluated in the current radix. If *list* is not given, the CodeView debugger prompts for new values, which must be entered in hexadecimal.

Examples

```
>EW 256 10 20 30
>
```

If the current radix is 10, the example above replaces the three words at DS:256, DS:258, and DS:260 with the decimal values `10`, `20`, and `30`. (These addresses correspond to the hexadecimal addresses DS:0100, DS:0102, and DS:0104.)

```
>EW 256

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

The example above replaces the integer at DS:256 (DS:0100 hexadecimal) with 10 (0A hexadecimal).

Enter Double Words Command

Syntax

ED *<address>* [*<list>*]

The Enter Double Words command enters one or more double-word values into memory at *address*. Double words are displayed and entered in the *segment:offset* address format; that is, two words separated by a colon (:). If the colon is omitted and only one word entered, only the offset portion of the address is changed.

The optional *list* can be entered as a list of expressions separated by spaces. The expressions are entered and evaluated in the current radix. If *list* is not given, the CodeView debugger prompts for new values, which must be entered in hexadecimal.

Examples

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

If the current radix is 10, the example above replaces the double words at DS:256 (DS:0100 hexadecimal) with the decimal address `8700:12008` (hexadecimal address 21FC:2EE8).

```
>ED 256

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

The example above replaces the offset portion of the double word at DS:256 (DS:0100 hexadecimal) with **2EE9** hexadecimal. Since the segment portion of the address is not provided, the existing segment (**21FC** hexadecimal) is unchanged.

Enter Short Reals Command

Syntax

ES <*address*> [<*list*>]

The Enter Short Reals command enters one or more short-real values into memory at *address*.

The optional *list* can be entered as a list of real numbers separated by spaces. The numbers must be entered in decimal, regardless of the current radix. If *list* is not given, the CodeView debugger prompts for new values, which must be entered in decimal. Short-real numbers can be entered either in floating-point format or in scientific-notation format.

Examples

```
>ES 256 23.479 1/4 -1.65E+4 235
>
```

The example above replaces the four numbers at DS:256, DS:260, DS:264, and DS:268 with the real numbers **23.479**, **0.25**, **-1650.0**, and **235.0**. (These addresses correspond to the hexadecimal addresses DS:0100, DS:0104, DS:0108, and DS:0112.)

```
>ES PI
3DA5:0064   42 79 74 65   7.215589E+022   3.141593
>
```

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

Enter Long Reals Command

Syntax

EL *<address>* [*<list>*]

The Enter Long Reals command enters one or more long-real values into memory at *address.*

The optional *list* can be entered as a list of real numbers separated by spaces. The numbers must be entered in decimal, regardless of the current radix. If *list* is not given, the CodeView debugger prompts for new values, which must be entered in decimal. Long-real numbers can be entered either in floating-point format or in scientific-notation format.

Examples

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

The example above 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 PI
3DA5:0064  42 79 74 65 DC OF 49 40   5.012391E+001  3.141593
>
```

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

Enter 10-Byte Reals Command

Syntax

ET *<address* [*list*]>

The Enter 10-Byte Reals command enters one or more 10-byte-real values into memory at *address*.

The optional *list* can be entered as a list of real numbers separated by spaces. The numbers must be entered in decimal, regardless of the current radix. If *list* is not given, the CodeView debugger prompts for new values, which must be entered in decimal. The numbers can be entered either in floating-point format or in scientific-notation format.

Examples

```
>ET 256 23.479 1/4 -1.65E+4 235
>
```

The example above replaces the four numbers at DS:256, DS:266, DS:276, and DS:286 with the real numbers **23.479**, **0.25**, **-1650.0**, and **235.0**. (These addresses correspond to the hexadecimal addresses DS:0100, DS:010A, DS:0114, and DS:011E.)

