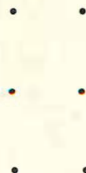


• • •
• •
•

SCO® UNIX® Development System

Debugging Tools Guide



SCO® UNIX®
Development System
Debugging Tools Guide



© 1983-1992 The Santa Cruz Operation, Inc.
© 1980-1992 Microsoft Corporation.
© 1989-1992 UNIX System Laboratories, Inc.
All Rights Reserved.

No part of this publication may be reproduced, transmitted, stored in a retrieval system, nor translated into any human or computer language, in any form or by any means, electronic, mechanical, magnetic, optical, chemical, manual, or otherwise, without the prior written permission of the copyright owner, The Santa Cruz Operation, Inc., 400 Encinal, Santa Cruz, California, 95060, U.S.A. Copyright infringement is a serious matter under the United States and foreign Copyright Laws.

The copyrighted software that accompanies this manual is licensed to the End User only for use in strict accordance with the End User License Agreement, which should be read carefully before commencing use of the software. Information in this document is subject to change without notice and does not represent a commitment on the part of The Santa Cruz Operation, Inc.

The following legend applies to all contracts and subcontracts governed by the Rights in Technical Data and Computer Software Clause of the United States Department of Defense Federal Acquisition Regulations Supplement:

RESTRICTED RIGHTS LEGEND: USE, DUPLICATION, OR DISCLOSURE BY THE UNITED STATES GOVERNMENT IS SUBJECT TO RESTRICTIONS AS SET FORTH IN SUBPARAGRAPH (c) (1) (ii) OF THE RIGHTS IN TECHNICAL DATA AND COMPUTER SOFTWARE CLAUSE AT DFARS 52.227-7013. "CONTRACTOR/SUPPLIER" IS THE SANTA CRUZ OPERATION, INC. 400 ENCINAL STREET, SANTA CRUZ, CALIFORNIA 95060, U.S.A.

Microsoft, MS-DOS, and XENIX are registered trademarks of Microsoft Corporation.
DEC, PDP, VAX, and VT100 are trademarks of Digital Equipment Corporation.
Intel is a registered trademark of Intel Corporation.
UNIX is a registered trademark of UNIX System Laboratories, Inc.

Date: 10 December 1991
Document Version: 3.2.4D

Preface 1

Using this guide	1
Documentation conventions	3
Development System documentation set	4
Using the Documentation	5
Commercial books and articles	6

Chapter 1 ***dbxtra*** 7

Tutorial session	8
Preparing files for dbxtra	8
Starting dbxtra	8
The dbxtra screen	9
The dbxtra help facility	10
Printing the value of variables	10
Viewing source code and aliasing	10
Viewing the call stack	11
Traversing the call stack	11
Monitoring the contents of variables	11
Analyzing the results	13
Examining a different function	13
Using a breakpoint	13
Continuing after a breakpoint	14
Exiting dbxtra	14
Invoking dbXtra	15
The Motif interface	17
The main window	17
The sidebar command buttons	18
Pull-down menus	19
The pop-up menu	27
Configurable resources	28
The curses interface	30
The screen interface	30
The keyboard interface	33
Editing commands	34
Using dbXtra	39
Expressions and conditions	39
Executing the program being debugged	41

Managing execution time events	45
Altering the execution environment	51
User interface commands	53
Debugging registers, memory locations, and expressions	56
Accessing source files	59
Miscellaneous commands	62
C++ specific debugging features	65
Usage requirements	66
Overloaded functions and operators	67
The cc command	67
Predefined aliases	69
Sample key bindings	69

Chapter 2

CodeView debugger

71

Restrictions	71
Preparing programs for CodeView	72
Programming considerations	72
CodeView compile options	72
CodeView link options	73
Preparing assembly programs	73
Starting CodeView	74
Using CodeView options	76
Specifying startup commands	77
Understanding CodeView windows	78
Displays are automatically updated	78
Adjusting the windows	79
Using window mode	79
Executing keyboard commands	81
Using menu selections	84
Using the help system	91
Using window commands	92
Entering commands and arguments	93
Using special keys	93
Using the command window buffer	93
Format for CodeView commands and arguments	94
CodeView expressions	95
C expressions	96
C symbols	97

C constants	97
C strings	98
Assembly expressions	99
Line numbers	101
Registers and addresses	101
Registers	102
Addresses	102
Address ranges	103
Memory operators	104
BY operator	105
WO operator	105
DW operator	105
Executing code	106
Trace command	107
Program step command	108
Go command	109
Execute command	111
Restart command	112
Examining data and expressions	113
Display expression command	113
Examine symbols command	116
Dump commands	119
Dump	121
Dump bytes	122
Dump ASCII	122
Dump integers (16-bit)	123
Dump integers (32-bit)	123
Dump unsigned integers (16-bit)	124
Dump unsigned integers (32-bit)	124
Dump words	125
Dump double words	126
Dump short reals	126
Dump long reals	127
Dump 10-byte reals	127
Compare memory command	128
Search memory command	129
Register command	130
8087 command	131
Managing breakpoints	132
Breakpoint set command	133
Breakpoint clear command	135

Breakpoint disable command	136
Breakpoint enable command	137
Breakpoint list command	137
Managing watch statements	138
Setting watch-expression	139
Deleting watch statements	140
Listing watchpoints	141
Assembly examples	142
Examining code	143
Set mode command	143
Unassemble command	145
View command	146
Current location command	147
Stack trace command	147
Modifying code or data	149
Assemble command	149
Enter commands	152
Enter command	155
Enter bytes command	156
Enter ASCII command	156
Enter integers command (16-bit)	157
Enter integers command (32-bit)	157
Enter unsigned integers command (16-bit)	158
Enter unsigned integers command (32-bit)	159
Enter words command	159
Enter double words command	160
Enter short reals command	161
Enter long reals command	161
Enter 10-byte reals command	162
Fill memory command	163
Move memory command	164
Register command	164
Using CodeView system-control commands	167
Help command	167
Quit command	167
Radix command	168
Redraw command	170
Screen exchange command	170
Search command	170
Shell escape command	172
Tab set command	173

Option command	174
Redirection commands	176
Redirecting CodeView input	176
Redirecting CodeView output	177
Resetting CodeView input and output	178
Commands used with redirection	178
Comment command	178
Delay command	179
Pause command	180

Chapter 3

sdb: the symbolic debugger 181

Using sdb	181
Printing a stack trace	183
Examining variables	183
Source file display and manipulation	185
Displaying the source file	186
A controlled environment for program testing	187
Setting and deleting breakpoints	187
Running the program	188
Calling functions	189
Machine language debugging	190
Displaying machine language statements	190
Manipulating registers	190
Other commands	191
An sdb session	191

Chapter 4

adb: Absolute debugger 195

Starting adb	195
Starting adb with a core image file	196
Exiting adb	196
Displaying instructions and data	197
Forming addresses	197
Forming expressions	197
Choosing data formats	201
Using the = command	203

Using the ? and / commands	204
An example: simple formatting	205
Debugging program execution	206
Executing a program	206
Setting breakpoints	207
Displaying breakpoints	208
Continuing execution	208
Stopping a program with interrupt and quit	208
Single-Stepping a program	209
Killing a program	209
Deleting breakpoints	209
Displaying the c stack backtrace	210
Displaying CPU registers	210
Displaying external variables	211
A 286 example: tracing multiple functions	212
A 386 example: tracing multiple functions	216
Using the adb memory maps	220
Displaying the memory maps	220
Changing the memory map	222
Creating new map entries	222
Validating addresses	223
Miscellaneous features	224
Combining commands on a single line	224
Creating adb scripts	224
Setting output width	225
Setting the maximum offset	225
Setting default input format	226
Using UNIX commands	226
Computing numbers and displaying text	227
An example: directory and inode dumps	227
Patching binary files	229
Locating values in a file	229
Writing to a file	229
Making changes to memory	230

Preface

Debuggers are, in general, designed to trace systematically a through program, statement-by-statement, to detect logic problems. By using a debugger, a programmer can step through a program one instruction at a time. This allows for each line of code to be tested, ensuring that the appropriate logic is executed.

Similarities exist between all debugger's. Most allow for the display of program instructions and data. Programs can be run through, single stepped or, monitored by setting breakpoints. In some instances, programs can be patched while being debugged, their re-executed.

The source information from a debugger can be displayed in either a line-by-line fashion or through a windowed environment.

This guide explains how to use the SCO UNIX System V/386 Development System to debug and maintain C and assembly language programs.

Using this guide

This guide is intended for programmers who are familiar with debugging C and assembly language programs. The following list briefly describes each chapter.

Chapter 1, "dbxtra," is a powerful source level debugger for C and assembly language programs. **dbxtra** is a full-screen windowing debugger.

There are two versions of **dbxtra**. The **dbxtra** program is a **curses** version available for use on any character-based terminal. The **dbXtra** program is an X windows based version.

dbxtra supports debugging of both C and C++ products in conjunction with SCO C++ products. This debugger can follow the child or parent process when a fork is executed. **dbxtra** allows for the suspension of the parent debugger while debugging the child, and returning to the parent session after exiting the child. When using the X-based version of **dbXtra**, a programmer can simultaneously debug parent and child processes.

Chapter 2, “Codeview Debugger,” is an executable program that helps you debug software written with C and assembly languages in a window-type environment. **CodeView**’s pull-down menus and simple keystrokes offer fast and easy access to common commands. In addition to numerous standard debugging functions, **CodeView** provides the ability to browse source or assembly code, indicating which line to execute next. **CodeView** can debug both COFF and *x.out* executable files.

Chapter 3, “sdb: symbolic debugger,” examines the core images of aborted programs. Through **sdb**, a program’s execution can be monitored and controlled. **sdb** provides a means for displaying variables to be accessed symbolically and viewed in the correct format.

Chapter 4, “adb: absolute debugger,” is a debugging tool for C and assembly language programs. Currently, **adb** is the only debugger available for debugging 286 binaries. **adb** provides a method for viewing non-ASCII data files with line mode commands. **adb** can single step through a program, allowing program patches to be applied to a program at the source level.

Documentation conventions

SCO documents use font changes and other typographic conventions to distinguish text elements. The following table shows these conventions:

Font conventions

Example	Entity
cc or cc(CP)	command. The "CP" indicates the manual page section in which the command is documented.
<i>libc.a</i>	filename
action	placeholder
⟨Esc⟩	keyname
open or open(S)	system calls, library routines, kernel functions, C keywords. The "S" indicates the manual page section in which the command is documented.
buf	structure
b.errno	structure member
\$HOME	environment or shell variable
SIGHUP	named constant
login	output
"clobber"	jargon
"adm3a"	data value
date	user input

Development System documentation set

The contents of the SCO UNIX System V/386 Development System documentation set are illustrated here. In addition to these books, one set of *Release and Installation Notes* is shipped with each copy of the Development System software.

Overview	<i>Developer's Overview</i>	
Reference	<i>Programmer's Reference Manual (2 volumes)</i>	<i>Encyclopedia</i>
Guides	<i>Programming Tools Guide</i>	<i>Developer's Topics</i>
	<i>Debugging Tools Guide</i>	<i>User Interfaces Guide</i>
Optional	<i>Device Driver Writer's Guide</i>	<i>Macro Assembler Writer's Guide</i>

The books included with the Development System are described here.

Developer's Overview

introduces the Development System facilities and gives general information about developing software to run on SCO UNIX systems and the supported cross-development environments

Programmer's Reference Manual

a two-volume set that includes manual pages for the entire Development System, including sections (CP), (DOS), (FP), (S), and (XNX). See the "Manual pages" article in the *Encyclopedia* for a description of these manual page sections.

Encyclopedia

contains articles that give background information about system internals, descriptions of facilities, and other general issues. Articles are arranged alphabetically.

Developer's Topics

a collection of technical papers about topics of interest to users of the Development System. Many of these papers include extensive examples that illustrate features that SCO added to the porting base.

Programming Tools Guide

provides generally useful information about programming tools and their use in developing software. It also provides implementation-specific details about the ANSI-conforming C compiler that is provided with the Development System.

Debugging Tools Guide

provides generally useful information about the debugging tools, and their use in tracking down and eradicating problems in C and assembly language programs. In addition, it includes information on using **dbXtra** to debug C and C++ programs in a Motif-based windowing environment.

User Interfaces Guide

introduces the facilities that are available for developing user interfaces, and gives detailed instructions for using **curses(S)**, the Extended Terminal Interface (ETI), and writing user interfaces that can be run on ASCII terminals. This book includes numerous examples of **curses** programs.

In addition, the following books are available for Development System customers who need them:

Device Driver Writer's Guide

Provides guide and reference information describing how to write device drivers for SCO UNIX systems. This book is sold separately.

Macro Assembler Writer's Guide

Provides information on the assembly language for the Intel 286, 386, and 486 processors. Developers who are writing device drivers or applications that have tight performance requirements may want to write portions in assembly language. Return the coupon enclosed with the Development System documentation to receive a copy.

Using the Documentation

We recommend the following approach to using this documentation set. In general, we suggest the following:

1. All users should read the *Developer's Overview* thoroughly, soon after installing the software. Experienced users may skim some sections, but this book gives a comprehensive overview of the Development System's capabilities.
2. Use the *Encyclopedia* to look up topics about which you need more information. The *Encyclopedia* gives both summary and background information on many topics, and includes cross-references to sources of more in-depth information.
3. The *Programmer's Reference Manual* contains complete reference information about all capabilities provided in the Development System. New users are warned, however, that the manual pages provide reference material that may be difficult to utilize if they do not understand basic concepts.

The *Programmer's Reference Manual* contains separate sections for Development System commands (CP), system services (S), Development System file formats (FP), and other topics. The pages in each section of the *Programmer's Reference Manual* are arranged alphabetically, although some related facilities are grouped together on a page. If you cannot locate the correct manual page, use the permuted index to find the one that contains the information you want. The "Manual pages" article in the *Encyclopedia* explains how to use the permuted index.

4. The *Programming Tools Guide* and *Debugging Tools Guide* are mainly for users who have not used one of these tools before, or who want to reacquaint themselves with some aspect of that tool's operation. These books are good places to look when the *Encyclopedia* and *Programmer's Reference Manual* are giving you lots of detail, but not enough guidance. Nonetheless, even experienced programmers will discover helpful hints and practical tips on the tools discussed in these books.

Commercial books and articles

A number of fine books and articles are published commercially that discuss how to develop software on the UNIX Operating System. We have not attempted to replicate guide information on all of these topics; manual pages are provided, and many articles in the *Encyclopedia* list additional sources of information on their subject matter. In addition, the *Encyclopedia* includes a large "Bibliography" that will interest users of the Development System.

In particular, we recommend that all developers have the following standard textbooks on their shelves:

Brian Kernighan and Dennis Ritchie, *The C Programming Language*, 2nd Edition.

Marc Rochkind, *Advanced UNIX Programming*.

W. Richard Stevens, *UNIX Network Programming*.

Maurice Bach, *Design of the UNIX Operating System*

See the "Bibliography" in the *Encyclopedia* for full citations if you are not familiar with these books.

Chapter 1

dbxtra

dbxtra is a powerful source level debugger for C and assembly language programs. Its functionality is a superset of the **dbx** and **sdb** debuggers available with most UNIX systems. It includes source code presentation through windows, as well as numerous powerful features that result in increased overall programmer efficiency and productivity. Some of the features of **dbxtra** include:

- conditional breakpointing and symbolic variable tracing
- keyboard configuration enables debugger commands to be issued with a single keystroke
- symbolic access to all active functions and variables
- online cross-referencing capabilities
- program input and output redirection to a different screen or terminal

Some of the benefits of **dbxtra** are:

- The windowed environment allows you to analyze and review program source systematically.
- You can use any alphanumeric terminal supported in your *terminfo* database: there is no need for expensive bit-mapped terminals if you do not have one. **dbxtra** puts up windows and processes commands on such terminals using the *curses* library.
- You can simultaneously examine source code, variable values, program output, and your interaction with **dbxtra**.

dbxtra is a tool for machine- and source-level debugging under the UNIX system. The facilities for machine-level debugging with **dbxtra** can be used on any program. The source-level facilities are available when the object file includes symbolic information.

Tutorial session

Using a sample program, designed for bidding in the card game “bridge”, we will explore some of the basic features of **dbxtra**.

Preparing files for **dbxtra**

Included on the media with your copy of **dbxtra** are files named *dealer.c*, *dealer.h*, and *deal.c* that you can use to participate in this session. These files are installed in the directory */usr/lib/dbxtra/demo*. Locate these files, copy them to a working directory, and compile them with:

```
$ cc -g dealer.c deal.c
```

NOTE To use **dbxtra** for source-level debugging, use the **-g** compiler option to place symbolic information, source line numbers, and variable names in the object file.

Execute *a.out*, and the system responds with:

```
a.out
Dealing 1 hands...
Illegal instruction - core dumped
$
```

NOTE The exact wording of the second line may vary from system to system. It indicates that a *core* file, used for debugging, has been written to the current directory.

Starting **dbxtra**

The program that you compiled has a serious problem and we are going to find it. To debug the executable file called *a.out*, enter the following at the shell prompt:

```
$ dbxtra a.out
```

This starts the **curses** based version of **dbxtra**. To initiate the **Motif** version, enter the following at the system prompt:

```
$ dbXtra a.out
```

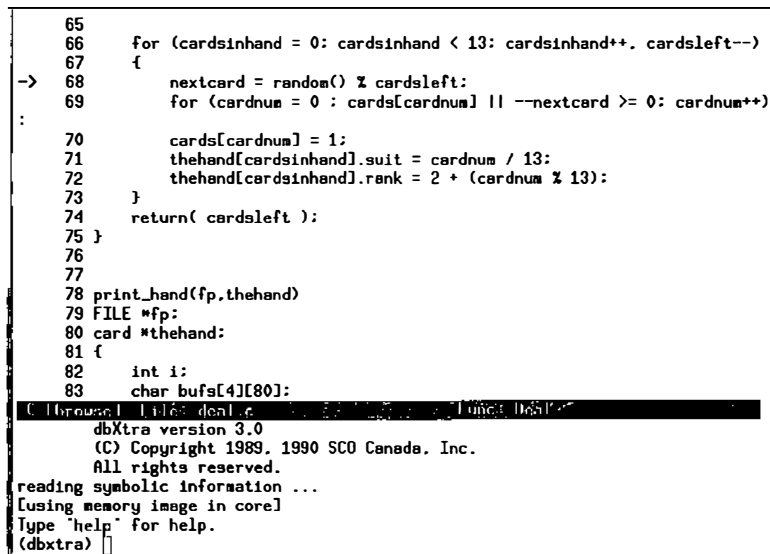
In this tutorial session, only the **curses** version of the debugger will be described. Although the method of presentation is different, the functionalities of the commands in both versions are identical. You may, of course, specify the name of any executable file you wish to debug. If you do not specify the name of an executable file, **dbxtra** prompts you for one.

At startup, **dbxtra** informs you that it is reading in the symbolic information in the executable file and that it is using the memory image in the core file.

The *dbxtra* screen

Figure 1-1 resembles a window with two panes separated by a narrow horizontal bar. The top pane contains lines of C code from the source file *deal.c*, and an arrow is pointing at line 68 where the illegal instruction that caused the core dump is located. The horizontal bar identifies the file (*deal.c*) and the function name (**Deal**). This bar constantly displays the name of the active function and the source file in which it appears.

The bottom pane, which contains the (**dbxtra**) prompt, is where you interact with **dbxtra**.



```

65
66     for (cardsinhand = 0; cardsinhand < 13; cardsinhand++, cardsleft--)
67     {
-> 68         nextcard = random() % cardsleft;
69         for (cardnum = 0 ; cards[cardnum] || --nextcard >= 0; cardnum++)
:
70             cards[cardnum] = 1;
71             thehand[cardsinhand].suit = cardnum / 13;
72             thehand[cardsinhand].rank = 2 + (cardnum % 13);
73         }
74     return( cardsleft );
75 }
76
77
78 print_hand(fp,thehand)
79 FILE *fp;
80 card *thehand;
81 {
82     int i;
83     char bufs[4][80];
C:\Program Files\IBM\dbx\bin\deal.c      Func: Deal
dbXtra version 3.0
(C) Copyright 1989, 1990 SCO Canada, Inc.
All rights reserved.
reading symbolic information ...
[using memory image in core]
Type 'help' for help.
(dbxtra)

```

Figure 1-1 Display contents after executing **dbxtra a.out**

The dbxtra help facility

Above the (dbxtra) prompt, Type help for help, is displayed. Help can be invoked in two ways, first by typing **help** at the prompt, a general list of dbxtra's commands are displayed. For a more detailed explanation of a specific command, invoke help with the dbxtra command name as an argument. For example:

```
(dbxtra) help when
forward [n]
backward [n]
Scroll the source window forward and backward n lines
(the default is one window).
```

The screen may look different from one terminal to another. dbxtra tries to take advantage of the emphasis modes available on each.

Printing the value of variables

After considering the line of code for a moment, you probably want to know the value of the variable **cardsleft**.

```
(dbxtra) print cardsleft
0
```

As suspected, the illegal instruction is the result of an attempt to divide by zero — by way of the modulus operator — but that is not the source of the problem. This variable should never have had a zero in it: we'll have to investigate why it did.

NOTE Besides simple variables like **cardsleft**, dbxtra can also print the contents of aggregates like arrays and structures. Aggregates can contain many elements. The bottom window is not large enough to display them all. dbxtra provides simple paging by stopping the output when the window is full and displaying a highlighted prompt -- MORE --.

Viewing source code and aliasing

From Figure 1-1, it is clear that **cardsleft** must have been assigned a zero before line 60. To see the source code that did this, we have to scroll the source code window back.

```
(dbxtra) backward
```

The **backward** command scrolls the source window back one full window. To scroll the window forward to see again how **cardsleft** is used, we use the command **forward**.

Typing these screen motion commands in full each time is tedious, so **dbxtra** provides the **alias** command to “rename” commands. For example, to rename the **backward** command to **b**, for example, type:

```
(dbxtra) alias b backward
```

dbxtra provides a set of default aliases. To view them, type **alias** itself at the (dbxtra) prompt.

Having moved back one screen, we see that **cardsleft** is a parameter of **Deal()**. Therefore, to see how it got passed the value zero, we have to find out which function called it. This can be done by looking at the *call stack*.

Viewing the call stack

To see all active functions within a program, use the **where** command to view the call stack.

```
(dbxtra) where
Deal(thehand = 0x401818, cards = 0x7ffff8c4, cardsleft = 0),
line 68 in "deal.c"
deal_hands(pdeal = 0x401678), line 51 in "deal.c"
main(argc = 1, argv = 0x7ffff9d4, 0x7ffff9dc), line 55 in
"dealer.c"
```

The **where** command produces a list of all active functions, their arguments, the line number in which they stopped, and the source file in which they appear. Each function is called by the function that appears below it. In this case, **Deal()** was called by **deal_hands()**, which in turn was called by **main()**.

Traversing the call stack

You can see from the stack trace that **cardsleft** is passed to **Deal()** by **deal_hands()**. Let's go there:

```
(dbxtra) up
```

Notice that the source window has changed. The filename is still *deal.c*, but the current function is now **deal_hands()**. The window now shows the code from **deal_hands()** with line r545 marked with an arrow.

Monitoring the contents of variables

The **dbxtra** commands **when**, **trace**, and **stop**, monitor the contents of a program's variables. The **trace** and **stop** commands are not as robust as the **when** command (for further details see “Managing execution time events”).

In our Bridge example, the variable `cardsleft` is passed to the function `Deal()`. It may be a good idea to have the value of `cardsleft` printed out every time the call to `Deal()`, at line 51, is made. To do this use the `when` command. At the (`dbxtra`) prompt, enter:

```
(dbxtra) when at 51 {print "at 51: cardsleft = ", cardsleft;}
[1] when at "deal.c":51 ( print "at 51: cardsleft = ",
deal.deal_hands.cardsleft;)
```

`dbxtra` acknowledges your command, and automatically prepends the names of the blocks containing the variable to the variable name and the filename to the line number. Notice the **W** beside line 51; this indicates a **when** has been issued for this function.

To verify how a variable is monitored, use the `status` command:

```
(dbxtra) status
[1] when at "deal.c":51 ( print "at 51: cardsleft = ",
deal.deal_hands.cardsleft;)
```

There are two things to note here:

- The symbol `[1]` is called an event-id. This is the first command that `dbxtra` need not execute immediately, but is kept in the “background” until a particular event occurs.
- `deal.deal_hands.cardsleft` is the *qualified name*, referring to the `cardsleft` variable. The qualified name indicates the respective scopes enclosing the variable. In this case, it refers to the `cardsleft` variable in function `deal_hands()` in `deal.c`. It is useful to ensure that the correct variable has been located, since a program can contain many variables with the same name.

Now let’s run `a.out` to see what happens:

```
(dbxtra) run
Dealing 1 hands...
at 51: cardsleft = 52
at 51: cardsleft = 39
at 51: cardsleft = 26
at 51: cardsleft = 13
at 51: cardsleft = 0
Floating point exception in Deal at line 55 in file "deal.c"
```

Analyzing the results

You are back where you started, at line 55 in *deal.c*, and at this point it helps to know two things about the game of Bridge:

- Bridge is a game for four players.
- Fifty-two cards are dealt.

Each time a player receives cards, there are 13 cards less in the pack to be dealt. From the output produced by **when**, we can see that at each line after the first, the number of cards is reduced by 13. Therefore, we conclude that **cardsleft** indicates the cards remaining after players receive cards, and that each time a line is printed, the program is about to deal cards to a player. Since five lines were printed, the program was trying to distribute cards to five players, clearly an error. Hence, it is likely that there is a loop somewhere that is out of bounds.

Examining a different function

Let's go back and examine the values in **deal_hands()**:

```
(dbxtra) func deal_hands
```

The code in the source window now includes the offending line:

```
for( player=0; player<=4; player++ )
```

The limit on the loop counter (**player**) should have been set to 3 instead of 4, so there is four iterations instead of five. How can we test this suspicion?

Using a breakpoint

One way to test our suspicion is to force the loop to iterate only four times. To do so we first need to stop the loop after it has iterated that many times. **dbxtra** allows us to do this by letting us set breakpoints within programs using the **stop** command.

```
(dbxtra) stop at 51 if player=3
[2] stop at "deal.c":51 if deal.deal_hands.player == 3
```

Line 51 is the last line of the loop after all statements within it have been executed and if **player** is 3, we know that it has iterated four times. Rerunning the program now produces:

```
(dbxtra) rerun
Dealing 1 hands...
at 51: cardsleft = 52
at 51: cardsleft = 39
at 51: cardsleft = 26
at 51: cardsleft = 13
[2] stopped in deal_hands at line 51 in file "deal.c"
```

To prevent the fifth iteration of the loop, we just need to increment **player** by one:

```
(dbxtra) assign player=player+1
(dbxtra) print player
4
```

The counter has been incremented, so there will not be another pass when you continue the program.

Continuing after a breakpoint

You may now continue the execution of the program using the **cont** command:

```
(dbxtra) cont

          *You*   LHO   Partner   RHO
Bidding
Your Hand
Spades
Hearts
Diamonds
Clubs

Enter your bid:
(dbxtra) cont
```

Exiting dbxtra

Our suspicions were correct! We, therefore, need to modify the function **deal_hands()** in the file *deal.c*. To do so, we first need to exit **dbxtra**. Before we can do so, however, notice that the cursor is now positioned after the last line of the program output shown above, and the (dbxtra) prompt has not appeared. This is because *a.out* is still running, so we need to interrupt it before we can exit **dbxtra**. This can be done by pressing the user-defined interrupt key. On most UNIX systems, it is the key. Once the (dbxtra) prompt is displayed, type **quit** to exit **dbxtra**.

Invoking dbXtra

There are a few command-line options that tailor the invocation of **dbXtra**. Options can appear in any order, but they must appear before *objfile*. Each option is specified as a separate argument. The syntax for the command is:

```
dbxtra [options] [objfile [corefile]]
dbXtra [options] [objfile [corefile]]
```

The first form of the command invokes the **curses** version of the debugger. This version is suitable for use with most common alphanumeric terminals as well as from within a terminal emulator window application (such as **xterm**) within X windows. The second form of the command can only be used within the X windowing system. It invokes the Motif version of the debugger, utilizing various X11 and Motif objects.

The object file specified by *objfile* contains the executable instructions of the program to be debugged. If it is not specified, **dbXtra** prompts for the name of the file. If this object file was compiled with the **-g** flag, the symbol table information needed to identify the variables by their source names, and the line numbers and filenames of the source code for each executable source line, are also included in the file. **dbXtra** gives a warning if the object file does not have this extra information.

The core file specified by *corefile* determines the state of the program when it faulted. If it is not specified, then the file named *core* in the current directory is used if it exists. If the core file does not coincide with the program being debugged, for example, if it was created when another program had faulted, it is ignored.

If the file *.dbxtrarc* exists in the current directory, then the debugger commands in it are executed. **dbXtra** also checks for a *.dbxtrarc* in the user's home directory if there is not one in the current directory.

One or more of the following options may be specified in any order:

-C++ Invoke **dbXtra** in C++ interaction mode. This option should be used if the program being debugged was developed using C++. It has no effect if the program was written in C. Use of this option activates the **cc** command, which toggles between C and C++ interaction modes. **dbXtra** assumes this flag by default.

For further information about debugging C++ programs, see "C++ specific debugging features".

-C Invoke **dbXtra** in C interaction mode only. C++ information is not read in and the **cc** command is not allowed.

- F This option loads only one symbol table entry for each unique definition of a structure or union or enum. This means that if more than one definition of an identifier, at the same scope, are present in a program, for example, if the definition is in an include file used by various modules, then all the definitions of the identifier are assumed to be the same.

The use of this option results in faster load time and requires significantly less memory for larger programs.

NOTE If the program contains non-unique definitions of the same identifier at the same scope, use of this option causes the symbol table of **dbXtra** to be different from that of the program.

- I *dir* By default, **dbXtra** looks for source files in the current directory and in the directory where *objfile* is located. The **-I** option adds the specified directory *dir* to the search path. Directories can also be added to the directory search path with the **use** command. You can specify multiple **-I** options on the command line. The order of the directories in the search path is the same as the order specified on the command line.
- c *file* Execute the **dbXtra** commands in the *file* before reading from standard input. These commands are read *after* any *.dbxtrarc* files are read.
- r Execute *objfile* immediately. If it terminates successfully, **dbXtra** exits. Otherwise, the reason for termination is reported and the user is offered the option of entering the debugger or letting the program fault.
dbXtra reads from */dev/tty* when **-r** is specified and standard input is not a terminal.
- s Do not invoke the screen manager. **dbXtra** works in line mode only and does not respond to any of the screen-related commands (including **screen**).
- i Force **dbXtra** to act as though standard input is a terminal.
- x Ignore *xref* file when initializing **dbXtra**.

NOTE Unless **-r** is specified, **dbXtra** prompts and waits for a command upon invocation.

The Motif interface

The Motif interface to the dbXtra debugger utilizes a number of the graphical capabilities of the X11 system as well as widgets utilized by most Motif applications. While emphasis has been placed on producing an interface that closely conforms with the Motif style guide, this version of the debugger will work on any workstation already running X11.

When dbXtra¹ is invoked, it creates one window called the Main window, with two panes within it. The pane at the top is called the source window, and the pane at the bottom is called the command window in this chapter. In addition to the two panes, various other widgets are created in the Main window, and these are discussed later in this chapter.

