User's Guide

HP B1467
68030/040
Debugger/Simulator

Notice

About this edition

Many product updates and fixes do not require manual changes, and manual corrections may be done without accompanying product changes. Therefore, do not expect a one-to-one correspondence between product updates and manual revisions.

Edition dates and the corresponding HP manual part numbers are as follows:

Edition 1   B1467-97003, January 1993

This manual incorporates information which previously appeared in B1467-92000, B1467-92001, B1467-97000, B1467-97001, B1467-97002, 64368-92003, 64368-97002, 64368-97003, and 64368-97004.

Certification and Warranty

Certification and warranty information can be found at the end of this manual on the pages before the back cover.

Debugging C Programs for 68030/040 Microprocessors

The HP B1467 68030/040 Debugger/Simulator is a debugging tool for 68030/040 microprocessor code. The debugger loads and executes C programs or assembly language programs using a simulator on your host system.

With the Debugger, You Can ...

- Browse and edit C and C++ source files.
- View C and C++ functions on the stack.
- Monitor variables as the program executes.
- View assembly language code with source lines.
- View registers and stack contents.
- Step through programs by C or C++ source lines or by assembly language instructions.
- Stop programs upon the execution of selected instructions or upon a read or write of selected memory locations.
- Create conditional breakpoints using macros.
- Patch C or C++ code without recompiling.
- Simulate input and output devices using your computer's keyboard, display, and file system.
- Tune code by using the clock cycle count pseudo register to time code modules during simulation.
- Save and execute command files.
- Log debugger commands and output.
- Examine the inheritance relationships of C++ classes.

With the Graphical Interface You Can ...

- Use the debugger under an X Window System that supports OSF/Motif interfaces.
- Enter debugger commands using pull-down or pop-up menus.
- Set source-level breakpoints using the mouse.
- Create custom action keys for commonly used debugger commands or command files.

- View source code, monitored data, registers, stack contents, and backtrace information in separate windows on the debugger's main display.
- Access on-line help information.
- Quickly enter commands using the guided syntax of the standard interface.

With the Standard Interface You Can ...

- Use the debugger with a terminal or terminal emulator.
- Quickly enter commands using guided syntax, command recall, and command editing.
- View source code, monitored data, registers, stack contents, and backtrace information in separate windows on the debugger's main display.
- Define your own screens and windows in the debugger's main display.
- Access on-line help information.

Compatibility with Other Products

The debugger/simulator has been designed to work with HP-UX version 7.0 or greater or SunOS version 4.1 or greater and the following Hewlett-Packard language products:

HP B1478 (HP 64907) 68030 C Cross Compiler, Version 3.50

HP B1463 (HP 64909) 68040 C Cross Compiler, Version 3.51

HP B1465 (HP 64874), 68030/40 Assembler, Linker, Librarian, Version 1.60

HP B1471, HP 64000-UX Operating Environment Software, Version 3.50

See the "Loading and Executing Programs" chapter if you are using the Microtec® language tools.

In This Book

This book describes the HP B1467 68030/040 Debugger/Simulator. It is organized into five parts:

Part 1. Quick Start Guide

An overview of the debugger and a short lesson to get you started.

Part 2. User's Guide

How to use the debugger to solve your problems.

Part 3. Concept Guide

Conceptual information on CPU simulation and on X resources.

Part 4. Reference

Descriptions of what each debugger command does, details of how the debugger works, and a list of error messages.

Part 5. Installation

How to install the debugger software on your computer.

Contents

Contents

2   Getting Started with the Standard Interface

The Standard Interface At a Glance   36

The Quick Start Tutorial   38

Contents

4   Loading and Executing Programs

Contents

Part 3 Concept Guide

9   X Resources and the Graphical Interface

Part 4 Reference

10 Debugger Commands

How Pulldown Menus Map to the Command Line   278

How Popup Menus Map to the Command Line   281

Command Summary   283

Contents

11 Expressions and Symbols in Debugger Commands

Expression Elements   431

Symbols   438

Contents

14 Debugger Error Messages

15 Debugger Versions

Version C.05.10   518

Contents

Part 1

Quick Start Guide

Part 1

1

Getting Started with the Graphical Interface

How to get started using the debugger's graphical interface.

When an X Window System that supports OSF/Motif interfaces is running on the host computer, the debugger has a *graphical interface* that provides features such as pull-down and pop-up menus, point and click setting of breakpoints, cut and paste, on-line help, customizable action keys and pop-up recall buffers.

The debugger also has a *standard interface* for several types of terminals, terminal emulators, and bitmapped displays. When using the standard interface, commands are entered from the keyboard. If you are using the debugger's standard interface, please skip to Chapter 2, "Getting Started with the Standard Interface".

Some advanced commands are not well-suited to menus. Those commands are entered through the *command line*. The command line allows you to enter standard interface commands in the graphical interface.

The Graphical Interface at a Glance

Pointer and cursor shapes

Arrow

The arrow mouse pointer shows where the mouse is pointing.

Hand

The hand mouse pointer indicates that a pop-up menu is available by pressing the right mouse button.

Hourglass

The hourglass mouse pointer means "wait." If the debugger is busy executing a program, you may stop it by pressing **< Ctrl> -C**.

Text

The "I-beam" keyboard cursor shows where text entered with the keyboard will appear in the entry buffer or in a dialog box.

Command-line

The "box" keyboard cursor on the command line shows where commands entered with the keyboard will appear.

The Debugger Window

Menu bar

Action keys

Entry buffer

Scroll bar

Display area

Status line

Command line

File	Display	Modify	Execution	Breakpoints	Window	Settings		Help

Action keys:	< Demo >	Disp Src ( )	C Expr ( )	Run	Run til ( )	Step Over
< Your Key >	Make	Disp Src PC	Monitor ( )	Step	Run Xfer	Step Out

( ): `main`   Recall

```
                         ─Monitor─────────────3─   ─Backtrace────────────4─
1   num_checks     0                                0. 00000400:crt0\entry          ▲
2   target_temp    0
3   current_temp   0
4   old_data      [00]:temp        0
5                      humid        0                                               ▮
6                      ave_temp     0.000000E+00
7                      ave_humid    0.000000E+00
8                 [01]:temp        0
```

```
                                 Code                                    2
    1   /**********************************************************************
    2   A Hewlett-Packard Software Product
    3   Copyright Hewlett-Packard Co. 1992
    4
    5   All Rights Reserved. Reproduction, adaptation, or translation without pri
    6   written  permission  is prohibited, except as allowed under copyright law
    7   **********************************************************************
    8   #include <stdio.h>
    9   #include <string.h>
   10   #include "update_sys.h"
   11   #include "proc_spec.h"
   12   /**********************************************************************
   13   * This typedef is also found in demo.h but since demo.h is not included in
   14   * this file, this declaration appears here by itself.
   15   **********************************************************************
   16   #define SHRINKFACTOR 1.3
   17   #define LISTLEN      NUM_OF_OLD*4+1
```

```
                                ─Journal──────────────────────1─
     Note: in startup routine. Press F8 to go to main.
>  File Command  cmdfiles/debug/Cmd_dbmac.com
```

```
STATUS: Command        68040    MODULE: crt0          BREAK #: 0    HELP=F5    ◄ ▶
```

```
> File Command cmdfiles/debug/Cmd_dbmac.com
```

Breakpt	Debugger	Expression	██	Memory	Program	Symbol	Window

Command   Error_Command   User_Fopen   Journal   Log   Window_Close   Startup

Command:	Return	Recall	Cursor:	Backup	Forward	Clear to end	Clear	Help

Menu Bar. Provides pull-down menus from which you select commands. When menu items are not applicable, they appear half-bright and do not respond to mouse clicks.

Action Keys. User-defined pushbuttons. You can label these pushbuttons and define the action to be performed. Action key labels and functions are defined by setting X resources (see the "Configuring the Debugger" chapter).

Entry Buffer. Wherever you see "()" in a pull-down menu, the contents of the entry buffer are used in that command. You can type values into the entry buffer, or you can cut and paste values into the entry buffer from the display area or from the command line entry area. You can also set up action keys to use the contents of the entry buffer.

Display Area. This area of the screen is divided into windows which display information such as high-level code, simulated input and output, and breakpoints. To activate a window, click on its border.

In this manual, the word "window" usually refers to a window inside the debugger display area.

Scroll Bar. Allows you to page or scroll up or down the information in the active window.

Status Line. Displays the debugger status, the CPU type, the current program module, and the number of the last breakpoint.

Command Line. The command line area is similar to the command line in the standard interface; however, the graphical interface lets you use the mouse to enter and edit commands. You can turn off the command line if you only need to use the pull-down menus.

Graphical Interface Conventions

This manual uses a shorthand notation for indicating that you should choose a particular menu item. For example, the following instruction

Choose **File→Load→Executable...**

means to select the **File** menu, then select **Load** from the File menu, then select the **Executable...** item from the Load menu.

```
┌─────┐
│File │
├─────────────┐
│Context    ▷ │
├─────────────┤──────────────────┐
│Load       ▷ │Executable ...     │
├─────────────┤──────────────────┤
│Store      ▷ │Program Only ...   │
├─────────────┤──────────────────┤
│Copy Window▷ │Symbols Only ...   │
├─────────────┤──────────────────┤
│Log        ▷ │User-Defined Macros ...│
├─────────────┤──────────────────┘
│Edit       ▷ │
├─────────────┤
│Term ...     │
├─────────────┤
│Exit       ▷ │
└─────────────┘
```
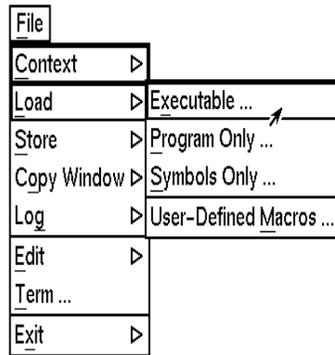
Refer to the "Entering Debugger Commands" for specific information about choosing menu items.

In this manual, the word "window" usually means a window inside the debugger display area, rather than an X window.

Mouse Buttons

Mouse Button Descriptions

Button Name	General Function
left	Selects pushbuttons. Pastes from the display area to the entry buffer.
middle	Pastes from the entry buffer to the command line text area. If you have a two-button mouse, press both buttons together to get the "middle button."
right	Click selects first item in pop-up menus. Click on window border activates windows. Press and hold displays menus.
command select	Displays pull-down menus. May be the left button or right button, depending on the kind of computer you have. *See* "Platform Differences" on page 10.

Platform Differences

A few mouse buttons and keyboard keys work differently between platforms. This manual refers to those mouse button and keyboard bindings in a general way. Refer to the following tables to find out the button names for the computer you are using to run the debugger.

Mouse Button Bindings

Generic Button Name	HP 9000	Sun SPARCsystem
command select	left	right

Keyboard Key Bindings

Generic Key Name	HP 9000	Sun SPARCsystem
menu select	extend char	extend char (diamond)
left-arrow	left arrow	left arrow[1]
right-arrow	right arrow	right arrow[1]

[1]These keys do not work while the cursor is in the main display area.

The Quick Start Tutorial

This tutorial gives you step-by-step instructions on how to perform a few basic tasks using the debugger.

Perform the tasks in the sequence given; otherwise, your results may not be the same as those shown here.

Some values displayed on your screen may vary from the values shown here.

The Demonstration Program

The demonstration program used in this chapter is a simple environmental control system (ECS). The system controls the temperature and humidity of a room requiring accurate environmental control. The program continuously looks at flags which tell it what action to take next.

Note

Some commands are printed on two lines in this chapter. When entering these commands, type the entire command on one line.

Note

The 68030/040 debugger/simulator can be started using either the command *db68030* or the command *db68040*. The demonstration files used in the examples in this chapter can be copied from directory:

`/usr/hp64000/demo/debug_env/sim68040`

To prepare to run the debugger

1 Check that the debugger has been installed on your computer. Installation is described in the "Installation" chapter.

2 Find out where the debugger software is installed. If it is not installed under "/usr/hp64000" then use "$HP64000" wherever "/usr/hp64000" is printed in this chapter.

3 Check that "/usr/hp64000/bin" and "." are in your $PATH environment variable. (Type "echo $PATH" to see the value of $PATH.)

4 If the debugger software is installed on a different kind of computer than the computer you are using, edit the "platformScheme" in the Xdefaults.sim file. This file is located in directory /usr/hp64000/demo/debug_env/sim68040. For example, if you are sitting at a Sun workstation which is networked to an HP 9000 Series 300 workstation, change the platformScheme to "SunOS".

To start the debugger

1 Change to the debugger demo directory:

```
cd /usr/hp64000/demo/debug_env/sim68040
```

2 Start the debugger by entering:

```
Startdebug
```

This will set some environment variables, start the debugger and load a program for you to look at.

The Startdebug script will ask you whether it should copy the demo files to another directory. Answer "y". (You cannot modify the files in /usr/hp64000/demo.)

Note   If you were debugging your own program, you would need to enter a command like:

```
db68040 -c mycmd ecs
```

This command starts the debugger, which executes the command file *mycmd.com* and loads the absolute file *ecs.x*. See the "Loading and Executing Programs" chapter for more details.

To activate display area windows

Notice there are several windows in the main display area of the debugger. The different windows contain different types of information from the debugger. The active window has the thicker border.

1 Use the right mouse button to click on the border of the Monitor window.

Be sure to click only once (do not "double-click"). The Monitor window should now have a thick border. Now activate the Code window:

2 Use the right mouse button to click on the border of the Code window.

If you click on the border of the active window, it will be expanded. Just click again to show the window in its normal size.

See the "Debugging Programs" chapter for a list of other ways to activate a window.

To run until main()

1 Click on the **Run til** () action key. | Run Til ( ) |

The Code window now shows the *main ()* routine.

Clicking on the **Run til** () action key runs the program until the line indicated by the contents of the *entry buffer*.