```
>ET PI
3DA5:0064  42 79 74 65 DC 0F 49 40 7F BD  -3.292601E-193 3.141593
>
```

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

Fill Memory Command

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

You can enter arguments to the Fill Memory command using any radix.

Keyboard

The Fill Memory command cannot be executed with a keyboard command.

Dialog

To fill an area of memory with values you specify, enter the Fill Memory command as follow:

F *<range> <list>*

The Fill Memory command fills the addresses in the specified *range* with the byte values specified in *list*. The values in the list are repeated until the whole range is filled. (Thus, if you specify only one value, the entire range is filled with that same value.) If the *list* has more values than the number of bytes in the *range*, then the command ignores any extra values.

Examples

```
>F 100 L 100 0    ;* hexadecimal radix assumed
>
```

The first example fills 255 (100 hexadecimal) bytes of memory starting at DS:0100 with the value 0. This command might possibly be used to reinitialize the program's data without having to restart the program.

```
>F table L 64 42 79 74 ;* hexadecimal radix assumed
>
```

The second example fills the 100 (64 hexadecimal) bytes starting at **table** with the following hexadecimal byte values: 42, 79, 74. These three values are repeated until all 100 bytes are filled.

Move Memory Command

The Move Memory command enables you to copy all the values in one block of memory directly to another block of memory of the same size. This command is of most interest to assembly programmers, but can be used by anyone who wants to do large data transfers efficiently. For example, you can use this command to copy all the values in one array to the elements of another.

Keyboard

The Move Memory command cannot be executed with a keyboard command.

Dialog

To copy the values in one block of memory to another, enter the Move Memory command with the following syntax:

M *<range> <address>*

The values in the block of memory specified by *range* are copied to a block of the same size beginning at *address*. All data in *range* are guaranteed to be copied completely over to the destination block, even if the two blocks overlap. However, 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(1) L arsize arr2(1)   ;* FORTRAN example
>
```

In the example above, the block of memory beginning with the first element of **arr1**, and **arsize** bytes long, is copied directly to a block of the same size beginning at the address of the first element of **arr2**. In C, this command would be entered as **M arr1[0] L arsize arr2[0]**.

Register Command

The Register command has two functions: it displays the contents of the central processing unit registers, and it can also change the values of those registers. The modification features of the command are explained in this section. The display features of the Register command are explained in Section 6.7.

Keyboard

The registers cannot be changed with keyboard commands.

Dialog

To change the value of a register with a dialog command, enter a command line with the following syntax:

R [<*registername*>[[=]<*expression*>]]

To modify the value in a register, type the command letter **R** followed by *registername*. The CodeView debugger displays the current value of the register and prompts for a new value. Press the <RETURN> key if you only want to examine the value. If you want to change it, type an expression for the new value and press the <RETURN> key.

As an alternative, you can type both *registername* and *expression* in the same command. You can use the equal sign (=) between *registername* and *expression*, but a space has the same effect.

The register name can be any of the following names: **AX**, **BX**, **CX**, **DX**, **CS**, **DS**, **SS**, **ES**, **SP**, **BP**, **SI**, **DI**, **IP**, or **F** (for flags). If you have a 386-based machine, then the register name can be one of the 32-bit register names shown in Table 5.11.

To change a flag value, supply the register name **F** when you enter the Register command. The command displays the current value of each flag as a two-letter name.

At the end of the list of values, the command displays a dash (-). Enter new values after the dash for the flags you wish to change, then press the <RETURN> key. You can enter flag values in any order. Flags for which new values are not entered remain unchanged. If you do not want to change any flags, simply press the <RETURN> key.

If you enter an illegal flag name, an error message is displayed. The flags preceding the error are changed; flags at and following the error are not changed.

The flag values are shown in Table 11.1.

Table 11.1

Flag-Value Mnemonics

Flag Name	**Set**	**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**

Examples

```
>R IP 256
>
```

The example above changes the **IP** register to the value **256** (0100 hexadecimal).

```
>R AX
AX OEOO
:_
```

The example above displays the current value of the **AX** register and prompts for a new value (the underscore represents the CodeView cursor). You can now type any 16-bit value after the colon.