The main window

The main window is the principal window for user interaction during a debugging session. dbXtra's main window is composed of a title bar, a menu bar on the top, command buttons on the left-hand side, and the source window pane as well as the command window pane. See Figure 1-2.

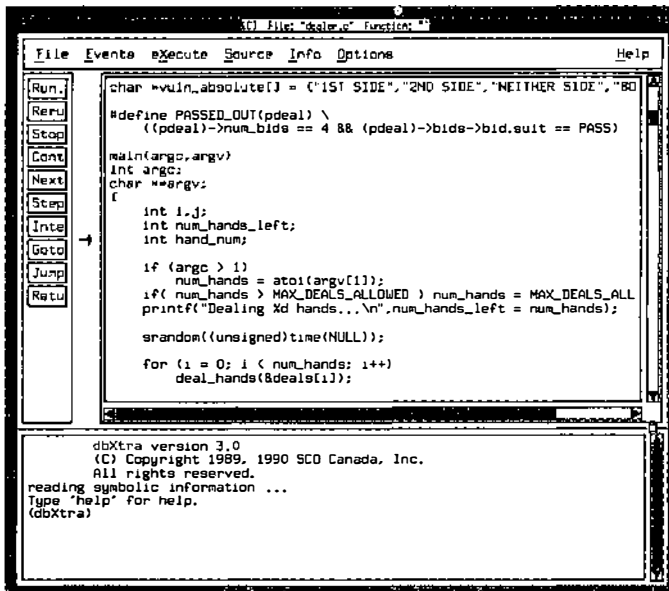


Figure 1-2 The main window

1. The Motif version is referred to as dbXtra, since this is the command used to invoke it. dbXtra also refers to the debugger in general.

The title bar

The title bar displays the current debugging mode. This is [C++] for the C++ mode and [C] for the C mode. It also displays the current file and function names.

The source window pane

The source window shows the current position in the current file. This is a read-only view of the file; the program cannot be modified or edited. The current line numbers and breakpoint status are displayed to the left of the source window. The current execution point is indicated by an arrow [->]. Breakpoints (or other events) are indicated by an [S] at the appropriate lines.

The command window pane

The command window provides direct access to the complexities of the underlying dbXtra debugger. Output from most debugger commands appear in this window and any dbXtra command can be entered using the keyboard from within this window.

The sidebar command buttons

The left-hand side of the main window contains the sidebar command buttons. These buttons provide quick access to frequently used commands.

- | | |
|-----------------|--|
| Run... | Start program execution. The user is prompted for the program command line parameters. The program is then run. Program execution continues to completion unless a breakpoint or other event is encountered. An input/output window is created for program interaction. |
| Rerun | Restart the program from its beginning. The program's previous command line parameters are used. |
| Stop | Set breakpoint event at the line indicated by the cursor in the source window. The line is flagged with the [S] symbol to the left of the line number. Breakpoints can only be set on lines containing executable code.

The executable code produced for certain C++ or C constructs do not correspond with the obvious source location for that code. In particular, for , switch and while statements display this problem.

The breakpoint can be deleted by moving the mouse pointer to the [S] symbol and clicking with the right mouse button. |
| Continue | Continue execution from the current breakpoint or event. |

Next	Execute one source line and then stop. Next does not follow function calls but does stop after a function call. Use Step to follow execution into called functions.
Step	Execute one source line and then stop. Step follows the execution into called functions. Use Next to avoid following execution into called functions.
Interrupt	Stop the execution of the program. An executing program is stopped and the source window is updated to display the current execution point.
Goto	Continue execution of the program until the specified line is reached.
Jump	Resume execution of the program from the selected line.
Return	Resume execution of the program until the first statement after control has returned from the current function.

Pull-down menus

The menu bar is located across the top edge of the source window. The menus along the bar can be accessed by selecting the name of the menu using the mouse, or by utilizing the keyboard accelerator sequence for the menu (simultaneously pressing the `<Alt>` key and the uppercase letter in the menu name). Each menu contains a logical group of buttons performing related operations.

The file menu

This menu has commands to save configuration information and exit dbXtra.

Write Configuration File

This command prompts for a filename, then creates a file containing dbXtra window geometry commands suitable for inserting in an *.Xdefaults* file.

Exit dbXtra

This command prompts to ensure the user really wants to quit, then exits **dbXtra** if so indicated.

The events menu

This menu contains a group of buttons to generate the various commands associated with breakpoint and trace events. The following is a brief description of the action performed by each button in the menu. The text between the “[” and the “] ” characters following the name of each button contain the **dbXtra** command constructed when the button is selected. For detailed information about the various **dbXtra** commands available, see section ‘Using **dbXtra**’.

Events [status]

This command displays the currently active events (breakpoints, trace expressions, and so on).

Call... [call <name>]

The **call** command calls a function within the user’s program. If the **\$unsafecall** variable is not set, the parameters supplied for the call must match in number and type to the function. When **\$unsafecall** is set, the parameters do not have to match in number or type. If C syntax calls C++ member functions; all hidden parameters must be explicitly specified.

The user is prompted via a dialog box to supply a string to use instead of the **<name>** field.

When... [when <rest-of-command>]

The **when** command allows the user to execute one or more **dbXtra** commands whenever a specific event occurs during the execution of the user’s program. For a complete explanation of the **when** command, see section “Managing execution time events”.

The user is be prompted via a dialog box to supply a string to use instead of the **<rest-of-command>** field.

Stop ->

This button generates a submenu to provide access to the various modes of the **stop** commands.

• if [stop if <conditional-expression>]

specifies via a dialog box a conditional expression to further qualify the event. Execution of the debugged program only halts if the conditional expression evaluates to true.

The user is prompted via a dialog box to supply a string to use instead of the **<conditional-expression>** field.

- **function** [stop in <function-name>]
specifies a function name that, when invoked, causes the execution of the program to halt. The user is prompted via a dialog box to supply a string to use instead of the <function-name> field.
- **address** [stop inst at <address>]
specifies an absolute address in the program space that, when encountered in the execution stream, causes the execution to halt. The user is prompted via a dialog box to supply a string to use instead of the <address> field.
- **line** [stop at <line-number>]
specifies a line number in the program that, when executed, causes the execution of the program to halt. The user is prompted via a dialog box to supply a string to use instead of <line-number>.
- **access** [stop access <variable> | <address>]
specifies a variable or absolute memory address. When the value at that address has been read from, or written to, the execution of the program halts.

NOTE Using this option requires the use of the debug registers.

The user is prompted via a dialog box to supply a string to use instead of the <variable> | <address> field.

- **change** [stop change <variable> | <address>]
specifies a variable or absolute memory address. When the value at that address has been written to, the execution of the program halts.

NOTE This option requires the use of the debug registers.

The user is prompted via a dialog box to supply a string to use instead of the <variable> | <address> field.

- **stop in main** [stop in main]
halts the program as it first starts running, that is before the first line in the function `main()` is executed. In C++ mode, all static initializers and constructors are executed before `main()` is called; use **stop in main** to regain control before static constructors are called.

- stop [stop <rest-of-command>]
specifies any variations of the **stop** command. The user is prompted via a dialog box to supply a string to use instead of the <rest-of-command> field.

Trace -> This menu generates a submenu providing access to the various modes of the **trace** commands.

- function [trace <function-name>]
specifies the name of a function that, when invoked, causes a message to be displayed. The user is prompted via a dialog box to supply a string to use instead of the <function-name> field.
- address [trace inst <address>]
specifies an absolute address in the program space that, when encountered in the execution stream, causes a message to be displayed. The user is prompted via a dialog box to supply a string to use instead of the <address> field.
- line [trace <line-number>]
specifies a line number of the program which, when executed, causes a message to be displayed. The user is prompted via a dialog box to supply a string to use instead of the <line-number> field.
- access [trace access <variable>|<address>]
specifies a variable or absolute memory address. When the value at that address has been read from, or written to, a message is displayed. This command requires the use of the debug registers. The user is prompted via a dialog box to supply a string to use instead of the <variable>|<address> field.
- change [trace change <variable>|<address>]
specifies a variable or absolute memory address. When the value at that address has been written to, a message is displayed. If the debug registers are not enabled, writes of the same value do not halt the program.

The user is prompted via a dialog box to supply a string to use instead of the <variable>|<address> field.

- **trace** [trace <rest-of-command>]

specifies any variations of the **trace** command. The user is prompted via a dialog box to supply a string to use instead of the <rest-of-command> field.

Interrupt This command stops the execution of the program. An executing program is stopped and the source window is updated to display the current execution point.

Catch... [catch <signal-name>]

The **catch** command catches signals destined for the user's program. Upon receipt of one of these signals, the program is stopped. The user may then elect to send the signal to the program with the **continue** command (see the description of the **Continue** command in the "The sidebar command buttons" section).

The signal may be specified by number or name (for example, SIGINT). The user is prompted via a dialog box to supply a string to use instead of the <signal-name> field.

Ignore... [ignore <signal-name>]

The **ignore** command ignores signals destined for the user's program. The signal may be specified by number or name (for example, SIGINT). The user is prompted via a dialog box to supply a string to use instead of the <signal-name> field.

Dbxtra This command allows the user to enter complex commands for direct execution by **dbXtra**. The output from the debugger is displayed in a separate, read-only window that can be removed when no longer required.

The execute menu

This menu controls the execution of the user's program.

Run... [run <command-line-parameters>]

This command executes a program. The user is prompted for the program command line parameters. The program is then run. Program execution continues to completion unless a breakpoint or other event is encountered. An input/output window is created for program interaction.

Rerun [rerun]

The **rerun** command restarts the program from its beginning. The previous command line parameters are used. Program execution continues to completion unless a breakpoint or other event is encountered. An input/output window is created for program interaction.

Continue [cont]	The continue command restarts execution from the current breakpoint or event.
Skip... [skip <number-of-events>]	This command pops up a dialog box, allowing the user to specify the number of events to skip. Execution continues from the current location until the specified number of events have occurred.
Next [next]	The next command executes one source line, then stops. next does not follow function calls but stops after a function call. Use Step to follow execution into called functions.
Step [step]	The step command executes one source line, then stops. step follows the execution into called functions.
Goto [goto <selected-line>]	goto inserts a temporary breakpoint at the selected source line, then continues execution of the program until it reaches that line.
Jump [jump <selected-line>]	Jump resumes execution of the program from the selected line.
Return [return]	This command returns from the current function and stops in the calling function. The source window displays the line from which the original function was called.
Instruction ->	This submenu provides access to single-stepping by machine instructions.

The source menu

This menu navigates the user's program source code.

Browse file... [file <source-file-name>]

The specified file is displayed in the source window.

Source Path... [use <directory-list>]

This command specifies the list of directories used by **dbXtra** to find source files.

Search... /regular-expression[/]

Search allows text searches within the currently displayed file. Regular expressions may be used in the search string, which defaults to the currently selected text within the source window.

Function... [func <function-name>]

Function prompts for a function within the current program, then displays the source text around the first executable statement within that function.

dbXref... [dbxref <arguments>]

dbXref prompts for the **dbxref** command line arguments, then generates cross reference information for the current executable.

Up [up]

The source window changes to display the function that called the current function. This function now becomes the current function and variable scoping rules default to this function.

Down [down]

The source window changes to display the function that was called by the current function. This function now becomes the current function and variable scoping rules default to this function.

The info menu

This menu obtains information about the user's program.

Show...

This command shows the type and value of a program variable or expression. The selected text or user entered text can be used as the input for this command. The actual variable shown is determined by the scope rules of **dbXtra**. Explicit qualification can be used to avoid the default scoping rules. The optional keywords **enum**, **union**, or **struct** can be used to obtain information about aggregates. A function can be indicated with the "()" characters appended to its name.

Where

The **where** command displays the functions called by the user's program. The **up** and **down** commands can change the current function to another one in the call stack.

Display...

The **display** command adds a variable or expression specified by the selected text or by the user to the display window. Information is updated and displayed as the program executes. The display window is created if it does not exist.

Assign...

The **assign** command assigns a value to a variable specified by the selected text or by the user.

Which...

The **which** command obtains a fully qualified name. The identifier selected or entered by the user is displayed in a fully qualified form with the names of the outer scopes prepended to the name.

Print...	The print command displays the value of an expression.
Xref...	The xref command prints cross-reference information for a specified variable, function, or macro.
Variables	This command displays the names and values of variables local to the current function.
Registers	This command displays the contents of the general and floating point registers of the processor.
Memory	This command displays the contents of program memory.

The options menu

This menu sets **dbXtra** options and the environment variables of the program being debugged.

C/C++	The C/C++ command allows the user to switch between the C++ source code and the C code that is a result of the compilation process. This is useful for programmers interested in the C language equivalent of a C++ construct. The C++ program must be compiled with the -dbxC flag. Since dbXtra is a C++ and C debugger, the C code that is produced during the compilation of a C++ program can be debugged with dbXtra.
Preferences ->	This submenu contains a number of dbXtra switches that can be toggled to suit user preferences. See the set command for detailed information about these switches.
Environment->	This submenu contains buttons to set and unset environment variables accessible by the program being debugged.

The help menu

This menu contains various buttons providing online help information about the debugger.

The pop-up menu

Associated with the source window is a pop-up menu. This appears when the right mouse button is pushed.

Show... This command shows the type and value of a program variable or expression. The selected text or user entered text can be used as the input for this command. The actual variable shown is determined by the scope rules of **dbXtra**. Explicit qualification can be used to avoid the default scoping rules. The optional keywords **enum**, **union**, or **struct** can be used to obtain information about aggregates. A function can be indicated with the "()" characters appended to its name.

Display... [**display** <expression>]

The **display** command adds a variable or expression specified by the selected text or by the user to the display window. Displayed information is updated and displayed as the program executes. The display window is created if it does not exist.

Assign...

This command assigns a value to a variable specified by the selected text or by the user. The format of the expression is <variable> = <expression>.

Which... [**which** <identifier>]

This button obtains a fully qualified name. The identifier selected or entered by the user is displayed in a fully qualified form with the names of the outer scopes prepended to the name.

Xref [**xref** <identifier>]

This button displays, in a special window, cross-reference information about the specified identifier. The window created is not used by any other command and can be removed when no longer needed.

Up [**up**]

This button changes the source window to display the function that called the current function. The new function now becomes the current function and variable and scoping rules default to this function.

Down [**down**]

This button changes the source window to display the function that is called by the current function. The new function now becomes the current function and variable and scoping rules default to this function.

Configurable resources

As with most X applications, you can configure a number of resources to modify the appearance of **dbXtra** to suit your needs. A number of the resources, for example, to modify the font utilized, or the colors used, are described elsewhere.² The values of the resources can be specified in the *.Xdefaults* file in your home directory, in a file called *dbXtra* in */usr/lib/X11/app-defaults*, or in the *app-defaults* directory in your home directory.

The resources that control the appearance of **dbXtra** have the following syntax:

dbXtra*resource: *value*

The following list describes the various **dbXtra** resources that can be used:

configFile

The value for this resource is a string denoting the full pathname of the file containing the definitions for the various buttons and menus appearing in the source window. The default value for this is */usr/lib/dbxtra/dbXtra.config*.

lineNumberDigits

The value for this resource is an integer number specifying the amount of space (in terms of digits) to be allocated for displaying line numbers in the source window. The default value for this resource is 4. A value of 0 suppresses the appearance of line numbers.

controlButtonWidth

The value for this resource is an integer number specifying the amount of space (in terms of characters) to be used as the size of each button defined in the configuration file. This value also defines the number of characters that appear in the name of the button. If this value is not defined, it is calculated to be the number of characters in the longest button-name in the configuration file.

controlPanel

The value for this resource controls the appearance of the control window. A value of "wide" means that the control window appears as a paned window underneath the source window. A value of "narrow" means that the control window appears as a paned window underneath the source window, and that the button column extends to the bottom of the control window. The value of "separate" means that the control window is completely separate from the source window. The default value for this resource is "wide".

2. See *OSF/Motif Style Guide* and *OSF/Motif User's Guide*.

font The value of this resource defines the font to be used to display characters. By default, this resource is not defined and, therefore, the default font used by the X session is used.

control.font

The value of this resource defines the font to be used to display characters in the control window only. By default, this resource is not defined and, therefore, the default font used by the X session is used.

display.font

The value of this resource defines the font to be used to display characters in the display window only. By default, this resource is not defined and, therefore, the default font used by the X session is used.

output.font

The value of this resource defines the font to be used to display characters in the output window only. By default, this resource is not defined and, therefore, the default font used by the X session is used.

control.wrap

The value of this resource determines whether or not line wraparound is to be performed in the control window. If the value is "true", line wraparound is performed. If the value is "false", wraparound does not take place and if any line exceeds the width of the window, a horizontal scroll bar appears to allow access to the parts of the line that cannot fit within the available space.

help.wrap

The value of this resource determines whether or not line wraparound is to be performed in the help window. If the value is "true", line wraparound is performed. If the value is "false", wraparound does not take place and if any line exceeds the width of the window, a horizontal scroll bar appears to allow access to the parts of the line that cannot fit within the available space.

information.wrap

The value of this resource determines whether or not line wraparound is to be performed in the information windows. If the value is "true", line wraparound is performed. If the value is "false", wraparound does not take place and if any line exceeds the width of the window, a horizontal scroll bar appears to allow access to the parts of the line that cannot fit within the available space.

In addition to the above, various resources can be set to manipulate the various Motif widgets utilized. See *OSF/Motif User's Guide* for more details.

The curses interface

The **curses** interface provides both full screen and fully configurable keyboard access to the **dbXtra** debugger. For the screen interface, the relevant debugging information is automatically segregated into separate regions of the screen so that it can be easily accessible. The keyboard interface consists of a rich set of user configurable commands that enable users to perform cursor movements, text scrolling, and command editing at a single keystroke. The screen as well as keyboard interfaces can be disabled by specifying the **-s** option when invoking **dbxtra**.³

The screen interface

A window in **dbxtra** consists of one or more lines of your terminal display. Windows are always placed adjacent to each other, and they are separated by a line displayed in reverse video (or highlighted) format. The sum of the lines in all the windows, as well as separator lines, make up all the lines of the screen. Therefore, it is not possible to have one window overlap another, nor is it possible to have any lines on the screen not used by any window or separator line.

Each window in **dbxtra** has associated with it one or more *capabilities*. A window capability describes the contents of that window and the functions that can be performed on it. The capability of a window is displayed at the extreme left of the separator line immediately below the window. The window at the bottom of the screen is always the interaction window, and since there is no separator line below that window, its capability is never displayed on the screen. See Figure 1-3.

3. The **curses** version is referred to as **dbxtra**, since this is the command used to invoke it.

```

65
66     for (cardsinhand = 0: cardsinhand < 13: cardsinhand++, cardsleft--)
67     {
-> 68         nextcard = random() % cardsleft;
69         for (cardnum = 0 : cards[cardnum] || --nextcard >= 0: cardnum++)
:
70             cards[cardnum] = 1;
71             thehand[cardsinhand].suit = cardnum / 13;
72             thehand[cardsinhand].rank = 2 + (cardnum % 13);
73     }
74     return( cardsleft );
75 }
76
77
78 print_hand(fp,thehand)
79 FILE *fp;
80 card *thehand;
81 {
82     int i;
83     char buf[4][80];
C Throuse] file: deal.c      func: Deal
1 deal.Deal.cardsleft = 0
2 cardsinhand = 0
display LJ
Dealing 1 hands...
Dealing 1 hands...

prog LJ      dealer
(dbxtra)
(dbxtra)
(dbxtra)
(dbxtra)
(dbxtra) run
Floating point exception in Deal at line 68 in file "deal.c"
(dbxtra) █

```

Figure 1-3 Sample screen

The following is a list of the different capabilities:

C execution

The window with this capability contains the relevant C source of the program being debugged. An arrow (->) marker indicates the line to be executed next. This capability is indicated by a "C" in the separator line.

C++ execution

If the optional C++ support is invoked and C++ debugging information is available, then the window contains the relevant C++ source of the program being executed.

Again, an arrow (->) marker indicates the next statement to be executed. This capability is indicated by a "C++".

Browse

This window displays other source files — typically by using the `file`, `func`, or `show_xref` command. This capability is indicated by the word "browse" in the separator line. If this window does not exist, the functionality is provided by the C/C++ execution window.

Interaction

This window submits commands to the debugger as well as displays messages from the debugger. If the output window (refer to **Output** below) has not been created, this window also contains the output from the program being debugged. This window is always the bottom-most window on the screen and since no separator line is present, no indicator displays its capability.

Output This window displays the output from the program being debugged. The word “prog” indicates this capability. If this window is also the interaction window, then “prog” does not appear because there is then no separator line for this window.

Display This window displays the expressions (and their values) that have been specified using the **display** command. It can only be automatically created, if it does not already exist, when the **display** command is invoked. Expressions displayed in this window may be removed using the **undisplay** command. When no expressions remain in this window, the window itself is removed. “Display” indicates this capability.

Any of the C++, C, and Browse capabilities can apply to the same window. However, they cannot apply to windows with the Interaction, Output, or Display capabilities. Windows with these latter capabilities also cannot have any other capabilities, except that the interaction window may also be the output window. Capabilities cannot be shared, for example, there cannot be two windows with the Browse capability.

When invoked in screen mode (for example, without -s), the terminal display is initially partitioned into two windows — a program source window with C/C++ execution and browse capabilities, and an interaction window. It is possible to create a number of other windows to segregate the information to a greater extent. For example, the previous figure contains an example of a screen with a number of the windows described above. The window on the top of the screen contains the C execution as well as the Browse capabilities. The next window contains the Display capability, while the third window contains the Output capability. Finally, the window at the bottom is the interaction window.

NOTE Neither of the initial windows can be removed using the **close** command. However, if another window with source code is displayed, then the capabilities of the original source window when closed are transferred to the new window.

The keyboard interface

The **bindkey** command binds one or more keys on your terminal to various **dbxtra** editing commands. The syntax of the command is:

```
bindkey [ <key-sequence> <editing commands> | > filename ]
```

The *key-sequence* can be any valid key designation defined in the *terminfo* database. It can also be a combination of characters enclosed in double quotes ("), including control characters as entered from the keyboard. Control characters can also be specified using the ^X notation or as an octal number using the \nnn notation. For example:

```
(dbxtra) bindkey ""[A" "cursor_up"
```

That is equivalent to:

```
(dbxtra) bindkey "\033[A" "cursor_up"
```

The *editing commands* must be enclosed in double quotes ("). Key sequences are interpreted as soon as they are typed in. If one sequence is a prefix of another, then it is interpreted first, effectively hiding the longer one.

If *key-sequence* is not specified, definitions of all *key-sequences* are printed. This output can be redirected to a file (using "> filename") so that the bindings can be used in subsequent invocations of **dbxtra**.

To unbind a key bound with **bindkey**, use:

```
unbindkey key-sequence
```

The syntax for *key-sequence* is the same as that for **bindkey**.

When trying to bind a key that is already bound, the first argument cannot be given verbatim unless key interpretation is disabled (using the **keys off** command). To use a different set of bindings, utilize the contents of the **TERM** environment variables as the name of the key bindings file for each terminal (for example, *key.at386* or *key.wy60*). This way the correct file is automatically loaded if the following line is included in your *.dbxtrarc* file:

```
source key.$TERM
```

If more than one *editing command* is specified, each command must be separated by the ";" character. If no editing commands are specified, the current definition of the *key-sequence* is printed.

Editing commands

The following is a listing and description of the various commands that can be specified in the **editing commands** parameter of the **bindkey** command:

- bof** Move cursor to the beginning of file. The screen is scrolled or redrawn as necessary. If the current window is not a source window (for instance, if it was the interaction window), then the cursor is moved to the head of the buffer. The buffer contains up to 128 lines, so if more than 128 lines were written in this window the old lines are lost.
- bol** Move cursor to beginning of line (not including line numbers if in source window).
- close** Close the current window. If the current window is a source window, then another source window (the top neighbor by priority) is expanded, inherits the capabilities of the current window, and becomes the current window. If there is no other source window, the current window cannot be closed.

If the current window is the **Output** window, then the **dbxtra** session window (that is, the **Interaction** window) is expanded and inherits the **Output** capability. The **Interaction** window cannot be closed.

cursor_down

Move the cursor one line down. If the cursor is at the bottom of the window but not at the end of file, then the text is scrolled one line.

cursor_left

Move the cursor one character to the left.

cursor_right

Move the cursor one character to the right.

cursor_up

Move the cursor one line up. If the cursor is at the top of the window but not at the beginning of file, then the text is scrolled one line.

- del_bol** If the cursor is on a line being entered as a **dbXtra** command, then delete from the cursor location to the beginning of the command.

del_char

If the cursor is on the line being entered as a **dbXtra** command, then delete the character at the cursor location.

del_word

If the cursor is on the line being entered as a **dbXtra** command, then delete the word (string delimited by spaces) to the left of the cursor.

- eof** Move cursor to end of file (or buffer). It is a complementary function to **bof**.

- eol** Move cursor to end of line. If the current line is the line being entered as a **dbXtra** command, then the cursor is moved one column beyond the last character on the line.
- expand** Expand the current window by two lines. If the current window has two neighbors (above and below, not necessarily immediate neighbors) that are larger than one line, then each of them is shrunk by one line. Otherwise, the boundary between the current window and its neighbor is moved by two lines.
- expand_lower**
Expand current window by moving its lower edge by one line (provided that there exists a window that can be shrunk).
- expand_upper**
Expand current window by moving its upper edge by one line (provided that there exists a window that can be shrunk).
- goto_line**
Move cursor to line number. The user is prompted to enter the line number. This function only works for source windows.
- goto_mark**
Move cursor to the window location marked by **set_mark** (see below).
- half_page_back**
Scroll half page backward.
- half_page_forw**
Scroll half page forward.
- load_file**
Load file into current window. **dbxtra** prompts for the filename and this is entered from the keyboard. This command can only be used in a window with the **browse**, **C**, and **C++** capabilities.
- line_back**
Scroll one line backward.
- line_forw**
Scroll one line forward.
- move_capability**
Move a capability to the current window. The capability must follow the command. The supported capabilities are **browse** and **C**. When debugging in C++ mode, the C++ capability is also recognized.
- next_command**
Display the next line entered by the user to the **dbxtra** interaction window.

next_window

Change current window to the next window below the current one. If the current window is the bottom window, then the next window is the top window. An asterisk (*) is displayed on the left side of the separator line of the window the cursor is in.

page_back

Scroll one page backward.

page_forw

Scroll one page forward.

prev_command

Display the previous line entered by the user to the **dbxtra** interaction window.

prev_window

Change current window to the previous window (above). If the current window is the top window, then the previous window is the bottom window. An asterisk (*) is displays on the left side of the separator line of the window the cursor is in.

quote_char

Accept next character as is, that is, do not perform key interpretation on it.

redraw Redraw the screen.**screen_function**

Prompt for the name of a screen function and invoke it.

This function can invoke screen functions that are not bound to keys.

send_edit

Display the argument as a command to **dbxtra** but do not execute the command so that it can be edited before being invoked. For example, suppose the string "send_edit stop in" is bound to the key "^X". Then typing (Ctrl)X causes **dbxtra** to echo "stop in" and then wait for the user to complete the command.

send_echo

Display the argument specified as a command to **dbxtra** and also execute the command. For example, binding "^X" to "send_echo stop in main" causes **dbxtra** to echo and execute the command "stop in main" when (Ctrl)X is typed.

send_noecho

Execute the command specified but do not display the command in the interaction window.

set_mark

Set a mark in a window. This mark is used by **goto_mark** (see above).

show_xref

Before using **show_xref**, the **xref** command must be used first on some identifier. When this has been done, move the cursor (using **cursor_up**, for example) to an identifier in the output of **xref** then use **show_xref**. This displays in the source window the lines around the identifier the cursor is at.

shrink

Shrink the current window by two lines. If the current window has two neighbors (above and below), then each of the neighboring windows is expanded by one line. Otherwise, the boundary between the current window and its neighbor is moved by two lines. A window cannot be shrunk to size zero.

split

Splitting a source window creates a new source window with no capabilities and the new window becomes the current window. Splitting the **dbxtra** session window creates a program window.

toggle_line_numbers

Toggle with/without line numbers in source windows.

toggle_more

Toggle paging for interaction window.

toggle_wrap_around

Toggle wraparound of long lines in source windows.

word_left

Move cursor one word (single non-alphanumeric character or a string of alphanumeric and underline characters) left.

word_right

Move cursor one word (single non-alphanumeric character or a string of alphanumeric and underline characters) right.

Some of the commands require parameters. In these cases, the string following the command name, until a ";" or the end of the line is encountered, is used as the parameter. If the ";" character is preceded by a "\" (backslash) character, it is interpreted as part of the parameter rather than a parameter terminator. Any number of *editing variables* can be specified in the parameter string. The following are the editing variables supported and their functions:

\$command

evaluates to the full **dbxtra** command at the cursor line. For example, the sequence "cursor_up; send_edit \$command" causes **dbxtra** to display the command at the previous line and wait for the user to edit it.

\$events

evaluates to the *event-ids* of breakpoints at the cursor line (applies to source windows only).

\$expr

evaluates to the expression at the cursor location. The expression is derived according to the following rules:

1. If the character at the cursor location is alphanumeric (or "_"), then all adjacent alphanumeric characters to the left are included.
2. Otherwise, if the character at the cursor location is the dereference operator ("*"), it is included along with any spaces on its right.
3. If the next character on the right is alphanumeric (or "_"), it is included.
4. If the next character on the right is "(" or "[", then it is included along with all the characters up to (and including) the matching ")" or "]".
5. If the next character on the right is the field operator ("." or "->"), it is included.
6. If the next two characters on the right are "::" (the C++ class member operator) they are included.

Steps 3 through 6 are repeated until no more suitable characters are found.

\$file evaluates to source filename in the current window

\$keyboard

causes **dbxtra** to prompt the user with a ":" to enter a line and evaluates that line. For example, "move_capability \$keyboard" prompts the user for the name of the capability to be moved.

\$line evaluates to the source line number at the cursor location. For example, "send_echo stop at \$line" sets a breakpoint at the current cursor location.

\$var evaluates to the longest string of alphanumeric and "_" characters at the cursor position.

Using dbXtra

This chapter describes the commands that can be used with both the **curses** and Motif interfaces of **dbXtra**. Although the latter interface provides buttons and menus to execute commands, the commands described here can still be entered via the keyboard by using, for example, the **Dbxtra** option in the events menu.

Expressions and conditions

Variables and constants can be combined with operators to form expressions and conditions. Expressions have a value that can be printed or assigned to a variable. Conditions have a Boolean value that can be used to issue **dbXtra** commands selectively.

Table 1-1 contains a list of operators that can form expressions for various **dbXtra** commands. If a C++ program being debugged has overloaded any of them, the *overloaded operators are not used*. Note that the semantics of the casting operator are slightly different from those of the C/C++ casting operator.

Identifier resolution

A variable in an expression must be one of the identifiers in the program being debugged, and **dbXtra** first checks to ensure that this is true before resolving it. Variable names are resolved by first using the static scope of the *current function*. By default, this function is the one in which control has stopped in (for example, due to a breakpoint). It can be redefined to another function by using the **up/down** or **func** commands. If the name is not defined in the static scope, then the dynamic scope is used. If static and dynamic searches do not yield a result, another variable with the same name is used and a message is printed showing the fully qualified name. The name resolution procedure may be overridden by qualifying an identifier with the scopes containing it, for example, *sfile.variable*, where *sfile* is the name of the source file (without the *.c*) that contains the global *variable*.