Locate the (): symbol. The area to the right of this symbol is the entry buffer. When you started the demonstration program, the debugger loaded the entry buffer with the value "main".

```
 File  Display  Modify  Execution  Breakpoints  Window  Settings                    Help

 Action keys:  [ < Demo > ]  [ Disp Src ( ) ]  [ C Expr ( ) ]  [    Run    ]  [ Run til ( ) ]  [ Step Over ]
 [< Your Key >]  [   Make   ]  [ Disp Src PC ]  [ Monitor ( ) ]  [   Step   ]  [  Run Xfer  ]  [ Step Out  ]

 ( ): main                                                                         [ Recall ]
 ┌──────────────Monitor──────────────3─┐ ┌───────────Backtrace──────────4─┐
 1   num_checks    0                     2. 0000043C?crt0\<unknown>          ▲
 2   target_temp   0                     1. 00000716 startup\_startup
 3   current_temp  0                     0. 00000D54*main\main               ▓
 4   old_data      [00]:temp       0
 5                      humid       0
 6                      ave_temp    0.000000E+00
 7                      ave_humid   0.000000E+00
 8                 [01]:temp        0
 └──────────────────────────────────┘ └────────────────────────────────┘
 ┌──────────────────────────Code──────────────────────────2─┐
 │  93   extern void do_sort();          /* sets up ascii array and calls combs
 │  94
 │  95   main()
 │▐96   {                                                     ▐
 │  97       init_system();
 │  98       proc_spec_init();
 │  99
 │ 100       while (true)                                     ▓
 │ 101       {
 │ 102           update_system();
 │ 103           num_checks++;
 │ 104           interrupt_sim(&num_checks);
 │ 105           if (graph)
 │ 106               graph_data();
 │ 107           proc_specific();
 │ 108       }                                                ▼
 │ 109   }
 └──────────────────────────────────────────────────────────┘
 ┌───────────────────────────Journal───────────────────────1─┐   ▼
 > Program Run Until main
   (Temp) Break module main line 96
 STATUS: Command      68040   MODULE: main      BREAK #: 1   HELP=F5  [◄][►]
```

To scroll the Code window

To see more of the program you can:

- Use the mouse to operate the vertical scroll bar:

Click to go to top of file ————————— ▲

Click to go up one line ———————

Click to go up one half page ———————

Drag slider to move incrementally ———————

Click to go down one half page ———————

Click to go down one line ———————

Click to go to end of file ————————— ▼
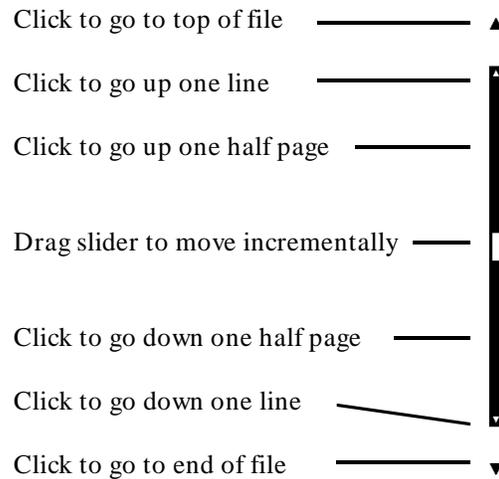
- Use the mouse to operate the horizontal scrolling buttons: ◄ | ►

- Use the < Page Up> and < Page Down> keys on your keyboard.

The scroll bar affects the contents of the active (highlighted) window.

You might notice that the scroll bar has a "sticky" slider which always returns to the center of the scroll bar. This is so that you can always do local navigation even in very large programs. Use the **Disp Src ()** action key or the **Display→Source ()** pull-down menu item to move larger distances.

To display a function

1 Position the cursor over the call to *init_system*.

2 Click the left mouse button.

This will place the string "init_system" into the entry buffer.

3 Click on the **Disp Src ()** action key.

4 Scroll up one line to see the "init_system()" line.

You should now see the source code for the *init_system()* routine in the Code window.

```
 File  Display  Modify  Execution  Breakpoints  Window  Settings                    Help

 Action keys:   < Demo >    Disp Src ( )   C Expr ( )     Run      Run til ( )   Step Over
 < Your Key >     Make      Disp Src PC   Monitor ( )     Step     Run Xfer     Step Out

 ( ):  init_system                                                              Recall
 ─────────────Monitor──────────────3─  ┌──────────Backtrace──────────4─
 1   num_checks     0                   2. 0000043C?crt0\<unknown>
 2   target_temp    0                   1. 00000716 startup\_startup        ▲
 3   current_temp   0                   0. 00000D54*main\main
 4   old_data      [00]:temp        0                                       ▲
 5                      humid       0
 6                      ave_temp    0.000000E+00
 7                      ave_humid   0.000000E+00
 8                 [01]:temp        0

 ──────────────────────────────────Code────────────────────────────2─
     30  init_system()
     31  {    /* FUNCTION init_system() */
     32       /* Initialize the target values for temperature and humidity */
     33       target_temp = 73;
     34       target_humid = 45;
     35
     36       /* Intialize the variables indicating the current environment */
     37       /* conditions */
     38       current_temp = 68;
     39       current_humid = 41;
     40
     41       /* Set starting directions for temp and humid */            ▼
     42       temp_dir = up;
     43       humid_dir = up;
     44
     45       /* Initialize the variables that depict the current status of the */
     46       /* computer room and what hardware needs to be on or off in the room *
 ──────────────────────────────────Journal────────────────────────────1─   ▼
 > Program Context Set init_system
 > Program Display_Source init_system
 STATUS: Command        68040   MODULE: init_system      BREAK #: 1    HELP=F5   ◄ ►
```

To run until a line

1 Position the cursor over line 34. The hand-shaped cursor means that a pop-up menu is available.

```
31  {    /* FUNCTION init_system() */
32       /* Initialize the target values for temperature and humidity */
33       target_temp = 73;
34       target_humid = 45;
```

2 Hold down the right mouse button to display the Code window pop-up menu. Move the mouse to **Run until**, then release the button.

```
34       target_humid = 45;
```

Debugger Display
Set/Delete Breakpoint
Edit source
Attach Macro ...
Edit Attached Macro ...
Run until
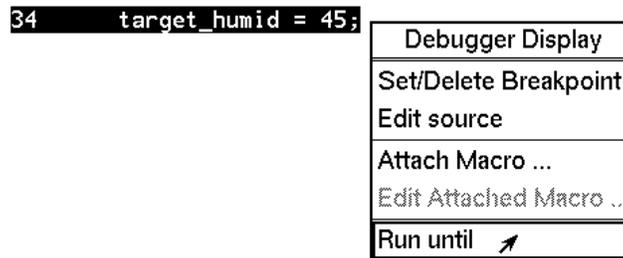
Line 34 should now be highlighted. Notice that "init_system" now appears in the Backtrace window at level 0, which means that the program counter is inside the *init_system()* function.

To edit the program

This step assumes you are using an HP Advanced Cross Language System compiler (HP B1478 or HP B1463). If you are using another compiler, skip this step.

Suppose we wanted the initial value of *target_temp* to be 74 instead of 73. The debugger makes it easy to change the source code:

1 Place the cursor over the assignment to *target_temp* (line 33).

```
33        target_temp = 73;
```

Debugger Display
Set/Delete Breakpoint
Edit source
Attach Macro ...
Edit Attached Macro ...
Run until
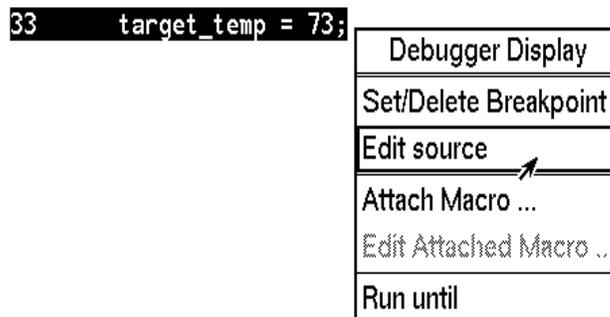
2 Hold the right mouse button and select **Edit Source** from the Code window pop-up menu.

An editor will appear in a new X window. The default text editor is **vi**. You can use a different text editor by editing X resources (described in the "Configuring the Debugger" chapter).

3 Change the "73" to "74".

4 Exit the editor.

5 Click on the **Make** action key.

The program will be re-compiled with the new value and reloaded.

To display init_system() **again**

- Click on the **Disp Src()** action key.

  Since "init_system" is still in the entry buffer, the *init_system( )* routine is displayed.

  You have now completed a edit-compile-load programming cycle.

To set a breakpoint

We want to run until just past the line that we changed.

1 Position the mouse pointer over line 42.

2 Click the right mouse button to set a breakpoint.

  The breakpoint window is displayed, showing the breakpoint has been added.

  An asterisk (*) appears in the first column of the Code window next to the location of the breakpoint. Dots apppear in front of any other lines (such as comments) associated with the breakpoint.

To run until the breakpoint

• Click on the **Run Xfer** action key to run the program from its transfer address.

While the program is executing, the menus and buttons are "grayed out," and an "hourglass" mouse pointer is displayed. You cannot enter debugger commands while the program is executing. If you need to stop an executing program, type **< Ctrl> -C** with the mouse pointer in the debugger X window.

After a few moments, line 42 will be highlighted, showing that program execution stopped there.

The Journal window shows that a break occurred and which breakpoint it was.

```
 File  Display  Modify  Execution  Breakpoints  Window  Settings                 Help

 Action keys:   < Demo >    Disp Src ( )    C Expr ( )      Run       Run til ( )    Step Over
 < Your Key >      Make     Disp Src PC    Monitor ( )      Step       Run Xfer      Step Out

 ( ):  init_system                                                              Recall

 ─────────────────Monitor──────────────3─    ─────────────Backtrace─────────4─
 1   num_checks      0                         3. 0000043C?crt0\<unknown>
 2   target_temp     74                        2. 00000716 startup\_startup
 3   current_temp    68                        1. 00000D5E main\main
 4   old_data      [00]:temp       0           0. 00001264 init_system\init_sy
 5                     humid       0
 6                     ave_temp    0.000000E+00
 7                     ave_humid   0.000000E+00
 8               [01]:temp         0

 ───────────────────────────────Code────────────────────────────────2─
    39        current_humid = 41;
 .  40
 .  41        /* Set starting directions for temp and humid */
 *  42        temp_dir = up;
    43        humid_dir = up;
    44
    45        /* Initialize the variables that depict the current status of the */
    46        /* computer room and what hardware needs to be on or off in the room *
    47        func_needed = 0;
    48        hdwr_encode = 0;
    49
    50        /*Initialize the count of calls to update_state_of_system() */
    51        num_checks = 0;
    52
    53        /* Initialize writing location in old_array */
    54        curr_loc = 0;
    55

 ───────────────────────────────Journal──────────────────────────────1─
 > Program Run
   Break # 1 on instr module init_system line 42

 STATUS: Command        68040    MODULE: init_system      BREAK #: 1    HELP=F5   ◄ ►
```

To patch code using a macro

1 Position the cursor over line 38.

2 Select **Attach Macro** from the Code window pop-up menu.

```
38      current_temp = 68;
39      current_humid = 41;
```

```
Debugger Display
Set/Delete Breakpoint
Edit source
Attach Macro ...
Edit Attached Macro ...
Run until
```
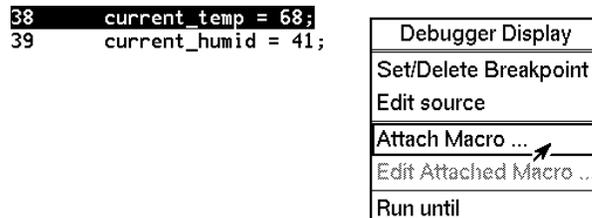
The Macro Operations dialog box appears. The macro "patch_temp" is already selected. Before we attach the macro, let's examine it:

```
Debugger/Emulator: Macro Operations

Defined Macros
writep(port, size, value)
write(fildes, buf, nbyte)
           <User-Defined Macros>_____
graph_on()
do_forward()
show_num()
patch_temp_arg(temp)
patch_temp()

Parameters:                          Recall
Selected Macro
patch_temp

  Edit          Attach          Cancel
```
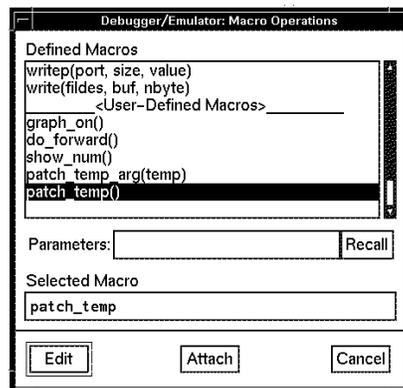
3 Click on the **Edit** button in the dialog box.

This macro will set *current_temp* to 71 each time the breakpoint is encountered. The macro skips over the assignment in the program source code by setting the program counter to line 39. The return value of 0 tells the macro to stop program execution after the macro.

```
Debugger Macro Add int patch_temp()
{
    /* set the current_temp to be 71 degrees instead of what the code says */
    current_temp = 71;

    /* Restart execution at line # 39 -- Skips over the code too!!  */
    $Memory Register @PC = #39$;

    /* Return value indicates continuation logic: 1=continue, 0=break */
    return(0);
}
.
```

4 Exit the editor.

5 Click on the **Attach** button in the dialog box.

The plus sign ("+ ") in front of line 38 indicates that a macro has been attached to a breakpoint at that line.

6 Click on the **Run Xfer** action key to run the program.

Notice that *current_temp*, as shown in the Monitor window, is 71, not 68.

To delete a single breakpoint

Once you set a breakpoint, program execution will break each time the breakpoint is encountered. If you don't want to break on a certain breakpoint again, you must delete the breakpoint. Suppose you want to delete the breakpoint that was previously set at line 42 in *init_system*.

1 Position the mouse over line 42.

2 Click the right mouse button to delete the breakpoint.

The breakpoint window shows the breakpoint has been deleted. The asterisk in front of line 42 disappears.

To delete all breakpoints

1 Position the mouse pointer in the Breakpoint window.

2 Hold down the right mouse button to select **Delete All Breakpoints** from the Breakpoint window pop-up menu.

All breakpoints are deleted.