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

The example above changes the value of **AX** to 256 (in the current radix).

```
>R F UP EI PL
```

The example above shows the command-line method of changing flag values.

```
>R F
NV(OV) UP(DN) EI(DI) PL(NG) NZ(ZR) AC(NA) PE(PO) NC(CY) -OV DI ZR
>R F
OV(NV) UP(DN) DI(EI) PL(NG) ZR(NZ) AC(NA) PE(PO) NC(CY) -
>
```

With the prompting method of changing flag values (shown above), the first mnemonic for each flag is the current value, and the second mnemonic (in parentheses) is the alternate value. You can enter one or more mnemonics at the dash prompt. In the example, the command is given a second time to show the results of the first command.

Chapter 12

Using CodeView System-Control Commands

Introduction

This chapter discusses commands that control the operation of the CodeView debugger. The commands in this category are listed below:

Command	**Action**
Help (**H**)	Displays help
Quit (**Q**)	Returns to System V
Radix (**N**)	Changes radix
Redraw (**@**)	Redraws screen
Screen Exchange (\)	Switches to output screen
Search (**/**)	Searches for regular expression
Shell Escape (**!**)	Starts new shell
Tab Set (**#**)	Sets tab size
Option (**O**)	Views or sets CodeView options
Redirection and related commands	Control redirection of CodeView output or input

The system-control commands are discussed in the following sections.

Help Command

The CodeView debugger has two help systems: a complete on-line-help system available only in window mode, and a syntax summary available with sequential mode.

Keyboard

If you are in window mode, press the **F1** key to enter the complete on-line-help system. If you are in sequential mode, a syntax-summary screen appears when you press **F1** .

Dialog

If you are in window mode, you can view the complete on-line-help system with the following command:

```
H
```

If you are in sequential mode, this command displays a screen containing all CodeView dialog commands with the syntax for each. This screen is the only help available in sequential mode.

Quit Command

The Quit command terminates the CodeView debugger and returns control to the operating system.

Keyboard

To quit the CodeView debugger with a keyboard command, press <ALT>**f** to open the File menu, and then press **X** to select Exit. The CodeView screen is replaced by the operating system screen, with the cursor at the operating system prompt.

Dialog

To quit the CodeView debugger with a dialog command, enter a command line with the following syntax:

```
Q
```

When the command is entered, the CodeView screen is replaced by the standard screen, with the cursor at the shell prompt.

Radix Command

The Radix command changes the current radix for entering arguments and displaying the value of expressions. The default radix when you start the CodeView debugger is 10 (decimal). Radixes 8 (octal) and 16 (hexadecimal) can also be set. Binary and other radixes are not allowed.

The following seven conditions are exceptions; they are not affected by the Radix command:

1. The radix for entering a new radix is always decimal.

2. Format specifiers given with the Display Expression command or any of the Watch Statement commands override the current radix.

3. Addresses output by the Assemble, Dump, Enter, Examine Symbol, and Unassemble commands are always shown in hexadecimal.

4. In assembly mode, all values are shown in hexadecimal.

5. The display radix for Dump, Watch Memory, and Tracepoint Memory commands is always hexadecimal if the size is bytes, words, or double words, and always decimal if the size is integers, unsigned integers, short reals, long reals, or 10-byte reals.

6. The input radix for the Enter commands with the prompting method is always hexadecimal if the size is bytes, words, or double words, and always decimal if the size is integers, unsigned integers, short reals, long reals, or 10-byte reals. The current radix is used for all values given as part of a list, except real numbers, which must be entered in decimal.

7. The register display is always in hexadecimal.

Keyboard

You cannot change the input radix with a keyboard command.