Identifier resolution becomes more complicated for C++ programs due to overloading.

Table 1-1 Operators in expressions

Operator	Description
. or ->	structure or union member
[]	array indexing
sizeof(<i>type</i>)	size in bytes of <i>type</i>
~	one's complement
!	logical not
& <i>identifier</i>	address of <i>identifier</i>
* <i>identifier</i>	dereference of <i>identifier</i>
(<i>type</i>)	type cast
*	multiplication
/	floating point division
div	integer division
% or mod	modulo (or remainder)
+	arithmetic addition
-	arithmetic subtraction
<<	bitwise left shift
>>	bitwise right shift
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
==	equal to
!=	not equal to
&	bitwise and
	bitwise or
&& or and	logical and
or or	logical or

Types and casting

The types of expressions are checked before being processed by **dbXtra**. They can be overridden by using the syntax *type(expression)*. As a special case, *expression* can be cast to a pointer to a named type by using "&", as in &*type(expression)*.

You can also use the casting format typical in C, **(type)expression**, to a limited extent. In such cases, the **type** specifier cannot be the name of a derived type but it can be a structure or union tag. For example, in the following program segment:

```
struct foo ( int i; );
typedef struct foo bar;

main()
{
    bar x;

    ...
}
```

The debugger recognizes the following casts:

```
(struct foo) x
bar(x)
```

But it does recognize:

```
(bar) x
```

Although the dbXtra and C/C++ casting operators have the same usage, they have different semantics. In particular, if a value is cast by dbXtra to be of another type, then the bits representing that value are treated by dbXtra as being of the new type. So, for example, **(int) x**, where **x** is of type **float**, is not converted to an integer by discarding its fractional part. Instead, the bits representing its value are treated as an integer.

Executing the program being debugged

This section describes the various commands controlling the execution of the program being debugged.

call

The **call** command executes the object code associated with a named function, passing parameters to it at the same time. The syntax is:

```
call function(parameters)
```

The number and types of parameters must match. Execution of the function halts if any breakpoints are encountered, and dbXtra reenters the command level.

cont

The **cont** command continues the execution of the program being debugged from where it was suspended. You can optionally specify that a signal be sent to the program upon resumption of execution. The syntax is:

```
cont number
cont signal-name
```

If a signal is specified, the function continues as though it received the signal. Otherwise, the execution of the function is resumed as though it had not been stopped.

NOTE The **cont** command can only be used if execution of the program being debugged has not completed. For example, the **cont** command can not be used if execution was stopped due to a breakpoint.

goto

The **goto** command resumes execution of the program from where it was suspended until a specified line is reached. The syntax is:

```
goto line-number
```

As intervening breakpoints are encountered, several **goto** commands may have to be issued, before the specified line is reached.

jump

The **jump** command allows the user to continue execution of the target program at another point in code, skipping (not executing) intervening lines in the process. If a breakpoint is present at that point, it is ignored. The syntax is:

```
jump line number
```

The line number must be in the function at the top of the call stack.

next

The **next** command executes one or more source lines,⁴ then stops. When a function call is encountered, it is treated as a single statement. Execution of the function is only stopped if there is a breakpoint set at a statement executed by the function or by any function that it calls. Its syntax is:

```
next [n]
```

If *n* is present, then execute up to the next *n* source lines (default 1).

4. More than one program statement can be in a source line.

nexti

The **nexti** command is functionally similar to the **next** command, except that one or more machine instructions, instead of source lines in the program, are executed. The syntax is:

nexti [*n*]

Here *n* specifies the number of instructions to be executed (default is 1). If a call to a function is encountered, then the call instruction as well as all the instructions until the return represent one instruction.

rerun

The **rerun** command restarts a program that was executed with **run** (see **run** later in this list). Its syntax is:

rerun [*args*] [< *filename*] [>[>] *filename*]

If any arguments *args* are passed to **rerun**, they are appended to the arguments passed to **run** before being passed on to the program being restarted. If < *filename* is present, then it directs input from a file to the program. > *filename* directs output from the program to a file, while >> *filename* appends the output to a file.

return

The **return** command continues execution of the program being debugged from where it was stopped until control returns to a specified function.

Its syntax is:

return [*function*]

If *function* is not specified, the calling function is assumed.

run

The **run** command starts execution of the program being debugged, clearing the current stack and all variables from memory. The syntax is:

run [*args*] [< *filename*] [>[>] *filename*]

The meaning of its arguments are the same as those of **rerun**, except that *args* is always treated as a new list of arguments, replacing any that came before.

skip

The **skip** command allows the user to continue execution of the target program and ignore a given number of breakpoints. The syntax is:

skip [*n*]

Here *n* is the number of breakpoints to ignore (default is 1).

step

The **step** command executes one or more source lines, then stops. The syntax is:

step [*n*]

Here *n* is the number of source lines (default is 1). Unlike **next**, the **step** command does not treat functions as single statements. Instead, each statement in a function is treated as a separate statement.

stepi

stepi is functionally similar to the **step** command, except that one or more instructions, instead of source lines in a program, are executed. The syntax is:

stepi [*n*]

Here *n* is the number of instructions (default is 1).

Summary of commands described in this section	
call	Execute a function call.
cont	Resume execution of the program being debugged.
goto	Resume execution of the program until a specified line is reached.
jump	Resume execution at a different line in the current function.
next	Resume execution of the program until the next line is reached.
nexti	Resume execution of the program until the next instruction is reached.
rerun	Restart execution of the program being debugged.
return	Resume execution until control returns to a specified function.
run	Start execution of the program being debugged.
skip	Resume execution and ignore a specified number of breakpoints.
step	Execute the next statement in the program.
stepi	Execute the next instruction in the program.

Managing execution time events

This section describes the commands to create, remove, and query debugging events.

delete

The **delete** command removes one or more currently active **trace**, **stop**, and **when** events. The syntax is:

```
delete event-id [event-id ...]
```

The events corresponding to the specified *event-id*'s are removed. These numbers are printed by the **status** command, which is described later in this section.

disable

The **disable** command temporarily removes one or more currently active events. If an event does not exist, an error is reported. The syntax is:

```
disable event-id [event-id ...]
```

Only the events corresponding to the specified *event-id*'s are disabled.

enable

The **enable** command reinstates one or more disabled events. If an event does not exist, an error message is printed.

status

The **status** command displays a listing of the currently active events together with their corresponding *event-id*'s. The syntax is:

```
status [> filename]
```

Disabled events are prefixed with an asterisk character. For example:

```
(dbxtra) status
[1] when at "deal.c":45 { print "at 45: cardsleft = ",deal.
deal_hands.cardsleft;}
[2] * stop at "deal.c":47 if deal.deal_hands.player == 3
```

Here the first event is active but the second has been temporarily disabled.

stop

The **stop** command sets breakpoints within a program. Its various forms cause program execution to stop depending on whether:

- A particular source line is reached.
- A function is called.
- A variable is accessed or changed.
- Some condition has become true.

The syntax for the simplest form is:

stop [inst] if condition

This command stops the execution of the program being debugged before the next statement is executed whenever the Boolean condition specified evaluates to **true**. If the **inst** keyword is used, then execution stops before the next machine instruction is executed.

If a 386 debug register is available, it monitors execution. Otherwise, the Boolean condition is evaluated after each statement or instruction has been executed.

The next two forms allow for specific locations where execution is to be stopped:

stop at *line-number* [if condition]

stop inst at *address* [if condition]

If the first form is used, then execution of the debugged program stops before the statement on line *line-number* is executed. If the second form is used, then execution stops before the instruction at the specified address is executed.

If the optional **if** clause is used, then when execution reaches the specified line or address, the **condition** specified is evaluated and, if it evaluates to **true**, execution stops.

stop [inst] in *routine* [if condition]

Using this variant, the execution of the program being debugged stops before the *first* statement in the specified **routine** is executed. If the **inst** keyword is used, then execution stops before the first instruction in the specified **routine** is executed.

NOTE The first instruction is typically in the **prologue** section of the routine and may not correspond to the first instruction in the program source.

In either case, if the optional **if** clause is used, then the condition is evaluated before the first instruction or statement is executed, and the execution is stopped only if the condition evaluates to **true**.

stop [inst] [change] *variable* [if *condition*]
stop [inst] [change] *address* [if *condition*]

The execution of the program being debugged stops whenever the contents of *variable* are modified. If *address* is specified instead of *variable*, it is interpreted as an address in the data section of the program, in which case execution stops whenever any of the four bytes starting at that address are modified.

If the **inst** keyword is specified, then execution stops before the next instruction is executed (instead of before the next statement is executed). The **change** keyword is included for the sake of consistency with the other variants of the **stop** command, and its inclusion does not alter the behavior described.

In either case, if the optional **if** clause is used, then the specified *condition* is evaluated whenever the contents of the variable or memory location are modified, and execution stops only if the *condition* evaluates to true.

If a 386 debug register is available, it monitors execution. Otherwise, variable or memory locations are tested after each instruction or statement (depending on whether the **inst** keyword was specified) to determine if execution should be stopped.

stop [inst] access *variable* [if *condition*]
stop [inst] access *address* [if *condition*]

The execution of the program being debugged stops whenever the contents of *variable* are accessed.⁵ If *address* is specified instead of *variable*, it is interpreted as an address in the data section of the program, in which case execution stops whenever any of the four bytes starting at that address are accessed.

If the **inst** keyword is specified, then execution stops before the next instruction is executed (instead of before the next statement is executed).

In either case, if the optional **if** clause is used, then the specified *condition* is evaluated whenever the contents of the variable or memory location are modified, and execution stops only if the *condition* evaluates to true.

This command is **only** accepted if a 386 hardware debug register is available. If no such registers are available, an error message is printed. To free up a debug register, you have to cancel a command that has been allocated a debug register. The **status** command can list all the commands that are active.

5. This command is also triggered when variables are modified since variables have to be accessed to be modified.

trace

The various forms of the **trace** command can have **dbXtra** display diagnostic messages whenever any of the four events described earlier for **stop** occur during execution of the program being debugged. As these commands continuously update the source window and/or print messages, it is useful to specify a time interval between updates or prints. This is achieved through the use of the **\$sleep dbXtra** variable, which is described in section "Set/unset". The commands this variable affects are marked with a "+" symbol. Each variant of **trace** has a trace region associated with it.

A trace region is the set of program statements or instructions executed while the specified events are monitored. By default, the trace region consists of all the statements (and instructions) in the program being debugged. A trace is triggered whenever a specified event occurs during the execution of the statements (or instructions) in the trace region.

This syntax for the simplest form of this command is:

```
trace [inst] [if condition]      +
```

This command causes the source window to be updated so that the source line pointer is placed at the line containing the next statement to be executed. If **inst** is specified, then each instruction is disassembled and printed in the interaction window before it is executed.

If the optional **if** clause is specified, then **condition** is evaluated after each statement (or instruction) in the tracing region is executed, and tracing information printed only when **condition** evaluates to true. The tracing region for this command is the whole program.

```
trace line-number [if condition]  
trace inst address [if condition]
```

The first form of this command causes a message to be displayed whenever *line-number* is executed. This line must be from the file displayed in the source window. The second form causes a message to be displayed whenever the instruction at *address* is executed.

In both cases, if the optional **if** clause is specified, the *condition* is evaluated before the specified *line-number* or *address* is executed, and if it evaluates to true, the trace message is displayed.

```
trace [inst] in routine [if condition]      +
```

This command is only slightly different from the first **trace** variant described above — it limits the tracing region to the statements that are executed while the specified *routine* is active.

```
trace routine [in routine2] [if condition]
```

A message is displayed every time the specified *routine* is called, and also every time the routine returns. If the optional *in* clause is used, then the tracing region is limited to those statements executed while *routine2* is active.

If the optional *if* clause is specified, then the *condition* is evaluated whenever the trace is triggered, and trace information is displayed only if the *condition* evaluates to true.

```
trace [inst] [change] variable [in routine] [if condition]
trace [inst] change address [in routine] [if condition]
```

If the first form is used, a message is displayed whenever the contents of *variable* are modified. If the second form is used, then the message is displayed whenever any of the four bytes starting at the specified *address* are modified.

In either case, if the *inst* keyword is specified, the message is displayed before the next instruction (as opposed to the next statement) is executed. Please note that in the first form of the command, the *change* keyword is optional and is included for consistency with the other variants of the *trace* command; its inclusion does not alter the behavior described.

If the "*in routine*" clause is specified, the tracing region is limited to the statements (instructions) executed while the specified *routine* is active. If the optional *if* clause is specified, then whenever the trace is triggered, the specified *condition* is evaluated, and the message is displayed only if the condition evaluates to true.

If a debug register is available, it monitors changes to the specified *variable* or *address*. Otherwise, the code in the tracing region is single stepped, and the contents of *variable* or *address* are monitored after each statement (or instruction) is executed.

```
trace [inst] access variable [in routine] [if condition]
trace [inst] access address [in routine] [if condition]
```

If the first form is used, a message is displayed whenever the contents of *variable* are accessed. If the second form is used, the message is displayed whenever any of the four bytes starting at the specified *address* are accessed.

In either case, if the *inst* keyword is specified, the message is displayed before the next instruction (as opposed to the next statement) is executed.

If the *in* clause is specified, the tracing region is limited to the statements (instructions) executed while the specified *routine* is active. If the optional *if* clause is specified, then whenever the trace is triggered, the specified *condition* is evaluated, and the message is displayed only if the condition evaluates to true.

This command is **only** accepted if a 386 hardware debug register is available. If no such registers are available, an error message is printed. To free up a debug register, you have to cancel a command that has been allocated a debug register. The **status** command can list all the commands that are active.

when

The various forms of the **when** command can execute one or more **dbXtra** commands whenever any of the four events described above for **stop** occur during execution of the program being debugged.

The simplest form executes commands depending on some arbitrary condition:

```
when [inst] condition { command ; [ command ; ... ] }
```

Before each statement is executed, *condition* is evaluated and if it is true, the specified command(s) are executed. If the **inst** keyword is specified, then *condition* is evaluated before each instruction is executed. If a debug register is available, it monitors the state of the *condition*.

```
when at line-number [if condition] { command-list }  
when inst at address [if condition] { command-list }
```

The first form of this command causes the specified command(s) to be executed whenever *line-number* line is executed. The second form causes the command(s) to be executed whenever the instruction at the specified *address* is executed.

In both cases, if the optional **if** clause is specified, the *condition* is evaluated before the specified *line-number* or *address* is executed, and the command(s) are executed only when the *condition* evaluates to true.

```
when in routine [if condition] { command ; [ command ; ... ] }
```

The specified command(s) are executed before the first statement in *routine* is executed. If the optional **if** clause is specified, then *condition* is evaluated and the command(s) are executed only if the condition evaluates to true.

```
when [inst] change variable [if condition] { command-list }  
when [inst] change address [if condition] { command-list }
```

If the first form is used, the specified commands are executed whenever the contents of the specified *variable* are modified. If the second form is used, the commands are executed whenever any of the four bytes starting at *address* are modified.

In either case, if the **inst** keyword is specified, the commands are executed before the next instruction (as opposed to the next statement) is executed.

If a debug register is available, it monitors changes to the specified *variable* or *address*. Otherwise, the contents of *variable* or *address* are monitored after each statement (or instruction) is executed.

```
when [inst] access variable [if condition] { command-list }
when [inst] access address [if condition] { command-list }
```

If the first form is used, the specified commands are executed whenever the contents of *variable* are accessed. If the second form is used, the commands are executed whenever any of the four bytes starting at *address* are accessed.

In either case, if the **inst** keyword is specified, the commands are executed before the next instruction (as opposed to the next statement) is executed.

This format for the **when** command can **only** be used if a debug register is available. If no debug registers are available, an error message is reported and the commands do not have any effect.

Summary of commands described in this section	
delete	Remove an event from the events list.
disable	Temporarily inhibit the triggering of specified events.
enable	Restore the triggering of previously disabled events.
status	List the contents of the events list.
stop	Set a breakpoint in the program.
trace	Generate trace information for specified situations during program execution.
when	Execute user-specified debugger commands during the execution of the program.

Altering the execution environment

This section describes the commands that can change the execution environment of the program being debugged.

catch

The **catch** command specifies that the debugger intercept the specified signals destined for the program being debugged. This command is useful when a program being debugged does not handle signals such as interrupts. Its syntax takes the following forms:

```
catch number
catch signal-name
```

A signal can be specified by a number or by a name (for example, SIGINT). Signal names are case insensitive and the SIG prefix is optional. By default, all

signals are caught except SIGCONT, SIGCHLD, SIGALRM, and SIGKILL. (See `signal(S)` for more information on signals.)

ignore

The `ignore` command specifies that the debugger send the specified signals on to the program being debugged. This command is useful when a program being debugged handles signals such as interrupts. Its syntax takes the following forms:

```
ignore number
ignore signal-name
```

A signal can be specified by a number or by a name (for example, SIGINT). Signal names are case insensitive and the SIG prefix is optional. By default, all signals are caught except SIGCONT, SIGCHLD, SIGALRM, and SIGKILL.

setenv

The `setenv` command can modify values of environment variables that are accessible by the program being debugged. For example, if the program's output is being redirected to another screen, then the **TERM** environment variable accessed by the program would contain a different value from that accessed by **dbXtra**. Its syntax is:

```
setenv name value
```

Set the value of the environment variable *name*, in the program being debugged, to a specific *value*.

unsetenv

The `unsetenv` command can nullify the values of environment variables, which are accessible by the program being debugged. The syntax is:

```
unsetenv name
```

Summary of commands described in this section	
catch	The debugger will intercept the specified signals when they occur.
ignore	The debugger will pass the specified signals to the program.
setenv	Set an environment variable in the program's environment.
unsetenv	Remove an environment variable from the program's environment.

User interface commands

The commands in this section are all effective when **dbXtra** is invoked with the **curses** interface. Their behavior may be different when the **Motif** interface is used, these differences are recorded as well.

display

The **display** command adds an expression or a series of expressions to the display window. Figure 1-4 shows a **dbXtra** screen image containing the display window. The value of the expression is evaluated every time a stop point is reached. The new value is then printed in the display window.

When used with the **trace**, **stop**, or **when** commands, the **display** command can constantly monitor the values of expressions. The syntax is:

```
display [expression [, expression ...]]
```

The values are updated whenever the prompt is printed. If no expression is specified, then a list of the current expressions being displayed is output to the interaction window.

If *expression* evaluates to a (char *) data type — that is, if a string expression is specified — the maximum number of characters displayed will depend on the value of the **\$stringlen** variable. The default value of this variable is 512.

```

65
66     for (cardsinhand = 0; cardsinhand < 13; cardsinhand++, cardsleft--)
67     {
-> 68         nextcard = random() % cardsleft;
69         for (cardnum = 0 ; cards[cardnum] || --nextcard >= 0; cardnum++)
:
70             cards[cardnum] = 1;
71             thehand[cardsinhand].suit = cardnum / 13;
72             thehand[cardsinhand].rank = 2 + (cardnum % 13);
73     }
74     return( cardsleft );
75 }
76
77
78 print_hand(fp,thehand)
79 FILE *fp;
80 card *thehand;
81 {
82     int i;
83     char bufs[4][80];
C Floating point exception in Deal at line 68 in file "deal.c"
2 cardsinhand = 0
3 cardsinhand = 0
4 cardsinhand = 0; deal.deal_hands.cardsleft = 0
5 nextcard = 4213968

display/ls
(dbxtra) display cardsinhand.cardsleft
(dbxtra) display nextcard

(dbxtra) run
Dealing 1 hands...

Floating point exception in Deal at line 68 in file "deal.c"
(dbxtra) █
```

Figure 1-4 The display window

The display expressions appear in a special window called the *display window*. This window appears only when at least one expression has been specified for display. Therefore, if no display window exists when the display command is invoked, it is created. On the other hand, if the window already exists, it to accommodates new expressions. Note that for the Motif interface, this window is created separately from the Main window.

All the scrolling capabilities available for other windows are also applicable to the display window. If more expressions are specified than can be accommodated by the window, some of the expressions scroll off the screen. Their space is accessible by manipulating the scrolling region or by increasing the size of the display window.

forward/backward

The **forward** and **backward** commands scroll the source window forward and backward *n* lines (the default is one window). The syntax for these commands takes the following forms:

```
forward [n]
backward [n]
```

keys

The **keys** command can toggle keyboard translation functionality of dbXtra. The syntax is:

```
keys [ on | off ]
```

If neither **on** nor **off** are specified, **on** is assumed. Note that keyboard translations are *never* attempted when the keyboard input is meant for the program being debugged.

This command has no effect in the Motif interface.

more

The **more** command can toggle pagination of output to the interaction window. The syntax is:

```
more [ on | off ]
```

If neither **on** nor **off** are specified, **on** is assumed. Note that only the output from the current command is paginated. This means that a page break only occurs if the output from a command exceeds the number of lines in the interaction window.

This command has no effect in the Motif interface. Scroll bars can be manipulated to view information that has scrolled off a window.

screen

The **screen** command changes between screen mode and line mode.

The syntax for this command is:

```
screen [ on | off ]
```

If neither **on** nor **off** are specified, **on** is assumed. The **screen** command can “clear up” or redraw the screen if the screen has become contaminated with unexpected output.

NOTE The **screen** command does not work if the **-s** option has been used on the command line when invoking dbXtra. This command has no effect in the Motif interface.

undisplay

The syntax is:

```
undisplay
undisplay number
undisplay [expression [, expression ...]]
```

The first form of the command removes all the expressions associated with the last line in the display window.

The second form of the command removes all the expressions associated with a specified *number* line of the display window. If the line does not exist, an error message is printed.

If the third form is used, the specified *expression* is removed from the first line in the display window in which the expression appears.

Summary of commands described in this section	
display	Add an expression to the display window.
backward	Scroll the contents of the source window backwards by one window.
forward	Scroll the contents of the source window forward by one window.
keys	Enable/disable translation of keyboard sequences.
more	Paginate output to the interaction window.
screen	Activate the screen mode.
undisplay	Remove an expression from the display window.

Debugging registers, memory locations, and expressions

In this section, we describe the commands that examine and modify registers and memory locations, and examine expressions.

assign

The **assign** command assigns the value of an expression to a named variable. The syntax is:

assign *variable* = *expression*

dump

The **dump** command obtains the names and values of variables in a function. The syntax is:

dump [*function*] [> *filename*]

If *function* is not specified, then the current one is used. If the function is given as "", then all active variables are dumped.

fpregs/regs

These commands print the contents of the hardware registers. Their syntax are:

fpregs
regs

fpregs examines the contents of the hardware floating-point registers. If no hardware registers are present, an error message is printed. **regs**, on the other hand, can print the contents of the general purpose registers used by the CPU.

print

The **print** command obtains the values of expressions. The syntax is:

print *expression* [, *expression* ...]

If *expression* evaluates to a (char *) data type — that is, if a string expression is specified — the maximum number of characters displayed depends on the value of the **\$stringlen** variable.

up/down

The **up** and **down** commands move up or down the call stack by a specified number of levels. The syntax for these commands takes the following forms:

up [*count*]
down [*count*]

The default *count* is 1. The function at the new level of the call stack arrived at is used in resolving identifiers.

whatis

The **whatis** command obtains the declaration of a specified name. The syntax is:

whatis [enum | union | struct | class] *name*

The *name* can be qualified with block names. By default, *name* is assumed to be the name of a variable in the program. Any of the optional keywords can specify that *name* be used as the name of an aggregate rather than a variable.

NOTE The “()” characters can specify that *name* be used as the name of a function rather than a variable.

where

The **where** command obtains a listing of the active functions and their arguments. The syntax is:

where

whereis

The **whereis** command obtains the full qualification of all the symbols whose name matches a given identifier. The syntax is:

whereis *identifier*

The order in which the symbols are printed is not meaningful.

which

The **which** command obtains the full qualification of a given identifier. The syntax is:

which *identifier*

dbXtra returns the name of the identifier prepended with the names of the enclosing scopes that the identifier is associated with.

Displaying absolute memory locations

Contents of memory locations within the program's address space can be printed using the following syntax:

start-address , *end-address* / [mode]
start-address / [count] [mode]

If the first method is used, the contents of memory starting at *start-address* and continuing up to the *end-address* is printed. If the second format is used, the contents of *count* items memory starting at memory location *start-address* is printed. If the *start-address* is “.”, the address following the one printed most recently is used.

Addresses can be integers (hexadecimal numbers must be prefixed with *0x*), identifiers prefixed with *&*, or the *."* character (for next address to be printed). They can also be expressions made up of these elements and the operators *"+"*, *"-"*, *"**"*, unary *"-"*, and unary *"**"* (de-reference), each operator assuming its usual precedence. This precedence can be over-ridden using parentheses *"("* and *")"*.

The *mode* specifies how memory is to be printed; if it is omitted, the *previously specified* mode is used. The initial mode is *"X"*. The set of supported modes are defined in Table 1-2.

Table 1-2 Formats used when printing contents of memory locations

Specifier	Description
i	Print the machine instruction.
d	Print a short word in decimal.
D	Print a long word in decimal.
u	Print a short word in unsigned decimal.
U	Print a long word in unsigned decimal.
o	Print a short word in octal.
O	Print a long word in octal.
x	Print a short word in hexadecimal.
X	Print a long word in hexadecimal.
b	Print a byte in octal.
c	Print a byte as a character.
s	Print a string of characters terminated by a null byte.
f	Print a single precision real number.
g	Print a double precision real number.

Summary of commands described in this section	
assign	Modify the value of a named variable.
dump	List the names and values of variables in a scope.
fpregs/regs	Display the contents of registers.
print	Display the value of an expression.
up/down	Traverse the call stack.
whatis	Print the declaration of an identifier.
where	Display the contents of the call stack.
whereis	List the functions which declare the variable.
which	Print the extended name of a variable.
<i>start-address, end-address</i> [<i>mode</i>]	Display memory locations.
<i>start-address</i> /[<i>count</i>][<i>mode</i>]	Display memory locations.

Accessing source files

This section describes the commands used to browse or scan program sources.

dbxref

The *dbxref* command generates detailed information about the nature of the programming elements used in the program sources. It is similar to the *cxref* command available on most UNIX systems and in fact, you may execute the *dbxref* command using the syntax described here from the UNIX shell or from within your *makefiles*.

The syntax for the command is:

dbxref [-i] [*options*] *files*

Generate the cross-reference database for the specified *files*. If the *-i* flag is used, then generate the cross-reference information incrementally. In this case, *files* should only be the ones that contain new information to be put into the database. This assumes that the cross-reference database for the program already exists.

The *options* allowed are *-I*, *-D*, and *-U*, and these have the same meanings as those for the C compiler and preprocessor.

The database is saved in the file *programname.xref*, where *programname* is the name of the executable file being debugged. This can be overridden using the *-ofile* flag, where *file* is the name of the new file to contain the database.

edit

The **edit** command invokes an editor on source files. The syntax is:

```
edit [filename]
edit function
```

This command invokes an editor on the current source file if no argument is specified, *filename* if it is specified, or the source file that contains *function*. Which editor is invoked by default depends on the installation. The default can be overridden by setting the environment variable **EDITOR** to the name of the desired editor.

file

The **file** command switches to another source file in the source window. If no argument is specified, then the current source filename is printed. The syntax is:

```
file [filename]
```

func

The **func** command switches to another function in the source file window. The new function then becomes the *current function*. The syntax is:

```
func [function]
```

If no argument is specified, then print the name of the current function.

list

The **list** command displays lines in a source file. The syntax is:

```
list [line-number [, line-number]]
list function
list .
```

The first form lists the lines in the current source file from the first line number to the second, inclusive. If no lines are specified, the next 10 lines are listed. If the name of a function is given, **dbXtra** displays its first line and an equal number above and below. If the argument is *."*, then the current execution location is listed. If the number of lines specified exceeds the size of the display, then the top-most lines are scrolled off the top of the display.

use

The **use** command sets the list of directories to be searched when looking for source files. The syntax is:

```
use directory-list
```

The specified list replaces the list currently being used to locate source files.

xref

The **xref** command prints out detailed cross-reference information for a specified identifier. The syntax for this command is:

xref *identifier*

The cross-reference information is displayed in up to three sections. A section containing a list of functions and file or line number pairs, where the identifier is referenced, is always displayed. The other two sections are displayed only if the specified *identifier* is a function or macro. The second section contains a list of functions or macros that are called by the specified *identifier*. The third section contains a list of global variables, as well as parameterless macros, which are referenced by the named function. In both of these sections, an “M” as the first character of a line indicates that the identifier listed on the line is a macro.

For example, the following is the sample output when a global variable is used:

```
(dbxtra) xref suit_strings
Referenced from:
    format_hand    "deal.c": 102 107
    log_deal       "dealer.c": 329
```

Whereas, the following is the sample output when a function name is used:

```
(dbxtra) xref deal_hands
Referenced from:
    main           "dealer.c": 49
References functions/macros: "deal.c"
    Deal          45
    qsort         46
References variables/macros: "deal.c"
    card_compare  46
```

The bindable dbXtra editing command **show_xref**, as described in section “Editing commands”, can list the sources surrounding each of the line numbers produced by the **xref** command.

Regular expression search

The search commands search forward or backward in the current source file for a given pattern. The syntax for these commands takes the following forms:

/regular expression[/]
?regular expression[?]

Here “/” searches forward and “?” searches backward.

Summary of commands described in this section	
dbxref	Build/update xref database.
edit	Invoke the editor on a file.
file	Switch to another source file in source window.
func	Switch to another function in source window.
list	Display lines from a source file.
use	Set the list of source file directories.
xref	Print cross-reference information for specified name.
/expression[/]	Use in forward regular expression search.
?expression[?]	Use in backward regular expression search.

Miscellaneous commands

This section describes commands that are not strictly required to use **dbxtra**. However, they can provide short-cuts.

alias/unalias

The **alias** and **unalias** commands create an alias for a command or string, or remove an alias. Their syntax is:

```
alias
alias name name
alias name "string"
alias name (parameters) "string"
unalias name
```

When a command is entered, **dbXtra** first checks to see if it is an alias for either a command or a string. If it is an alias, **dbXtra** treats the input as though the string corresponding to it had been entered (with values substituted for any parameters). For example, to define an alias **rr** for the command **rerun**, you can say:

```
(dbxtra) alias rr rerun
```

To define an alias called **b** that sets a stop at a particular line you can say:

```
(dbxtra) alias b(x) "stop at x"
```

Subsequently, the command **b(12)** expands to **stop at 12**. You can get a listing of all of the currently defined aliases by using **alias** with no argument. The section “Predefined aliases”, contains a list of the predefined aliases in **dbXtra**.

help

This prints out a synopsis of **dbXtra** commands. The syntax is:

```
help [ command-name ]
```

If no *command-name* is specified, a brief description of all **dbXtra** commands is printed in the interaction window. Otherwise, a detailed description of the specified command is printed.

quit

quit exits **dbXtra**. The syntax is:

```
quit
```

set/unset

The **set** command defines values for **dbXtra** variables; **unset** deletes the debugger variable associated with *name*. Their syntax is:

```
set name [= expression]
```

```
unset name
```

The names of these variables cannot conflict with names in the program being debugged, and are expanded to the corresponding expression within other commands.