To step through a program

You can execute one source line (high-level mode) or one instruction (assembly-level mode) at a time by stepping through the program.

- Click on the **Step** action key a few times.

- If you want to try using a pull-down menu, select **Execution→Step→from PC** a few times.

  As the debugger steps through the program, you can see the PC progress through the source code, as shown by the inverse video line in the Code window.

To run until a stack level

Now we need to go back to *main( )*. You can run the program until it enters *main( )* by running to a stack level.

1 Position the mouse pointer over the line containing "main\main" in the Backtrace window.

2 Select **Run Until Stack Level** from the Backtrace pop-up menu.

The program counter is now back in *main( )*, on the call to *proc_spec_init( )*.
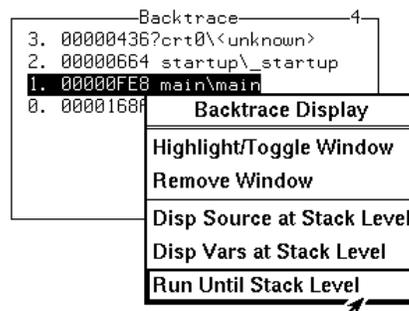
```
┌─────Backtrace──────4─┐
3. 00000436?crt0\<unknown>
2. 00000664 startup\_startup
1. 00000FE8 main\main
0. 0000168┌──────────────────────────┐
         │   Backtrace Display      │
         ├──────────────────────────┤
         │ Highlight/Toggle Window  │
         │ Remove Window            │
         ├──────────────────────────┤
         │ Disp Source at Stack Level│
         │ Disp Vars at Stack Level  │
         ├──────────────────────────┤
         │ Run Until Stack Level    │
         └──────────────────────────┘
```

To step over functions

You can either step through functions or step over functions. When you step over a function, it is executed as a single program step.

- Click on the **Step Over** action key.

The next line in *main( )* is highlighted. The routine *proc_spec_init( )* was executed as a single program step.

To step out of a function

1 Click on the **Step** action key until the program counter is in *update_system( )*.

2 Click on the **Step Out** action key.

The program will execute until it returns from *update_system( )*.

To display the value of a variable

1 Use the left mouse button to highlight "num_checks" in the Code window.

2 Click on the **C Expr ()** action key.

In the Journal window, the current value of the variable is displayed in its declared type (int). Notice that this is the same as the value displayed in the Monitor window.

To change the value of a variable

1 In the entry buffer, add "= 10" after "num_checks".

2 Click on the **C Expr ()** action key.

The new value is displayed in the Journal window and in the Monitor window.

```
 File  Display  Modify  Execution  Breakpoints  Window  Settings                Help

 Action keys:  │ < Demo >  │ Disp Src ( ) │ C Expr ( ) │    Run   │ Run til ( ) │ Step Over
 │< Your Key >│    Make    │Disp Src PC │ Monitor ( ) │   Step   │  Run Xfer  │ Step Out

 ( ): num_checks=10                                                         │Recall│
 ┌──────────────────────Monitor──────────────────3─┬─────────Backtrace────────4─┐
 │1   num_checks   10                               │2. 0000043C?crt0\<unknown>  │  ▲
 │2   target_temp  76                               │1. 00000716 startup\_startup│
 │3   current_temp 70                               │0. 00000D6C main\main        │  ▲
 │4   old_data     [00]:temp       70               │                             │
 │5                humid           43               │                             │
 │6                ave_temp        0.000000E+00      │                             │
 │7                ave_humid       0.000000E+00      │                             │
 │8                [01]:temp       65               │                             │
 ├──────────────────────────────Code───────────────────────────────2─┐         │
 │    100       while (true)                                          │         │
 │    101       {                                                     │         │
 │    102           update_system();                                 │         │
 │    103           num_checks++;                                    │         │
 │    104           interrupt_sim(&num_checks);                      │         ││
 │    105           if (graph)                                       │         ││
 │    106               graph_data();                                │         ││
 │    107           proc_specific();                                 │         │
 │    108       }                                                    │         │
 │    109   }                                                        │         │
 │    110                                                            │         │
 │    111   /************************************************************************│
 │    112    * FUNCTION: interrupt_sim                               │         │
 │    113    * PARMS:    counter -- loop counter passed in from main │         │
 │    114    * DESCRIPTION:                                          │         │
 │    115    *   create a simulation of a (usually) long interrupt service routine tha│  ▼
 │    116    *   also has a duration profile to use with a SPA duration trigger.       │
 ├──────────────────────────────Journal───────────────────────────1─┐         ▼
 │> Expression C_Expression num_checks=10                            │
 │ Result is: 10  0x0A                                               │
 STATUS: Command        68040   MODULE: main          BREAK #: 1   HELP=F5  ◄ ►
```

To recall an entry buffer value

1 Click on the **Recall** button.

2 In the Recall dialog box, click the left mouse button on "num_checks".

3 In the Recall dialog box, click the left mouse button on **OK**.

The string "num_checks" is now in the entry buffer.
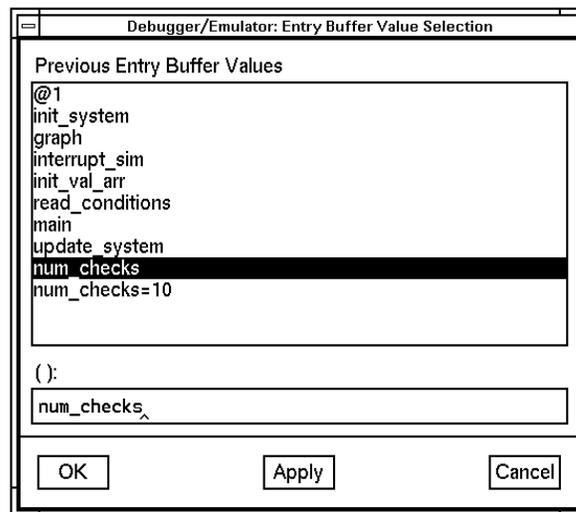
```
┌──────────────────────────────────────────────┐
│ ─  Debugger/Emulator: Entry Buffer Value Selection │
│ ┌──────────────────────────────────────────┐ │
│   Previous Entry Buffer Values               │
│  ┌────────────────────────────────────────┐ │
│  │@1                                       │ │
│  │init_system                              │ │
│  │graph                                    │ │
│  │interrupt_sim                            │ │
│  │init_val_arr                             │ │
│  │read_conditions                          │ │
│  │main                                     │ │
│  │update_system                            │ │
│  │num_checks                               │ │
│  │num_checks=10                            │ │
│  │                                         │ │
│  └────────────────────────────────────────┘ │
│   ( ):                                        │
│  ┌────────────────────────────────────────┐ │
│  │num_checks                               │ │
│  └────────────────────────────────────────┘ │
│  ┌────┐        ┌─────┐        ┌──────┐      │
│  │ OK │        │Apply│        │Cancel│      │
│  └────┘        └─────┘        └──────┘      │
└──────────────────────────────────────────────┘
```

To display the address of a variable

You can use the C address operator (&) to display the address of a program variable.

1 Position the mouse pointer in the entry buffer.

2 Type "&" in the entry buffer so that it contains "&num_checks".

| ( ): | &num_checks | Recall |

3 Click on the **C Expr ()** action key.

The result is the address of the variable *num_checks*. The address is displayed in hexadecimal format.
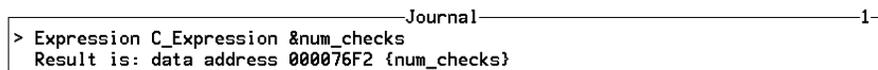
```
────────────────────────Journal──────────────────────────1─
> Expression C_Expression &num_checks
  Result is: data address 000076F2 {num_checks}
```

To break on an access to a variable

You can also set breakpoints on a read, a write, or any access of a variable. This helps to locate defects due to multiple functions accessing the same variable. Suppose you want to break on the access of the variable *num_checks*. ("&num_checks" should still be in the entry buffer.)

1 Set the breakpoint by selecting **Breakpoints→Set→Read/Write ()**.

2 Run the program by clicking on the **Run** action key.
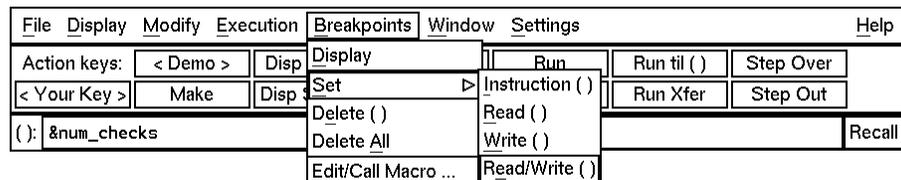
When the program stops, the code window shows that the program stopped at the next reference to the variable *num_checks*.

Try running the program a few more times to see where it stops. (Notice that *num_checks* is passed by reference to *interrupt_sim*. Since *counter* points to the same address as *num_checks*, the debugger stops at references to *counter*.)

3 Delete the access breakpoint. Select **Window→Breakpoints**, place the mouse in the Breakpoint window, press and hold the right mouse button, and choose **Delete All Breakpoints**.

To use the command line

1 Select **Settings→Command Line** from the menu bar.

The command line area which appears at the bottom of the debugger window can be used to enter complex commands using either the mouse or the keyboard.

2 Build a command out of the command tokens which appear beneath the command line entry area.

To use the command line with the mouse, click on the button for each command token.

3 When the command has been built, type or select < Return> .

To use a C printf command

The command line's Expression Printf command prints the formatted output of the command to the Journal window using C format parameters. This command permits type conversions, scaling, and positioning of output within the Journal window.

- Using the command line, enter:

Expression **P**rintf "%010d",num_checks

In this example, the value of *num_checks* is printed as a decimal integer with a field width of 10, padded with zeros.

```
┌─────────────────────────────Journal────────────────────────────────1─┐
│> Expression Printf "%010d",num_checks                                 │
│  0000000011                                                           │
│                                                                       │
```

31

To turn the command line off

1 Move the mouse pointer to the Status line.

2 Hold down the shift key and click the right mouse button.

The shift-click operation selects the second item from a pop-up menu, which in this case is **Command Line On/Off**.

You can turn the command line on and off from the Settings pull-down menu, the Status pop-up menu, and the command line pop-up menu.

To see on-line help

1 Select **Help→General Topic ...**

2 Select **To Use Help**, then click on the **OK** button.

Spend a few minutes exploring the help topics, so that you can find them when you need them.

To end the debugging session

● Use the *command select* mouse button to choose **File→Exit→Yes**.

```
┌─File─┐
│ Context      ▷ │
│ Load         ▷ │
│ Store        ▷ │
│ Copy Window  ▷ │
│ Log          ▷ │
│ Edit         ▷ │
│ Term ...       │
│ Exit         ▷ │ Yes   <Ctrl>X │
```
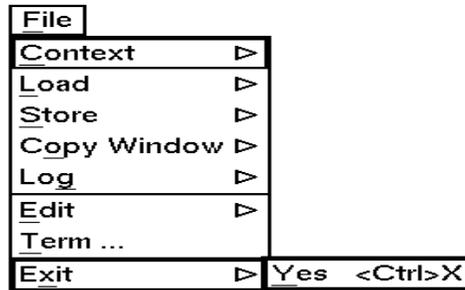
The debug session is ended and your system prompt is displayed.

This completes your introduction to the 68030/040 Debugger/Simulator. You have used many features of the debugger. For additional information on performing tasks with the debugger, refer to the "User's Guide" part of this manual. For more detailed information on debugger commands, error messages, etc., refer to the "Reference" part of this manual.

2

Getting Started with the Standard Interface

How to get started using the debugger's character-based interface.

The Standard Interface At a Glance

The debugger has a *standard interface* for several types of terminals, terminal emulators, and bitmapped displays. When using the standard interface, commands are entered from the keyboard.

This interface will be started if the DISPLAY environment variable is not defined, an X window cannot be opened, or the **-t** command line option is used.

Display area ——

```
┌──────────────────Monitor──────────────3─┐┌────────Backtrace────────4─┐
│ 1   num_checks   0                       ││ 2. 0000043C?crt0\<unknown> │
│ 2   target_temp  0                       ││ 1. 00000716 startup\_startup│
│ 3   current_tem  0                       ││ 0. 00000D58 main\main      │
│ 4   old_data     [00]:temp      0        ││                            │
│ 5                     humid      0        ││                            │
│ 6                     ave_temp   0.000000E+00│                        │
└──────────────────────────────────────────┘└────────────────────────────┘
  ┌──────────────────────────Code──────────────────────────2─┐
  │    93   extern void do_sort();        /* sets up ascii array and calls c│
  │    94                                                     │
  │    95   main()                                            │
  │    96   {                                                 │
  │    97        init_system();                               │
  │    98        proc_spec_init();                            │
  │    99                                                     │
  │   100        while (true)                                 │
  │   101        {                                            │
  │   102            update_system();                         │
  └───────────────────────────────────────────────────────────┘
```

Status line ——

```
STATUS: Command          68040   MODULE: main              BREAK #: 0    HELP=F5
> Program Step
  Breakpoint  Debugger  Expression  File  Memory  Program  Symbol  Window
  Display_Source  Find_Source  Run  Interrupt  Load  Context  Pc_Reset  Step
```

Command line   Command tokens   Command look ahead

Display area. Can show assembly level screen, high-level screen, simulated I/O screen, or user-defined screens. These screens contain windows that display code, variables, the stack, registers, breakpoints, etc. You can use the UP ARROW, DOWN ARROW, PAGE UP, and PAGE DOWN cursor keys to scroll or page up or down the information in the active window.

Status line. Displays the debugger status, the CPU, the current program module, and the number of the last breakpoint.

Command line. Commands are entered on the command line at the debugger prompt (>) and executed by pressing the **< Return>** key. Command tokens are placed on the command line by typing the first uppercase letter of the token. The Tab and Shift-Tab keys allow you to move the cursor on the command line forward or backward. The Clear line key clears from the cursor position to the end of the line. The `<Ctrl>-U` key clears the whole command line.