Dialog

To change the input radix with a dialog command, enter a command line with the following syntax:

N[<*radixnumber*>]

The *radixnumber* can be 8 (octal), 10 (decimal), or 16 (hexadecimal). The default radix when you start the CodeView debugger is 10 (decimal), unless your main program is written with the Macro Assembler, in which case the default radix is 16 (hexadecimal). If you give the Radix command with no argument, the debugger displays the current radix.

Examples

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

>N8   ;
>? prime
0153
>

>N16  ;
>? prime
0x006b
>
```

The example aboves show how 107 decimal, stored in the variable *prime*, would be displayed with different radixes.

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

Dialog

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

G [<*breakaddress*>]

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

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

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

Examples

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

```
>G

Program terminated normally (0)
>
```

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

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

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

Redraw Command

The Redraw command can be used only in window mode; it redraws the CodeView screen. This command is seldom necessary, but you might need it if the output of the program being debugged disturbs the CodeView display temporarily.

Keyboard

You cannot redraw the screen using a keyboard command.

Dialog

To redraw the screen with a dialog command, enter a command line with the following syntax:

```
@
```

Screen Exchange Command

The Screen Exchange command allows you to switch temporarily from the debugging screen to the output screen.

The CodeView debugger uses either screen flipping or screen swapping to store the output and debugging screens. See Chapter 2, "Getting Started," for an explanation of flipping and swapping.

Keyboard

To execute the Screen Exchange command with a keyboard command, press the **F4** key. Press any key when you are ready to return to the debugging screen.

Dialog

To execute the Screen Exchange command from the dialog window, enter a command line with the following syntax:

```
\
```

The output screen appears. Press any key when you are ready to return to the debugging screen.

Search Command

The Search command allows you to search for a regular expression in a source file. The expression being sought is specified either in a dialog box or as an argument to a dialog command. Once you have found an expression, you can also search for the next or previous occurrence of the expression.

Regular expressions are patterns of characters that may match one or many different strings. The use of patterns to match more than one string is similar to the shell method of using wild-card characters in file names.

You can use the Search command without understanding regular expressions. Since text strings are the simplest form of regular expressions, you can simply enter a string of characters as the expression you want to find. For example, you could enter **COUNT** if you wanted to search for the word "COUNT" in the source file.

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

For example, you would use * to find *x*y*. The periods in the relational operators must also be preceded by a backslash.

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

Note

When you search for the next occurrence of a regular expression, the CodeView debugger searches to the end of the file, and then wraps around and begins again at the start of the file. This can have unexpected results if the expression occurs only once. When you give the command repeatedly, nothing seems to happen. Actually, the debugger is repeatedly wrapping around and finding the same expression each time.

Search Command

Keyboard

To find a regular expression with a keyboard command, press <ALT>s to open the Search menu, and then press **F** to select Find. A dialog box appears, asking for the regular expression to be found. Type the expression and press the <RETURN> key. The CodeView debugger starts searching at the current cursor position and puts the cursor at the next line containing the regular expression. An error message appears if the expression is not found. If you are in assembly mode, the debugger automatically switches to source mode when the expression is found.

After you have found a regular expression, you can search for the next or previous occurrence of the expression. Press <ALT>s to open the Search menu and then press **N** to select Next or **P** to select Previous. The cursor moves to the next or previous match of the expression.

You can also search the executable code for a label (such as a routine name or an assembly-language label). Press <ALT>s to open the Search menu and then press **L** to select Label. A dialog box appears, asking for the label to be found. Type the label name and press the <RETURN> key. The cursor moves to the line containing the label. This selection differs from other search selections because it searches executable code rather than source code. The CodeView debugger switches to assembly mode, if necessary, to display a label in a library routine or assembly-language module.

Dialog

To find a regular expression using a dialog command, enter a command line with the following syntax:

/[<*regularexpression*>]

If *regularexpression* is given, the CodeView debugger searches the source file for the first line containing the expression. If no argument is given, the debugger searches for the next occurrence of the last regular expression specified.

In window mode, the CodeView debugger starts searching at the current cursor position and puts the cursor at the next line containing the regular expression. In sequential mode, the debugger starts searching at the last source line displayed. It displays the source line in which the expression is found. An error message appears if the expression is not found. If you are in assembly mode, the CodeView debugger automatically switches to source mode when the expression is found.

You cannot search for a label with the dialog version of the Search command, but you can use the View command with the label as an argument for the same effect.

Shell Escape Command

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

Keyboard

To open a shell with a keyboard command, press <ALT>**f** to open the File menu, and then press **D** to select Shell. When you are ready to return to the debugging session, type the command **exit**. The debugging screen appears with the same status it had when you left it.

Dialog

To open a shell using a dialog command, enter a command line with the following syntax:

!*[<command>]*

If you want to exit to the system and execute several programs or commands, enter the command with no arguments. The standard screen appears. You can run programs or shell commands. When you are ready to return to the debugger, type the command **exit**. The debugging screen appears with the same status it had when you left it.

If you want to execute a program or shell command from within the CodeView debugger, enter the Shell Escape command (**!**) followed by the name of the command or program you want to execute. The output screen appears, and the debugger executes the command or program. When the output from the command or program is finished, the message `Press any key to continue...` appears at the bottom of the screen. Press a key to make the debugging screen reappear with the same status it had when you left it.

Examples