The following variables have a special meaning:

\$allmatch

When **\$allmatch** is set, the choosing of the “correct” symbol in a given context can not be done. In this case, all symbols are equally “reasonable”.

\$hexchars

\$hexints

\$hexoffsets

\$hexstrings

When set, **dbXtra** prints out characters, integers, offsets from registers, or character pointers, respectively, in hexadecimal.

\$listwindow

The value of this variable specifies the number of lines to list around a function or when the **list** command is given without any parameters. Its default value is 10.

\$machinecode

Setting **\$machinecode** affects the contents of disassembly output: the “function:line number” field in this output is replaced with the hexadecimal bit pattern representing the instructions decoded.

\$noclobber

If set, then redirecting the output of a command into an existing file fails.

\$nodebugregs

When **\$nodebugregs** is set, **dbXtra** can not attempt to use the 386 hardware debug registers. The **trace** access, **when** access, and **stop** access commands need this variable to be enabled to function. The hardware debug registers can not detect access or modification performed by the underlying UNIX operating system (such as those resulting from **read** and **write** system calls), and it may be desirable to turn off use of the debug registers when debugging program segments that make many system calls.

NOTE If floating-point hardware is not available, the operating system has to emulate floating-point operations and so certain operations may not be detected by the debug registers.

\$nodebugwarn

When **dbXtra** attempts to use the 386 debug registers, and there are no free registers, a warning message is printed, and slow emulation is performed. If **\$nodebugwarn** is set, the warning message is suppressed.

\$pccformat

Setting **\$pccformat** causes the disassembly instruction format to be generated by the AT&T C compiler (**pcc**). Under SCO UNIX, the default format is generated by the Microsoft C compiler.

\$printbases

When set, members of base classes inherited by the current object or class are printed in **print** or **whatis**.

\$sleep

This variable specifies the number of seconds that **dbXtra** sleeps at each statement when tracing the whole program in single-step (in screen mode only). The default value is 1 second.

\$stringlen

This variable specifies the maximum number of characters to be output for each string expression specified as a parameter to the **print** or **display** commands. The default value is 512.

\$unsafecall

When **\$unsafecall** is set, strict type checking is turned off for arguments to subroutine or function calls (for example, in the **call** statement). When **\$unsafeassign** is set, strict type checking between the two sides of an **assign** statement is turned off. These variables should be used only with great care, because they severely limit **dbXtra**'s usefulness for detecting errors.

\$windowshell

If set, then the **sh** command invokes a shell in the interaction window instead of turning the screen off automatically.

sh and !

The **sh** command passes a command line to the shell for execution. The **SHELL** environment variable determines which shell is used. The character **!** can be used instead of **sh**. The syntax is:

sh *shell command*

! *shell command*

This command has no effect in the Motif interface.

source

source reads and executes **dbXtra** commands from a named file. The syntax is:

source *filename*

Each line of the named file is read and interpreted by **dbXtra** as if it were entered from the standard input stream. When all the lines have been read, commands are again read from the standard input stream. Any input to the program being debugged is always read from the standard input stream.

version

This command simply prints version information about **dbXtra**.

Summary of commands described in this section	
alias/unalias	Create or remove an alias to a dbXtra command.
help	Display information about the debugger.
quit	Exit dbXtra session.
set/unset	Assign value to debugger variable.
sh/!	Execute a shell command.
source	Read and execute dbXtra commands from a file.
version	Print version information about dbXtra .

C++ specific debugging features

dbXtra is available in two versions — one facilitates debugging of programs written in C, and the other facilitates debugging programs written in either the C or the C++ programming languages. This section details the features that (because of the nature and/or the implementation of the C++ language) only apply to the version of **dbXtra** that facilitates debugging programs written using the C++ programming language.

Usage requirements

To debug C++ programs properly, the `-C++` command line option must have been used when `dbXtra` was invoked. This option is assumed by default. In addition, the proper C++ compiler flags must be used to compile the program. The following describes `dbXtra`'s behavior (when `-C++` is used) for the different compiler flags used by the SCO C++ compiler.

If the `-dbxC` flag was used to compile the C++ program, then the compiler also generates the C source that is derived from the C++ program. This enables `dbXtra` to display both C and C++ sources (via the `cc` command described later), and to display comprehensive information about C++ constructs. Commands that take expressions are also able to recognize both C++ identifiers and the C identifiers that resulted from them. These capabilities allow us to debug both C and C++ sources. This flag, however, does result in additional files being created and, therefore, requires more hard disk space.

If the `-dbx` flag was used, then the compiler does not produce C source. Hence, C source cannot be displayed. Commands, however, can recognize both C++ identifiers and their C derivatives, although the latter is hard to obtain since C source cannot be displayed. C++ information produced by `dbXtra` when this flag is used is as good as when the `-dbxC` flag is used. Although additional files are created, their number is less than the number created when `-dbxC` is used. This flag is useful when C++ source alone is to be debugged.

Like `-dbx`, the compiler does not produce C source with the `-g` flag. `dbXtra` cannot display C source, although C++ identifiers and their C derivatives are recognized by `dbXtra` commands. Again, this is of limited use, since the C derivatives are hard to obtain. However, the very limited amount of C++ debugging information produced prevents `dbXtra` from displaying much about C++ constructs. Hence, among the three flags, `-g` is the most restrictive in terms of allowing `dbXtra` to debug C++ programs.

Overloaded functions and operators

Overloaded functions and operators are recognized by **dbXtra**. If an overloaded function or operator is specified as an argument to a **dbXtra** command, all of its instances are listed and the user is asked to select those which should be used as arguments to the command. For example:

```
(dbxtra) whatis print
print is overloaded or ambiguous; do you mean
1 : card.Hand::print()
2 : card.CardGroup::print()
3 : card.Card::print()
Your choice (1-3/all/n,m,...) :
```

Responding with a digit to the prompt selects one of the instances of **print**. If some are required, then a comma separated list of digits is entered at the prompt. Otherwise, typing "all" selects all the instances.

An overloaded operator is specified as a function whose name is specified as **operator<op>**, where **<op>** is replaced by the actual overloaded operator. The exception to this is that the **new** and the **delete** operators are not prefixed with the **operator** string.

The **cc** command

When debugging C++ programs, this command toggles **dbXtra** between the C and C++ interaction modes. In the C++ interaction mode, variables as well as other identifiers produced by **dbXtra** commands appear as the C++ identifiers.

Whereas, in C interaction mode, the identifiers appear as translated (or mangled) identifiers. Regardless of the interaction mode, either the translated or untranslated identifiers can be specified as parameters to any of the **dbXtra** commands.

If the C++ program was compiled using the **-dbxC** compiler flag, use of the **cc** command also toggles the appearance of the contents of the program source window between C++ source and C source. This option can be useful to examine the translation of specific code segments as well as to execute segments of the C code rather than the C++ code.

In addition to having either the C++ or the C sources displayed in the source window, it is possible to view both the C++ as well as the generated C sources at the same time as shown in Figure 1-5.

```

53 {
54     i += x;
55 }
56
57 struct subclass : virtual base {
58     int x;
59     subclass () : base(0,0) { x = 0;};
60     subclass (int i) : base(0,0) { x = i;};
61     void f();
62 };
63
S-> 64 void subclass::f()
65 {
66     x++;
67 }
68
69 struct subclass {          /* sizeof subclass == 24 */
70     int x__subclass ;
71
72     struct __mptr *__vptr__subclass :
73     struct base *Pbase:
74     struct base Obase:
75 };
76
77
78
79
80
81
82
83 char f__subclassFv (__0this )
84 struct subclass *__0this :
85
86
87
88
89
90
91
92
93
94
95

```

File: classes.c Func: main

```

(dbxtra)
(dbxtra)
(dbxtra)
(dbxtra)
(dbxtra)
(dbxtra) stop in main
[2] stop in main
(dbxtra) run
[1] stopped in main at line 64
(dbxtra)

```

Figure 1-5 Display C++ and C sources at the same time

This command is only effective for C++ modules in the program being debugged. If the program being debugged is a C program, dbXtra attempts to unmangle any identifiers that appear to be C++ identifiers.

Predefined aliases

The following table shows the aliases that are defined when **dbXtra** is started up:

Alias	Command
c	cont
d	delete
h	help
e	edit
l	list
n	next
p	print
q	quit
r	run
rr	rerun
s	step
st	stop
j	status
t	where
sc	screen
noscreen	screen off
di	display
f	forward
b	backward
stopi	stop at instruction
tracei	trace instruction

Sample key bindings

The following is a set of sample key bindings. They are from your distribution of **dbXtra** and are a subset of those in the file called */usr/lib/dbxtra/keys/keys.at386*. They are known to work with the at386 *terminfo* entry included with your distribution of **dbXtra**. You can use the **source** command to load them into your **dbXtra** session.

Key name/sequence	Editing command
key_up	"cursor_up"
key_left	"cursor_left"
key_right	"cursor_right"
key_down	"cursor_down"
"[[U"	"page_forw"
"[[V"	"page_back"
"[[G^[[H"	"bof"
"[[G^[[Y"	"eof"
"[[H"	"bol"
"[[Y"	"eol"
"[[G^[[A"	"send_edit \$command"
"[[G^[[OY"	"close"
"[[OY"	"split; move_capability C"
"[[G^[[G"	"next_window"
"[[G^[[V"	"expand"
"[[G^[[U"	"shrink"
"[[T"	"send_echo next"
"[[G^[[T"	"send_echo step"
"[[S"	"send_echo cont"
"[[G^[[S"	"send_echo rerun"
"[[@"	"send_echo stop at \$line"
"^?"	"send_echo delete \$events"
"[G"	"goto_line"
"^P"	"prev_command"
"^N"	"next_command"

Chapter 2

CodeView debugger

CodeView debugger is an executable program that helps you debug software written in C and assembly languages.

CodeView 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. **CodeView** 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 is 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**.

Restrictions

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

Preparing programs for CodeView

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 can only use the debugger in assembly mode. This means that **CodeView** does not display source code or understand source-level symbols, such as symbols for functions and variables.

CodeView uses two methods to display program information. This first is source mode, in which a program's content is displayed as source lines. The second is assembly mode, in which a program's content is displayed as assembly-language instructions. **CodeView** can mix both methods, allowing a program's content to be displayed in both source lines and assembly-language instructions.

Programming considerations

Any source code that is legal in C or assembler can be compiled or assembled to create an executable file, and then debugged with **CodeView**. 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. The preferred method of developing programs is to create separate object modules, then link the object modules with your program. **CodeView** supports the debugging of separate object modules in the same session.

Also, **CodeView** 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 does not prevent **CodeView** 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.

CodeView compile options

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

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

If you are working with a high-level language, you may 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 to the source lines. After debugging, you can compile a final version of the program with the optimization level you prefer.

You cannot debug a program until you compile it successfully. **CodeView** 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.

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. 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 *Developer's Topics* for information about the **strip** command.

Preparing assembly programs

To use all the features of **CodeView** with assembly programs, you need to assemble with the Macro Assembler, **masm**.

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 **CodeView**. (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.

Since the assembler does not have its own expression evaluator, you have to use the C-expression evaluator. 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. CodeView, in turn, does not properly evaluate the value of the variable.

- Avoid the use of special characters in symbol names.
- Assemble with `-Mx` or `-MI` 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, CodeView 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` options. After assembling, link with the `-g` option to produce an executable file in CodeView format.

Starting CodeView

Before starting the debugger, make sure all the files it requires are available in the proper places:

<code>/usr/bin/cv</code>	The CodeView program file is located in the <code>/usr/bin</code> directory.
<code>/usr/lib/codeview/cv.hlp</code>	The CodeView help file is located in <code>/usr/lib/codeview</code> . If CodeView 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.
<code>program</code>	The executable file for the program that you wish to debug must be in the current directory or in a specified directory on the CodeView command line.

source.ext

Normally, source files should be in the current directory. However, if you specify a source file during compilation, that specification becomes part of the symbolic information stored in the executable file. 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, 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 used for this module. If no source file is specified, you must debug in assembly mode.

The **CodeView** 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". The *executablefile* is the name of an executable file to be loaded by the debugger. If you try to load a non-executable 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 **CodeView** format, the debugger starts in assembly mode and displays the following message:

No symbolic information

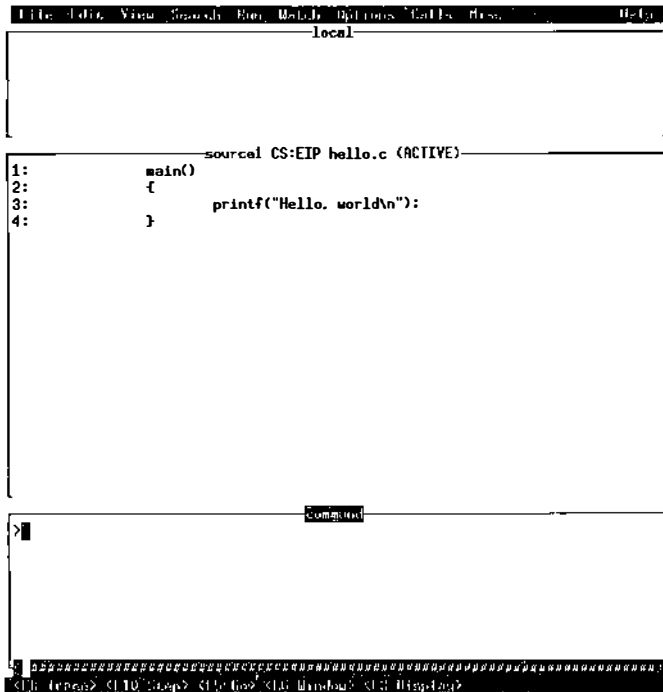
If you do not specify an executable file, **CodeView**, attempts to load *a.out*.

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 *hello*, you could start the debugger with the following command line:

cv hello

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



You can use **CodeView** options, as described in the section “Using CodeView Options” to override the default startup mode.

If your program is written in a high-level language, CodeView is now at the beginning of the startup 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 startup code to the beginning of your program. At this point, you are ready to start debugging your program.

Using CodeView options

The startup behavior of the debugger can be changed by specifying options in the command line. An option is a sequence of characters preceded by a dash (-). CodeView command-line options are case sensitive.

To use a file whose name begins with a dash, it first must be either preceded with a `/` or renamed so that CodeView does not interpret the dash as an option designator. To use more than one option in a command line, a dash must be associated with each option and a space must separate each element.

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

Specifying startup commands

Option

-c commands

The **-c** option specifies one or more *commands* to execute automatically upon startup. 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 startup 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 startup 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 is interpreted by CodeView rather than by the shell.

Examples

```
cv -cGmain calc calc.dat
```

This example loads CodeView with *calc* as the executable file and *calc.dat* as the argument. Upon startup, the debugger executes the high-level-language startup 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
```

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

This example loads the same file and argument, but the startup command directs the debugger to accept input from the file *input.fil* rather than 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 CodeView rather than by the shell.

Understanding CodeView windows

CodeView divides the screen into logically separate sections called windows. Each window is a discrete section of the display that operates independently of the other windows. Each window has a distinct function. The name of each window described here appears in the top of the window's frame:

- The source window displays the source code.
- The command window accepts debugging commands.
- The watch window displays the current value of selected variables.
- The local window lists the values of all variables local to the current function.
- The memory window displays the contents of memory.
- The register window displays the contents of the microprocessor's registers, as well as the processor flags.
- The 8087 window displays the registers of the coprocessor or its software emulator.

CodeView begins running with three windows displayed. The Local window is located at the top, the Source window in the middle, and the Command window at the bottom.

There are two methods available for opening other windows: choose a window through the View menu, and select some operations, for example, Watch, to open the appropriate window automatically.

Displays are automatically updated

CodeView continually and automatically updates the contents of all windows. If you want to interact with a particular window (to enter a command, to set a breakpoint), the window must be selected. The selected window is referred to as the "current" window. The "current" window can be identified by either the window's name being highlighted or by the presence of the "text" cursor.

To select a new current window, press the <F1> or <Shift><F6> to move the focus from one window to the next.

Entering commands into the source window temporarily causes CodeView to shift its focus to the command window. Whatever is typed is appended to the last line in the command window.

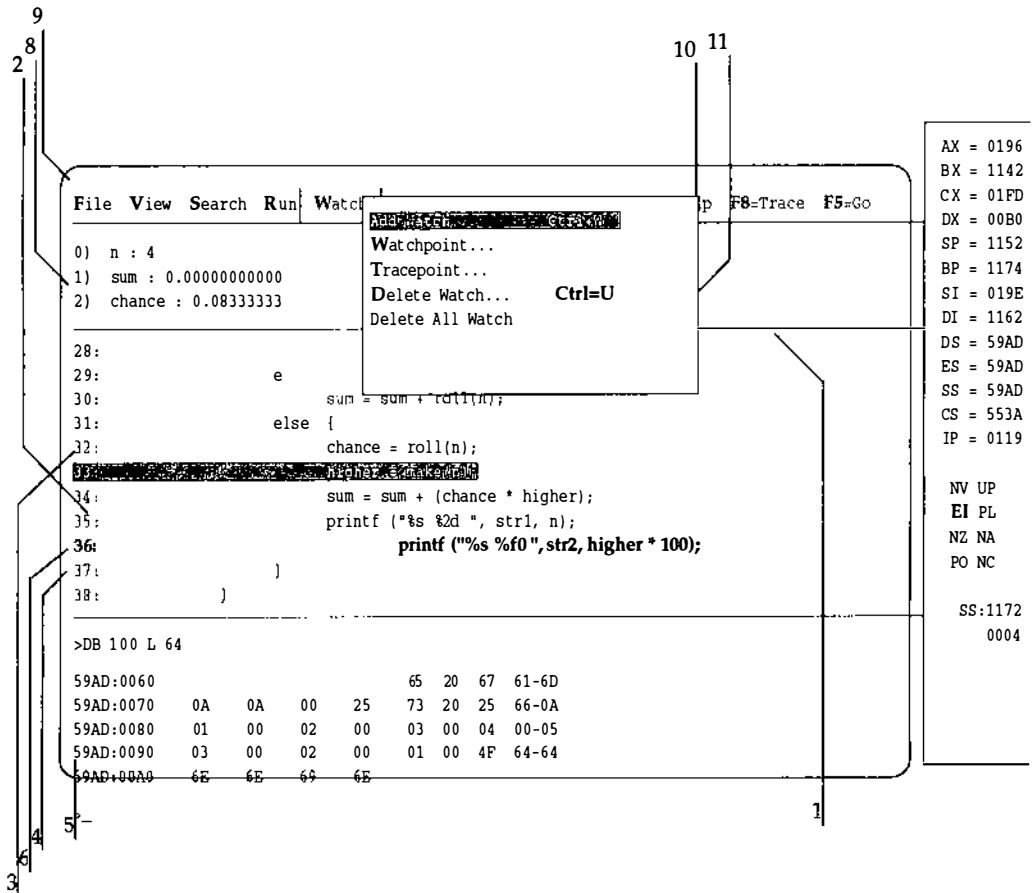
NOTE If the Command window is closed, CodeView beeps in response to any entries, and ignores the input.

Adjusting the windows

The relative position of a window is fixed. However, Maximize changes the window size and Close removes a window altogether. These command are accessed through the VIEW menu.

Using window mode

Figure 2-1 shows elements of the CodeView display. A description of each of the elements follows the figure.



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

2. The current location line (the next line the program will execute) is displayed 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.
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 automatically whenever you create watch statements.
9. The menu bar shows titles of menus and commands that you can activate with the keyboard.
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 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.

Executing keyboard commands

In **CodeView**, commonly used debugging and display manipulation commands are available through simple keyboard combinations. Commands can be entered with function keys, (Ctrl)-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 a window.

- | | |
|--------------|---|
| (F6) | Move the cursor between windows.
If the cursor is in the 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 window. |
| (Up Arrow) | Moves the cursor up one line in a window. |
| (Down Arrow) | Moves the cursor down one line in a window. |
| (PgUp) | Scrolls up one page.
If the cursor is in the source window, the source lines or assembly-language instructions scroll up. If the cursor is in the command window, the buffer of commands entered during the session scrolls up. The cursor remains at its current position in the window. |
| (PgDn) | Scroll down one page.
If the cursor is in the source window, the source lines or assembly-language instructions scroll down. If the cursor is in the command window, the buffer of commands entered during the session scrolls down. The cursor remains at its current position in the window. |
| (Ctrl)(Home) | Scroll to the top of the file or command buffer.
If the cursor is in the source window, the text scrolls to the start of the source file or program instructions. If the cursor is in the command window, the commands scroll to the top of the command buffer. |
| (Ctrl)(End) | Scroll to the bottom of the file or command buffer.
If the cursor is in the source window, the text scrolls to the end of the source file or program instructions. If the cursor is in the command window, the commands scroll to the bottom of the command buffer, and the cursor moves to the CodeView prompt (>) at the end of the buffer. |



Changing the screen

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

⟨Shift⟩F1) Display initial online 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”.

⟨F2) Toggle 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.

⟨F3) Switch among 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.

⟨F4) Switch to the output screen.

The output screen shows the output, if any, from your program. The CodeView screen returns after 3 seconds.

Controlling program execution

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

⟨F5) Execute 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) Set a temporary breakpoint on the line with the cursor, and execute 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), CodeView sounds a warning and ignores the command. This window command corresponds to the Go dialog command when it is given with a destination breakpoint.

⟨F8) Execute a Trace command.

CodeView 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 does not trace into function calls.

(F9) Set or clear 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, **CodeView** sounds a warning and ignores the command. This window command corresponds to the Breakpoint Set and Breakpoint Clear dialog commands.

(F10) Execute the Program Step command.
CodeView 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 A program's execution can be interrupted by pressing (or the user-defined kill key). This key can exit endless loops or interrupt loops that are slowed by the Watchpoint or Tracepoint commands (see "Managing watch statements"). The keystroke may not work if your program has a special use for that key.

Selecting from menus with the keyboard

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. Hold down the <Alt> key, simultaneously 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.
2. There are two ways to make a selection from an open menu:
 - 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.

- Press the key corresponding to the menu-selection mnemonic. The mnemonic is simply a single letter that represents the selection. In most cases, but not all, the letter is simply the first letter of the selection name.

3. After a selection is made from the menu, one of three things happens:
 - For most menu selections, the choice is executed immediately.
 - The items on the View and Options menus have an asterisk next to them if the option is on, or no asterisk if the option is off. Choosing the item toggles the option.
 - 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 right or left arrow keys 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 from the keyboard, as described in "Selecting form menus with the keyboard" The File menu includes selections for working on the current source or program file. The File menu selections are explained here.

Open Source... Load a source file from disk into the active source window.

When making this selection, a dialog box appears asking for the name of the new file 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 in assembly mode, **CodeView** switches to source mode). When finished viewing the file, the original file can be reopened, with the last location and breakpoints still marked.

It may not be necessary to open a new file to see source files for a different module of a program. **CodeView** 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 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.

- Open Module... Open one of the source files used to create the program currently being debugged. It reads that file into the active source window.
- Unix Shell Exit to a shell. This leaves **CodeView** temporarily to generate a new UNIX shell. To return to **CodeView**, type **exit** at the operating system command prompt.
- Print... Use the Print command to write selected text to a list file.
- Exit Terminate the debugger and return to the system.

The Edit menu

The Edit menu provides selections for manipulating text within a source window.

- Undo Reverse the most recent edit. Undo restores the current line to its previous condition. This only works if the cursor is still on the edited line.
- Copy Copy any highlighted (selected) line from the active window to the Clipboard. After **copy** is executed, the line remains in the active window but is no longer highlighted.
- Paste Copy text from the Clipboard to the active window.

The View menu

The View menu includes selections for opening, closing, sizing, and 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 “Executing Code”.

Source	Display the source code or assembly-language instructions of a program currently being debugged.
Memory	Display a contiguous region of memory.
Register	Display the current status of processor registers and flags.
8087	Display the contents of the coprocessor registers.
Local	Display all parameters of the currently executing function.
Watch	Display the values of selected variables and expressions.
Command	Enter command(s)
Help	Display help information about CodeView features and language keywords.
Output	Toggles between the CodeView screen and the program-output screen.
Maximize	Toggles between "maximized" and "normal" window modes.
Size	Is used to expand or contract the active window.
Close	Is used to close a window where the cursor is located.

The Search menu

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

Find... Search the current source file or other text file for a specified regular expression.

When selected, a dialog box opens. Enter the expression to be searched for, then press the `<Return>` key. **CodeView** begins searching from the current cursor position. If found, **CodeView** places the cursor on the line containing the expression. If the entry is not found, a message box opens informing you of the problem.

The following characters have a special meaning in regular expressions:

backslash	\
asterisk	*
left bracket	[
right bracket	]
period	.
dollar sign	\$
caret	^

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`.

NOTE The Case Sense selection from the Options menu has no effect on searching for regular expressions. If you are in assembly mode, the debugger automatically switches to source mode when the expression is found.

- Repeat Last Find** Repeat the last FIND string searched, without retyping the text being searched for. This selection is meaningful only after you have used the Search command to specify the current regular expression. If CodeView searches to the end of the file without finding another match for the expression, it wraps around and starts searching at the beginning of the file.
- Label/Function...** Search the current source window for an assembly-language label or a function name.

The Run menu

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

- Restart** Restart the current program, but do not begin executing it.
- Animate** Cause CodeView to highlight each statement in the source window following the CS:IP address as it is executed.

NOTE Although Restart retains breakpoints, along with pass count, and arguments (see “Executing code”), any instructions entered with the **Assemble** command are 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 here.

- Add Watch...** Display an expression and its value in the watch window.
CodeView updates the display whenever the value of the expression changes. A watch expression can be any valid expression or memory address in **<segment> : <offset>** form.
- Delete Watch...** Remove a watch expression from the watch window.
- Set Breakpoint...** Create a breakpoint at a specified source line or address.
- Edit Breakpoint...** Add, remove, change, enable, or disable breakpoints. It creates a breakpoint at a specified source line or address.
- Quick Watch...** Display the value of any selected expression. The current statement is selected automatically.

The Options menu

The Options menu allows options to be set that affect various aspects of the behavior of **CodeView**. Selections on the Options menu have an asterisk to the left of the selection name when the option is on. The status of the option is reversed each time the option is selected. By default, the Save Output and Bytes Coded options are on when **CodeView** is started. **CodeView** automatically turns Case Sense on (if the program is in C) or off (if the program is in another language) when debugging starts.

The selections from the Options menu are discussed here.

Source Window	Define the display characteristics of the active source window.
Memory Window	Define the displays characteristics of the active memory window.
Trace Speed...	Set tracing and execution speed for programs being debugged.
Language	Specify the expression evaluator used for the program being debugged. 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 to determine which expression evaluator to select. The Auto selection uses the C expression evaluator if the current source file does not have a <i>.bas</i> , <i>.f</i> , <i>.for</i> , or <i>.pas</i> extension. If you change to a source module with an <i>.asm</i> 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.
Screen Swap	Toggle screen swapping on or off.
Ignore Signals	Direct CodeView to trap or not trap signals before the signal is sent to the child process.
Emulator Scroll	CodeView 3.0 for the SCO UNIX system supports the child process screen via a VT100 emulator. When the program being debugged is executed it behaves as if its output is being written to a VT100 terminal.

NOTE For curses (program/library) based programs the Emulator Scroll, in most cases, should be disabled. For UNIX system line-oriented programs, Emulator Scroll should be enabled.

Case Sensitivity Toggle case sensitivity on or off.

When on, CodeView treats symbol names as case sensitive, lowercase letters are different from their corresponding uppercase ones. If off, symbol names are not case sensitive.

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 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 SCO UNIX C programs, the bottom routine should always be `start()`.

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 Misc menu

The Menu provides screen movement for those user who do not have access to keyboard function keys.

Next Window Move to the next window in CodeView.

Prev window Move to a previous window.

Execute to cursor Execute the program to the cursor position in the source window.

Memory mode Change mode of the memory window.

The Help menu

The Help menu lists the major topics in the help system. For help, open the Help menu, then select the topic that you want to view. Each topic may have any number of subtopics. Major topics must be selected 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 Help on Help command.

Using the help system

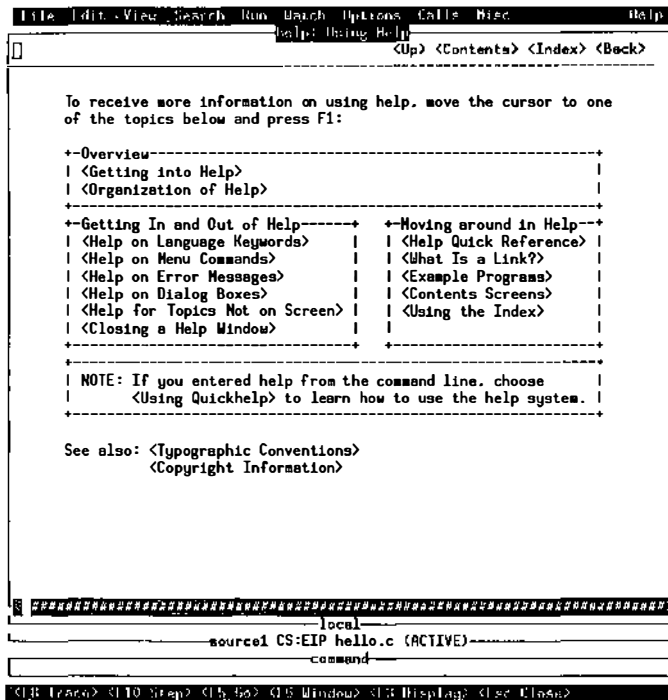
The **CodeView** online help system uses tree-structured menus providing quick access to help screens on a variety of subjects.

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

There are three methods available for requesting help:

<code><Shift><F1></code>	Keyboard access
H	Command line access
Help Menu	Window menu access

The Help screens are arranged in a hierarchical fashion. By selecting the `<Up Arrow>` or `<Down Arrow>` keys, allows movement through the hierarchy. Pressing `<PgDn>`, or `<PgUp>`, achieves scrolling in the help window, and `<Esc>` exits from the help screen. The Help system is cursor sensitive, by pressing either the `<F1>` or `<Return>` keys, while the cursor is on a help list item, information is be displayed. The following screen is an example of the Help system:



Using window commands

The command window allows commands requiring arguments or having no corresponding keyboard initiators to be entered.

There are many duplications in window and keyboard commands. Generally, the keyboard version of a command is more convenient, but the dialog version is more powerful. For example, to set a breakpoint on a source line, 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.

Entering commands and arguments

Command window commands are entered at the **CodeView** prompt (>). Type the command and arguments, then press the <Return> key.

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 command 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:

- | | |
|--------|---|
| | If the key is used in the CodeView window, it acts as a delete key (deleting characters). If a program is initiated through the CodeView command window, and the is used, it behaves as the user-defined interrupt key. For example, if you are watching a long display from a Dump command, you can press to interrupt the output and return to the CodeView prompt. |
| <Bksp> | Delete the previous character on the command line and move the cursor back one space. For example, if an error is made while typing a command, the <Bksp> key deletes the characters back to the error, and then retypes the rest of the command. |

Using the command window buffer

CodeView has a command buffer where the last two to four screens of commands and command output are stored.

When the cursor is in the command 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 "Using window mode."

Scrolling through the buffer is particularly useful for viewing the output from commands such as **dump** or **examine symbols**, whose output can 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.

However, 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 any warnings at startup time. 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:

```
"<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 Label1      ;* 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 separates 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 **Label1** 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 (*****).