Command tokens. The second line under the status line shows the tokens that you can enter at the current location in the command line.

Command look ahead. The third line under the status line shows tokens that are available if you select the highlighted command token above.

The Quick Start Tutorial

This tutorial gives you step-by-step instructions on how to perform basic tasks using the debugger.

Perform the tasks in the sequence given; otherwise, your results may not be the same as those shown here.

Some values displayed on your screen may vary from the values shown here.

Before You Begin

This chapter assumes you have already installed the debugger as described in the "Installation" chapter.

Note   Some commands are printed on two lines in this chapter. When entering these commands, type the entire command on one line.

Note   The 68030/040 debugger/simulator can be started using either the command *db68030* or the command *db68040*. The demonstration files used in the examples in this chapter can be copied from directory:

```
/usr/hp64000/demo/debug_env/sim68040
```

The Demonstration Program

The demonstration program used in this chapter is a simple environmental control system (ECS). The system controls the temperature and humidity of a room requiring accurate environmental control. The program continuously looks at flags which tell it what action to take next.

To copy the demonstration files

Before you can run the demonstration program, you must copy the debugger demo files to a new subdirectory. Perform the following steps to make the subdirectory and copy the demo files into it.

1 Make a subdirectory for the debugger demo files.

Make sure the present working directory is the one in which you wish to create the subdirectory for the debugger demo files. Then, enter the command:

```
mkdir mydirectory
```

where *mydirectory* is the name of your debugger demo subdirectory.

2 Change to the debugger demo subdirectory:

```
cd mydirectory
```

3 Copy the demo files:

```
cp -r /usr/hp64000/demo/debug_env/sim68040/* .
```

All files that you need to run the demonstration program should now be in the working directory.

To start the debugger

- Start the debugger by entering:

  ```
  db68040 -t -c cmdfiles/debug/Cmd_dbstart ecs
  ```

  The **-t** option starts the debugger's standard interface.

  The **-c cmdfiles/debug/Cmd_dbstart** option tells the debugger to execute the *Cmd_dbstart.com* command file.

  The **ecs** argument tells the debugger to load the *ecs.x* absolute file.

  This command starts execution of the debugger, which executes the command file *Cmd_dbstart.com* and loads the absolute file *ecs.x.*

Note
Do not start the standard interface as a background process. Doing this has unpredictable results, especially under SunOS.

In other words, never use both the "-t" option and the "&" symbol when you start the debugger.

To enter commands

1 Type the first letter of one of the command tokens listed below the command entry line.

2 Type the first letter of the next command token, if any.

3 Type any necessary parameters. They are indicated on the command token line in < angle brackets> .

4 Press **<Return>** to enter the command.

You may edit the command you are entering. See the section "To edit the command line" in the "Entering Debugger Commands" chapter for more information on how to enter and edit commands.

Note

If a portion of the previous command is still visible on the command line, press **<Ctrl>-E** to clear to the end of the command line before pressing **<Return>**.

To activate display area windows

Notice there are several windows in the main display area of the debugger. The different windows contain different types of information from the debugger. The active window has the thicker border. It is in the active window that the scroll bar and the PAGE UP and PAGE DOWN keys have an effect. There are several ways to activate a window in the display area.

• Press the F1 function key to activate the next higher numbered window or the F2 function key to activate the next lower numbered window.

Or:

• Using the command line, enter the Window Active command.

Try changing the active window a few times. Activate the Code window when you are done.

To display main()

• Enter the following command:

```
Program Display_Source main
```

Remember, to enter this command all you need to type is "P" then "D" then "main" then **<Return>**.

The *main()* function is displayed. Use the UP ARROW key to see the "main()" statement.

To display a function

Notice that *main()* calls *update_system()*.

• Enter the following command:

```
Program Display_Source update_system
```

The *update_system()* function is displayed. Use the UP ARROW key to see the "update_system()" statement.

To set a breakpoint

Suppose you want to execute up to the call to *update_system()*. To do this you could set a breakpoint at the statement "update_system()" and run the program.

• Using the command line, enter:

Breakpt **I**nstr update_system

The breakpoint window is displayed, showing the breakpoint has been added.

An asterisk (*) appears in the first column of the Code window next to the location of the breakpoint. The dot (.) in the first column of the previous lines show the source lines associated with that breakpoint.

To run the demo program

• Using the command line, enter:

Program **R**un

The journal window shows that a break occurred and which breakpoint it was.

Notice that the source file line at which the breakpoint was set is now in inverse video. The inverse video line shows the current program counter. You should now be viewing the *update_system()* routine.

To step through the program

You can execute one source line (high-level mode) or one instruction
(assembly-level mode) at a time by stepping through the program.

• Using the command line, enter:

Program **S**tep

You can step again by just pressing **<Return>**.

As the debugger steps through the program, you can see the PC progress
through the source code, as shown by the inverse video line in the Code
window.

To step over functions

You can either step through functions or step over functions. When you step
over a function, it is executed as a single program step.

• Using the command line, enter:

Program **S**tep **O**ver

To delete a breakpoint

Once you set a breakpoint, program execution will break each time the
breakpoint is encountered. If you don't want to break on a certain breakpoint
again, you must delete the breakpoint. Suppose you want to delete the
breakpoint that was previously set at the statement "update_system()".

1 Run the program up to the breakpoint:

Program **R**un

2 Using the command line, enter:

Breakpt **D**elete 1

The breakpoint window is displayed, showing the breakpoint has been deleted.
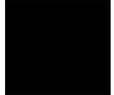
To display variables in their declared type

Whenever you specify a variable name without a C or debugger operator prefix, it is displayed in its declared type.

• Using the command line, enter:

Expression **D**isplay_Value current_temp

In the Journal window, the current value of the variable is displayed in its declared type (int).

To display the address of a variable

You can use the C address operator (&) to display the address of a program variable.

• Using the command line, enter:

Expression **D**isplay_Value &current_temp

The result is the address of the variable *current_temp*. The address is displayed in hexadecimal format.

To use a C printf command

The Expression Printf command prints the formatted output of the command
to the Journal window using C format parameters. This command permits
type conversions, scaling, and positioning of output within the Journal window.

• Using the command line, enter:

Expression **P**rintf "%010d",current_temp

In this example, the value of *current_temp* is printed as a decimal integer with a
field width of 10, padded with zeros.

The Expression Fprintf command can be used to print formatted output to a
file or user-defined window.

To break on an access to a variable

You can also set breakpoints on a read, a write, or any access of a variable.
This helps to locate defects due to multiple functions accessing the same
variable. Suppose you want to break on the access of the variable
&current_temp.

1 Set the access breakpoint:

Breakpt **A**ccess &current_temp

2 Run the program:

Program **R**un

When the program stops (after approximately ten seconds), the code window
shows that the program stopped at the next reference to the variable
current_temp.

Try running the program (just press **<Return>**) a few more times to see where it stops. If the program had a pointer to the variable, it would stop there, too.

3 Delete the access breakpoint.

Using the command line, enter:

Breakpt **D**elete 1

To display blocks of memory

You can display structures and arrays in memory as well as ranges of memory locations that encompass several variables.

• Using the command line, enter:

Memory **D**isplay **B**yte &current_temp

The debugger displays a block of memory starting at the address of the variable *current_temp*.

The C address operator & is used because the Memory Display command is an assembly-level command and expects a memory address as its argument.

To monitor variables

The Expression Monitor Value command allows you to monitor a variable's value during execution of your program.

• Using the command line, enter:

Expression **M**onitor **V**alue current_temp

The value of *current_temp* is now displayed in the Monitor window.

To modify a variable by entering a C expression

The Expression C_Expression command calculates the value of a C expression or modifies a C variable if the C expression contains the assignment operator (=). This command recognizes variable types and the assignment expression specified behave according to the rules of C.

• Using the command line, enter:

Expression **C**_Expression current_temp = 99

Notice that the value of *current_temp* in the Monitor window has changed to the number you entered.

To end the debugging session

- Enter:

Debugger **Q**uit **Y**es

The debug session is ended and your system prompt is displayed. The emulator is released so that other people can use it.

The Yes option causes the debugging session to be terminated immediately. Otherwise, the debugger prompts you with the question Are you sure?. You must then answer **yes** to end the debugging session.

This completes your introduction to the debugger. For additional information on performing tasks with the debugger, refer to the "User's Guide" part of this manual. For more detailed information on debugger commands, error messages, and so on, refer to the "Reference" part of this manual.
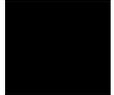
Part 2

User's Guide

Part 2

3

Entering Debugger Commands

How to enter debugger commands using the mouse or the keyboard.

Entering Debugger Commands

This chapter shows you how to enter debugger commands using the graphical interface or the standard interface. The tasks are grouped into the following sections:

- Starting the debugger.

- Using menus, the entry buffer, and action keys.

- Using the command line with the mouse.

- Using the command line with the keyboard.

- Viewing debugger status.

The *graphical interface* provides an easy way to enter commands using a mouse. It lets you use pull-down and pop-up menus, point and click setting of breakpoints, cut and paste, on-line help, customizable action keys and pop-up recall buffers, and other advanced features. To use the graphical interface, your computer must be running an X Window System that supports OSF/Motif interfaces.

The debugger also has a *standard interface* for several types of terminals, terminal emulators, and bitmapped displays. When using the standard interface, commands are entered from the keyboard.

When using the graphical interface, the *command line* portion of the interface gives you the option of entering commands in the same manner as they are entered in the standard interface. If you are using the standard interface, you can only enter commands from the keyboard using the command line.

Function Key Commands

You can enter commonly used commands quickly and easily by pressing the function keys F1 through F8 on your keyboard. Function keys can be used in the graphical interface as well as the standard interface. The following table and figure describe the commands associated with the function keys.

If you are using the debugger on a Sun SPARCsystem, refer to the "Installation" chapter for information on mapping function keys.

Function Key Commands

Function Key	Graphical Equivalent, Command Line Equivalent	Description
F1	**Display→Next Window**, Window Active Next	Activate the next higher numbered window.
F2	**Display→Previous Window**, Window Active Previous	Activate the next lower numbered window.
F3	**Settings→High Level Debug** or **Settings→Assembly Level Debug**, Debugger Level	Switch between assembly-level and high-level mode.
F4	Right click on active window border, Window Toggle_View	Select the alternate display of the active window.
F5	**Help→Command Line...**, Debugger ? (Help)	Access on-line help.
F6	**Display→Simulated I/O**, Window Screen_On Next	Access the standard I/O screen. Also access any existing user-defined screens.
F7	**Execution→Step Instruction→from PC**, Program Step	Execute one C source line (high-level mode), or execute one microprocessor instruction (assembly-level mode).
F8	**Execution→Step Source→from PC**, Program Step Over	Execute one C source line, but treat whole functions as a single line (high-level mode); execute one microprocessor instruction, but treat whole subroutines as a single instruction.

Command Line Control Character Functions

Press the control key `<Ctrl>` simultaneously with the **B**, **C**, **E**,**F**, **G**, **L**, **Q**, **R**, **S**, **U**, or **** keys to execute the operations listed in the following table. (The letter keys may be upper- or lower-case.)

Command Line Control Character Functions

Control	Function
`<Ctrl> B`	Recall command reverse.
`<Ctrl> C`	Abort the current command and return to debugger command mode.
`<Ctrl> E`	Clear to end of command line.
`<Ctrl> F`	Shift contents of active window to right.
`<Ctrl> G`	Shift contents of active window to left.
`<Ctrl> L`	Redraw screen.
`<Ctrl> Q`	Resume output to screen (standard interface only).
`<Ctrl> R`	Recall previous command.
`<Ctrl> S`	Suspend output to screen (standard interface only).
`<Ctrl> U`	Clear command line
`<Ctrl> \`	End the debug session (same as Debugger Quit Yes command)

The Journal Window

The debugger displays debugger commands entered from the keyboard in the Journal window. The Journal window also displays warning and informational messages from the debugger and output generated by commands. This window is available in both the high-level and assembly-level screens.

Starting the Debugger

Use the *db68040* or *db68030* command to start the debugger.

See Also   The "Getting Started with the Graphical Interface" chapter for information about starting the graphical interface.

The "Getting Started with the Standard Interface" chapter for information about starting the standard interface.

The "Loading and Executing Programs" chapter for information about loading programs as you start the debugger.

The "Using Macros and Command Files" chapter for information about loading command files as you start the debugger.

The "Configuring the Debugger" chapter for information about using debugger startup files.

The on-line "manual page" for information about the *db68040* command and its command-line options. To see this information, type the following operating system command:

```
man db68040
```

Using Menus, the Entry Buffer, and Action Keys

This section describes the tasks you perform when using the debugger's graphical interface to enter commands. This section describes how to:

- Choose a pull-down menu item using the mouse.

- Choose a pull-down menu item using the keyboard.

- Use the pop-up menus.

- Use action keys.

- Use the entry buffer.

- Copy and paste to the entry buffer.

- Use dialog boxes.

- Access help information.

To choose a pull-down menu item using the mouse (method 1)

1 Position the mouse pointer over the name of the menu on the menu bar.

2 Press and hold the *command select* mouse button to display the menu.

3 While continuing to hold down the mouse button, move the mouse pointer to the desired menu item. If the menu item has a cascade menu (identified by an arrow on the right edge of the menu button), then continue to hold the mouse button down and move the mouse pointer toward the arrow on the right edge of the menu. The cascade menu will display. Repeat this step for the cascade menu until you find the desired menu item.

4 Release the mouse button to select the menu choice.

If you decide not to select a menu item, simply continue to hold the mouse button down, move the mouse pointer off of the menu, and release the mouse button.

Some menu items have an ellipsis ("...") as part of the menu label. An ellipsis indicates that the menu item will display a dialog or message box when the menu item is chosen.

Note   The *command select* button can be either the left or right button, depending on the computer you are using. The "Getting Started with the Graphical Interface" chapter has a table which explains which button to use.