```
>!
```

In the above example, the CodeView debugger saves the current debugging context. The standard screen appears, and you can enter any number of commands. To return to the debugger, enter **exit**.

```
>!ls /tmp
```

In the example above, the command **ls** is executed with the argument */tmp*. The directory listing is followed by a prompt telling you to press any key to return to the CodeView debugging screen.

Tab Set Command

The Tab Set command sets the width in spaces that the CodeView debugger fills for each tab character. The default tab is eight spaces. You might want to set a smaller tab size if your source code has so many levels of indentation that source lines extend beyond the edge of the screen. This command has no effect if your source code was written with an editor that indents with spaces rather than with tab characters.

Keyboard

You cannot set the tab size by using a keyboard command.

Dialog

To set the tab size with a dialog command, enter a command line with the following syntax:

#<number>

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

Example

```
>.
32:                     IF (X(I)) .LE. X(J)) GOTO 301
>#4
>.
32:             IF (X(I)) .LE. X(J)) GOTO 301
>
```

In the example above, the Source Line command (.) is used to show the source line with the default tab width of eight spaces. Next the Tab Set command is used to set the tab width to four spaces. The Source Line command then shows the same line.

Option Command

The Option command allows you to view the state of options in the Option menu (Save Output, Bytes Coded, and Case Sense), and to turn any of the these options on or off.

For each different kind of source module that you debug, there is a different set of default settings. However, the use of the Option command overrides any of these settings.

Keyboard

To view the state of the Options menu with a keyboard command, press <ALT>**o** to open the Options menu. Each option is then displayed. Those options that are turned on have a double arrow immediately to the left. Options that are turned off have no double arrow.

To change one of the Option settings, press the letter key corresponding to the option's mnemonic. This reverses the state of the option. (An option that was on is turned off and vice versa.) You can also reverse an option by moving the highlight down with the arrow key, and then pressing <RETURN> .

Dialog

To view or change options with a dialog command, enter a command line with the following syntax:

O[<*option*> [+ | -]]

In the above display, *option* is one of the following characters: **F**, **B**, **C**, or **3**. If used, there must be no spaces between the character and the **O**. These characters correspond to options as shown below:

Option Command

Command	Correspondence
OF	Save Output option
OB	Bytes-Coded option
OC	Case-Sense option
O	All options

The **O** form of the command (all options) takes no arguments. It simply displays the state of all four options. The other forms of the command (**OF**, **OB**, and **OC**) can be used either with no arguments (in which case they simply display the state of the option) or they can take the argument + or –.

The + argument turns the option on. The – argument turns the option off.

Examples