CodeView expressions

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 assembler. **CodeView** has an expression evaluator for C that computes the value of source-level expressions.

Each expression evaluator 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. An expression evaluator can be changed. If a language is specified, other than the one used in the source file, then the expression evaluator still, if possible, recognizes the program symbols.

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 section 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 section describes how to switch the expression evaluator.

NOTE When you use a variable in an expression where that variable is not defined, **CodeView** 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. Also supported are the three memory operators (BY, WO, and DW), which are discussed in the section “Memory operators”. The memory operators are primarily useful for debugging assembly source code. The CodeView C-expression operators are listed in Table 2.1 in order of precedence.

Table 2-1 CodeView c-Expression operators

Precedence
(Highest)
1
2
3
4
5
6
7
8
9
10
(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 CodeView.

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:

<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. CodeView 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` can 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 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, CodeView 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 radices can 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”.

If the current radix is 16 (hexadecimal) or 8 (octal), you can enter decimal numbers in the special **CodeView** format **0ndigits**. 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. **CodeView** 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 (**0xdigits**).

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

Table 2.2 shows how a sample number (63 decimal) would be represented in each radix.

Table 2-2 C radix examples

Input radix	Octal
8	77
10	077
16	077

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 **CodeView**. 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 “Memory operators”. It generally is the appropriate expression evaluator with which to debug assembly code 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 “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.

- 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

- 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

- Take the address of a variable.

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

OFFSET var	&var
------------	------

- Use the **PTR** operator.

With **CodeView**, 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

- Access 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, **CodeView** treats an array name like any other variable.

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

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

In the second and third examples 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. **CodeView** 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 (.). **CodeView** 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.

CodeView 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 an extension to the C-expression evaluator.

Registers

Syntax

`[@@]<register>`

By specifically naming a register, the current value stored there can be accessed. Registers are rarely needed in source-level debugging, but they are used frequently for assembly-language debugging.

When an identifier is specified, **CodeView** first checks the symbol table for a symbol with that name. If the symbol is not found, it checks to see if the identifier is a valid register name. By using the “at” sign (@@) as a prefix, the identifier considers a register, regardless of any name in the symbol table. For example, if a program had both a symbol and register with the same name **AX**, to reference the register, @@AX would have to be used.

The register names known to **CodeView** are shown in Table 2-3.

Table 2-3 Registers

Type	Names	
8-bit high byte	AH	BH
8-bit low byte	AL	BL
16-bit general purpose	AX	BX
16-bit segment	CS	DS
16-bit pointer	SP	BP
16-bit index	SI	DI
32-bit general purpose	EAX	EBX
32-bit pointer	ESP	EBP
32-bit index	ESI	EDI

Addresses

Syntax

`[<segment>:]<offset>`

Addresses can be specified in **CodeView** 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 this example, the Dump Bytes command (DB) dumps 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 this example, the Dump Bytes command dumps memory starting at a point 10 bytes beyond the symbol `label`.

```
>DB ES:200
```

In this example, the Dump Bytes command dumps 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.

A range can be specified in two way. The first 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 a second address is not supplied, **CodeView** usually assumes the default range. Each command has its own default range. (The most common default range is 128 bytes.)

A range can also be specified by giving its starting point and the number of objects to be 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
```

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

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

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

```
>DB buffer L 20
```

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

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

The 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. Memory operators are used in assembly mode and are not necessary for high-level debugging.

The following operators are only available with the C-expression evaluator. They have the lowest precedence of any C operators and should not be confused with CodeView commands. They only build expressions used as arguments in commands. Each of the following operators are the same syntactically:

operator <address>

NOTE The examples that follow make use of the Display Expression (?) Command described in "Display expression command". The x format specifier causes the debugger to produce output in hexadecimal.

BY operator

This operator displays the byte of a requested address. **BY** operator can simulate the “**BYTE PTR**” operation of the Macro Assembler. This is useful for watching the byte pointed to by a register. The result is a short integer containing the value of the first byte stored at **address**.

Examples

```
>? BY sum
101
```

This example 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.

WO operator

This operator displays a word at a requested address. **WO** operator can simulate the “**WORD PTR**” operation of the assembler. This is useful for watching the word pointed to by a register, such as the stack pointer. The result is a short integer containing the value of the first two bytes stored at **address**.

Examples

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

This example returns the first word at the address of *sum*.

```
>? WO sp,
>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).

DW operator

This operator displays a double word at a requested address. **DW** operator can simulate the “**DWORD PTR**” operation of the Macro Assembler. This is useful for watching the word pointed to by a register. The result is a long integer containing the value of the first four bytes stored at **address**.

Examples

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

This example 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.

NOTE Do not 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**, 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.

Executing code

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.

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 a Trace, Program Step, or Go command is executed. 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.

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 “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.

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

NOTE If a section of code is executing with the Go or Program Step command, the program can be interrupted by pressing (Del). This terminates endless loops.

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 source mode, if the current source line contains a call, **CodeView** executes the first source line of the called routine. In this mode, **CodeView** 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 debugging Macro Assembler programs in source mode, the debugger still traces into the function. The debugger does not trace into an **INT** or **REP** sequence when in source mode.

To Trace into calls, use the Trace command. To execute calls without tracing into them, 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.

Command window

To execute the Trace command through a command window, 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 from the command window. If invoked from the keyboard, there would be no output from the commands, but the display would be updated to show changes caused by the command.

```
>S+      ; * c example
source
>.
9:          input (data,n,infmt);
>T3
34:          myopen(1, file, &status);
35:          for (i = 0; i < 100; i++)
36:              myread(1, infmt, &data[i]);

>
```

This C example sets the display mode to source, then uses the Source Line command to display the current source line. (See “Examining code,” for a further explanation of the Set Source and Source Line commands.) Note that the current source line calls the function `input()`. The Trace command then executes the next three source lines. These lines are the first three lines of the function `input()`.

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 EIP registers.

In source mode, if the current source line contains a call, CodeView 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.

Command window

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 invoked from the command window. In window mode, there would be no output from the commands, but the display would be updated to show changes.

```
>S+      ;* C example
source
>.
9:          input(data, n, inpfmt);
>P 3
10:         bubble(data, n);
11:         stats(data, n);
12:         return;
>
```

This example sets the display mode to source, then uses the Source Line command to display the current source line. (See “Examining code,” for a further explanation of the Set Source and Source Line commands.) Notice that the current source line calls the function **input()**. The Program Step command then executes the next three source lines. The first program step executes the entire function **input()**. The next two steps execute the functions **bubble()** and **stats()**, also in their entirety.

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), **CodeView** executes to the end of the program.

If the Go command is entered 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.

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 required, unless a breakpoint is encountered first). CodeView sounds a warning and takes no action if you try to go to a comment line or other source line that does not correspond to the code.

If the line required is in another module, use the Load command from the File menu to load the source file for the other module. Then move the cursor to the destination line and press the <F7> key.

Command Window

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 a *breakaddress* is given 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 invoked from the command window. If invoking from the keyboard, 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)
>
```

This example passes control to the instruction pointed to by the current values of the CS and IP registers. No breakpoint is encountered, so **CodeView** 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 this example, the display mode is first set to source (S+). (See “Examining code,” for information on setting the display mode.) When the Go command is entered, **CodeView** starts program execution at the current address and continues until it reaches the start of the subprogram BUBBLE.

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 watchpoint is reached. Pressing any key also stops automatic program execution.

Keyboard

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

Command Window

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 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.

NOTE 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 **CodeView**. 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, although they can be changed with the dialog version of the command.

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

Keyboard

To restart the program with a keyboard command, press <Alt>r to open the Run menu, then press Restart. The program is restarted. The program is ready to start executing from the beginning (but not actually running).

Command window

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*.

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
>
```

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

```
>L 6
>
```

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

Examining data and expressions

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

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 80387 register



Display expression command

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

Each of the expression evaluators (C, FORTRAN, and BASIC) accepts a different set of symbols, operators, functions, and constants, as explained earlier in this chapter in “CodeView expressions.” The resulting expressions can contain the intrinsic functions listed for the FORTRAN- and BASIC-expression evaluators. They can 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 can 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, use `N=N+1`, and with the BASIC-expression evaluator, use `LET N=N+1`. After being incremented, the new value is displayed.

The way in which the value of expressions are displayed is specified with using the Display Expression command. Type a comma after the expression, followed by a **CodeView** format specifier. The format specifiers used in **CodeView** are a subset of those used by the C `printf` function. They are listed in Table 2.4.

Table 2-4 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**.

NOTE The following format specifiers are not supported by **CodeView**. However, they are supported by the C **printf** function:

n
p
F
H

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

```
CALL PROCESS (s-11,*10,J,*20,*30)
```

This call must be called from the debugger as:

```
?PROCESS(IARG1,0,IARG2,0,0)
```

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

NOTE Do not use a type specifier when evaluating strings in FORTRAN or BASIC. 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.

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.

Command window

To view the addresses of symbols with a dialog command, enter a command line in one of the following formats, where **routine** is in a program unit, such as a C function or a BASIC subprogram, capable of having its own local variables.

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

The syntax combinations are listed in more detail below.

<code>X?<module>!><function>.<symbol></code>	the specified <i>symbol</i> in the specified <i>function</i> in the specified <i>module</i>
<code>X?<module>!<function>.*</code>	all symbols in the specified <i>function</i> in the specified <i>module</i>
<code>X?<module>!<symbol></code>	the specified <i>symbol</i> in the specified <i>module</i> (symbols within function are not found)
<code>X?<module>!*</code>	all symbols in the specified <i>module</i>
<code>X?<function>.<symbol></code>	the specified <i>symbol</i> in the specified <i>function</i> (looks for <i>function</i> first in the current module, and then in other modules from first to last)
<code>X?<function>.*</code>	all symbols in the specified <i>routine</i> (looks for <i>routine</i> first in the current module, then in other modules from first to last)
<code>X?<symbol></code>	look for the specified <i>symbol</i> 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

XL
displays the names, addresses, types, and values of all local variables for the current function

NOTE When you debug an assembly module, you cannot use the **function** 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 3D37:19B4 char _t[]
3D37:2956 int _scratch2 3D37:19B0 int _q
3A79:0010 int _main() 3A79:0010 int main()
3D37:19B2 int scratch0
3D37:0A10 char p[]
3D37:2954 int scratch1
3D37:19B4 char t[]
3D37:2956 int scratch2
3D37:19B0 int q
>
```

Example 3 shows all the symbols in the **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 would not 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

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

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

DI	Dump Integers (16-bit)
DJ	Dump Integers (32-bit)
DU	Dump Unsigned Integers (16-bit)
DV	Dump Unsigned Integers (32-bit)
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.

Command window

To execute a Dump command through a command window, 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 command 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 32 Kbytes.

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 command window.

The current dump address is the byte following the last byte specified in the previous Dump command. If no Dump command was 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, CodeView 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.

Each form of the Dump command displays data in a similar presentation. The command displays one or more lines, depending on the address or range supplied and the type of data being dumped. Each line displays the address of the first data element in the line, followed by the corresponding data. The data format depends on the specific command used: hexadecimal values and/or ASCII character representation of bytes, signed or unsigned decimal values of 16- or 32-bit integers, hexadecimal values of words or double words, or hexadecimal and decimal values of reals.

Hexadecimal values of bytes, integers, and words are separated by spaces. The words of each hexadecimal double-word value are colon-separated. ASCII characters in the range of ASCII 32 to ASCII 126 are printed without separation; unprintable characters, such as carriage returns and line feeds, are displayed as dots. Decimal values are separated by spaces.

The following sections discuss the variations of the Dump commands in order of the size of data they display. Each section includes an explanation and example of the presentation format.

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, or Watch Memory command. If none of these commands has entered during the session, the default type is bytes.

The Dump command displays one or more lines, depending on the address or range specified. 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 values and characters until the end of the *range* or, if no *range* is given, until the first 128 bytes have been displayed.

Each line displays up to 16 hexadecimal byte values, 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 (-).

Example

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

This example 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 specified.

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. 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.

Examples

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

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

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

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

Dump integers (16-bit)

Syntax

DI [<address> | <range>]

The Dump Integers command displays the signed decimal values of the words (2-byte values) starting at *address* or in the specified *range* of addresses. Each line displays up to eight signed decimal words. The command displays values until the end of the *range* or until the first 64 2-byte integers have been displayed, whichever comes first.

NOTE In this manual, an integer is considered a 2-byte value, since Code-View assumes that integer size.

Example

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

This example 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 integers (32-bit)

Syntax

DJ [<address> | <range>]

The Dump Integers command displays the signed decimal values of the double words (4-byte values) starting at *address* or in the specified *range* of addresses. The command displays values until the end of the *range* or until the first 64 2-byte integers have been displayed, whichever comes first. Each line displays up to eight signed decimal words.

NOTE In this manual, an integer is considered a 2-byte value, since CodeView assumes that integer size.

Example

```
>DJ 0 36
>dj &ldd[60]
001F:7FFFF694          -4          -3          -2          -1
001F:7FFFF6A4           0           1           2           3
001F:7FFFF6B4           4           5           6           7
>
```

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 (16-bit)

Syntax

DU [<address> | <range>]

The Dump Unsigned Integers command displays the unsigned decimal values of the words (2-byte values) starting at *address* or in the specified *range* of addresses. The command displays values until the end of the *range* or until the first 64 unsigned integers have been displayed, whichever comes first. Each line displays up to eight decimal words.

Example

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

This example 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 earlier sections on Dump integers.

Dump unsigned integers (32-bit)

Syntax

DV [<address> | <range>]

The Dump Integers command displays the signed decimal values of the words (2-byte values) starting at *address* or in the specified *range* of addresses. The command displays values until the end of the *range* or until the first 64 2-byte integers have been displayed, whichever comes first. Each line displays up to eight signed decimal words.

NOTE In this manual, an integer is considered a 2-byte value, since Code-View assumes that integer size.

Example

```
>DV 0 36
>dv &ldd[60]
001F:7FFFF694 4294967292 4294967293 4294967294 4294967295
001F:7FFFF6A4          0          1          2          3
001F:7FFFF6B4          4          5          6          7
>
>
```

This example 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 words

Syntax

DW [<address> | <range>]

The Dump Words command displays the hexadecimal values of the words (2-byte values) starting at *address* or in the specified *range* of addresses. The command displays values until the end of the *range* or until the first 64 words have been displayed, whichever comes first. Each line displays up to eight hexadecimal words.

Example

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

This example displays the word values from DS:0 to DS:36 (24 hexadecimal).

Dump double words

Syntax

DD [<address> | <range>]

The Dump Double Words command displays the hexadecimal values of the double words (4-byte values) starting at *address* or in the specified *range* of addresses. The command displays values until the end of the *range* or until the first 32 double words have been displayed, whichever comes first. Each line displays up to four hexadecimal double-word values; the words of each double word are separated by a colon.

Example

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

This example displays the double-word values from DS:0 to DS:36 (24 hexadecimal).

Dump short reals

Syntax

DS [<address> | <range>]

The Dump Short Reals command displays the hexadecimal and decimal values of the short (4-byte) floating-point numbers at *address* or in the specified *range* of addresses. If a *range* is specified, all values in the range are displayed. The command displays at least one value; only one value is displayed per line. Each line displays the hexadecimal values of the bytes in the number, followed by the decimal value of the number.

Example

```
>DS SPI
5E68:0000 0100 DB 0F 49 40 3.141593E+000
>
```

This example displays the short-real floating-point number at the address of the variable **SPI**.

Dump long reals

Syntax

DL [<address> | <range>]

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

Each line displays the hexadecimal values of the bytes in the number, followed by the decimal value of the number. The hexadecimal values are separated by spaces. If a *range* is specified, all values in the range are displayed. The command displays at least one value; only one value per line is displayed.

Example

```
>DL LPI
5E68:0000 0200 11 2D 44 54 FB 21 09 40 3.141593E+000
>
```

This example displays the long-real floating-point number at the address of the variable **LPI**.

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.

Each line displays the hexadecimal values of the bytes in the number, followed by the decimal value of the number. If a *range* is specified, all values in the range are displayed. The command displays at least one value; only one number per line is displayed.

Example

```
>DT TPI
5E68:0000 0300 DE 87 68 21 A2 DA 0F C9 00 40 3.141593E+000
>
```

This example displays the 10-byte floating-point number at the address of the variable **TPI**.

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.

Command window

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 address `arr1[0]`, with the 100 bytes starting at address `arr2[0]`. CodeView 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 "Using Codeview system-control commands" 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.

Command window

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 CodeView.

```
>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 “Modifying code or data.”

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.

Command window

To display the registers in the dialog window, 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 **EIP** register values is also shown. (The Register command can also be given with arguments, but only when used to modify registers, as explained in “Modifying code or data.”)

If an operand of the instruction contains memory expressions or immediate data, CodeView evaluates operands and shows 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 **EIP** registers are currently at a breakpoint location, the register display indicates the breakpoint number.

Examples

```
>S&
mixed
>R
>r
EAX=7FFFF8F4  EBX=00000000  ECX=00000048  EDX=00000000
ESP=7FFFF8A8  EBP=7FFFF8B8  ESI=7FFFF8EC  EDI=7FFFF8F4
DS=001F  ES=0003  FS=441140  GS=0001  SS=001F  CS=0017  EIP=00D4
NV UP EI PL NZ NA PE NC
0017:000000D4 8B4510  MOV     EAX,DWord Ptr [envp]    ;BR0  SS:7FFFF8C8=7FFFF8F4
>
```

This example displays all register and flag values, as well as the instruction at the address pointed to by the **CS** and **EIP** registers. The example is from a FORTRAN program, but applies equally well to BASIC and C programs.

8087 command

The 8087 command dumps the contents of the 8087 registers. If you do not have an 8087, 80287, 80387, or 80486 coprocessor chip on your system, then this command is invalid. This is due to the lack of kernel support for getting floating-point emulator information.

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. 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.

Command window

To display the status of the math coprocessor 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
cControl 1272 (Closure=affine      Round=nearest Precision=53-bit  )
              IEM=0 PM=1 UM=1 OM=0 ZM=0 DM=1 IM=0
cStatus  2820 cond=0000 top=5 PE=1 UE=0 OE=0 ZE=0 DE=0 IE=0
cTag      4BFF Instr Ptr=0017:00E4 Data Ptr=001F:7FFFF464
Stack      Exp  Mantissa      Value
cST(0) invalid 7FFF 8000000000000000 = +INF
cST(1) valid  4000 C924924924924800 = +3.14285714285714280000E+00
cST(2) zero    0000 0000000000000000 = +0.000000000000000000E+00
>
```

In this example, the first line of the dump shows the current closure method, rounding method, and the precision. The number 1272 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 affine (as in the example) or projective. 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, 53 bits (as in the example), 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 (2820 in the example), then interprets the bits of the register. The condition code (cond) in the example is the binary number 0000. The top of the stack (top) is register 5 (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 (4BFF in the example). It then gives the hexadecimal values of the instruction address (0017:00E4), the operand address (001F:7FFFF464).

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 three types of numbers that may be held in these registers. Starting from the top of the list, which is the top of the stack, register 0 contains infinity. This number is considered to be invalid. Register 1 contains a valid real number. Its exponent (in hexadecimal) is 4000 and its mantissa is C924924924924800. The number is shown in scientific notation in the rightmost column. Register 2 contains zero. A register may also be classed as empty, which means a position on the stack is occupied, but contains no data.

Managing breakpoints

CodeView 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:

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

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.

CodeView 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 can 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, **CodeView** 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.

Command window

To set a breakpoint, enter a command line in the command window 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 command window version of the command is more powerful than the keyboard version in that it allows a *passcount* and a string of *commands* to be given. 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"`, CodeView automatically displays the value of the variable *code*, then executes the Trace command each time the breakpoint is encountered. The Trace and Display Expression commands are described in "Executing code" and "Examining data and expressions," respectively.

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).

Examples

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

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

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

This example creates a breakpoint at the address of the routine STATS. The breakpoint is passed over nine times before being taken on the tenth 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.

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 **e** to select Edit Breakpoints. Use the Edit Breakpoints dialog box to clear all the breakpoints.

Command window

To clear breakpoints through a command window, 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. Use the Breakpoint List command (*BL*) if the numbers for each existing breakpoint is required. If an asterisk (*) is given as the argument, all breakpoints are removed.

Examples

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

This example removes breakpoints 0, 4, and 8.

```
>BC *
>
```

This example 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 “Executing code.”

Keyboard

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

Command Window

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

```
BC <list>
BC *
```

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
>
```

This example disables breakpoints 0, 4, and 8.

```
>BD *
>
```

This example 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, 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.

Command window

To enable breakpoints through a command window, enter a command line with the following syntax:

```
BC <list>
BC *
```

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. CodeView ignores all or part of the command if you try to enable a breakpoint that is not disabled.

Examples

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

This example enables breakpoints 0, 4, and 8.

```
>BE*
>
```

This example 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.

Command window

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 "qecho.c"!..7
1) D "qecho.c"!..11, 10-10, "? envp,s"
2) E "qecho.c"!0x17:0x00F3
>
```

In this example, breakpoint 0 is enabled at line 7 of file *qecho.c*. No pass count or break commands are set.

Breakpoint 1 is currently disabled, as indicated by a **D** after the breakpoint number. It also has a pass count of 10, meaning that the breakpoint is not taken until the tenth time it is encountered. The command string at the end of the line indicates that each time the breakpoint is taken, the value of *envp* is treated as a pointer to a string and displayed.

The line number for breakpoint 2 has an offset. This breakpoint was set in assembly mode.

Managing watch statements

Watch statement commands are among CodeView’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:

Watch (W)	sets an expression or range of memory to be watched
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 **CodeView**. They are not canceled when you restart the program being debugged. Therefore, you can set a complicated series of watch statements once, 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.

NOTE 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, **CodeView** displays the message *watch expression not in context*. When you exit from a function containing a local variable referenced in a watch statement, the value of the statement is displayed as *watch expression not in context*. 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

The Watch command sets a watch statement that specifies an expression (watch-expression 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.

When setting a watch expression, you can specify the format in which the value is displayed. Type the expression followed by a comma and a format specifier. If you do not give a format specifier, **CodeView** displays the value in a default format. See "Display expression command" 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, then type `A` (uppercase or lowercase) to select Add Watch. You can also select the Add Watch command directly by pressing `(Ctrl)w`. A dialog box appears, asking for the expression to be watched. Type the expression and press the `(Return)` key.

Command window

To set a watch-expression statement through a command window, enter a command line with the following syntax:

W? <expression>[,<format>]

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" for more information on format arguments.

Examples

The following example displays a watch statement in the watch window.

W? n

Deleting watch statements

The Watch Delete command enables the user to delete watch statements previously set with the Watch command.

When deleting a watch statement, 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, then type **D** (uppercase or lowercase) to select Delete Watch. You can also select the Delete Watch command directly by pressing <Ctrl>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, then press the <Return> key. The dialog box disappears, and the watch window is redrawn without the watch statement.

By selecting Clear in the dialog box, all the statements in the watch window can be deleted at once.

Command window

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

Y <number>

(The **Y** is a mnemonic for “yank.”) When setting 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.

To delete existing watch statements, specify the *number* of the statement to delete by using the Delete Watch command.

Use the asterisk (*) to represent all watch statements.

Examples

```
>Y 2
>
```

This command deletes watch statement 2.

```
>Y *
>
```

This command deletes all watch statements and closes the watch window.

Listing watchpoints

The Watch List command lists all previously set watchpoints and trace points 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.

Command window

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
>

```

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 commands.

Examples

```

>W? wo bp+4,d
>W? by bp-2,d
>

```

To watch this parameter, use the expression **bp+4** ; a parameter cannot be referenced by a specified name. The assembler does not emit symbolic information for parameters. The second command, **W? by bp-2,d**, is similar to the first, 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]**. See the following screen example for output.

```

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 0100020003 ..... SP = 09A2
BP = 09A4
70: ; First parameter largest SI = 0098
71: ; DI = 0A8C
72: mov BYTE PTR [bp-2],1 ; Load indicator value DS = 531C
73: ; of 1 into local variabl ES = 531C
74: jmp SHORT finished ; and finish up SS = 531C
75: next_test: CS = 52D7
76: mov ax,[bp+8] ; Load 3rd parm into ax IP = 005D
77: cmp [bp+6],ax ; If 2nd parm <= 3rd parm
78: jle last_test ; go to last test NV UP
79: ; EI NG
NZ AC
PE CY
>WW sp L 8
>WW bp L 8
>W? wo bp+4,d SS:09AA
>W? by bp-2,d 000F
>TPB arr L 5
>_

```

Examining code

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

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 where code is displayed. The two basic display modes are source mode, where 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, where the program is displayed with both source lines and assembly-language instructions.

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.

Command window

To set the display mode from the command 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.

When entering the dialog version of the Set Mode command, **CodeView** outputs the name of the new display mode: source, assembly or mixed.

Examples

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

These examples show the source mode being changed to source, assembly and mixed.

Unassemble command

The Unassemble command displays the assembly-language instructions of the program being debugged. It can 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 CodeView 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”, for more information.

Command window

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

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

By specifying an *address*, the disassembly starts at that address and shows the next eight lines of instructions. If a *range* is specified, the instructions within the range are displayed.

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.

View command

The View command can page through the source file or 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 O 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” for more information.

Command window

To display source lines through a command window, 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>]
```

If the View command is entered, while the display mode is assembly, **Code-View** automatically switches back to source mode. If a *linenumber* or *expression* is supplied, the display window is redrawn so that the source line corresponding to the given *address* appears at the top of the source window. If a *filename* with a *linenumber* is specified, that file is loaded.

If the View command is entered 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.

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.

Command window

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

The command puts the current program location (marked with 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, executing the program being debugged to somewhere near the start of the program, but the display has scrolled to a point near the end, the Current Location command returns the display to the current program location.

Stack trace command

The Stack Trace command displays routines that have been called during program execution (see the following note). 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 command window 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 uses the stack to transfer control to an independent program unit. In assembly mode, the term “procedure” may be more accurate.

If **CodeView** is used to debug assembly-language programs, the Stack Trace command works only if the procedures are called with the appropriate calling convention for that 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.

Command window

To display a stack trace through a command window, 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.

To view code at the point where the routine was called, enter a line number as an argument to either the View or Unassemble commands.

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

C example

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

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

Each routine is shown with the arguments it was passed, along with the last source line that it had been executing.

NOTE `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.

Modifying code or data

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

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 **CodeView**, 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 are assembled are lost as soon as the program is exited.

Assembled instructions are also lost when a program is restarted with the **Start** or **Restart** command. This occurs because the original code is reloaded on top of memory that may have been altered.

To test the results of an **Assemble** command, the **EIP** register (and possibly the **CS** register may have to be manipulated) to the starting address of the assembled instructions. If this is done, 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.

Command Window

To assemble code through a command window, 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 EIP registers). However, when the Assemble command is used, the assembly address is set to the address immediately following the last assembled instruction. When any command that executes code (Trace, Program Step, Go, or Execute) is entered, the assembly address is reset to the current address.

When the Assemble command is initiated, the assembly address is displayed. **CodeView** then waits for a new instruction to be entered, in the standard 8086-family instruction-mnemonic form. Instructions can be entered in uppercase, lowercase, or both.

To assemble a new instruction, type the desired mnemonic and press the <Return> key. **CodeView** 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, redisplay the current assembly address, and waits for you to enter a correct instruction. The caret symbol in the message points to the first character **CodeView** could not interpret.

The following eight principles govern entry of instruction mnemonics:

- The far-return mnemonic is **RETF**.
- 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.
- **CodeView** 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. These examples use the C notation for hexadecimal numbers.

- **CodeView** 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
```

- **CodeView** 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 here:

```
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**.

- **CodeView** supports all forms of indirect register instructions, as shown in the following examples:

```
ADD      BX, [BP+2] . [SI-1]
POP      [BP+DI]
PUSH     [SI]
```

- All instruction-name synonyms are supported. If you assemble instructions and then examine them with the **Unassemble** command (**U**), **CodeView** may show synonymous instructions, rather than the ones you assembled, as shown in the following examples:

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

- 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 crashes.

Example

```

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

```

The Unassemble command (U) shows 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

CodeView 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 here:

E	Enter (size is the default type)
EB	Enter Bytes
EA	Enter ASCII
EI	Enter Integers (16-bit)
EJ	Enter Integers (32-bit)
EU	Enter Unsigned Integers (16-bit)
EV	Enter Unsigned Integers (32-bit)
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.

Command window

To enter data (or code) to memory through a command window, 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 the given value 10 is specified with the Enter Words command in a list, 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 2-byte data are being entered for each command.

If *list* is not given, **CodeView** 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, **CodeView** 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, **CodeView** displays a new address to start a new display line.

- To return to the preceding value, type a backslash (\). When you return to the preceding value, 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

This example shows how to enter two word-sized values at the address PLACE.

>EW PLACE

3DA5:0B20 00F3._

This example illustrates the prompting method of entering data. When you supply the address where you want to enter data but supply no data to be entered there, **CodeView** displays the current value of the address and waits for you to enter a new value. The underscore in this example and the following examples represent 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 here:

>EW PLACE

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

You can then skip to the next value by pressing the **<Space>** key. **CodeView** responds by displaying the next value, as shown here:

>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. **CodeView** 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 here:

>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, **CodeView** 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."

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
>
```

This example 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." The *list* version of this command can be used to enter a string expression.

Example

```
>EA message "File cannot be found"
>
```

In this example, 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 (16-bit)

Syntax

EI <address> [<list>]

The Enter Integers command enters one or more word values into memory at *address* using the signed-integers format. With **CodeView**, 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, **CodeView** prompts for new values, which must be entered in decimal.

Examples

```
>EI 256 -10 10 -20
>
```

If the current radix is 10, this example replaces the three integers at DS:256, DS:258, and DS:260 with the decimal values **-10**, **10**, and **-20**. (The three addresses correspond to the three hexadecimal addresses DS:0100, DS:0102, and DS:0104.)

```
>EI 256
3DA5:0100 130F.-10
>
```

This example replaces the integer at DS:256 (hexadecimal address DS:0100) with **-10**.

Enter integers command (32-bit)

Syntax

EJ <address> [<list>]

The Enter Integers command enters one or more word values into memory at *address* using the signed-integers format. With **CodeView**, 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, **CodeView** prompts for new values, which must be entered in decimal.

Examples

```
>EJ 256 -10 10 -20
>
```

If the current radix is 10, this example replaces the three integers at DS:256, DS:258, and DS:260 with the decimal values -10, 10, and -20. (The three addresses correspond to the three hexadecimal addresses DS:0100, DS:0102, and DS:0104.)

```
>EJ 256
```

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

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

Enter unsigned integers command (16-bit)

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 CodeView, 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, CodeView prompts for new values, which must be entered in decimal.

Examples

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

If the current radix is 10, this example replaces the three unsigned integers at DS:256, DS:258, and DS:260 with the decimal values 10, 20, and 30. (These addresses correspond to the hexadecimal addresses DS:0100, DS:0102, and DS:0104.)

```
> EU 256
```

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

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

Enter unsigned integers command (32-bit)

Syntax

EV <address> [<list>]

The Enter Unsigned Integers command enters one or more word values into memory at *address* using the unsigned-integers format. With **CodeView**, 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, **CodeView** prompts for new values, which must be entered in decimal.