To choose a pull-down menu item using the mouse (method 2)

1 Position the mouse pointer over the menu name on the menu bar.

2 Click the *command select* mouse button to display the menu.

3 Move the mouse pointer to the desired menu item. If the menu item has a cascade menu (identified by an arrow on the right edge of the menu button), then repeat the previous step and then this step until you find the desired item.

4 Click the mouse button to select the item.

If you decide not to select a menu item, simply move the mouse pointer off of the menu and click the mouse button.

Some menu items have an ellipsis ("...") as part of the menu label. An ellipsis indicates that the menu item will display a dialog or other box when the menu item is chosen.

To choose a pull-down menu item using the keyboard

- To initially display a pull-down menu, press and hold the *menu select* key (for example, the "Extend char" key on a HP 9000 keyboard) and then type the underlined character in the menu label on the menu bar. (For example, "f" for "File". Type the character in lower case.)

- To move right to another pull-down menu after having initially displayed a menu, press the **right-arrow** key.

- To move left to another pull-down menu after having initially displayed a menu, press the **left-arrow** key.

- To move down one menu item within a menu, press the **down-arrow** key.

- To move up one menu item within a menu, press the **up-arrow** key.

- To choose a menu item, type the character in the menu item label that is underlined. Or, move to the menu item using the arrow keys and then press the **< RETURN>** key on the keyboard.

- To cancel a displayed menu, press the **Escape** key.

The interface supports keyboard mnemonics and the use of the arrow keys to move within or between menus. For each menu or menu item, the underlined character in the menu or menu item label is the keyboard mnemonic character. Notice the keyboard mnemonic is not always the first character of the label. If a menu item has a cascade menu attached to it, then typing the keyboard mnemonic displays the cascade menu.

Some menu items have an ellipsis ("...") as part of the menu label. An ellipsis indicates that the menu item will display a dialog or other box when the menu item is chosen.

Dialog boxes support the use of the keyboard as well. To direct keyboard input to a dialog box, you must position the mouse pointer somewhere inside the boundaries of the dialog box. That is because the interface *keyboard focus*

policy is set to *pointer*. That just means that the window containing the mouse pointer receives the keyboard input.

In addition to keyboard mnemonics, you can also specify keyboard accelerators which are keyboard shortcuts for selected menu items. Refer to the "Setting X Resources" chapter and the "Debug.Input" scheme file for more information about setting the X resources that control defining keyboard accelerators.

To choose pop-up menu items

1 Move the mouse pointer to the area whose pop-up menu you wish to access. (If a pop-up menu is available, the mouse pointer changes from an arrow to a hand.)

2 Press and hold the right mouse button.

3 After the pop-up menu appears (while continuing to hold down the mouse button), move the mouse pointer to the desired menu item.

4 Release the mouse button to select the menu choice.

If you decide not to select a menu item, simply continue to hold the mouse button down, move the mouse pointer off of the menu, and release the mouse button.

Some pop-up menus which are available include:

- Display-area Windows.

- Status Line.

- Command Line.

To use pop-up menu shortcuts

- To choose the first item in a pop-up menu, click the right mouse button.

- To choose the second item in a pop-up menu, hold down the **< Shift>** key and click the right mouse button.

To place values into the entry buffer using the keyboard

1 Position the mouse pointer within the text entry area. (An "I-beam" cursor will appear.)

2 Enter the text using the keyboard.

To clear the entry buffer text area from beginning until end, press the **< Ctrl> U** key combination.

To copy-and-paste to the entry buffer

- To copy and paste a "word" of text, position the mouse pointer over the word and click the left mouse button.

- To specify the exact text to copy to the entry buffer, position the mouse pointer over the first character to copy, then hold the left mouse button while dragging the mouse pointer over the text. When you release the mouse button, the highlighted text will appear in the entry buffer.

  You can copy-and-paste from the display area, the status line, and from the command line entry area.

On a memory display, you may need to scroll the display to show more characters of a symbol.

The interface displays absolute addresses as hex values. If you copy and paste an address from the display to the entry buffer, you must add a trailing "h" to make the interface interpret it as a hex value when you use the entry buffer contents with a command.

Text pasted into the entry buffer replaces that which is currently there. You cannot use paste to append text to text already in the entry buffer. You can retrieve previous entry buffer values by using the **Recall** button.

See "To copy-and-paste from the entry buffer to the command line entry area" for information about pasting the contents of the entry buffer into the command line entry area.

Example

To paste the symbol "init_system" into the entry buffer from the interface display area, position the mouse pointer over the symbol and then click the left mouse button.

```
(): init_system                                                      Recall
```

```
     96   {
     97       init_system();
     98       proc_spec_init();
     99
```
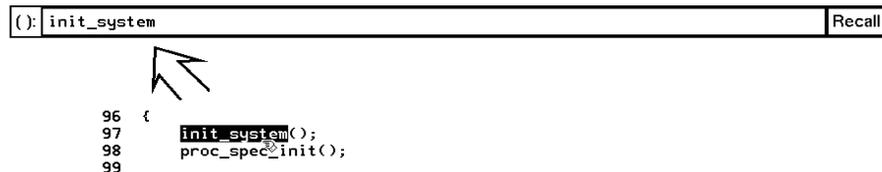
To recall entry buffer values

1 Position the mouse pointer over the **Recall** button just to the right of the entry buffer text area, and click the mouse button to bring up the Entry Buffer Value Selection dialog box.

2 In the dialog box, click on the string you want.

3 In the dialog box, click on the "OK" button.

The Entry Buffer Value Selection dialog box contains a list of previous values from the entry buffer. You can also predefine entries for the Entry Buffer Value Selection dialog box and define the maximum number of entries by setting X resources (refer to the "Setting X Resources" chapter).

If you decide not to change the contents of the entry buffer, click on the "Cancel" button in the dialog box.

If you want the Entry Buffer Value Selection dialog box to remain visible after you make a selection, press "Apply" instead of "OK". You may drag the dialog box to another location on your display so that it does not cover the debugger window.

See the following "To use dialog boxes" section for information about using dialog boxes.

To edit the entry buffer

- To position the keyboard cursor, click the left mouse button or use the arrow keys.

- To clear the entry buffer, type **< Ctrl> -U**.

- To delete characters, press the **< Backspace>** or **< Delete char>** keys.

- To delete several characters, highlight the characters to be deleted using the left mouse button, then press the **< Backspace>** or **< Delete char>** keys.

To use the entry buffer

1 Place information into the entry buffer (see the previous "To place values into the entry buffer using the keyboard", "To copy-and-paste to the entry buffer", or "To recall entry buffer values" task descriptions).

2 Choose the menu item, or click the action key, that uses the contents of the entry buffer.

The contents of the entry buffer will be used wherever the "()" symbol appears in a menu item or action key.

To copy-and-paste from the entry buffer to the command line entry area

1 Position the mouse pointer within the command line text entry area.

2 If necessary, reposition the keyboard cursor to the location where you want to paste the text.

3 If necessary, choose the insert or replace mode for the command entry area.

4 Click the middle mouse button to paste the text into the command line entry area at the current cursor position.

Note   You should paste to the command line *only* when the command line is expecting an address or a string. The characters from the entry buffer will be treated as if they were typed from the keyboard. If the command line is expecting keyword tokens, pasting can have unexpected results. For example, pasting "delta" into an empty command line will generate a "Debugger Execution Load_State ta" command!

Although a paste from the display area to the entry buffer affects all displayed entry buffers in all open windows, a paste from the entry buffer to the command line only affects the command line of the window in which you are currently working.

See "To copy-and-paste to the entry buffer" for information about pasting information from the display into the entry buffer.

To use the action keys

1 If the action key uses the contents of the entry buffer, place the desired information in the entry buffer.

2 Position the mouse pointer over the action key and click the action key.

Action keys are user-definable pushbuttons that perform interface or system functions. Action keys can use information from the entry buffer — this makes it possible to create action keys that are more general and flexible.

Several action keys are predefined when you first start the debugger's graphical interface. You can use the predefined action keys to make, load,

run, and step through the demo program. You'll really appreciate action keys when you define and use your own.

Action keys are defined by setting an X resource. Refer to the chapter "Setting X Resources" for more information about creating action keys.

To use dialog boxes

1 Click on an item in the dialog box list to copy the item to the text entry area.

2 Edit the item in the text entry area (if desired).

3 Click on the "OK" pushbutton to make the selection and close the dialog box, click on the "Apply" pushbutton to make the selection and leave the dialog box open, or click on the "Cancel" pushbutton to cancel the selection and close the dialog box.

The graphical interface uses a number of dialog boxes for selection and recall:

Directory Selection Selects the working directory. You can change to a previously accessed directory, a predefined directory, or specify a new directory.

File Selection From the working directory, you can select an existing file name or specify a new file name.

Entry Buffer Recall You can recall a previously used entry buffer text string, a predefined entry buffer text string, or a newly entered entry buffer string, to the entry buffer text area.

Command Recall You can recall a previously executed command, a predefined command, or a newly entered command, to the command line.

The dialog boxes share some common properties:

- Most dialog boxes can be left on the screen between uses.

- Dialog boxes can be moved around the screen and do not have to be positioned over the graphical interface window.
- If you iconify the interface window, all dialog boxes are iconified along with the main window.

Except for the File Selection dialog box, predefined entries for each dialog box (and the maximum number of entries) are set via X resources (refer to the "Setting X Resources" chapter).

In file names, you may use a tilde as shorthand for your home directory.

Examples   To use the File Selection dialog box:

The file filter selects
specific files.

A list of
filter-matching files
from the current
directory.

A list of files
previously accessed
during the debugger
session.

A single click on a
file name from
either list highlights
the file name and
copies it to the text
area. A double click
chooses the file and
closes the dialog
box.

```
┌──────────────────────────────────────────────────────┐
│ ▭   Debugger/Simulator: File Selection                │
├──────────────────────────────────────────────────────┤
│ File Filter                                            │
│ ┌──────────────────────────────────────────────────┐  │
│ │ /users/myproj/*.c                                │  │
│ └──────────────────────────────────────────────────┘  │
│                                                        │
│ Files                                                  │
│ ┌──────────────────────────────────────────────┐ ┌─┐ │
│ │ /users/myproj/init_system.c                   │ │▲│ │
│ │ /users/myproj/main.c                          │ ├─┤ │
│ │ /users/myproj/proc_spec.c                     │ │ │ │
│ │ /users/myproj/update_sys.c                    │ │ │ │
│ │         <Previous Files>                      │ │ │ │
│ │ /usr/hp64000/lib/X11/app-defaults/HP64_Debug  │ │ │ │
│ │ /users/myproj/.Xdefaults                      │ │ │ │
│ │ /users/myproj/Xdefaults.sim.man               │ │ │ │
│ │                                               │ ├─┤ │
│ │                                               │ │▼│ │
│ └──────────────────────────────────────────────┘ └─┘ │
│ ┌─┐────────────────────────────────────────────┌─┐   │
│ │◄│                                             │►│   │
│ └─┘────────────────────────────────────────────└─┘   │
│ vi                                                     │
│ ┌──────────────────────────────────────────────────┐  │
│ │ /users/myproj/main.c                             │  │
│ └──────────────────────────────────────────────────┘  │
├──────────────────────────────────────────────────────┤
│ ┌──────┐          ┌────────┐          ┌────────┐      │
│ │  OK  │          │ Filter │          │ Cancel │      │
│ └──────┘          └────────┘          └────────┘      │
└──────────────────────────────────────────────────────┘
```

Label informs you
what kind of file
selection you are
performing.

Text entry area. Text is
either copied here
from the recall list, or
entered directly.
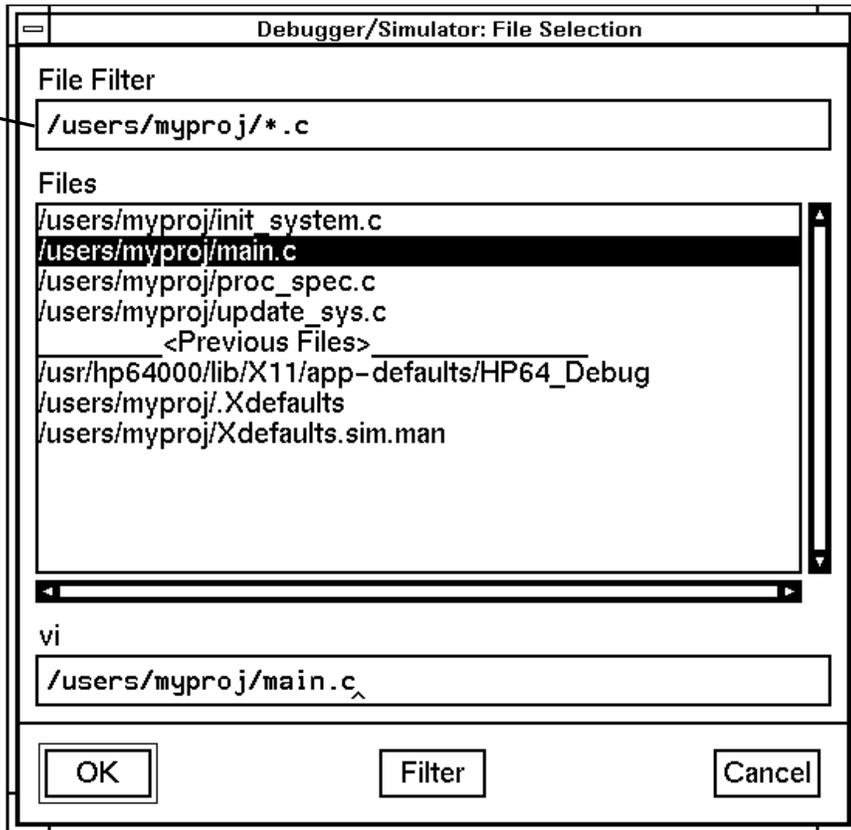
Clicking this button
chooses the file name
displayed in the text
entry area and closes
the dialog box.

Entering a new file
filter and clicking this
button causes a list of
files matching the new
filter to be read from
the directory.

Clicking this button
cancels the file
selection operation
and closes the dialog
box.

To use the Directory Selection dialog box:

Label informs you of the type of list displayed.