```
>O
Save Output on
Bytes Coded on
Case Sense off
>OF
Save Output on
>OF-
Save Output off
```

In the example above, the **O** and **OF** commands are used simply to view the current state of an option. The **OF-** command modifies an option and then reports the results of the modification.

The dialog version of the Option command is particularly useful for redirected CodeView commands (which cannot access menus) and for making CodeView startup with certain options. For example, the following shell-level command line brings up CodeView with the Bytes Coded off:

```
CV /c"OB-" test
```

This command line could be put into a shell script for convenient execution.

Redirection Commands

The CodeView debugger provides several options for redirecting commands from or to devices or files. Furthermore, the debugger provides several other commands, which are relevant only when used with redirected files. The redirection commands and related commands are discussed in the following sections.

Keyboard

None of the redirection or related commands can be executed with keyboard commands.

Dialog

The redirection commands are entered with dialog commands, as shown in the following sections.

Redirecting CodeView Input

Syntax

< *devicename*

The Redirected Input command causes the CodeView debugger to read all subsequent command input from a device, such as another terminal or a file.

Examples

```
></dev/tty1a
```

The example above redirects commands from the device (probably a remote terminal) designated as */dev/tty1a* to the CodeView terminal.

```
><infile.txt
```

The example above redirects command input from file *infile.txt* to the CodeView debugger. You might use this command to prepare a CodeView session for someone else to run. You create a text file containing a series of CodeView commands separated by carriage-return-line-feed combinations or semicolons. When you redirect the file, the debugger executes the commands to the end of the file. One way to create such a file is to redirect commands from the CodeView debugger to a file (see the section ''Redirecting CodeView Input and Output'') and then edit the file to eliminate the output and add comments.

Redirecting CodeView Output

Syntax

[T]>[>] <*devicename*>

The Redirected Output command causes the CodeView debugger to write all subsequent command output to a device, such as another terminal, a printer, or a file. The term ''output'' includes not only the output from commands, but the command characters that are echoed as you type them.

The optional **T** indicates that the output should be echoed to the CodeView screen. Normally, you want to use the **T** if you are redirecting output to a file, so that you can see what you are typing. However, if you are redirecting output to another terminal, you may not want to see the output on the CodeView terminal.

The second greater-than symbol (optional) appends the output to an existing file. If you redirect output to an existing file without this symbol, the existing file is replaced.

Examples

```
>>/dev/tty1a
```

In the example above, output is redirected to the device designated as */dev/tty1a* (probably a remote terminal). You might want to enter this command, for example, when you are debugging a graphics program and want CodeView commands to be displayed on a remote terminal while the program display appears on the originating terminal.

```
>T>outfile.txt
.
.
.
>>/dev/tty
.
.
.
```

In the example above, output is redirected to the file *outfile.txt*. You might want to enter this command in order to keep a permanent record of a CodeView session. Note that the optional **T** is used so that the session is echoed to the CodeView screen as well as to the file. After redirecting some commands to a file, output is returned to the console (terminal) with the command >*/dev/tty*.

```
>T>>outfile.txt
```

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

Redirecting CodeView Input and Output

Syntax

= <*devicename*>

Redirection Commands

The Redirected Input and Output command causes the CodeView debugger to write all subsequent command output to a device and simultaneously to receive input from the same device. This command is practical only if the device is a remote terminal.

Redirecting input and output works best if you start in sequential mode (using the **-T** option). The CodeView debugger's window interface has little purpose in this situation, since the remote terminal can act only as a sequential (nonwindow) device.

Example