Examples

```
>EV 256 10 20 30
>
```

If the current radix is 10, this example replaces the three unsigned integers at DS:256, DS:258, and DS:260 with the decimal values 10, 20, and 30. (These addresses correspond to the hexadecimal addresses DS:0100, DS:0102, and DS:0104.)

```
> EV 256

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

This example 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, **CodeView** prompts for new values, which must be entered in hexadecimal.

Examples

```
>EW 256 10 20 30
>
```

If the current radix is 10, this example 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
>
```

This example 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, CodeView prompts for new values, which must be entered in hexadecimal.

Examples

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

If the current radix is 10, this example 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, CodeView 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
```

```
>
```

This example replaces the four numbers at DS:256, DS:260, DS:264, and DS:268 with the real numbers **23.479**, **0.25**, **-1650.0**, and **235.0**. (These addresses correspond to the hexadecimal addresses DS:0100, DS:0104, DS:0108, and DS:0112.)

```
>ES PI
```

```
3DA5:0064 42 79 74 65 7.215589E+022 3.141593
```

```
>
```

This example 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, CodeView 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
>
```

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

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

This example 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, CodeView 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
>
```

This example replaces the four numbers at DS:256, DS:266, DS:276, and DS:286 with the real numbers **23.479**, **0.25**, **-1650.0**, and **235.0**. (These addresses correspond to the 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
>
```

This example 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.

Command window

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 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.

Command window

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.

Example

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.

```
>M arr1[0] L arsize arr2[0] ;* c example
>
```

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

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 the previous Register command section.

Keyboard

The registers can be changed by moving to the register window, and typing on the register required.

Command window

To change the value of a register through a command window, 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**. **CodeView** 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 2.3. 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 2.5.

Table 2-5 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
>
```

This example changes the IP register to the value 256 (0100 hexadecimal).

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

This example 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
>_
```

This example changes the value of AX to 256 (in the current radix).

```
>R F UP EI PL
```

This example 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, 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.

Using CodeView system-control commands

This section discusses commands that control the operation of **CodeView**. The commands in this category are listed below:

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	controls redirection of CodeView output or input

The system-control commands are discussed in the following sections.

Help command

CodeView has a complete online-help system available.

Keyboard

In window mode, press <Shift><F1> to enter the online-help system.

Command Window

View the online-help system with the following command:

H

Quit command

The Quit command terminates **CodeView** and returns control to the operating system.

Keyboard

To quit **CodeView**, press <Alt>f to open the File menu, then press X to select Exit. The standard shell prompt is displayed.

Command Window

To quit CodeView through a command window, enter a command line with the following syntax:

Q

The standard screen shell prompt is displayed.

Radix command

The Radix command changes the current radix for entering arguments and displaying the value of expressions. The default radix when you start CodeView is 10 (decimal). Radixes 8 (octal) and 16 (hexadecimal) can also be set. Binary and other radices are not allowed.

The following seven conditions are exceptions; they are not affected by the Radix command:

- The radix for entering a new radix is always decimal.
- Format specifiers given with the Display Expression command or any of the Watch Statement commands override the current radix.
- Addresses output by the Assemble, Dump, Enter, Examine Symbol, and Unassemble commands are always shown in hexadecimal.
- In assembly mode, all values are shown in hexadecimal.
- 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.
- 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.
- The register display is always in hexadecimal.

Keyboard

You cannot change the input radix with a keyboard command.

Command window

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 **CodeView** 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
>
```

This example shows how 107 decimal, stored in the variable *prime*, is displayed with different radices.

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

In this example, the same number is entered in different radices, but the *i* format specifier displays the result as a decimal integer in all three cases. See "Examining data and expressions," for more information on format specifiers.

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.

Command window

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.

Keyboard

To execute the Screen Exchange command with a keyboard command, press the <F4> key. The CodeView screen returns in 3 seconds.

Command window

To execute the Screen Exchange command through a command window, enter a command line with the following syntax:

[time]

The output screen appears for the number of seconds specified. The default is 3 seconds.

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 command window. 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 filenames.

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, **CodeView** 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.

Keyboard

To find a regular expression with a keyboard command, press `<Alt>s` to open the Search menu, 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. **CodeView** 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 of the expression. Press `<Alt>s` to open the Search menu then press **R** to select Repeat Last Find. 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, 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. CodeView switches to assembly mode, if necessary, to display a label in a library routine or assembly-language module.

Command window

To find a regular expression through a command window, enter a command line with the following syntax:

`/[<regularexpression>]`

If **regularexpression** is given, CodeView 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, CodeView starts searching at the current cursor position and puts the cursor at the next line containing the regular expression. If you are in assembly mode, CodeView 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 CodeView 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, then press S 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.

Command window

To open a shell through a command window, 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 **CodeView**, 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 this example, **CodeView** 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 this example, 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 **CodeView** screen.

Tab set command

The Tab Set command sets the width in spaces that **CodeView** fills for each tab character. The default tab is eight spaces. You might want to set a smaller tab size if your source code has so many levels of indentation that source lines extend beyond the edge of the screen. This command has no effect if 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.

Command window

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.

Example

```
>.
32:                IF (X(I)) .LE. X(J)) GOTO 301
>#4
>.
32:                IF (X(I)) .LE. X(J)) GOTO 301
>
```

In this example, the Source Line command (.) shows the source line with the default tab width of eight spaces. Next the Tab Set command sets 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 Options menu and to turn any of 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. Selected options are denoted by an asterisk.

To change one of the Option settings, press the letter key corresponding to the option's mnemonic. This reverses the state of the option. Options can also be reversed by moving the highlight down with the arrow key, then pressing <Return>.

Command window

To view or change options with a dialog command, enter a command line with the following syntax:

O[<option> [+ | -]]

In this 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 here:

OF	Flip/Swap
OB	Bytes-Coded option
OC	Case-Sense option
OS	Symbols Off
OL	Show Symbol Address On
OI	Ignore Signals Off
OE	Emulator Scroll On
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
Flip/Swap Off
Bytes Coded On
Case Sense On
Show Symbol Address On
Symbols On
Ignore Signals Off
Emulator Scroll On
>of
Flip/Swap Off
>of+
Flip/Swap On
>
Bytes Coded on
Case Sense off
>OF
Save Output on
>OF-
Save Output off
```

In this example, 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 command window version of the Option command is particularly useful for redirected **CodeView** commands (which cannot access menus) and for making **CodeView** start up 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

CodeView provides several options for redirecting commands from, or to, devices or files. Furthermore, the debugger provides several other commands that 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.

Command window

The redirection commands are entered through the command window, as shown in the following sections.

Redirecting CodeView input

Syntax

```
< devicename
```

The Redirected Input command causes **CodeView** to read all subsequent command input from a device, such as another terminal or a file.

Examples

```
></dev/tty1a
```

This example redirects commands from the device (probably a remote terminal) designated as */dev/tty1a* to the **CodeView** terminal.

```
><infile.txt
```

This example redirects command input from file *infile.txt* to **CodeView**. You might use this command to prepare a **CodeView** session for someone else to run. You create a text file containing a series of **CodeView** commands separated by carriage-return-line-feed combinations or semicolons. When

you redirect the file, the debugger executes the commands to the end of the file. One way to create such a file is to redirect commands from **CodeView** to a file (see the section “Redirecting CodeView input and output”), then edit the file to eliminate the output and add comments.

Redirecting CodeView output

Syntax

[T]>[>] <devicename>

The Redirected Output command causes **CodeView** 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 this example, 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 this example, 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.

Resetting CodeView input and output

Syntax

```
= dev/tty
```

The Resetting Input and Output command causes **CodeView** to read all subsequent command input from, the console and write all subsequent output to the terminal.

Example

```
>=/dev/tty
```

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.

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. **CodeView** 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 this example, the user is sending a copy of a **CodeView** session to file *output.txt*. Comments are added to explain the purpose of the command. The text file contains commands, comments, and command output.

```
* Dump first 20 bytes of screen buffer
D #B800:0 L 20
.
.
.
< /dev/tty
```

This example 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 **CodeView**. 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 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 this example from a text file that might be redirected into **CodeView**, the **Delay** command slows 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 this example from a text file that might be redirected into **CodeView**, a **Comment** command prompts the user to press a key. The **Pause** command then halts execution until the user responds.

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

This example shows the output when the text is redirected into the debugger. The next **CodeView** prompt does not appear until the user presses a key.

Chapter 3

sdb: the symbolic debugger

The **sdb** program is useful both for examining core images of aborted programs and for providing an environment in which execution of a program can be monitored and controlled.

The **sdb** program allows interaction with a debugged program at the source-language level. When debugging a core image from an aborted program, **sdb** reports which line in the source program caused the error and allows all variables to be accessed symbolically and displayed in the correct format.

When executing, breakpoints may be placed at selected statements, or the program may be single-stepped on a line-by-line basis. To facilitate specification of lines in the program without a source listing, **sdb** provides a mechanism for examining the source text. Procedures may be called directly from the debugger. This feature is useful both for testing individual procedures and for calling user-provided routines, which give formatted printouts of structured data.

Using sdb

In order to use **sdb** to its full capabilities, it is necessary to compile the source program with the **-g** option. This causes the compiler to generate additional information about the variables and statements of the compiled program. When the **-g** option has been specified, **sdb** can be used to obtain a trace of the called functions at the time of the abort and interactively display the values of variables.

A typical sequence of shell commands for debugging a core image is:

```
cc -g prgm.c -o prgm
prgm
Bus error - core dumped
sdb prgm
main:25:      x[i] = 0;
*
```

The program **prgm** was compiled with the **-g** option and then executed. An error occurred, causing a core dump. The **sdb** program is then invoked to examine the core dump to determine the cause of the error. It reports that the bus error occurred in function **main** at line 25 (line numbers are always relative to the beginning of the file) and outputs the source text of the offending line. The **sdb** program then prompts the user with an *****, which shows that it is waiting for a command.

It is useful to know that **sdb** has a notion of current function and current line. In this example, they are initially set to **main** and 25, respectively.

Here **sdb** was called with one argument, **prgm**. In general, this takes three arguments on the command line:

1. The first argument is the name of the executable file that is to be debugged. It defaults to **a.out** when not specified.
2. The second argument is the name of the core file, defaulting to **core**.
3. The third argument is the list of the directories (separated by colons) containing the source of the program being debugged. The default is the current working directory.

In the example, the second and third arguments defaulted to the correct values, so only the first was specified.

If the error occurred in a function that was not compiled with the **-g** option, **sdb** prints the function name and the address at which the error occurred. The current line and function are set to the first executable line in **main**. If **main** was not compiled with the **-g** option, **sdb** will print an error message, but debugging can continue for those routines that were compiled with the **-g** option.

Example 3-1, at the end of the chapter, shows a more extensive example of **sdb** use.

Printing a stack trace

It is often useful to obtain a listing of the function calls that led to the error. This is obtained with the `t` command. For example,

```
*t
sub(x=2,y=3)      [prgm.c:25]
inter(i=16012)    [prgm.c:96]
main(argc=1,argv=0x7fffff54,envp=0x7fffff5c) [prgm.c:15]
```

This indicates that the program was stopped within the function `sub` at line 25 in file `prgm.c`. The `sub` function was called with the arguments `x=2` and `y=3` from `inter` at line 96. The `inter` function was called from `main` at line 15. The `main` function is always called by a startup routine with three arguments often referred to as `argc`, `argv`, and `envp`.

NOTE `argv` and `envp` are pointers, so their values are printed in hexadecimal.

Examining variables

The `sdb` program can be used to display variables in the stopped program. A variable is displayed by typing its name followed by a slash.

```
*errflag/
```

This causes `sdb` to display the value of variable `errflag`. Unless otherwise specified, variables are assumed to be either local to or accessible from the current function. To specify a different function, use the following to display variable `i` in function `sub`:

```
*sub:i/
```

The `sdb` program supports a limited form of pattern matching for variable and function names. The symbol `*` is used to match any sequence of characters of a variable name and `?`, to match any single character. Consider the following commands:

```
*x*/
*sub:y?/
**/
```

The first prints the values of all variables beginning with `x`, the second prints the values of all two-letter variables in function `sub` beginning with `y`, and the last prints all variables. In the first example, only variables accessible from the current function are printed. The last example, prints global program and `lib.c` variables. The following command displays the variables for each function on the call stack and `lib.c` variables.

```
**:* /
```

The **sdb** program normally displays the variable in a format determined by its type as declared in the source program. To request a different format, place a specifier after the slash. The specifier consists of an optional length specification followed by the format. The length specifiers are:

- b** one byte
- h** two bytes (half word)
- l** four bytes (long word)

The length specifiers are effective only with the formats **d**, **o**, **x**, and **u**. If no length is specified, the word length of the host machine is used. A number can be used with the **s** or **a** format to control the number of characters printed. The **s** and **a** formats normally print characters until either a null is reached or 128 characters have been printed. The number specifies exactly how many characters should be printed. There are a number of format specifiers available:

- c** character
- d** decimal
- u** decimal unsigned
- o** octal
- x** hexadecimal
- f** 32-bit single-precision floating point
- g** 64-bit double-precision floating point
- s** assume variable is a string pointer and print characters starting at the address pointed to by the variable, until a null is reached
- a** print characters starting at the variable's address until a null is reached
- p** pointer to function
- i** interpret as a machine-language instruction

For example, the variable **i** can be displayed with the following to print out the value of **i** in hexadecimal:

```
*i/x
```

The **sdb** program also knows about structures, arrays, and pointers, so that all of the following commands work.

```
*array[2][3]/
*sym.id/
*psym->usage/
*xsym[20].p->usage/
```

The only restriction is that array subscripts must be numbers.

NOTE As a special case, the following displays the structure pointed to by **psym** in decimal.

```
*psym[0]
```

Core locations can also be displayed by specifying their absolute addresses.

```
*1024/
```

The above command displays location 1024 in decimal. As in C language, numbers may also be specified in octal or hexadecimal, so the above command is equivalent to both:

```
*02000/
```

```
*0x400/
```

It is possible to mix numbers and variables so that the following refers to an element of a structure starting at address 1000:

```
*1000.x/
```

Where as the following refers to an element of a structure whose address is at 1000. For commands of the types `*1000.x/` and `*1000->x/`, the **sdb** program uses the structure template of the last structure referenced.

```
*1000->x/
```

The address of a variable is printed with `=`. The following displays the address of **i**.

```
*i=
```

Another feature whose usefulness will become apparent later is the following command which redisplay the last variable typed:

```
*./
```

Source file display and manipulation

The **sdb** program is designed to make it easy to debug a program without constant reference to a current source listing. Facilities are provided that perform context searches within the source files of the program being debugged and display selected portions of the source files. The commands are similar to those of the UNIX System text editor **ed**(C), documented in the *User's Reference*. Like the editor, **sdb** has a notion of current file and line within the current file. The **sdb** program also knows how the lines of a file are partitioned into functions, so it also has a notion of current function.

Displaying the source file

Four commands exist for displaying lines in the source file. They are useful for perusing the source program and determining the context of the current line. The commands are as follows:

p	prints the current line
w	window; prints a window of ten lines around the current line
z	prints ten lines starting at the current line; advances the current line by ten
control-d	scrolls; prints the next ten lines and advances the current line by ten; used to display cleanly long segments of the program

When a line from a file is printed, it is preceded by its line number. This not only gives an indication of its relative position in the file, but it is also used as input by some **sdb** commands.

Changing the current source file or function

The **e** command is used to change the current source file.

```
*e function
*e file.c
```

Either of above forms may be used. The first causes the file containing the named function to become the current file, and the current line becomes the first line of the function. The other form causes the named file to become current. In this case, the current line is set to the first line of the named file. Finally, an **e** command with no argument causes the current function and file named to be printed.

Changing the current line in the source file

The **z** and **control-d** commands have a side effect of changing the current line in the source file. The following paragraphs describe other commands that change the current line. There are two commands for searching for instances of regular expressions in source files. They are:

```
*/regular expression/
*?regular expression?
```

The first command searches forward through the file for a line containing a string that matches the regular expression, and the second searches backwards. The trailing **/** and **?** may be omitted from these commands. Regular expression matching is identical to that of **ed(C)**.

The `+` and `-` commands may be used to move the current line forward or backward by a specified number of lines. Typing a new-line advances the current line by one, and typing a number causes that line to become the current line in the file. These commands may be combined with the display commands so that, the following advances the current line by 15 and then prints ten lines.

```
*+15z
```

A controlled environment for program testing

After entering `sdb`, breakpoints can be set at certain lines in the source program. The program is then started with an `sdb` command. Execution of the program proceeds as normal until it is about to execute one of the lines at which a breakpoint has been set. The program stops and `sdb` reports the breakpoint where the program stopped. `sdb` commands may be used to display the trace of function calls and the values of variables. If the user is satisfied that the program is working correctly to this point, some breakpoints can be deleted and others set; then program execution may be continued from the point where it stopped.

A useful alternative to setting breakpoints is single-stepping. The `sdb` program can be requested to execute the next line of the program and then stop. This feature is especially useful for testing new programs, so they can be verified on a statement-by-statement basis. If an attempt is made to single-step through a function that has not been compiled with the `-g` option, execution proceeds until a statement in a function compiled with the `-g` option is reached. It is also possible to have the program execute one machine-level instruction at a time. This is particularly useful when the program has not been compiled with the `-g` option.

Setting and deleting breakpoints

Breakpoints can be set at any line in a function compiled with the `-g` option. The command format is as follows:

```
*12b  
*proc:b  
*b
```

The first form sets a breakpoint at line 12 in the current file. The line numbers are relative to the beginning of the file as printed by the source-file display commands. The second sets a breakpoint at the first line of `proc`. The last sets a breakpoint at the current line.

Similarly, breakpoints are deleted with the **d** command:

```
*12d
*proc:12d
*proc:d
```

In addition, if the command **d** is given alone, the breakpoints are deleted interactively. Each breakpoint location is printed, and a line is read from the user. If the line begins with a **y** or a **d**, the breakpoint is deleted.

The **B** command prints a list of the current breakpoints and the **D** command deletes all breakpoints. It is sometimes desirable to have **sdb** automatically perform a sequence of commands at a breakpoint and then have execution continue. This is achieved with another form of the **b** command:

```
*12b t;x/
```

This causes both a trace back and the value of *x* to be printed each time execution gets to line 12. The **a** command is a variation of the above command. There are two forms:

```
*proc:a
*proc:12a
```

The first prints the function name and its arguments each time it is called, and the second prints the source line each time it is about to be executed. For both forms of the **a** command, execution continues after the function name or source line is printed.

Running the program

The **r** command is used to begin program execution. It restarts the program as if it were invoked from the shell.

```
*r args
```

The above command runs the program with the given arguments as if they had been typed on the shell command line. If no arguments are specified, then the arguments from the last execution of the program within **sdb** are used. To run a program with no arguments, use the **R** command.

After the program is started, execution continues until a breakpoint is encountered, a signal such as **INTERRUPT** or **QUIT** occurs, or the program terminates. In all cases, after an appropriate message is printed, control returns to the user.

The **c** command may be used to continue execution of a stopped program. A line number may be specified, for example:

```
*proc:12c
```

This places a temporary breakpoint at the named line. The breakpoint is deleted when the **c** command finishes. There is also a **C** command that continues but passes the signal that stopped the program back to the program.

This is useful for testing user-written signal handlers. Execution may be continued at a specified line with the `g` command. For example:

```
*17 g
```

Will continue at line 17 of the current function. One use for this command is to avoid executing a section of code that is known to be bad. The user should not attempt to continue execution in a function different from that of the breakpoint.

The `s` command is used to run the program for a single statement. It is useful for slowly executing the program to examine its behavior in detail. An important alternative is the `S` command. This command is like the `s` command but does not stop within called functions. It is often used when you are confident that the called function works correctly but are interested in testing the calling routine.

The `i` command is used to run the program, one machine-level instruction at a time, while ignoring the signal that stopped the program. Its uses are similar to the `s` command. There is also an `I` command that causes the program to execute one machine-level instruction at a time, but also passes the signal that stopped the program back to the program.

Calling functions

It is possible to call any of the functions of the program from `sdb`. This feature is useful both for testing individual functions with different arguments and for calling a user-supplied function to print-structured data. There are two ways to call a function:

```
*proc(arg1, arg2, ...)  
*proc(arg1, arg2, ...)/m
```

The first simply executes the function. The second is intended for calling functions. (It executes the function and prints the value that it returns.) The value is printed in decimal unless some other format is specified by `m`. Arguments to functions may be integer, character or string constants, or variables that are accessible from the current function.

An unfortunate bug in the current implementation is that if a function is called when the program is not stopped at a breakpoint (such as when a core image is being debugged) all variables are initialized before the function is started. This makes it impossible to use a function that formats data from a dump.

Machine language debugging

The `sdb` program has facilities for examining programs at the machine-language level. It is possible to print the machine-language statements associated with a line in the source and to place breakpoints at arbitrary addresses. The `sdb` program can also be used to display or modify the contents of the machine registers.

Displaying machine language statements

To display the machine language statements associated with line 25 in function `main`, use the command:

```
*main:25?
```

The `?` command is similar to the `/` command except that it searches backwards. The default format for printing text space is the `i` format, which interprets the machine-language instruction. The `control-d` command may be used to print the next ten instructions. Absolute addresses may be specified instead of line numbers by appending a `:` to them, so that the following example displays the contents of address `0x1024` in text space:

```
*0x1024:?
```

The command displays the instruction corresponding to line `0x1024` in the current function:

```
*0x1024?
```

It is also possible to set or delete a breakpoint by specifying its absolute address, so that the following sets a breakpoint at address `0x1024`:

```
*0x1024:b
```

Manipulating registers

The `x` command prints the values of all the registers. Also, individual registers may be named by appending a `%` sign to their name. For example, on the 80286 the following displays the value of register `ax`:

```
*ax%
```

On the 80386 the following displays the value of register `eax`:

```
*eax%
```

Other commands

To exit **sdb**, use the **q** command.

The **!** command (when used immediately after the ***** prompt) is identical to that in **ed(C)** and is used to have the shell execute a command. The **!** can also be used to change the values of variables when the program is stopped at a breakpoint.

```
*variable!value
```

This sets the variable to the given value. The value may be a number, character constant, register, or the name of another variable. If the variable is of type **float** or **double**, the value can also be a floating-point constant (specified according to the standard C language format).

An sdb session

An example of a debugging session using **sdb** is shown in Example 3-1. Comments (preceded by a pound sign, **#**) have been added to help you see what is happening.

Example 3-1 Example of sdb Usage (Page 1 of 3)

```
sdb myoptim - ../common      # enter sdb command
Source path: ../common
No core image
*window:b                    # set a breakpoint at start of window
0x80802462 (window:1459+2) b
*r < m.s > out.m.s           # run the program
Breakpoint at
0x80802462 in window:1459: window(size, func) register int size;
boolean(*func)(); (
*t                             # print stack trace
window(size=2,func=w2opt)    [optim.c:1459]
peep()    [peep.c:34]
pseudo(s=.def^Imain;^I.val^I.;^I.scl^I-1;^I.endif)    [local.c:483]
yylex()    [local.c:229]
main(argc=0,argv=0xc00201bc,-1073610300)    [optim.c:227]
```

Example 3-1 Example of sdb Usage (Page 2 of 3)

```

*z                                     # print 10 lines of source
1459: window(size, func) register int size; boolean (*func)(); {
1460:
1461:  extern NODE *initw();
1462:  register NODE *pl;
1463:  register int i;
1464:
1465:  TRACE(window);
1466:
1467:  /* find first window */
1468:
*s                                     # step
window:1459: window(size, func) register int size; boolean (*func)(); {
*s                                     # step
window:1465:  TRACE(window);
*s                                     # step
window:1469:  wsize = size;
*s                                     # step
window:1470:  if ((pl = initw(n0.forw)) == NULL)
*s                                     # step through procedure call
window:1475:  for (opf = pf->back; ; opf = pf->back) {
*pl                                  # show variable pl
0x80886b38
*x                                     # print the 80286 register contents
    es/ 0x5f                        dx/ 0x67                        ds/ 0x5f
    ax/ 0x1cea                      cd/ 0x57                        ip/ 0x40
    di/ 0xfbd4                      cs/ 0x57                        bx/ 0xce6
    flgs/ 0x202                     bp/ 0x1cd6                     sp/ 0x1cce
    si/ 0x5b50                      ss/ 0x5f
0x570040 (main+4): lea -8(%bp),%d: [0x7fffffff]
*x                                     # print the 80386 register contents
    eax/ 1                          ecx/ 0x402e2c                  edx/ 0xbffffca8
    eox/ 0x17                       esp/ 0xbffffc54                  edp/ 0xbffffc60
    esi/ 0x16                       edi/ 0x15                      eip/ 0x16b
    flag/ 0x13296                   trap/ 0xe                      err/ 7
0x16b (main+4):  lea  -8(%ebp),%edi  [0x7fffffff]

```

Example 3-1 Example of sdb Usage (Page 3 of 3)

```

*pl[0]                                # dereference the pointer
pl[0].forw/ 0x80886b6c
pl[0].back/ 0x80886ac8
pl[0].ops[0]/ pushw
pl[0].uniqid/ 0
pl[0].op/ 123
pl[0].nlive/ 3588
pl[0].ndead/ 4096
*pl->forw[0]                          # dereference the pointer
pl->forw[0].forw/ 0x80886ca0
pl->forw[0].back/ 0x80886b38
pl->forw[0].ops[0]/ call
pl->forw[0].uniqid/ 0
pl->forw[0].op/ 9
pl->forw[0].nlive/ 3584
pl->forw[0].ndead/ 4099
*pl!pl->forw                          # replace pl with pl->forw
*pl                                  # show pl
0x80886b6c
*c                                  # continue
Breakpoint at
0x80802462 in window:1459: window(size, func) register int size;
boolean (*func)(); {
*s                                  # step
window:1459: window(size, func) register int size; boolean (*func)(); {
*s                                  # step
window:1465: TRACE(window);
*size                              # show function argument size
3
*D                                  # delete all breakpoints
All breakpoints deleted
*c                                  # continue
Process terminated
*q                                  # quit sdb
$

```


Chapter 4

adb: Absolute debugger

adb(CP) is a source level debugger for C and assembly language programs. It provides a set of commands to examine, debug and repair executable binary files as well as examine non-ASCII data files. **adb** provides a method of looking at the contents of files in a variety of formats and structures. The following aspects of **adb** are outlined in this chapter:

- start **adb**
- display program instructions and data
- run, breakpoint, and single-step a program
- patch program files and memory

Starting adb

By using the following form, any executable C or assembly language program file can initiate the debugger:

adb [*filename*]

filename is the name of the file to be debugged. The type of file can be data, program or core. The **adb** program opens the file and prepares its text (instructions) and data for subsequent debugging. By initiating **adb** with the **-w** option, changes and corrections can be made in a program file. See "Patching Binary Files" later in this chapter for details. Once started, **adb** prompts with an asterisk (*) and waits for commands to be entered.

If the filename provided does not exist or is in the wrong format, **adb** displays an error message. For example:

```
adb sample
adb: cannot open 'sample'
```

adb can be started without a filename. In this case, **adb** searches for the default file *a.out* in the current working directory. The executable file *a.out* is created by the C compiler when a program is compiled and linked successfully. **adb** can be invoked by either of the following methods:

```
adb
or
adb a.out
```

Starting adb with a core image file

Core image files, that cause fatal system errors can be examined through **adb**. Core image files contain the contents of the CPU registers, stack, and memory areas of the program at the time the error occurred.

To examine a core image file with its corresponding program, provide both the core and the program file names. If no *corefile* name is provided, **adb** searches for the default *corefile* in the current directory and use it regardless of whether or not it belongs to the program. In the following example the *programfile* is the filename of the program causing the error and *corefile* is the filename of the core image file generated by the system:

```
adb programfile corefile
```

To prevent **adb** from opening this file, use a hyphen (-) in place of the core filename. For example:

```
adb sample -
```

Exiting adb

To exit **adb** and return to the system shell use the **\$q** or **\$Q** command. To stop the debugger press (Ctrl)D.

NOTE **adb** cannot be stopped by pressing the Quit or Interrupt key. **adb** ignores Quit; Interrupt is caught by **adb** and causes it to wait for a new command.

Displaying instructions and data

The `adb` program provides several commands for displaying the instructions and data of a given file.

```
address [, count ] = format
address [, count ] ? format
address [, count ] / format
```

In the above command lines *address* is a value or expression giving the location of the instruction or data item. *count* is an expression giving the number of items to be displayed, and *format* is an expression defining how to display the items. The equal sign (=), question mark (?), and slash (/) tell `adb` from what source to take the item for display.

Forming addresses

. In `adb`, every address has the following form:

```
[ segment :] offset
```

In the above command line *segment* is an expression giving the address of a specific segment of 8086/286/386 memory, and *offset* is an expression giving an offset from the beginning of the specified segment to the desired item. Segments and offsets are formed by combining numbers, symbols, variables, and operators. The following are some valid addresses:

```
0:1
0x0bce:772
```

The *segment*: is optional. If not given, the most recently typed segment is used.

Forming expressions

Expressions contain decimal, octal, and hexadecimal integers, symbols, `adb` variables, register names, and a variety of arithmetic and logical operators.

Symbol names

A symbol is the name of a global variable or function defined within the program being debugged, and is equal to the address of the given variable or function. Symbols are stored in the program's symbol table, and are available if the symbol table has not been stripped from the program file.

Expressions that include references to a function, can be evaluated by specifying the function's name or its symbol table name. Symbols in the symbol table are no more than eight characters long, and those defined in C programs are prefixed with an underscore (_). The following are examples of symbols:

```
main    _main    hex2bin __out_of
```

NOTE If the spelling of any two symbols is the same, **adb** will ignore the second symbol and allow references only to the first. Also, using an underscore (_) in front of a name does not make a different symbol. For example, if both "main" and "_main" exist in a program, then **adb** accesses only the first to appear in the source and ignores the other.

By using question mark (?) command **adb** uses the symbols found in the symbol table of the program file to create symbolic addresses. Sometimes the command gives a function name when it displays data. This does not happen if the ? command for text (instructions) and the slash (/) command is use for data. Local variables cannot be addressed.

adb variables

adb automatically creates a set of its own variables whenever the debugger is initiated. These variables are set to the addresses and sizes of various parts of the program file as defined below:

Variable	Definition
b	base address of data segment
d	size of data
e	entry address of the program
m	execution type
n	number of segments
s	size of stack
t	size of text

A user can access storage locations using the **adb** defined variables. The following request prints these variables:

```
$v
```

adb reads the program file to find the values for these variables. If the file does not seem to be a program file, then **adb** leaves the values undefined.

To use the current value of an **adb** variable in an expression, preceding the variable name with a less than (<) sign. For example, the current value of the base variable *b* is:

<b

To create or change the value of an existing variable by assigning a value to a variable name, use the greater than (>) sign. The assignment has the following form:

expression > *variable-name*

In the above command line *expression* is the value to be assigned to the variable, and *variable-name* must be a single letter. For example, the following assignment gives the hexadecimal value "0x2000" to the variable *b*:

0x2000>b

To display the values of all currently defined **adb** variables, followed by their values in the current format, use the **\$v** command. The command displays any variable whose value is not zero. If a variable also has a nonzero segment value, the variable's value is displayed as an address; otherwise it is displayed as a number.