A single click on a directory name from the list highlights the name and copies it to the text area. A double click chooses the directory and closes the dialog box.

A list of predefined or previously accessed directories.

Debugger/Simulator: Directory Selection

Previous Working Directories

```
# To customize the initial list of entries look for "dirSelectSub"
# in the file $HP64000/lib/X11/app-defaults/HP64_Debug
# and add your own definition to your .Xdefaults file.
# Use the File->Edit->File pulldown to edit these files.
#
$HOME
..
$HP64000/demo/debug_env
/users/myproj
```

Selection

```
/users/myproj
```

| OK | | Apply | | Cancel |

Text entry area. Directory name is either copied here from the recall list, or entered directly.

Clicking this button chooses the directory displayed in the text entry area and closes the dialog box.

Clicking this button chooses the directory displayed in the text entry area, but keeps the dialog box on the screen instead of closing it.

Clicking this button cancels the directory selection operation and closes the dialog box.

To access help information

1 Display the Help Index by choosing **Help→General Topic ...** or
Help→Command Line ....

2 Choose a topic of interest from the Help Index.

The Help Index lists topics covering operation of the interface as well other
information about the interface. When you choose a topic from the Help
Index, the interface displays a window containing the help information. You
may leave the window on the screen while you continue using the interface.

Examples

To see more information on how to use the on-line help, click on **Help**, then
click on **General Topics ...**, then click on "To Use Help", then click on the
"OK" button.
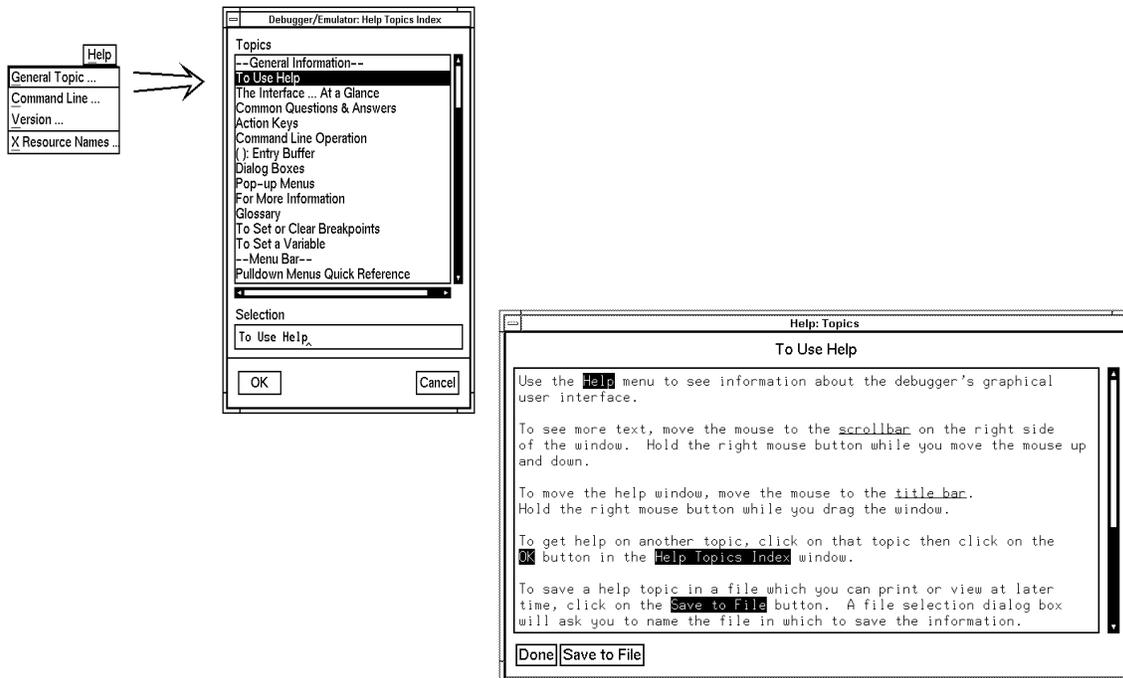
Using the Command Line with the Mouse

When using the graphical interface, the *command line* portion of the interface gives you the option of entering commands in the same manner as they are entered in the standard interface. Additionally, the graphical interface makes the command tokens pushbuttons so commands may be entered using the mouse.

If you are using the standard interface, the command line is the only way to enter commands.

This section describes how to:

- Turn the command line off/on.

- Enter commands.

- Edit commands.

- Recall commands.

- Display the help window.

To turn the command line on or off

- To turn the command line on or off using the pull-down menu, choose **Settings**→**Command Line**.

- To turn the command line on or off using the status line pop-up menu: position the mouse pointer within the status line area, press and hold the right mouse button, and choose **Command Line On/Off** from the menu.

- To turn the command line on or off with a single mouse click, hold the **< Shift>** key and click on the status line.

- To turn the command line off using the command line entry area pop-up menu: position the mouse pointer within the entry area, press and hold the right mouse button, and choose **Command Line On/Off** from the menu.

- To turn the command line on with the keyboard: place the mouse pointer in the display area and press any alphanumeric key.

  "On" means that the command line is displayed and you can use the command token pushbuttons, the command return and recall pushbuttons, and the cursor pushbuttons for command line editing. "Off" means the command line is not displayed and you can use only the pull-down and pop-up menus and the action keys to control the interface.

  The command line area begins just below the status line and continues to the bottom of the debugger window. The status line is not part of the command line and continues to be displayed whether the command line is on or off.

  Choosing certain pull-down menu items while the command line is off causes the command line to be turned on. That is because the menu item chosen requires some input at the command line that cannot be supplied another way.

To enter a command

1 Build a command using the command token pushbuttons by successively positioning the mouse pointer on a pushbutton and clicking the left mouse button until a complete command is formed.

2 Execute the completed command by clicking the **Return** pushbutton (found near the bottom of the command line in the "Command" group).

Or:

Execute the completed command using the Command Line entry area pop-up menu: Position the mouse pointer in the command line entry area; press and hold the right mouse button until the Command Line pop-up menu appears; then, choose the **Execute Command** menu item.

You may need to combine pushbutton and keyboard entry to form a complete command.

A complete command is a string of partial commands or command tokens. You know a command is complete when "< return> " appears on one of the command token pushbuttons. The interface does not check or act on a command, however, until the command is executed. (In contrast, commands resulting from menu choices and action keys are supplied with the needed carriage return as part of the command.)

To edit the command line using the command line pushbuttons

- To clear the command line, click the **Clear** pushbutton.

- To clear the command line from the cursor position to the end of the line, click the **Clear to end** pushbutton.

- To move to the right one command word or token, click the **Forward** pushbutton.

- To move to the left one command word or token, click the **Backup** pushbutton.

- To insert characters at the cursor position, press the **Insert char** key to change to insertion mode, and then type the characters to be inserted.

- To delete characters to the left of the cursor position, press the **< Backspace>** key.

When the cursor arrives at the beginning of a command word or token, the softkey labels change to display the possible choices at that level of the command.

When moving by words left or right, the **Backup** pushbutton is grayed out and unresponsive when the cursor reaches the beginning of the command string.

See "To edit the command line using the mouse and the command line pop-up menu" and "To edit the command line using the keyboard" for information about additional editing operations you can perform.

To edit the command line using the command line pop-up menu

- To clear the command line: position the mouse pointer within the Command Line entry area; press and hold the right mouse button until the Command Line pop-up menu appears; choose **Clear Entire Line** from the menu.

- To clear the command line from the cursor position to the end of the line: position the mouse pointer at the place where you want the clear-to-end to start; press and hold the right mouse button until the Command Line pop-up menu appears; choose **Clear to End of Line** from the menu.

- To position the cursor at the next token or the previous token: press and hold the right mouse button until the Command Line pop-up menu appears; choose **Forward Tab** or **Backward Tab** from the menu.

When the cursor arrives at the beginning of a command word or token, the softkey labels change to display the possible choices at that level of the command.

See "To edit the command line using the mouse and the command line pushbuttons" and "To edit the command line using the keyboard" for information about additional editing operations you can perform.

To recall commands

1 Click the pushbutton labeled **Recall** in the Command Line to display the dialog box.

2 Choose a command from the buffer list. (You can also enter a command directly into the text entry area of the dialog box.)

Because all command entry methods in the interface — menus, action keys, and command line entries — are echoed to the command line entry area, the

contents of the Command Recall dialog box is not restricted to commands entered directly into the command line entry area.

The Command Recall dialog box contains a list of interface commands executed during the debugger session as well as any predefined commands present at interface startup.

You can predefine entries for the Command Recall dialog box and define the maximum number of entries by setting X resources (refer to the "Setting X Resources" chapter).

See "To use dialog boxes" for information about using dialog boxes.

To get help about the command line

- To display the help topic explaining the operation of the command line, select **Help→General Topic ...→Command Line Operation**.

- To display the command line help menu, select **Help→Command Line ...**.

Using the Command Line with the Keyboard

Commands are entered on the command line at the debugger prompt (>) and executed by pressing the **< Return>** key. Command tokens are entered by typing a single letter, typically the first uppercase letter of the token.

The third and fourth lines of the status window display command tokens. The third line shows the tokens that you can enter at the current location in the command line. The fourth line shows tokens that are available if you select the highlighted command token on the third line. The command token lines provide you with a look ahead feature, showing you the debugger commands available to you at any time.

This section describes how to:

- Enter commands.

- Edit commands.

- Recall commands.

- Access on-line help information.

To enter debugger commands from the keyboard

1 Build a command using direct keyboard entry by successively typing letters corresponding to command tokens until a complete command is formed.

2 Execute a completed command using the keyboard, press the **< Return>** key on the keyboard.

You can enter commands any time the cursor is displayed on the command line. You can enter only one debugger command at a time.

Debugger commands have the following syntax:

command   [qualifier...]   *[parameter...]*

To enter a command keyword, type the first letter of the keyword. For example, to enter the command *Debugger Level Assembly*, type the letters D, L, and A. The following command will appear on the command line:

Debugger **L**evel **A**ssembly

Press **< Return>** to enter (execute) the command.

In command examples, the letter you must type is highlighted in bold type.

Note

In cases where you can select from more than one keyword beginning with the same letter, type the first uppercase letter of the desired keyword. For example, type **O** to select **O**n and **F** to select o**FF**.

Enter qualifier keywords in the same way as command keywords. Qualifiers provide the debugger with information on how to execute the command. Qualifiers are normally single words that immediately follow the command name. For example, in the command:

Program **F**ind_Source **N**ext **B**ackward

the qualifier *Backward* causes the debugger to search the file from the current position in the file towards the beginning of the file for a specified string.

Type parameters in their entirety from the keyboard. Parameters must be separated from the command or qualifier keyword by at least one space. Parameters describe the object of the command and are typically C expressions that represent values or addresses used by the command. For example, in the command:

Expression **D**isplay_Value &system_is_running

the parameter *&system_is_running* specifies the address of the variable *system_is_running.*

To edit the command line

- To clear the command line, press **< Ctrl> U**.

- To clear the command line from the cursor position to the end of the line, press **< Ctrl> E**.

- To move to the right one command word, press **< Tab>** .

- To move left or right character-by-character, press the ←and →keys.

- To delete characters to the left of the cursor position, press the **< BACKSPACE>** key.

When the cursor arrives at the beginning of a command word or token, the softkey labels change to display the possible choices at that level of the command.

To recall commands using the command line recall feature

- To recall commands from the command line, press the **< Ctrl> R** key combination. Continue to press **< Ctrl> R** to move from the most recently executed commands backward to earlier commands.

- To move forward in the recall list, press **< Ctrl> B**.

The command line recall feature is available to you, but it is not as easy to use or as flexible as the Command Recall dialog box in the graphical interface. You must search through commands in a linear fashion instead of going directly to the command you want in the dialog box. The depth of the recall list is predefined and cannot be controlled by you. The recall list may contain duplicate entries that you must scroll past and that take up room in the recall

list. Finally, you cannot predefine entries for the recall list — the list only
contains the most recent commands executed during the debugger session.

To display the help window

- Press the function key **F5**.

  Or:

- Enter the command

  Debugger **?**

  This command displays a menu of debugger commands, command parameters,
  function keys, and other debugger features. Descriptions for each topic may be
  obtained by positioning the cursor on the first letter of any topic in the help
  menu and pressing the **< Return>** key.

  The debugger's help window is context sensitive. When you display the help
  window, the cursor is located on the last command you entered before
  displaying the help window. The debugger assumes you need help with this
  command. Press **< Return>** to display information about the command.

  Pressing **< Return>** or **< Down>** displays information on the next item in the
  help menu. Pressing **< Up>** displays information about the previous item in
  the help menu.

  You can move the cursor to the first command of a command type
  (Breakpoint, Debugger, etc.) by entering the first letter of the command type.
  For example, to move the cursor to the entry for the first window command,
  enter:

  W

The cursor will be positioned at the Window Active command entry. Then you can use the cursor keys to select the window command you need help with and press **< Return>** to display information on that command.

Press the **F5** function key one time or press the escape (**< Esc>**) key twice to exit the help window. (Note that you cannot exit the graphical interface help window this way.)

Viewing Debugger Status

The status line shows you what the debugger is doing. The status line:

- Contains information about the operation being performed by the debugger.

- Contains indicators to warn you about special conditions.

- Shows the microprocessor being simulated.

- Shows the program module associated with the current program counter.

- Shows the number of the last breakpoint that occurred.

The status line is always present in both the graphical interface and the standard interface.

The debugger displays the status line in the following format:

```
STATUS:<Status> [J][L][W]  CPU   MODULE: <module>        BREAK #: <#>   TRC:<Trc_status>
                   [R]
```

Debugger Status

The Status field on the status line shows the current state of the debugger. The possible values for this field are:

Command	The debugger is ready to accept a command or a macro definition.
Execute	The debugger is executing target environment instructions. The debugger displays *Execute* on the status line when you enter the Program Run command or the Program Step command.
ComFile	The debugger is reading commands from a command file.
Macro	The debugger is executing a macro.
Paused	The debugger is in the paused state after execution of the Debugger Pause command.