```
>=/dev/tty1a
```

In the example above, output and input are redirected to the device designated as */dev/tty1a*. This command would be useful if you wanted to enter debugging commands and see the debugger output on a remote terminal, while entering program commands and viewing program output on the terminal where the debugger is running.

Commands Used with Redirection

The following commands are intended for use when redirecting commands to or from a file. Although they are always available, these commands have little practical use during a normal debugging session.

Command	**Action**
Comment (*)	Displays comment
Delay (:)	Delays execution of commands from a redirected file
Pause (")	Interrupts execution of commands from a redirected file until a key is pressed.

Comment Command

Syntax

**<comment>*

The Comment command is an asterisk (*) followed by text. The CodeView debugger echoes the text of the comment to the screen (or other output device). This command is useful in combination with the redirection commands when saving a commented session, or when writing a commented session that is redirected to the debugger.

Examples

```
>T>output.txt
>* Dump first 20 bytes of screen buffer
>D #B800:0 L 20
B800:0000 54 17 6F 17 20 17 72 17 65 17 74 17 75 17 72 17 T.o. .r.e.t.u.r.
B800:0010 6E 17 20 17                                     n. .
>
```

In the example above, the user is sending a copy of a CodeView session to file *output.txt*. Comments are added to explain the purpose of the command. The text file contains commands, comments, and command output.

```
    * Dump first 20 bytes of screen buffer
    D #B800:0 L 20
    .
    .
    .
    < /dev/tty
```

The example above illustrates another way to use the Comment command. You can put comments into a text file of commands that are executed automatically when you redirect the file into the CodeView debugger. In this example, an editing program was used to create the text file called *input.txt*.

```
><input.txt
>* Dump first 20 bytes of screen buffer
>D #B800:0 L 20
B800:0000 54 17 6F 17 20 17 72 17 65 17 74 17 75 17 72 17 T.o. .r.e.t.u.r.
B800:0010 6E 17 20 17                                     n. .
.
.
.
></dev/tty
>
```

When you read the file into the debugger by using the Redirected Input command, you see the comment, the command, and then the output from the command, as in the example above.

Delay Command

Syntax

```
:
```

The Delay command interrupts execution of commands from a redirected file and waits about half a second before continuing. You can put multiple Delay commands on a single line to increase the length of the delay. The delay is the same length, regardless of the processing speed of the computer.

Example

```
        : ;* That was a short delay...
        ::::: ;* That was a longer delay...
```

In the example above from a text file that might be redirected into the CodeView debugger, the Delay command is used to slow execution of the redirected file.

Pause Command

Syntax

```
"
```

The Pause command interrupts execution of commands from a redirected file and waits for the user to press a key. Execution of the redirected commands begins as soon as a key is pressed.

Example

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

In the example above from a text file that might be redirected into the CodeView debugger, a Comment command is used to prompt the user to press a key. The Pause command is then used to halt execution until the user responds.

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

The example above shows the output when the text is redirected into the debugger. The next CodeView prompt does not appear until the user presses a key.

Index

Suggestions – Criticisms – Corrections

Are you happy with this manual? If so, let us know.
If not, help us improve it by informing us
- where you have noticed mistakes
- where the content is unclear.

From:

Name

Company/department

Address

Postal Code

Telephone ( )

Local Siemens office

Contact person

Siemens AG
DI ST QM2
Manualredaktion
Otto-Hahn-Ring 6
Postfach 83 09 51

D-8000 München 83

From:

Name

Company/department

Address

Postal Code

Telephone ( )

Local Siemens office

Contact person

Siemens AG
DI ST QM2
Manualredaktion
Otto-Hahn-Ring 6
Postfach 83 09 51

D-8000 München 83

Hit a "stumbling block"? Tell us where.

Manual title: SINIX Open Desktop V1.0, U5760-J-Z95-1-7600

Page	Problem:

I am ☐ a programmer
☐ a system administrator
☐ an ordinary user
☐ ____________________

I use the manual ☐ frequently
☐ occasionally for reference
☐ ____________________

Manual title: SINIX Open Desktop V1.0, U5760-J-Z95-1-7600

Page	Problem:

I am ☐ a programmer
☐ a system administrator
☐ an ordinary user
☐ ____________________

I use the manual ☐ frequently
☐ occasionally for reference
☐ ____________________