Current address

The **adb** program has two special variables that keep track of the last address to be used in a command and the last address to be typed with a command. The dot (.) variable, also called the current address, contains the last address to be used in a command. The double quotation mark (") variable contains the last address to be typed with a command. The dot and double quote variables are usually the same except when you use implied commands, such as the Newline and caret (^) characters. (These automatically increment and decrement dot, but leave " unchanged.)

Both the dot and the double quote can be used in any expression. The less than (<) sign is not required. For example, the following command displays the value of the current address:

. =

The following command displays the last address to be typed:

" =

Register names

adb allows the use of the current value of the CPU registers when evaluating expressions. To specify the value of a register, precede its name with the less than (<) sign. For a list of existing register names see eP article "**adb** register names". All 286 and 386 registers can be evaluated in expressions on XENIX 386, but only 286 registers can be evaluated in expressions on XENIX 286.

For example, the value of the 286 **ax** register can be evaluated in an expression by specifying the register as follows:

```
<ax
```

NOTE Register names cannot be used unless **adb** has been started with either a *core* file, or the program is currently being run under **adb** control.

Operators

Integers, symbols, variables, and register names can be combined with the following operators:

Unary	Meaning
~	Not
-	Negative
*	Contents of location
Binary	Meaning
+	Addition
-	Subtraction
*	Multiplication
%	Integer division
&	Bitwise And
	Bitwise inclusive Or
^	Modulo
#	Round up to the next multiple

Unary operators have higher precedence than binary operators. Expressions are evaluated from left to right. All binary operators have the same precedence. Therefore, the following expression evaluates to 10:

```
2*3+4
```

Where as the following expression evaluates to 18:

```
4+2*3
```

To can change the precedence of the operations in an expression use parentheses. For example, the following expression evaluates to 10:

```
4+(2*3)
```

NOTE The `adb` program uses 32-bit arithmetic. Values that exceed 2,147,483,647 (decimal) are displayed as negative values.

The unary `*` operator treats an expression as a pointer to an address. An expression using this operator evaluates to the value stored at the given address. For example, the following expression evaluates to the value stored at the address `"0x1234"`:

```
*0x1234
```

Whereas the following is just equal to `"0x1234"`:

```
0x1234
```

Choosing data formats

Data of different forms can be displayed by specifying a string of format commands. A format command is a letter that specifies the format in which data is displayed. One or more letter commands can be concatenated with an integer to specify the number of times the letter commands are displayed.

The following partial list illustrates each letter command and associated data format displayed. For the complete list see **adb**(CP) manual page:

Letter	Format
o	2 bytes in octal
d	2 bytes in decimal
D	4 bytes in decimal
x	2 bytes in hexadecimal
X	4 bytes in hexadecimal
u	2 bytes as an unsigned integer
f	4 bytes in floating point
F	8 bytes in floating point
c	1 byte as a character
s	A null terminated character string
i	Machine instruction
b	1 byte in octal
a	The current symbolic address
A	The current absolute address
n	A Newline
r	A blank space
t	A horizontal TAB

A letter command can be used by itself or combined with other commands to present a combination of data in different forms.

Use **D** and **X** to display long variables or 32-bit values. The **f** and **F** commands can be used to display single- and double-precision floating-point numbers. The **c** command displays **char** type variables, and the **s** command is for arrays of **char** that end with a null character (null terminated strings).

The **i** command displays machine instructions in 8086/286/386 mnemonics. The **b** command displays individual bytes and is useful for displaying data associated with instructions, or the high or low bytes of registers.

To make the display more readable, combine the **a**, **r**, and **n** commands with other commands. For example, the following format commands display the current address after each instruction:

```
ia
```

To repeat each format, precede each format with a number. For example, the following format commands display four ASCII characters:

```
4c
```

Also, combine format requests to provide elaborate displays. For instance, the following commands display four octal words followed by their ASCII interpretation from the data space of the core image file:

```
<b, -1/4o4^8Cn
```

In this example, the display starts at the address "<b," the base address of the program's data. The display continues until the end-of-the-file since the negative count "-1" causes an indefinite execution of the commands until an error condition, such as the end of the file, occurs. The command 4o displays the next four words (16-bit values) as octal numbers. The command 4^ then moves the current address back to the beginning of these four words and the C command redisplay them as eight ASCII characters. Finally, n sends a Newline character to the terminal. The C command causes values to be displayed as ASCII characters if they are in the range 32 to 126. If the value is in the range 0 to 31, it is displayed as an at sign (@) followed by a lowercase letter. For example, the value 0 is displayed as @. The at sign itself is displayed as a double at sign (@@).

Using the = command

The equal sign (=) command displays a given address in a given format. This is used primarily to display instruction and data addresses in simpler form, or to display the results of arithmetic expressions. For example, the following displays the absolute address of the symbol "main" (giving the segment and offset):

```
main=A
```

The following displays (in decimal) the sum of the variable *b* and the hexadecimal value 0x2000:

```
<b+0x2000=D
```

If a count is given, the same value is repeated that number of times. For example, the following displays the value of "main" twice:

```
main,2=x
```

If no address is given, the current address is used instead. This is the same as the following command:

```
.=
```

If no format is specified, the previous format, specified for this command, is used. For example, in the following sequence, both “main” and “_start” are displayed in hexadecimal:

```
main=x
_start=
```

Using the ? and / commands

Use ? and / commands to display the contents of a text or data segment. The commands have the following form:

```
[ address ] [ , count ] ? [ format ]
[ address ] [ , count ] / [ format ]
```

In the above command lines *address* is an address with the given segment. *count* is the number of items to display, and *format* is the format of the items displayed.

Use the ? command to display instructions in the text segment. For example, the following command displays five instructions starting at the address “main,” and the address of each instruction displays immediately before it:

```
main,5?ia
```

The following command displays the instructions, with no addresses other than the starting address:

```
main,5?i
```

Use the / command to check the values of variables in a program, especially variables for which no name exists in the program’s symbol table. For example, the following command displays the value (in hexadecimal) of a local variable:

```
<bp-4?x
```

Local variables are generally at some offset from the address indicated by the **bp** register.

An example: simple formatting

The following example illustrates how to combine formats in `?` or `/` commands, to display different types of values when stored together in the same program. This program has the following source statements:

```
char    str1[ ] = "This is a character string" ;
int     one      = 1 ;
int     number   = 456 ;
long    lnum      = 1234 ;
float    fpt      = 1.25 ;
char    str2[ ] = "This is the second character string" ;

main()
{
    one = 2;
}
```

The program is compiled and stored in a file named *sample*. To start the session, type:

```
adb sample -
```

To display the value of each individual variable by specifying its name and corresponding format in a `/` command. For example:

```
str1/s
```

Displays the contents of *str1* as a string:

```
_str1: This is a character string
```

The following command:

```
number/d
```

displays the contents of *number* as a decimal integer:

```
_number:      456.
```

You can choose to view a variable in a variety of formats. For example, you can display the long variable *lnum* as a 4-byte decimal, octal, and hexadecimal number by typing the following:

```
lnum/D
_lnum: 1234
lnum/O
_lnum: 02322
lnum/X
_lnum: 0x4d2
```

You can also examine all variables as a whole. For example, if you wish to see them all in hexadecimal, type:

```
str1,5/8x
```

This command displays eight hexadecimal values on a line, and continues for five lines.

Since the data contains a combination of numeric and string values, it is worthwhile to display each value as both a number and a character to see where the actual strings are located. You can do this with one command by typing:

```
str1,5/4x4^8Cn
```

In this case, the command displays four values in hexadecimal, then the same values as eight ASCII characters. The caret (^) is used four times, immediately before displaying the characters to set the current address back to the starting address for that line.

To make the display easier to read, insert a tab between the values and characters, and give an address for each line, for example:

```
str1,5/4x4^8t8Cna
```

Debugging program execution

The *adb* program provides a variety of commands to control the execution of programs being debugged. The following sections explain how to use these commands as well as how to display the contents of memory and registers.

Executing a program

Execute a program using the *:r* or *:R* command. The command has the following form:

```
[ address ] [count] :r [ arguments ]
[ address ] [count] :R [ arguments ]
```

In the above command lines *address* gives the address at which to start execution. *count* is the number of breakpoints to be skipped before one is taken,

and *arguments* are the command line arguments, such as filenames and options, to be passed to the program.

If no *address* is given, then the start of the program is used. Thus, to execute the program from the beginning, type:

```
:r
```

If a *count* is given, **adb** will ignore all breakpoints until the given number have been encountered. For example, the following command causes **adb** to skip the first 5 breakpoints:

```
,5:r
```

To specify arguments, separated each one by at least one space. The arguments are passed to the program in the same way the system shell passes command-line arguments to a program.

The **:R** command passes the command arguments through the shell before starting program execution. This allows the use of shell metacharacters in the arguments, to refer to multiple files or other input values. The shell expands arguments containing metacharacters before passing them on to the program.

The **:R** command is especially useful if the program expects multiple filenames. For example, the following command passes the argument "[a-z]*.s" to the shell where it is expanded to a list of the corresponding filenames before being passed to the program:

```
:R [a-z]*.s
```

The **:r** and **:R** commands remove the contents of all registers and destroy the current stack before starting the program. This kills any previous copy of the program that may have been running.

Setting breakpoints

Breakpoints are typically set to stop program execution at a specific place in the program, such as the beginning of a function, so that the contents of registers and memory can be examined. To set a breakpoint in a program, use the **:br** command. Breakpoints cause execution of the program to stop when it reaches the specified address. Control then returns to **adb**. The command has the following form:

```
address [, count ] :br command
```

In the above command line *address* must be a valid instruction address. *count* is a count of the number of times the breakpoint is to be skipped before it causes the program to stop, and *command* is the **adb** command to be executed when the breakpoint is taken.

The following command sets a breakpoint at the start of the function named “main”:

```
main:br
```

The breakpoint is taken just as control enters the function and before the function’s stack frame is created.

A breakpoint with a count is typically used within a function, that is called several times during execution of a program, or within the instructions that correspond to a **for** or **while** statement. Such a breakpoint lets the program continue to execute until the given function or instructions have been executed for the specified number of times. For example, the following command sets a breakpoint at the fifth repetition of the function “light”:

```
light,5:br
```

The breakpoint does not stop the function until it has been called at least five times.

NOTE No more than 16 breakpoints at a time are allowed.

Displaying breakpoints

To display the location and count of each currently defined breakpoint, use the **\$b** command. This command displays a list of the breakpoints given by address. If the breakpoint has a count or a command, these are given as well.

Continuing execution

To continue a program execution, after it has been stopped by a breakpoint, use the **:co** command. The command has the following form:

```
[ address ] [ ,count ] :co [ signal ]
```

In the above command line *address* is the address of the instruction at which to continue execution. *count* is the number of breakpoints to ignore, and *signal* is the number of the signal to send to the program.

If an *address* is not specified, the program starts at the next instruction after the breakpoint. If *count* is specified, **adb** ignores the first *count* breakpoints.

Stopping a program with interrupt and quit

By pressing the INTERRUPT, (Ctrl)\, or Quit keys a program’s execution can be stop at any time. These keys stop the current program and return control to **adb**. The keys are especially useful for programs that have infinite loops or other program errors.

NOTE By pressing INTERRUPT (Ctrl)\, or Quit keys, to stop a program, **adb** automatically saves the signal and passes it to the program, if the program is re-started by using the **:co** command. To continue a program's execution, without sending the signals, specify:

```
:co 0
```

0 prevents a pending signal from being sent to the program.

Single-Stepping a program

To execute one instruction at a time, use the **:s** command. The command executes one instruction and returns control to **adb**. The command has the following form:

```
[address] [, count] :s
```

In the above command line *address* must be the address of the instruction to execute, and *count* is the number of commands to execute.

If *address* is not specified, **adb** uses the current address. If *count* is specified, **adb** continues to execute each successive instruction until *count* instructions have been executed. For example, the following command executes the first 5 instructions in the function *main*:

```
main,5:s
```

Killing a program

To kill the program currently being debugging, use the **:k** command. The command kills the process created for the program and returns control to **adb**. The command is typically used to clear the current contents of the CPU registers and stack and begin the program again.

Deleting breakpoints

To delete a breakpoint from a program, use the **:dl** command. The command has the following form:

```
address :dl
```

In the above command line *address* is the address of the breakpoint to be deleted. The **:dl** command is typically used to delete breakpoints no longer required. The following deletes the breakpoint set at the start of the function "main":

```
main:dl
```

Displaying the c stack backtrace

To trace the path of all active functions, use the `$c` command. The command lists the names of all functions that have been called but have not yet returned control, as well as the address from which each function was called, and the arguments passed to it.

For example, the following command displays a backtrace of the C language functions called:

```
$c
```

By default, the `$c` command displays all calls. To display a specific amount, supply a count of the number of calls required. For example, the following command displays upto 25 calls in the current call path:

```
,25$c
```

NOTE Function calls and arguments are put on the stack after the function has been called. By setting breakpoints at the entry point to a function, the function will not appear in the list generated by the `$c` command. To remedy this, place breakpoints a few instructions into the function.

Displaying CPU registers

To display the contents of all CPU registers, use the `$r` command. The command displays the name and contents of each register in the CPU as well as the current value of the program counter, and the instruction at the current address.

Registers for UNIX 286

The `adb` program displays registers in the following format when executing in UNIX 286; the value of each register is given in the current default format:

ax	0x0	fl	0x0
bx	0x0	ip	0x0
cx	0x0	cs	0x0
dx	0x0	ds	0x0
di	0x0	ss	0x0
si	0x0	es	0x0
sp	0x0	sp	0x0
0:0:	addb	al,bl	

Registers for UNIX 386

The `adb` program displays registers in the following format when executing in UNIX 386; the value of each register is given in the current default format:

```

eax    0x81000 efl    0x246
ebx    0x0          eip    0x142
ecx    0x0          cs     0x3f
edx    0x8          ds     0x47
edi    0x0          es     0x47
esi    0x0          fs     0x47
ebp    0x0          gs     0x47
esp    0x7fef8 ss    0x47
0x3f:0x142:  push    ebp

```

Displaying external variables

To display the values of all external variables in a program, use the `$e` command. External variables are variables in a program that have global scope, or have been defined outside of any function. This may include variables defined in library routines used by your program. For example, use the `$e` command to display the following external variables and their values in hexadecimal format in a program:

```

_environ:      0xff08
_fcnt: 0x0
_gcnt: 0x0
_hcnt: 0x0
_errno: 0x0
__bufend:      0x0
__xcstar:      0x0
__xistar:      0x0
_end:          0x0
__stdbuf:      0x1b0
_iob:          0x0
_edata: 0x0
__argv:        0xff00
__acrtus:      0x0
__xcend:       0x0
__sibuf:       0x0
__lastbu:      0x130
__xiend:       0x0
__smbuf:       0x0
__sobuf:       0x0

```

A 286 example: tracing multiple functions

The following example illustrates how to execute a program under **adb** control. In particular, it shows how to set breakpoints, start the program, and examine registers and memory. The program to be examined has the following source statements:

```
int fcnt,gcnt,hcnt;
h(x,y)
{
    int hi; register int hr;
    hi = x+1;
    hr = x-y+1;
    hcnt++ ;
    hj:
    f(hr,hi);
}

g(p,q)
{
    int gi; register int gr;
    gi = q-p;
    gr = q-p+1;
    gcnt++ ;
    gj:
    h(gr,gi);
}

f(a,b)
{
    int fi; register int fr;
    fi = a+2*b;
    fr = a+b;
    fcnt++ ;
    fj:
    g(fr,fi);
}

main()
{
    f(1,1);
}
```

The program is compiled and stored in a file named *sample*. To start the session, type:

```
adb sample
```

This starts **adb** and opens the corresponding program file. There is no core image file.

The first step is to set breakpoints at the beginning of each function. Use the `:br` command to do this. For example, to set a breakpoint at the start of function `"f,"` type:

```
f:br
```

Use similar commands for the `"g"` and `"h"` functions. Once the breakpoints are created, display their locations by typing:

```
$b
```

This command lists the address, optional count, and optional command associated with each breakpoint. In this case, the command displays:

```
breakpoints
count  bkpt      command
1      _h
1      _g
1      _f
```

The next step is to display the first five instructions in the `"f"` function. Type:

```
f,5?ia
```

This command displays five instructions, each preceded by its symbolic address.

```
_f:      push      bp
_f+1.:   mov       bp,sp
_f+3.:   mov       ax,4
_f+6.:   call      near __chkstk
_f+9.:   push      di
_f+10.:  
```

Display five instructions in the `"g"` function without their addresses by typing:

```
g,5?i
```

The system displays:

```
_g:      push      bp
         mov       bp,sp
         mov       0x4
         call      near __chkstk
         push      di
```

To begin program execution, type:

```
:r
```

adb displays the following message and begins to execute:

```
sample: running
```

As soon as **adb** encounters the first breakpoint (at the beginning of the `"f"` function), it stops execution and displays the following message:

```
breakpoint  _f:      push      bp
```

Since execution to this point caused no errors, remove the first breakpoint by typing:

```
f:dl
```

To continue the program, type:

```
:co
```

The following message is displayed and the program begins execution at the next instruction:

```
sample: running
```

Execution continues until the next breakpoint, where **adb** displays the following message:

```
breakpoint      _g:      push    bp
```

To trace the path of execution, type:

```
$c
```

The display shows that only three functions are active: “f,” “main,” and “start”:

```

_f (1., 1.)                                from _main+18.
_main (1., 5922., 5926.)                  from __start+50.
__start                                     from start0+5.

```

The values 18, 5922, and 5926 will vary.

Although the breakpoint has been set at the start of function “g,” it will not be listed in the backtrace until its first few instructions have been executed. To execute these instructions, type:

```
,5:s
```

adb responds with a message notifying that the execution has stopped after **adb** single-stepped the first five instructions. To list the backtrace again. Type:

```
$c
```

This time, the list shows four active functions:

```

_g (2.,3.)                                from _f+39
_f (1.,1.)                                from _main+18
_main (1., 5922., 5926) from __start+50
__start ()                                from start0+5

```

To display the contents of the integer variable *fcnt* type:

```
fcnt/D
```

The number should be 1. To continue execution of the program and skip the first 10 breakpoints, type:

```
,10:co
```

In response, **adb** starts the program and then displays the running message again. The program does not stop until **adb** encounters exactly 10 breakpoints, when it displays the following message:

```
breakpoint      _g:      push    bp
```

To show that these breakpoints have been skipped, display the backtrace again, by typing:

```
$c
```

For UNIX 286, system displays:

```
_f (2., 11.)           from _h+36.
_h (10., 9.)           from _g+38.
_g (11., 20.)          from _f+39.
_f (2., 9.)            from _h+36.
_h (8., 7.)            from _g+38.
_g (9., 16.)           from _f+39.
_f (2., 7.)            from _h+36.
_h (6., 5.)            from _g+38.
_g (7., 12.)           from _f+39.
_f (2., 5.)            from _h+36.
_h (4., 3.)            from _g+38.
_g (5., 8.)            from _f+39.
_f (2., 3.)            from _h+36.
_h (2., 1.)            from _g+38.
_g (2., 3.)            from _f+39.
_f (1., 1.)            from _main+18.
_main(1., 5922., 5926.) from __start+50.
__start()              from start0+5.
```

Exit **adb** by typing:

```
$q
```

A 386 example: tracing multiple functions

The following example illustrates how to execute a program under **adb** control. In particular, it shows how to set breakpoints, start the program, and examine registers and memory. The program to be examined has the following source statements:

```
int fcnt,gcnt,hcnt;
h(x,y)
{
    int hi; register int hr;
    hi = x+1;
    hr = x-y+1;
    hcnt++ ;
    hj:
    f(hr,hi);
}

g(p,q)
{
    int gi; register int gr;
    gi = q-p;
    gr = q-p+1;
    gcnt++ ;
    gj:
    h(gr,gi);
}

f(a,b)
{
    int fi; register int fr;
    fi = a+2*b;
    fr = a+b;
    fcnt++ ;
    fj:
    g(fr,fi);
}

main()
{
    f(1,1);
}
```

The program is compiled and stored in a file named *sample*. To start the session, type:

```
adb sample
```

This starts **adb** and opens the corresponding program file. There is no core image file.

The first step is to set breakpoints at the beginning of each function. Use the `:br` command. For example, to set a breakpoint at the start of function “f,” type:

```
f:br
```

Use similar commands for the “g” and “h” functions. Once the breakpoints are created, display their locations by typing:

```
$b
```

This command lists the address, optional count, and optional command associated with each breakpoint. In this case, the command displays:

```
breakpoints
count  bkpt      command
1      h
1      g
1      f
```

The next step is to display the first five instructions in the “f” function. Type:

```
f,5?ia
```

This command displays five instructions, each preceded by its symbolic address:

```
f:      push  ebp
f+0x1:  mov   ebp,esp
f+0x3:  sub   esp,0x8
f+0x9:  push  ebx
f+0xa:  push  edi
f+0xb:
```

Display five instructions in the “g” function without their addresses by typing:

```
g,5?i
```

In this case, the display is:

```
g:      push  ebp
        mov   ebp,esp
        sub   esp,0x8
        push  ebx
        push  edi
```

To begin program execution, type:

```
:r
```

then `adb` displays the following message and begins to execute:

```
sample: running
```

As soon as **adb** encounters the first breakpoint (at the beginning of the “f” function), it stops execution and displays the following message:

```
breakpoint      f:      push    ebp
```

Since execution to this point caused no errors, you can remove the first breakpoint by typing:

```
f:dl
```

To continue the program, type:

```
:co
```

adb displays the following message and begins program execution at the next instruction:

```
sample: running
```

Execution continues until the next breakpoint, where **adb** displays the following message:

```
breakpoint      g:      push    ebp
```

To trace the path of execution, type:

```
$c
```

The commands show that only three functions are active: “f,” “main,” and “start”:

```
f (0x1, 0x1)                                from _main+0x15
main(0x1, 0x187ef20, 0x187ef28) from __start+0x39
```

The values 0x187ef20, 0x187ef28, and 0x39 will vary.

Although the breakpoint has been set at the start of function “g,” it will not be listed in the backtrace until its first few instructions have been executed. To execute these instructions, type:

```
,5:s
```

The **adb** program responds with a message indicating it has single-stepped the first five instructions. Now you can list the backtrace again. Type:

```
$c
```

This time, the list shows four active functions:

```
g (0x2,0x3)                                from _f+0x2c
f (0x1,0x1)                                from _main+0x15
main (0x1, 0x187ef20, 0x187ef28) from __start+0x39
```

To display the contents of the integer variable *fcnt*, type:

```
fcnt/D
```

This command displays the value of *fcnt* found in memory. The number should be 1. To continue execution of the program and skip the first 10 breakpoints, type:

```
,10:co
```

adb starts the program; then it displays the running message again. It does not stop the program until it encounters exactly ten breakpoints. It displays the following message:

```
breakpoint      g:      push      ebp
```

To show that these breakpoints have been skipped, display the backtrace again, by typing:

\$c

For UNIX 386, system displays:

```
f (0x2,0x11)          from _h+0x29
h (0x10,0xf)          from _g+0x2b
g (0x11,0x20)          from _f+0x2c
f (0x2,0xf)           from _h+0x29
h (0xe,0xd)           from _g+0x2b
g (0xf,0x1c)          from _f+0x2c
f (0x2,0xd)           from _h+0x29
h (0xc,0xb)           from _g+0x2b
g (0xd,0x18)          from _f+0x2c
f (0x2,0xb)           from _h+0x29
h (0xa,0x9)           from _g+0x2b
g (0xb,0x14)          from _f+0x2c
f (0x2,0x9)           from _h+0x29
h (0x8,0x7)           from _g+0x2b
g (0x9,0x10)          from _f+0x2c
f (0x2,0x7)           from _h+0x29
h (0x6,0x5)           from _g+0x2b
g (0x7,0xc)           from _f+0x2c
f (0x2,0x5)           from _h+0x29
h (0x4,0x3)           from _g+0x2b
g (0x5,0x8)           from _f+0x2c
f (0x2,0x3)           from _h+0x29
h (0x2,0x1)           from _g+0x2b
g (0x2,0x3)           from _f+0x2c
f (0x1,0x1)           from _main+0x15
main (0x1,0x187ef20,0x187ef28) from __start+0x39
```

Exit adb by typing:

\$q

Using the adb memory maps

adb prepares a set of maps for the text and data segments in a program and uses these maps to access items that are requested for display. The following sections describe how to view these maps, and how they are used to access the text and data segments.

Displaying the memory maps

To display the contents of the memory maps, use the **\$m** command. The command has the following form:

\$m [*segment*]

In the above command line *segment* is the number of a segment used in the program.

The command displays the maps for all segments in the program using information taken from either the program and core files or directly from memory.

Displays for UNIX 286

If **adb** is started but the program has not executed, the **\$m** command display has the following form:

Text Segments				
Seg #	File Pos	Vir Size	Phys Size	'sample' - File
63.	160.	3712.	2462.	
Data Segments				
Seg #	File Pos	Vir Size	Phys Size	'sample' - File
71.	160.	3712.	2462.	

If the program has been executed, the command displays the following form:

Text Segments				
Seg #	File Pos	Vir Size	Phys Size	'sample' - memory
63.	160.	3712.	2462.	
Data Segments				
Seg #	File Pos	Vir Size	Phys Size	'sample' - memory
71.	160.	3712.	2462.	

Displays for UNIX 386

The `$m` command has the following form:

Text Segments File - 'sample'				
Seg #	File Pos	Vir Size	Phys Size	Reloc Base
0x3f	0x400	0xb48	0xb48	0x0
Data Segments File - 'sample'				
Seg #	File Pos	Vir Size	Phys Size	Reloc Base
0x47	0x1000 0xe90	0x460	0x1880000	

Each entry provides the segment number, file position, and physical size of a segment. The segment number is the starting address of the segment. The file position is the offset from the start of the file to the contents of the segment. The physical size is the number of bytes the segment occupies in the program or core file. The filenames to the right of the display are the program and core filenames.

If the program is executed, the command displays the following form:

Text Segments File - 'sample'				
Seg #	File Pos	Vir Size	Phys Size	Reloc Base
0x3f	0x400	0xb48	0xb48	0x0
Data Segments File - 'sample'				
Seg #	File Pos	Vir Size	Phys Size	Reloc Base
0x47	0x1000 0x1880e90	0x460	0x1880000	

Virtual size is the number of bytes the segment occupies in memory. This size is sometimes different from the size of the segment in the file and will often change as the program is executed. This is due to expansion of the stack or allocation of additional memory during program execution. The filenames to the right always name program files. The file position value is ignored.

Giving segment numbers

By providing a segment number with the command, *adb* displays information only about that segment. For example, the following command displays a map for segment 63 only:

```
$m 63
```

The display has the following form for XENIX 286:

```
Segment #= 63.  
Type= Text  
File position= 160.  
Virtual Size= 3712.  
Physical Size= 2048.
```

The display has the following form for XENIX 386:

```
Segment # = 0x3f  
Type = Text  
File position = 0x3f  
Virtual size = 0x400  
Physical Size = 0x400  
Reloc Base = 0x0
```

Changing the memory map

To change the values of a memory map, use the **?m** and **/m** commands. These commands assign specified values to the corresponding map entries. The commands have the following forms:

?m *segment-number file-position size*
/m *segment-number file-position size*

In the above command lines *segment-number* gives the number of the segment map to be changed. *file-position* gives the offset in the file to the beginning of the given address, *size* is the segment size in bytes, **?m** assigns values to a text segment entry, and **/m** assigns values to a data segment entry.

For example, the following command changes the file position for segment 0x3f in the text map to 0x2000:

```
?m 0x3f 0x2000
```

The following command changes the file position for segment 0x47 in the data map to 0x0:

```
/m 0x47 0x0
```

Creating new map entries

To create new segment maps and add them to a memory map, use the **?M** and **/M** commands. Unlike **?m** and **/m**, these commands create a new map instead of changing an existing one. These commands have the following form:

?M *segment-number file-position size*
/M *segment-number file-position size*

In the above command lines *segment-number* is the number of the segment map to be created, *file-position* is the offset in the file to the beginning of the given address, and *size* is the segment size in bytes.

The **?M** command creates a text segment entry; **/M** creates a data segment entry. The segment number must be unique.

NOTE A new map entry cannot be created if that has the same number as an existing one.

The **?M** and **/M** commands are useful in accessing segments that are otherwise allocated to your program. For example, the following command creates a text segment entry for segment 0x47 whose size is 0x9c8 bytes:

```
?M 0x47 0x0 0x9c8
```

Validating addresses

adb checks each address to ensure it is valid. To validate the address, **adb** uses the segment number, file position, and size values in each map entry. If an address is correct, **adb** carries out the command; otherwise, it displays an error message.

The first step **adb** takes when validating an address is to check the segment value to make sure it belongs to the appropriate map. Segments used with the **?** command must appear in the text segments map; segments used with the **/** command must appear in the data segments map. If the value does not belong to the map, **adb** displays a bad segment error.

The next step is to check the offset to see if it is in range. The offset must be within the following range:

$$0 \leq \text{offset} \leq \text{segment-size}$$

If it is not in this range, **adb** displays a bad address error.

If **adb** is currently accessing memory, the validating segment and offset are used to access a memory location and no other processing takes place. If **adb** is accessing files, it computes an effective file address like the following, then uses this effective address to read from the corresponding file:

$$\text{effective-file-address} = \text{offset} + \text{file-position}$$

Miscellaneous features

The following sections explain a number of useful commands and features of **adb**.

Combining commands on a single line

To specify more than one command on a line, separate each commands with a semicolon (;). The commands are performed one at a time, starting at the left. Changes to the current address and format are carried to the next command. If an error occurs, the remaining commands are ignored. For example, the following command searches for and displays a string that begins with the characters *Th*:

```
?l 'Th'; ?s
```

Creating adb scripts

By redirecting **adb**'s standard input file at invocation, **adb** can be directed to read commands from a text file instead of the keyboard. To redirect the standard input, use the standard redirection symbol **<** and supply a filename. For example:

```
adb sample <script
```

The file supplied must contain valid **adb** commands. Such files are called script files, and can be used with any invocation of the debugger.

Scripts are typically used to display the contents of core files after a program error. For example, this file containing the following commands will display most relevant information about a program error:

```
120$w
4095$s
$v
=3n
$m
=3n"C Stack Backtrace"
$C
=3n"C External Variables"
$e
=3n"Registers"
$r
0$s
=3n"Data Segment"
<b, -1/8xna
```

Setting output width

To set the maximum width (in characters) of each line of output created by **adb**, use the **\$w** command. The command has the following form:

***n*\$w**

where *n* is an integer number giving the width in characters of the display. The default width, when **adb** is first invoked, is 80 characters.

The command is typically used when redirecting output to a lineprinter or special terminal. For example, the following command sets the display width to 120 characters, a common maximum width for lineprinters:

```
120$w
```

Setting the maximum offset

The **adb** program normally displays memory and file addresses as the sum of a symbol and an offset. This helps associate the instructions and data being viewed with a given function or variable. When first invoked, **adb** sets the maximum offset to 255. This means instructions or data that are no more than 255 bytes from the start of the function or variable are given symbolic addresses. Instructions or data beyond this point are given numeric addresses.

In many programs, the size of a function or variable is actually larger than 255 bytes. For this reason, **adb** lets you change the maximum offset to accommodate larger programs. To change the maximum offset, use the **\$s** command. The command has the following form where *n* is an integer giving the new offset:

n\$s

For example, the following command increases the maximum possible offset to 4095:

4095\$s

All instructions and data that are no more than 4095 bytes away are given symbolic addresses.