Reading	The debugger is reading an executable file or a C source file into the debugger's memory.
Working	The debugger is executing internal debugger operations.

Indicator Characters

The Warning indicator (W) indicates that the program counter is not on a C source line boundary. The debugger displays a warning when it detects a breakpoint, an instruction halt, or an instruction error between lines.

The Log indicator (L) indicates that commands are being logged to a log file.

The Journal indicator (J) indicates that everything appearing in the Journal window is being written to a journal file.

The Register indicator (R) indicates that a register variable is being used, but its lifetime is not known by the debugger. The debugger displays an R when the variable is referenced, indicating that the values being used for this variable may not be valid.

CPU Simulated

The CPU entry indicates which microprocessor is being simulated.

Current Module

The MODULE: entry names the current module (< module>). The current module is the module pointed to by the program counter. If the program counter points outside of the known code area associated with the program, this entry displays *???????*.

Last Breakpoint

The BREAK # entry indicates the number of the last breakpoint that occurred, or (0) zero if execution was not terminated with a breakpoint.

If pop-up menus don't pop up

When you hold the right mouse button down, a pop-up menu does not appear. Here are some things to check:

☐ Check that the mouse pointer is hand-shaped.

Some areas of the screen do not have pop-up menus.

☐ Check that your mouse buttons are not being redefined by your window manager.

If you are using **mwm** to redefine your mouse buttons, delete the redefinition from your .mwmrc file.

☐ If you are using an older window manager such as **mwm**, look in

`/usr/hp64000/lib/X11/HP64_schemes/HP-UX/Debug.Input`

Copy the line

`HP64_Debug*whichButton: Button5`

to your .Xdefaults file. Change the 5 to a 3.

4

Loading and Executing Programs

How to load a program into the debugger and control its execution.

Compiling Programs for the Debugger

Writing programs for simulation

Several microprocessor features work differently in the simulator:

- Some instructions and exception conditions cause the microprocessor to read or write data to CPU space. CPU space references are intended for communication with external hardware (such as a floating point coprocessor or security-checking device). For this reason, the memory simulator ignores CPU space writes and returns $FF for CPU space reads. Thus, there is no mechanism for simulating coprocessor communication.

- Since the debugger provides complete step and breakpoint control, the trace bits in the Status Register are ignored; they do not cause a trace exception to occur after each instruction.

- The BKPT instruction causes an *illegal instruction* exception. Use normal debugger commands to set breakpoints.

- Function codes are used to select memory protection based on the MMU table, but all simulated function code memory spaces overlap. Therefore, they cannot be used as an extra three bits of address when referencing memory in the simulator.

- Data and instruction caches are not simulated. The 68040 instructions CPUSH and CINV are simulated as NOPs.

Using the 68040 FPU

FPU behavior is simulated with the following differences from the actual processor.

Rounding Mode

No postprocessing is performed to affect rounding of results. Neither mode (to nearest, toward zero, toward plus infinity, toward minus infinity) nor precision (extended, single, double) are performed. Only round to the nearest (mode bits = 00) and the double rounding precision (precision bits = 10) are supported. If the single or extended rounding precision is used, the floating-point register values may differ from the values obtained by executing

the program on the MC68040. For those familiar with the operation of the FPCR, this means the mode control byte is not used by the simulator.

Instructions for which rounding to single precision can be specified do not perform rounding, either; however, they decode and disassemble correctly. They are FSMOVE, FDMOVE, FSSQRT, FDSQRT, FSABS, FDABS, FSNEG, FDNEG, FSDIV, FDDIV, FSADD, FDADD, FSMUL, FDMUL, FSSUB, and FDSUB.

Extended Precision

Floating point registers are simulated with 64 bits instead of 80; however, they are displayed in the Registers window as 80-bit patterns. Memory-to-register operations with an extended precision source format are converted to double precision before they are stored. Register-to-memory operations with an extended precision destination format are stored with an extended precision representation of the double precision register value.

FPU State Frames

The contents of the state frames (triggered by FSAVE) can only be approximated because, by Motorola's definition, they contain information specific to the floating point part of the chip that is not visible to the user. The version number generated by the simulator is presently 0xff, the version used by Motorola's 68881 is 0x1f, and the version used by the 68040 is 0x40.

Packed Decimal

Unlike the 68040 chip, which signals exception vector 55 (Unimplemented Data Type), the simulator signals exception vector 11 (F-line Exception).

FPSR Quotient Byte

This portion of the floating point status register is not set. Users who make use of this data will get incorrect program results.

Inexact Results

The INEX1 bit of the FPSR is not set. This is moot because the bit is set during conversion of packed decimal inputs, which are not supported by the simulator.

Using the 68040 MMU

All protection features of the 68040 MMU are fully simulated. The simulated MMU makes use of MMU registers, the data and instruction ATCs, and table walks as appropriate. Address translation is not performed, however. The simulated ATC has only a single entry instead of the architecture described by Motorola.

68040 bus error exceptions caused by an MMU-detected fault are simulated according to the @exc pseudoregister.

None of the 68030 MMU functionality is simulated. The 68030 MMU instructions are simulated as NOPs.

Using a Hewlett-Packard C Cross Compiler

Use the default compile mode when compiling your target programs for use with the debugger. The default settings generate executable files (.x file extension) in the HP-MRI IEEE-695 file format required by the debugger. The default option settings force a stack frame to be built for every function call, which is required for stack backtracing.

The "Getting Started" chapter of the *68030 C Cross Compiler User's Guide* or the *68040 C Cross Compiler User's Guide* gives an example of how to compile a simple program and execute it in the debugger/simulator environment.

Note   Do not use the *–h* option when compiling and linking your program for the debugger. The *–h* option causes the compiler to generate HP 64000 file formats. Use the default settings which generate executable files in the file format required by the debugger. The debugger extracts all symbolic information from the executable (.x) file.

Using Environment Dependent Files

The HP B1478 and HP B1463 C Cross Compilers provide environment dependent files that support the HP B1467 68030/040 Debugger/Simulator environment. These environment dependent routines affect the following areas of C programming:

- program setup

- dynamic memory allocation

- program input and output

The "Environment Dependent Routines" chapter of the *68030 C Cross Compiler Reference* or the *68040 C Cross Compiler Reference* describes the environment dependent routines supplied with the compiler.

Using Optimizing Modes

If you use the optimizing modes (–O or –OT), function calls that do not have automatic variables may not have stack frames. As a result, the stack backtrace window will not contain entries for such functions. Additionally, the optimizing modes will cause the compiler to generate code which is not easily debugged.

Note

When initially compiling a program for the debugger, you should turn off all optimizations to avoid confusion when using the debugger. After program flow and all basic algorithms have been debugged, you can recompile the program with all optimizations turned on.

When you compile with all optimizations on, one or more of the following problems may occur while using the debugger:

- Target program execution in the debugger may not appear to correctly reflect the logical flow of the program.

- The debugger may not stop execution at a high-level breakpoint or may stop execution at the wrong location in the program.

- The debugger may not be able to display local variables.

Forcing Variables to be Placed in Memory

The default compiler settings automatically create register variables for statics and frequently used variables. Some debugger functions such as access breakpoints will not work with register variables. The compiler option *-Wc, -F* turns off the compiler's automatic creation of register variables, forcing the compiler to assign these variables to memory. This enables greater functionality of some debugger commands. After debugging your code, you can then recompile your code without these options for greater efficiency.

Using Math Libraries

Although FPU instructions can be executed in the target system, the debugger/simulator cannot execute these instructions. To generate code that will run interchangeably in both the debugger/emulator and debugger/simulator, use the C compiler's floating point library routines. These libraries contain routines that do not use FPU instructions, thereby allowing them to execute properly in both debugging environments.

References

The "Getting Started" chapter of the *68030 C Cross Compiler User's Guide* gives an example of how to compile a simple program and execute it in the debugger environment.

The "Command Syntax" chapter of the *68030 C Cross Compiler User's Guide* gives detailed descriptions of compiler options.

The "Environment Dependent Routines" chapter of the *68030 C Cross Compiler Reference* describes the environment dependent routines supplied with the compiler.

Loading Programs and Symbols

This section shows you how to:

- Specify the location of C source files.

- Load programs.

- Load programs only (without symbols).

- Load symbols only (without the program).

- Load additional programs.

- Specify demand loading of symbols.

To specify the location of C source files

- Before you start the debugger, set the HP64_DEBUG_PATH environment variable.

The location of C source files can be defined to the debugger with the UNIX shell variable *HP64_DEBUG_PATH*. If HP64_DEBUG_PATH is defined, the debugger only searches for the files in the path(s) specified in HP64_DEBUG_PATH, in the order in which they are listed.

The % character can be included in the path to cause the debugger to search the location of the source files recorded in the absolute file.

If HP64_DEBUG_PATH is not defined, the debugger searches for source files in the following sequence:

1 their location at compile time (this information is recorded in the absolute file)
2 the current directory (if the required source files are not found in their compile location)

Example   The shell variable definition:

```
HP64_DEBUG_PATH=/users/proj/src:/users/proj/mysrc:%
export HP64_DEBUG_PATH
```

causes the debugger to search paths for C source files in the following order:

1   /users/proj/src
2   /users/proj/mysrc
3   the paths specified in the absolute file at compile time

If you use the csh shell (most Sun systems), use **setenv** instead of **export** to set the variable.

To load programs

- When starting the debugger, enter the executable file name as the last term in the db68040 or db68030 command line.

  $ **db68040** <abs_file>

  Or:

- Select **File→Load→Executable**, then use the File Selection dialog box to select the executable file.

  Or:

- Using the command line, enter:

  Program **L**oad **D**efault <file_name>

  When you load an absolute file using these commands, the debugger:

  1   Removes all previous program symbols.

  2   Removes all previously set breakpoints.

  3   Resets the program counter (PC).

  4   Loads the full symbol set.

5   Loads the new executable module.

Absolute files contain executable object code. They must have a file name extension of *.x*. You do not need to specify the *.x* file extension when entering the absolute file name.

The **P**rogram **L**oad **D**efault command is equivalent to the **P**rogram **L**oad **N**ew **A**ll **P**c_Set command.

Examples   To load the executable file *ecs.x*:

```
$ db68040 ecs
```

Or:

```
Program Load Default ecs
```

To load programs code only

- Select **File→Load→Program Only ...**, then use the File Selection dialog box to select the absolute file.

  Or:

- Using the command line, enter:

  ```
  Program Load New Code_only No_Pc_Set <absolute_name>
  ```

  Enter the name of the absolute file whose code is to be loaded, and press the **<Return>** key.

  The code image will be loaded without loading symbols or resetting the PC.

  If you are re-loading a program, you may need to re-specify variables for the Monitor window. To re-load a program without clearing the Monitor window, enter:

  ```
  Program Load Append Code_only No_Pc_Set <absolute_name>
  ```

To load symbols only

- Use the -I option to the db68040 command when starting the debugger.

  ```
  $ db68040 -I <absolute_file> <RETURN>
  ```

  Or:

- Select **File→Load→Symbols Only ...**, then use the File Selection dialog box to select the absolute file.

  Or:

- Using the command line, enter:

  ```
  Program Load New Symbols_only No_Pc_Set <absolute_file>
  ```

  Enter the name of the absolute file whose symbols are to be loaded, and press the **< Return>** key.

  Only symbolic information is loaded from the absolute file.

To load additional programs

- Using the command line, enter:

  ```
  Program Load Append
  ```

  Select either All, Code_Only, or Symbols_Only. Then, select either Pc_Set or No_Pc_Set. Finally, enter the name of the absolute file to be appended, and press the **< Return>** key.

  All   both code and symbols are loaded.

Code_Only   only code from the absolute file is loaded.

Symbols_Only   only symbols from the absolute file are loaded.

Pc_Set   the program counter (PC) is set to the transfer address
                 found in the absolute file.

No_Pc_Set   the program counter (PC) is not changed.

When you append a program, it is loaded without deleting the existing
program. The new symbols will be added in a tree with the executable file
name as the root.

Examples   To append the program "module2.x" to the current program without setting
                 the program counter:

Program **L**oad **A**ppend **A**ll **N**o_Pc_Set module2

To specify demand loading of symbols

• Use the -d option when starting the debugger.

The -d option turns on demand loading of symbols, loading some symbol
information on an as-needed, demand basis rather than during the initial load
of the .x file.

Symbol information for global symbols, local symbols in the source module
containing main, and local symbols in assembly modules are loaded during the
initial load of the .x file. Local symbols in C source modules other than that
module which contains main are loaded either when the user explicitly
references the module or when the program is stopped with the program
counter in the module. The primary advantage of demand load is that it lets
you load and debug programs that otherwise would not be loadable because of
very large amounts of symbol information.

There are several side effects of demand loading. The debugger command
Memory Unload_BBA is disabled. Type mismatch errors may not be detected

during the initial load of the .x file. Global symbols may have leading underscores stripped depending on whether they were defined/referenced in a C or assembly source module.

Examples

To specify demand loading of symbols when starting the debugger:

$ **db68040 −d** <RETURN>

To prevent demand loading of symbols

- Use the -doff option when starting the debugger.

The -doff option turns off demand loading of symbols. This option wil override the option in the startup file.

Stepping Through and Running Programs

The various Program Run command options can be combined to make complex run-time control commands for your program.

This section shows you how to:

- Step through programs.

- Step over functions.

- Run from the current PC address.

- Run from a start address.

- Run until a stop address.
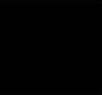
To step through programs

- Click on the **Step** action key.

  Or:

- Select **Execution→Step→from PC**.

  Or:

- Using the command line, enter:

  Program **S**tep

  And press the **< Return>** key.

  Your program executes one C source line (high-level mode) or one machine instruction (assembly-level mode) at a time from the address contained in the program counter (PC). When the program calls a function, stepping continues in the called function.

You can specify a starting address with the Program Step command. You can also specify a step count to cause the debugger to step multiple lines or instructions in your program.

The debugger updates the screen after each instruction or line is executed. The highlighted line in the Code window (which indicates the value of the program counter) is the location of the next line to be executed. If a breakpoint is encountered, single-stepping is halted.