NOTE To disable all symbolic addressing, set the maximum offset to zero. All addresses will be given numeric values instead.

Setting default input format

To set the default format for numbers used in commands, use the **\$d** (decimal), **\$o** (octal), and **\$x** (hexadecimal) commands. The default format tells **adb** how to interpret numbers that do not begin with 0 or 0x, and how to display numbers when no specific format is given.

The commands are useful when working with a combination of decimal, octal, and hexadecimal numbers. For example, if the following combination is used, hexadecimal addresses can be specified without prepending each with 0x. Furthermore, **adb** displays all numbers in hexadecimal except those that are specifically requested to be in some other format:

\$x

adb's default format is decimal. To change or restore this at any time, use the **\$d** command.

Using UNIX commands

To execute UNIX commands without leaving **adb**, use the **adb** escape command **!**. The escape command has the following form:

!command

Where *command* is the UNIX command to be executed. When finished, the shell returns control to **adb**. For example, to display the date, type:

! date

The system displays the date at your terminal and restores control to **adb**.

Computing numbers and displaying text

To perform arithmetic calculations while in `adb`, use the `=` command. This command directs `adb` to display the value of an expression in a given format.

Use the `=` command to convert numbers in one base to another, to double-check the arithmetic performed by a program, and to display complex addresses in easier form. For example, the following command displays the hexadecimal number `"0x2a"` as the decimal number 42:

```
0x2a=d
```

However, the following command displays it as the ASCII character asterisk (*):

```
0x2a=c
```

Expressions in a command may have any combination of symbols and operators. To compute the value of external symbols, type:

```
main+5=X
```

NOTE The `=` command can also be used to display literal strings. This is useful in an `adb` script where you may wish to display comments about the script as it performs its commands. For example, the following command spaces three lines:

```
=3n"C Stack Backtrace"
```

The system then displays the following message:

```
C Stack Backtrace
```

An example: directory and inode dumps

This example illustrates how to create `adb` scripts to display the contents of a directory file and the inode map of a UNIX file system. The directory file is assumed to be named *dir*, and contains a variety of files. The UNIX file system is assumed to be associated with the device file */dev/src*, and has the necessary permissions for you to read it.

To display a directory file, create an appropriate script, then start `adb` with the name of the directory, redirecting its input to the script.

First, create a script file named *script*. A directory file normally contains one or more entries. Each entry consists of an unsigned *inumber* and a 14-character filename. To display this information, add the following command to the script file:

```
0,-1?ut14cn
```

This command displays one entry for each line, separating the number and

filename with a tab. The display continues to the end of the file. If this command is placed at the beginning of the script, **adb** displays the strings as headings for the columns of numbers:

```
= "inumber"8t"Name"
```

Once you have the script file, type:

```
adb dir - <script
```

(The dash (-) is used to prevent **adb** from attempting to open a core file.) The **adb** program reads the commands from the script and displays the following:

```
inumber name
652          .
82           ..
5971         cap.c
5323         cap
0            pp
```

To display the inode table of a file system, you must create a new script, then start **adb** with the filename of the device associated with the file system (such as the hard disk drive).

The structure of an inode table entry is defined in the file */usr/include/sys/ino.h*. Each inode entry includes:

- an unsigned short containing the mode and type of the file
- a short containing the number of links to the file
- an unsigned short containing the owner's user ID
- an unsigned short containing the owner's group ID
- a long containing the size of the file in bytes
- an array of 40 bytes containing the disk block addresses (only 39 of the 40 address bytes are in use: 13 addresses of 3 bytes each)
- a long giving the time the file was last accessed
- a long giving the time the file was last modified
- a long giving the time the file was created

The inode table starts at the address "04000" of the filesystem. This is the address of the beginning of block 2 of the file system.

Display the first entry by typing:

```
04000, -1?onororon2un40b3Y2na
```

Several Newlines are inserted within the display to make it easier to read.

To use the script on the inode table of */dev/src*, type:

```
adb /dev/src - <script
```

(Again, the dash (-) is used to prevent an unwanted core file.) Each entry in the inode table display has the following form:

```
LO.: 040755
      046    03      03
      640    0
      0121   06      0      0      0      0      0
      0      0      0      0      0      0      0
      0      0      0      0      0      0      0
      0      0      0      0      0      0      0
      0      0      0      0      0      0      0
      1986 Dec  4 13:55:49    1986 Nov 20 20:46:13    1986 Nov 20 20:46:13
```

Patching binary files

To make corrections or changes to any file, including executable binary files, use the **w** and **W** commands and invoke **adb** with the **-w** option. The following sections describe how to locate and change values in a file.

Locating values in a file

To locate specific values within a file, use the **l** and **L** commands. The commands have the following form:

```
[ address ] ?l value
[ address ] ?L value
```

In the above command lines *address* is the address at which to start the search, **l** command searches for 2 byte values, **L** command searches for 4 byte values, and *value* is the value (given as an expression) to be located.

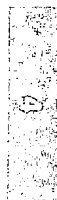
The following command starts the search at the current address, and continues until the first match or the end of the file. If the value is found, the current address is set to that value's address:

```
?l
```

Writing to a file

To write to a file, use the **w** and **W** commands. The commands have the following form:

```
[ address ] ?w value
[ address ] ?W value
```



In the above command lines *address* is the address of the value to be changed, **w** command writes 2 byte values, **W** writes 4 bytes, and *value* is the new value. For example, the following commands change the word “This” to “The”:

```
?l 'Th'  
?W 'The'
```

NOTE **W** is used to change all four characters.

Making changes to memory

Changes to memory whenever a program has been executed can also be made. If you have used an **:r** command with a breakpoint to start program execution, subsequent **w** commands cause **adb** to write to the program in memory rather than the file. This is useful if you wish to make changes to a program’s data as it runs, for example, to change the value of program flags or constants temporarily.

Index

A

Aborted program debugging, sdb, 181
 Absolute
 address, sdb, 185
 debugger. *See* adb
 memory location display, dbXtra, 57
 memory location print, dbXtra, 57
 Accessing array elements, CodeView, 100
 Active functions
 dbXtra, 57
 dbxtra, 9, 11
 adb
 ? and / command display, 204
 = command display, 203
 286 example: tracing multiple functions, 212
 386 example: tracing multiple functions, 216
 backtrace, 210
 breakpoints, 207
 combining commands on a single line, 224
 core image file, 196
 creating scripts, 224
 current address, 199
 data formats, 201
 deleting breakpoints, 209
 displaying instructions and data, 197
 exiting, 196
 forming addresses, 197
 forming expressions, 197
 introduced, 195
 killing a program, 209
 memory maps, 220
 operators, 200
 options, 195, 207, 229
 patching binaries, 229
 program execution, 206
 program preparation, 195
 register names, 200
 setting default input format, 226
 single-stepping a program, 209
 starting, 195
 stopping, 196
 symbols, 197
 using UNIX commands, 226

adb (*continued*)

 validating addresses, 223
 variables, 198
 Add Watch command, CodeView, 88, 139
 Address
 CodeView, 101-102
 arguments, CodeView, 102
 ranges, CodeView, 103
 validation, adb, 223
 Adjusting CodeView windows, 79
 alias command, dbxtra, 10
 Aliases, predefined, dbXtra, 69
 alias/unalias command, dbXtra, 62
 Animate command, CodeView, 88, 111
 Array element assessment, CodeView, 100
 Arrays, sdb, 184
 Assemble command, CodeView, 149
 Assembler source code for CodeView, 73
 Assembly
 expressions, CodeView, 99
 language program debugger. *See* dbxtra
 mode, CodeView, 86, 143
 assign command, dbXtra, 56

B

Backtrace display, adb, 210
 backward command
 dbXtra, 54
 dbxtra, 10
 backward command, dbxtra, 10
 bindkey command, dbXtra, 33
 Breakpoint
 CodeView, 82
 address, CodeView, 110
 clear command, CodeView, 135
 conditional, CodeView, 133
 dbxtra, 14
 delete command, CodeView, 135
 delete command, adb, 209
 delete command, dbXtra, 18
 delete command, sdb, 187
 disable command, CodeView, 136
 display command, CodeView, 80
 display command, adb, 208

Breakpoint (*continued*)

- display command, dbXtra, 18, 20
 - enable command, CodeView, 137
 - event-ids, dbXtra, 38
 - go command, CodeView, 109
 - list command, CodeView, 137
 - list command, sdb, 187
 - management, sdb, 187
 - sdb, 181
 - set command, CodeView, 83, 133
 - set command, adb, 207
 - set command, dbXtra, 18, 38
 - set command, dbxtra, 13
 - set command, sdb, 187
 - status, dbXtra, 18
 - temporary, dbXtra, 24
- Browse capabilities, dbXtra, 31
- BY operator, CodeView, 105
- BY PTR, CodeView, 151
- BYTE PTR, CodeView, 151

C

C

- constants, CodeView, 97
- execution capabilities, dbXtra, 31
- expressions, CodeView, 96
- source level debugger. *See* dbxtra

C++

- debugging, overloaded functions and operators, 67
 - debugging, usage requirements, 66
 - debugging, cc command, 67
 - debugging, specific features, 65
 - execution capabilities, dbXtra, 31
 - interaction mode, dbXtra, 15
 - strings, CodeView, 98
 - symbols, CodeView, 97
- C/C++ toggle, dbXtra, 67
- call command, dbXtra, 41
- Call stack, dbxtra, 11
- Call stack movement, dbXtra, 56
- Calling functions, sdb, 189
- Calls menu, CodeView, 90
- Case Sensitivity command, CodeView, 90
- catch command, dbXtra, 51
- cc command, C++ debugging, 67
- Changing
- current line in source file, sdb, 186
 - function, sdb, 186

Changing (*continued*)

- memory map, adb, 222
 - source file, sdb, 186
- Close command, CodeView, 86
- CodeView
- C expressions, 96
 - Calls menu, 90
 - Dump command, 119
 - Edit menu, 85
 - Help menu, 90
 - Misc menu, 90
 - Set Mode command, 143
 - Unassemble command, 145
 - View command, 146
 - breakpoint management, 132
 - command window, 92-93
 - commands, 75, 94
 - compile options, 72
 - cursor control, 81
 - enter commands, 152
 - executing code, 106
 - expressions, 95
 - help file, 74
 - introduced, 71
 - keyboard commands, 81
 - memory operators, 104
 - menu, 84
 - menu bar, 80
 - modifying code or data, 149
 - options, 76-77, 89, 174
 - program execution, 82
 - program file, 74
 - program preparation, 72
 - redirection command, 176
 - restrictions, 71
 - screen command, 82
 - screen example, 75
 - selecting menus with keyboard, 83
 - starting, 74
 - syntax, 75
 - windows, 78
- Command
- Motif, dbXtra, 39
 - curses, dbxtra, 39
 - dialog, CodeView, 81
 - entry, CodeView, 86
 - execution, dbXtra, 50
 - format, CodeView, 94
 - redirect, CodeView, 77
 - syntax, dbXtra, 33
 - window, CodeView, 78, 92

Command (*continued*)
 window buffer, CodeView, 93
 window, dbXtra, 17

Command line
 adb, 197, 199
 arguments, adb, 207
 arguments, sdb, 182
 options, CodeView, 75
 options, dbXtra, 15
 sdb, 182
 syntax, CodeView, 75
 syntax, dbXtra, 15

Compare Memory command, CodeView, 128

Compile options
 CodeView, 72
 dbXtra, 15
 dbxtra, 8
 sdb, 181

Compiling source program for sdb, 181

Computing numbers, adb, 227

Configurable keyboard interface, dbXtra, 30

Configurable resources, dbXtra, 28

cont command, dbXtra, 42

Context search, sdb, 185

Continuing
 after breakpoint, dbxtra, 14
 program execution, adb, 208

Copy command, CodeView, 85

Core
 dump examined, sdb, 182
 file, dbXtra, 15
 file, dbxtra, 8
 file, sdb, 182
 image file, adb, 196
 image file, sdb, 181
 location, sdb, 185

CPU register, adb, 200, 210

Creating or changing variable value,
 adb, 199

Cross reference information, dbXtra, 61

Current line change, sdb, 186

Current Location command, CodeView, 147

curses
 debugger. *See* dbxtra
 display window, dbXtra, 53
 interface introduced, dbXtra, 30

Cursor control, CodeView, 81

D

Data
 format, adb, 201
 segment display, adb, 204

dbxref command, dbXtra, 59

dbXtra
 C++. *See* C++ debugging
 C interaction mode, 15
 C++ interaction mode, 15
 Motif interface, 17
 command line options, 15
 compile options, 15
 core file, 15
 curses interface, 30
 exiting, 63
 help, 62
 help menu, 26
 key binding, 69
 object file, 15
 options, 15
 options menu, 26
 predefined aliases, 69
 program preparation, 15
 quitting, 63
 source, 65
 source file, 16
 starting, 15, 43
 stopping, 63
 using, 39
 windows, 30

dbxtra
 breakpoint, 14
 help, 10
 introduced, 7
 program preparation, 8, 15
 screen, 9
 setting breakpoint, 13
 source code, 10
 starting, 8, 15
 stopping, 14
 tutorial, 8
 variables, 10

Debugger
 CodeView, 71, 77
 Motif version. *See* dbXtra
 absolute. *See* adb
 curses version. *See* dbxtra
 dbxtra, 7
 symbolic. *See* sdb

Debugging

- C++. *See* C++ debugging
- described, 1
- machine language, sdb, 190
- separate object modules, CodeView, 72
- Definitions, dbXtra, 16
- Delay command, CodeView, 179
- Delete breakpoint
 - adb, 209
 - dbXtra, 18, 45
 - interactively, sdb, 188
 - sdb, 187-188, 190
- Delete Watch command, CodeView, 88
- Denormalized-operand mask, CodeView, 132
- Dialog window, CodeView, 80
- Directories, sdb, 182
- disable command, dbXtra, 45
- Disassembly, CodeView, 145
- Display
 - UNIX 286, adb, 220
 - UNIX 386, adb, 221
 - c stack backtrace, adb, 210
 - command =, adb, 203
 - command /, adb, 204
 - command ?, adb, 204
 - command, dbXtra, 32, 53
 - coprocessor registers command, CodeView, 86
 - CPU registers, adb, 210
 - expression command, CodeView, 86, 113
 - flags command, CodeView, 86
 - instructions and data, sdb, 197
 - machine language statements, sdb, 190
 - memory command, CodeView, 86
 - memory maps, adb, 220
 - parameters command, CodeView, 86
 - register command, CodeView, 86
 - source command, CodeView, 86
 - source command, sdb, 186
 - text, adb, 227
 - variables command, CodeView, 86
 - window, CodeView, 79
- Double word operator, CodeView, 105
- Dump command
 - CodeView, 93, 119-125
 - dbXtra, 56

E

Edit

- breakpoint, CodeView, 88
- command, dbXtra, 60, 69
- commands, dbXtra, 33-34
- menu, CodeView, 85
- Emulator Scroll command, CodeView, 89
- enable command, dbXtra, 45
- Enter commands, CodeView, 152
- Environment variables, dbXtra, 52
- Error
 - logic and syntax, CodeView, 73
 - message, CodeView, 74, 91, 101, 153, 165, 171
 - message, adb, 196, 223
 - message, dbxtra, 45, 47, 50, 55-56
 - message, sdb, 181-182
 - reporting in programs tracked with sdb, 181-182
 - tracked with CodeView, 71
- Event status, dbXtra, 45
- event-id, dbxtra, 12
- Events menu, dbXtra, 20
- Examine
 - code, CodeView, 143
 - data, CodeView, 113
 - expressions, CodeView, 113
 - symbols, CodeView, 116
- Executable file
 - CodeView, 71-72, 75, 77, 113-114
 - adb, 195-196
 - dbxtra, 8, 59
 - sdb, 182
- Execute command
 - CodeView, 111
 - adb, 207, 209, 226
 - dbXtra, 16, 39
 - dbxtra, 39
 - sdb, 188-189, 191
- Execute menu, dbXtra, 23
- Execute to cursor command, CodeView, 90
- Executing
 - code, CodeView, 106
 - program, adb, 206
 - program, dbXtra, 41
- Execution environment management, dbXtra, 51
- Execution time events, dbXtra, 45
- Exit command
 - CodeView, 83, 85, 112, 172-173

Exit command (*continued*)

adb, 196
 dbXtra, 19, 63
 dbxtra, 14
 sdb, 191

Expressions

C, 96
 CodeView, 80, 86-87, 94-95, 98-99, 104, 106, 113, 170
 adb, 197, 200
 dbXtra, 20, 32, 39-40, 53-54, 56, 58, 66
 dbxtra, 24
 sdb, 186

External variable display, adb, 211

F

FAR, CodeView, 150
 Fatal system error examined, adb, 196
 File
 menu, dbXtra, 19
 preparation for dbxtra, 8
 file command, dbXtra, 60
 Files required for CodeView, 74
 Fill Memory command, CodeView, 163
 Find command, CodeView, 87
 Format
 C language, 98
 CodeView, 74-75, 89, 94, 97-98, 104, 114
 adb, 201-202, 204-205, 226
 dbXtra, 27, 41, 51, 64
 specifier, CodeView, 114
 specifier, sdb, 184
 forward command
 dbXtra, 54
 dbxtra, 10
 fregs/regs command, dbXtra, 56
 Fullscreen interface, dbXtra, 30
 func command, dbXtra, 60
 Function call
 CodeView, 82
 dbXtra, 11, 44
 sdb, 183, 187, 189

G

-g compile option
 CodeView, 72
 dbxtra, 8
 sdb, 181

Go command, CodeView, 109
 goto command, dbXtra, 42

H

Hardware register display, dbXtra, 56
 Help
 command, CodeView, 82, 86, 91, 167
 command, dbXtra, 62
 command, dbxtra, 65
 dbxtra, 10
 file, CodeView, 74
 menu, CodeView, 90-91
 menu, dbXtra, 26
 window, dbxtra, 29

I

Identifier
 qualification, dbXtra, 57
 resolution, dbXtra, 39
 ignore command, dbXtra, 52
 Ignore Signals command, CodeView, 89
 Indirect register instructions, CodeView, 151
 Info menu, dbXtra, 25
 Input format, adb, 226
 Interaction capabilities, dbXtra, 32
 Interactive
 debugging, sdb, 181
 display, sdb, 181
 Interface commands, dbXtra, 53
 Interrupt loops, CodeView, 83
 Interrupt-enable mask, CodeView, 132
 Interrupting program execution, adb, 208
 Invalid-operator mask, CodeView, 132

J

jump command, dbXtra, 42

K

Key binding, dbXtra, 69
 Keyboard
 command, CodeView, 92
 commands, CodeView, 81
 interface, dbXtra, 30, 33
 translation, dbXtra, 54

Keyname/sequence editing command,
dbXtra, 69
keys command, dbXtra, 54

L

Label/Function command, CodeView, 88
Language command, CodeView, 89
Length specifier, sdb, 184
Line mode
 dbXtra, 16, 55
Line number
 CodeView, 101
 dbXtra, 15
 dbxtra, 8
 sdb, 181, 187
Line-by-line debugging, sdb, 181
list command, dbXtra, 60
Local
 command, CodeView, 86
 variables, CodeView, 73, 95
 window, CodeView, 78
Locating values in a file, adb, 229
Location display, sdb, 185

M

Machine
 language, sdb, 190
 level debugger. *See* dbxtra
 level instruction, sdb, 187
Main window, dbXtra, 17
Managing breakpoints, CodeView, 132
Manipulating registers, sdb, 190
masm assembly for CodeView, 73
Maximize, CodeView windows, 79
Maximize command, CodeView, 86
Maximum offset, adb, 225
Memory
 Window, CodeView, 89
 command, CodeView, 86
 location, dbXtra, 56-57
 map, adb, 220, 222
 mode command, CodeView, 90
 operators, CodeView, 104
 window, CodeView, 78
Menu
 CodeView, 84
 bar, CodeView, 80
 dbXtra, 19

Menu (*continued*)

 selection from keyboard, CodeView, 83
Message box, CodeView, 80
Misc menu, CodeView, 90
Modifying code or data, CodeView, 149
more command, dbXtra, 54
Motif interface, dbXtra, 17
Motif version debugger. *See* dbXtra
MOVSB, CodeView, 150

N

NEAR, CodeView, 150
next command, dbXtra, 42
Next Window command, CodeView, 90
nexti command, dbXtra, 43
Non-ASCII data file examined, adb, 195

O

Object file
 CodeView, 71-73
 dbXtra, 15
 dbxtra, 8
objfile, dbXtra, 16
Open Module menu command,
 CodeView, 85
Open Source menu command, CodeView, 84
Operators
 adb, 200
 in expressions, dbXtra, 39
Optimization, CodeView, 73
Options
 CodeView, 76-77
 adb, 195, 207, 229
 dbXtra, 15
 menu, CodeView, 89
 menu, dbXtra, 26
 sdb, 181-182, 187
Output
 capabilities, dbXtra, 32
 command, CodeView, 86
 screen, CodeView, 170
 stopping, dbxtra, 10
 width, adb, 225
Overflow mask, CodeView, 132
Overloaded functions and operators, C++
 debugging, 67

P

Pagination, dbXtra, 54
 Paging, dbxtra, 10
 Paste command, CodeView, 85
 Patching binaries, adb, 229
 Path, dbXtra, 16
 Pattern matching, sdb, 183
 Pause command, CodeView, 180
 Pointers, sdb, 184
 Pop up menu, dbXtra, 27
 Precision mask, CodeView, 132
 Predefined aliases, dbXtra, 69
 Prev window command, CodeView, 90
 Print
 command, CodeView, 85
 command, adb, 225
 command, dbXtra, 56
 command, dbxtra, 10
 memory location content, dbXtra, 57
 stack trace, sdb, 183
 variables, dbxtra, 10
 Program
 debugged with sdb, 181
 errors tracked with CodeView, 71
 execution, CodeView, 82
 execution, adb, 206
 execution, dbXtra, 8
 execution, dbxtra, 8
 execution, sdb, 183
 file, CodeView, 74
 preparation for CodeView, 72
 preparation for adb, 195
 preparation for dbXtra, 15
 preparation for dbxtra, 8, 15
 preparation for sdb, 181
 Program Step command, CodeView, 83, 86, 108
 PTR operator, CodeView, 100
 Pull-down menus, dbXtra, 19

Q

Qualified name, dbxtra, 12
 Quick Watch command, CodeView, 88
 Quitting
 CodeView, 167
 adb, 196
 dbXtra, 63
 dbxtra, 14
 sdb, 191

R

Radix command, CodeView, 168
 Redirection command, CodeView, 77, 176
 Register
 CodeView, 101-102
 command, CodeView, 86, 130, 164
 debugging, dbXtra, 56
 display UNIX 286, adb, 210
 display UNIX 386, adb, 210
 indirection, CodeView, 100
 manipulation, sdb, 190
 names, adb, 200
 value change, sdb, 191
 window, CodeView, 78, 80
 Regular expression
 CodeView, 170-171
 sdb, 186
 search, CodeView, 87, 170
 search, dbXtra, 61
 Repeat Last Find command, CodeView, 88
 rerun command, dbXtra, 43
 Restart command, CodeView, 88, 112
 Restrictions, CodeView, 71
 RETF, CodeView, 150
 return command, dbXtra, 43
 Routine display, CodeView, 90
 Routines, CodeView, 148
 run command, dbXtra, 43

S

Screen
 command, CodeView, 82
 corr.mand, dbXtra, 55
 dbxtra, 9
 example, CodeView, 75
 interface, dbXtra, 30
 manager, dbXtra, 16
 mode, dbXtra, 55
 movement commands, CodeView, 81
 scroll command, dbxtra, 10
 view, CodeView, 82
 Screen Swap command, CodeView, 89
 Script creation, adb, 224
 Scroll command, dbxtra, 10
 Scrolling windows, dbXtra, 54
 sdb
 breakpoint management, 187
 calling functions, 189
 changing source file, 186

sdb (*continued*)

- command line arguments, 182
- displaying variables, 183
- exiting, 191
- format specifier, 184
- introduced, 181
- length specifier, 184
- machine language debugging, 190
- options, 181-182, 187
- program preparation, 181
- running a program, 188
- session example, 191
- shell command, 191
- single stepping, 187
- source file display, 185-186
- stack trace command, 183
- startup routine, 183

Search

- command, CodeView, 88, 129, 170-171
- command, adb, 224, 229
- command, dbXtra, 61
- path, dbXtra, 16
- sdb, 185

Segment numbers, adb, 221

Set breakpoint

- CodeView, 88, 109
- adb, 207
- dbXtra, 18
- dbxtra, 13
- sdb, 187

Set Mode command, CodeView, 143

setenv command, dbXtra, 52

set/unset command, dbXtra, 63

sh and ! commands, dbXtra, 65

Shell

- commands, sdb, 181, 191
- escape command, CodeView, 172
- execution, dbXtra, 65

Sidebar command buttons, dbXtra, 18

Signals, dbXtra, 51

Single-stepping

- adb, 209
- sdb, 187

Size command, CodeView, 86

skip command, dbXtra, 43

Source

- code, dbXtra, 15
- code, dbxtra, 10
- command, CodeView, 86
- command, dbXtra, 65
- file, CodeView, 101

Source (*continued*)

- file access, dbXtra, 59
- file manipulation, sdb, 185-0186
- files, dbXtra, 16
- level debugger. *See* dbxtra
- list, sdb, 182
- menu, dbxtra, 24
- mode, CodeView, 86, 143
- programs debugged with sdb, 181
- window, CodeView, 78
- window, dbXtra, 17

Source Window command, CodeView, 89

Specified name, dbXtra, 57

Specifying register value, adb, 200

Stack trace

- CodeView, 90, 147
- adb, 210
- dbxtra, 11
- sdb, 183

Starting

- CodeView, 74
- adb, 195
- dbXtra, 8, 15, 43
- dbxtra, 8, 15
- sdb, 183

status command, dbXtra, 45

Status register, 132

step command, dbXtra, 44

stepi command, dbXtra, 44

stop command, dbXtra, 46

Stopping

- CodeView, 111, 132
- adb, 207-209
- dbXtra, 63
- dbxtra, 11, 13-14
- output, dbxtra, 10
- sdb, 191

Structures, sdb, 184

Symbol

- C, 97
- examining, CodeView, 116
- names, CodeView, 74
- names, adb, 197
- qualification, dbXtra, 57
- sdb, 183
- table, adb, 197
- table, dbXtra, 15

Symbolic debugger. *See* sdb

Syntax

- CodeView, 75, 167
- dbXtra, 15

Syntax (*continued*)

dbxtra, 15
 error, CodeView, 73
 System-control commands, CodeView, 167

T

Tag register, 132

Text

display, adb, 227
 segment display, adb, 204

Trace command

CodeView, 82, 86, 107
 adb, 210
 dbXtra, 48
 dbxtra, 11
 sdb, 181, 183

Trace Speed command, CodeView, 89

Tutorial, dbxtra, 8

Types and casting, dbXtra, 40

U

Unassemble command, CodeView, 145

Underflow mask, CodeView, 132

undisplay command, dbXtra, 55

Undo command, CodeView, 85

UNIX 286

display, adb, 220
 register display, adb, 210

UNIX 386

display, adb, 221
 register display, adb, 210

UNIX commands, adb, 226

UNIX Shell menu command, CodeView, 85

unsetenv command, dbXtra, 52

up/down command, dbXtra, 56

Usage requirements, C++ debugging, 66

User interface commands, dbXtra, 53

V

Validating addresses, adb, 223

Value set/unset, dbXtra, 63

Variable

CodeView, 95
 adb, 198
 address, CodeView, 100
 assign, dbXtra, 56

Variable (*continued*)

contents, dbxtra, 11
 current address, adb, 199
 current value, adb, 199
 dbXtra, 15
 last address, adb, 199
 local, CodeView, 73, 139
 names, dbxtra, 8
 printing, dbxtra, 10
 sdb, 183

value change, sdb, 191
 value display, adb, 199

version command, dbXtra, 65

View command, CodeView, 146

W

Warning message, dbXtra, 15

Watch command

CodeView, 86, 138
 menu selections, CodeView, 88
 set, CodeView, 139

Watch Delete command, CodeView, 88, 140

Watch menu, CodeView, 88

Watch Set Breakpoint command,
 CodeView, 88

Watch window, CodeView, 78, 80

whatis command, dbXtra, 57

when command, dbXtra, 50

where command, dbXtra, 57

where command, dbxtra, 11

whereis command, dbXtra, 57

which command, dbXtra, 57

Window

CodeView, 78-79
 Maximize, 79
 command, dbXtra, 17
 commands, CodeView, 81, 92
 dbXtra, 30
 described, CodeView, 80
 scrolling, dbXtra, 54
 sdb, 186, 191
 source, dbXtra, 17

WO operator, CodeView, 105

WO PTR, CodeView, 151

WORD PTR, CodeView, 151

X

xref command, dbXtra, 61

Z

Zero-divide mask, CodeView, 132

0-9

286 memory address, adb, 197

286 register, adb, 200

386 memory address, adb, 197

386 register, adb, 200

8086 memory address, adb, 197

8087 command, CodeView, 86, 131

8087 coprocessor, CodeView, 131

80286 coprocessor, sdb, 190

80287 coprocessor, CodeView, 131

80386 coprocessor, sdb, 190

80387 coprocessor, CodeView, 131

80486 coprocessor, CodeView, 131

Please help us to write computer manuals that meet your needs by completing this form. Please post the completed form to the Technical Publications Research Coordinator nearest you: The Santa Cruz Operation, Ltd., Croxley Centre, Hatters Lane, Watford WD1 8YN, United Kingdom; The Santa Cruz Operation, Inc., 400 Encinal Street, P.O. Box 1900, Santa Cruz, California 95061, USA or SCO Canada, Inc., 130 Bloor Street West, 10th Floor, Toronto, Ontario, Canada M5S 1N5.

Volume title: _____
(Copy this from the title page of the manual, for example, SCO UNIX Operating System User's Guide)

Product: _____
(for example, SCO UNIX System V Release 3.2 Operating System Version 4.0)

How long have you used this product?

- ☐ Less than one month ☐ Less than six months ☐ Less than one year
☐ 1 to 2 years ☐ More than 2 years

How much have you read of this manual?

- ☐ Entire manual ☐ Specific chapters ☐ Used only for reference

	Agree			Disagree		
The software was fully and accurately described	☐	☐	☐	☐	☐	☐
The manual was well organized	☐	☐	☐	☐	☐	☐
The writing was at an appropriate technical level (neither too complicated nor too simple)	☐	☐	☐	☐	☐	☐
It was easy to find the information I was looking for	☐	☐	☐	☐	☐	☐
Examples were clear and easy to follow	☐	☐	☐	☐	☐	☐
Illustrations added to my understanding of the software	☐	☐	☐	☐	☐	☐
I liked the page design of the manual	☐	☐	☐	☐	☐	☐

If you have specific comments or if you have found specific inaccuracies, please report these on the back of this form or on a separate sheet of paper. In the case of inaccuracies, please list the relevant page number.

May we contact you further about how to improve SCO UNIX documentation? If so, please supply the following details:

Name _____ Position _____

Company _____

Address _____

City & Post/Zip Code _____

Country _____

Telephone _____ Facsimile _____



10 December 1991



BH01 405P000

56756

CLUSTER



BJ01401P000

MANUAL



AU01409P000