You can also use function key *F7* to single-step.

If the debugger steps into an HP library routine, run until the stack level above the level of the library routine. Use the Program Run Until command or the Backtrace window pop-up menu.

To step over functions

- Click on the **Step Over** action key.

  Or:

- Select **Execution→Step Over→from PC**.

  Or:

- Using the command line, enter:

  Program **S**tep **O**ver

  And press the **< Return>** key.

  The debugger steps through the program one line or one instruction at a time. However, if the debugger encounters a C function or assembly-level JSR or CALL instruction, it stops stepping, executes the JSR or CALL instruction, and then continues stepping when the called subroutine returns.

  You can also use function key *F8* to step over functions.

To run from the current PC address

- Click on the **Run** action key.

  Or:

- Select **Execution→Run→from PC**.

  Or:

- Using the command line, enter:

  Program **R**un

  And press the **< Return>** key.

  The program runs until:
  - The program encounters a permanent or temporary breakpoint.
  - An error occurs.
  - A STOP instruction is encountered.
  - You press **< Ctrl> -C**.
  - The program terminates normally.

  You can run from the current program counter address to resume program execution after the program has been stopped.

To run from a start address

1 Enter the start address into the entry buffer.

2 Select **Execution→Run→from ()**.

  Or:

- Using the command line, enter:

Program **R**un **F**rom <start_addr>

Type in the start address, and press the **< Return>** key.

The program runs until:
- The program encounters a permanent or temporary breakpoint.
- An error occurs.
- A STOP instruction is encountered.
- You press **< Ctrl> -C**.
- The program terminates normally.

Running from a start address in high-level mode may cause unpredictable results if the compiler startup routine is bypassed.

To run until a stop (break) address

1 Enter the stop address into the entry buffer.

2 Select **Execution**→**Run**→**until ()** or click on the **Run Til ()** action key.

Or:

- Using the command line, enter:

Program **R**un **U**ntil <break_addr>

Type in the stop address and, optionally, a pass count, and press the
< Return> key.

The break address (< break_address>) acts as a temporary instruction breakpoint. It is automatically cleared when program execution is halted.

The pass count (< pass_count>) parameter specifies the number of times the break address is executed before the program is halted. For example, a pass

count of three will cause the program to break on the fourth execution of the break address.

Multiple break addresses are OR'ed. In other words, if you specify more than one break address, the program runs until either address is encountered.

Examples   To run the program until either line 20 or line 90 is encountered, whichever occurs first.

Program **R**un **U**ntil #20,#90

To run from the current program counter address until the break address *update_state_of_system* is encountered twice:

Program **R**un **U**ntil update_state_of_system %%2

The Until option in the command sets a temporary breakpoint at address *update_state_of_system*. The pass count parameter % % 2 specifies that the debugger is to stop program execution on the second access to address *update_state_of_system*.

To count simulated clock cycles

- Look at the *cycles* value in the register window.

When using the simulator, the *cycles* value in the register window is the cumulative number of clock cycles executed since this value was last reset (by means of the command Memory Registers @cycles = 0, for example). This count is based on 68030 or 68040 cycle counts, with the assumption that no instructions overlap in the pipeline. Actual cycle counts on a pipelined processor may be significantly less.

If an instruction causes an exception, cycle counts for exception processing are included even if exception processing is disabled. Partially executed instructions receive full cycle counts.

Cycle times are shown with the cache disabled. Enabling the cache has no effect on cycle times.

You can use the @cycles pseudoregister in expressions and macros. Keep in mind, however, that @cycles exists only in the simulator. Macros which will be used with a debugger/emulator must not refer to @*cycles*.

To add simulated wait states

- Set the value of the @wait_state pseudoregister.

The maximum value of @wait_state is 255 (0xFF).

Example

To add three cycles to simulated memory accesses, type "@wait_state= 3" in the entry buffer, then click on the **C EXPR ()** action key.

Using Breakpoints

The debugger implements breakpoints using shadow bits. The number of access breakpoints available to the debugger/simulator is unlimited.

This section shows you how to:

- Set a memory access breakpoint (read, write, or either).

- Set an instruction breakpoint.

- Clear selected breakpoints.

- Clear all breakpoints.

- Display breakpoint information.
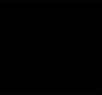
To set a memory access breakpoint

- Enter the address (which may be a symbol) in the entry buffer. Select **Breakpoints→Set** and select **Read**, **Write**, or **Read/Write**.

  Or:

- Using the command line, enter **Breakpt** select the type of access to break on (Read, Write, or Access), enter the address of the memory location, and press the **< Return>** key.

The access types have the following meanings:

Read   break on read accesses.

Write   break on write accesses.

Access   break on either read or write accesses.

Access breakpoints cause the debugger to halt program execution each time the target program reads from or writes to the specified memory location(s). Memory locations can contain code or data.

Examples

To cause execution to halt each time the program reads from or writes to the variable current_temp:

Breakpt **A**ccess &current_temp

To cause execution to halt each time the program reads from the variable current_temp:

Breakpt **R**ead &current_temp

To cause execution to halt each time the program writes to the variable current_temp:

Breakpt **W**rite &current_temp

To set an instruction breakpoint

- Position the mouse pointer in the code window over the line at which you wish to set a breakpoint. Either click the right mouse button, or press and hold the right mouse button to display the Debugger Display pop-up menu and choose **Set/Clear Breakpoint** from the menu.

  Or:

- Enter the instruction address into the entry buffer, then select **Breakpoints**→**Set**→**Instruction ()**.

  Or:

- Using the command line, enter:

  Breakpt **I**nstr <addr>

Enter the address of the instruction location, and press the **< Return>** key.

The instruction breakpoint causes the debugger to halt program execution each time the target program attempts to execute an instruction at the specified memory location(s). The debugger halts program execution before the program executes the instruction at the breakpoint address.

If you specify a range, the debugger sets breakpoints on the first byte of each instruction within the specified range.

Set breakpoints are marked with asterisks "*" in the code window. In the high-level mode, dots "." show the source lines associated with a breakpoint.

Examples

To set an instruction breakpoint at line 82 of the current module:

```
Breakpt Instr #82
```

To set a breakpoint for a C+ + object instance

• Use the dot or arrow operator to specify the object and the member function.

This allows you to set a breakpoint for a member function only when it is invoked for a given object or instance.

Example

To break when function *cfunc* is invoked by object instance *cobj1*, enter:

```
Breakpoint Instr cobj1.cfunc
```

To do this the hard way, you could enter:

```
Breakpoint Instr C::cfunc\@entry;when (C::cfunc\this==
&cobj1)
```

To set a breakpoint for overloaded C+ + functions

- To set a breakpoint at one of the functions when you know the argument type, supply the argument type following the function name.

- To set a breakpoint at one of the functions when you don't know which argument type you want, just use the name of the function. The debugger will list the choices with a menu in the Journal window.

Example

To set a breakpoint for the function *print* (which is not in a class) for **float** arguments, enter **print (float)** in the entry buffer and select **Breakpoints→Set ()**.

Another way to set a breakpoint for the function *print* is to enter **print** in the entry buffer, select **Breakpoints→Set ()**, then type the number of "print (float);" from the menu in the Journal window.

To set a breakpoint for C+ + functions in a class

- Set a breakpoint for the C+ + class.

Examples

To set breakpoints for all member functions of the class *classname*, enter "classname::" in the entry buffer, then select **Breakpoints→Set ()** from the menu bar.

Or, using the command line, enter:

Breakpoint **I**nstr classname::

To clear selected breakpoints

- Position the mouse pointer in the Code window over the line at which you wish to clear a breakpoint. Click the right mouse button.

  Or:

- Position the mouse pointer in the Code window over the line at which you wish to clear a breakpoint. Hold the right mouse button and select **Set/Clear Breakpoint**.

  Or:

- Position the mouse pointer in the Breakpoint window over the breakpoint you wish to clear. Hold the right mouse button and select **Delete Breakpoint**.

  Or:

- Enter the breakpoint number into the entry buffer, the select **Breakpoints→Delete ()**.

  Or:

- Using the command line, enter:

  Breakpt **D**elete <brkpt_nmbr>

  Enter the breakpoint number, and press the **< Return>** key.

  The debugger assigns a breakpoint number to each breakpoint. The debugger uses this number to remove the breakpoint.

  The < brkpt_nmbr> is the number of the breakpoint displayed in the debugger breakpoint window. Enter a range of breakpoint numbers (< brkpt_nmbr> ..< brkpt_nmbr>) to remove more than one breakpoint at a time. When you delete a breakpoint, all following breakpoints are renumbered.

Examples

To delete breakpoint number 1:

Breakpt **D**elete 1

To clear all breakpoints

- Select **Breakpoints→Delete All**.

  Or:

- Select **Delete All Breakpoints** from the Breakpoints window pop-up menu.

  Or:

- Using the command line, enter:

  Breakpt **C**lear_All

  And press the < **Return**> key.

To display breakpoint information

- Select **Window→Breakpoints**.

  Or:

- Using the command line, enter:

  Window **A**ctive **B**reakpoint

  And press the < **Return**> key.

The debugger displays the breakpoint window when:

- You enter a breakpoint command.
- You execute the Window Active Breakpoint command.
- You use function keys F1/F2 to activate next/previous windows.

The Breakpoint window temporarily overlays the top portion of the screen.

When made active, this window displays breakpoint information including:

- Breakpoint number.
- Breakpoint address.
- Name of the module or function containing the breakpoint (in high-level mode).
- Module line number (in high-level mode).
- Breakpoint type.
- Command arguments entered with the breakpoint command.

The following paragraphs describe each field in the breakpoint window.

Breakpoint number

The debugger assigns a breakpoint number (#) when you execute a breakpoint command. The debugger uses this number as a label to reference or clear each breakpoint.

Breakpoint address

The breakpoint address (ADDRESS) shows the memory location of the breakpoint. The debugger displays the address as a hexadecimal value.

Module/function

The module/function field (MOD/FNCT) displays either the name of the module containing the breakpoint or the name of a function if you qualified the breakpoint with a function name. If you specify a module name with a breakpoint command, the name must be followed by a line number (for example: *main\#80*). The field width is eight characters. The debugger truncates field entries greater than eight characters in length to eight characters.

Line number

The line number entry (LINE) displays a module line number if you set a breakpoint in a high-level module. If the compiler did not generate executable code for the C statement at the line number specified, the debugger examines the source code and sets a breakpoint on the next line number for which the compiler generated executable code.

In the code window, the debugger places asterisks beside all line numbers that are associated with breakpoints. The debugger places period symbols (.) beside line numbers that are specified as breakpoints, but have no code associated with them.

Breakpoint type

The breakpoint type (TYPE) describes what type of breakpoint is set: instruction, read, write, or access. In assembly-level mode, the debugger sets instruction breakpoints on microprocessor instruction addresses. In high-level mode, the debugger sets instruction breakpoints on source line numbers. The debugger flags instruction breakpoints with /A (assembly-level) or /H (high-level). When switching between modes, these flags are useful for differentiating between the different types of breakpoints.

Command argument

The debugger records arguments (COMMAND ARGUMENT) in the breakpoint window as you entered them on the command line. Line numbers, addresses, symbol names, and macro names all appear in this field. For more information about breakpoints, see the specific breakpoint command descriptions in the "Debugger Commands" chapter.

To halt program execution on return to a stack level

- Select **Run Until Stack Level** from the Backtrace window pop-up menu.

Or:

1 Set a stack level breakpoint.

2 Run the program.

3 If desired, delete the breakpoint that was just encountered.

Example

Assume that you want to run the program until it returns to the *main( )* function. You can determine where to set a breakpoint on return to main by using the stack level information in the backtrace window (you may have to activate this window in order to see the information in it).

There is a number next to the function *main( )* in the backtrace window. This is the current stack level of *main( )*. This is the address of the machine level instruction immediately following the call to *initialize_system*.

Place the mouse pointer over the line in the backtrace window that lists "main." Hold the right button and select **Run Until Stack Level**.

Or, using the command line and assuming *main( )* is at stack level 1, enter:

```
Breakpoint Instr @1
```

This command will cause program execution to stop when the program returns to the function *main*. The at sign (@) is a debugger operator that causes the debugger to interpret the number 1 as a stack level.

Executing the Breakpt Instr command causes the debugger to update and display the Breakpoint window. The breakpoint you just entered is shown in the Breakpoint window. Now use the appropriate commands to run the program and delete the breakpoint.

Using Simulated Interrupts

The debugger can simulate program interrupts to your target program. The debugger lets you specify a delay between interrupts in terms of a clock cycle count. The pseudoregister @*cycles* maintains the current clock cycle count.

Caution

The pseudoregister @cycles is not implemented in the emulation environment. Macros written for execution in both the simulation and emulation environments must not refer to @*cycles*.

This section shows you how to:

- Define simulated interrupts.

- Remove simulated interrupts.

To define simulated interrupts

- Using the command line, enter:

Program **I**nterrupt **A**dd

Select Repetitive or Once, specify how often the interrupt should occur, specify the interrupt level number and exception vector number; then, press the **< Return>** key.

Use the Program Interrupt Add command to cause a simulated program interrupt to occur after a specified number of clock cycles have been executed. You can define simulated interrupts to be repetitive or to occur only once.

Examples

To set a one-time interrupt to occur in ten cycles at interrupt level 7 through interrupt vector 66:

Program **I**nterrupt **A**dd **O**nce 10,7,66

To set a repetitive interrupt occurring every 70000 cycles at interrupt level 7 to interrupt vector 64:

Program **I**nterrupt **A**dd **R**epetitive 70000,7,64

To remove simulated interrupts

- Using the command line, enter:

Program **I**nterrupt **R**emove

Enter the level number of the interrupts to be removed, and press the < **Return**> key.

The Program Interrupt Remove command cancels any pending interrupts.

Examples   To remove all level 7 interrupts:

Program **I**nterrupt **R**emove 7

To remove all interrupts:

Program **I**nterrupt **R**emove

Restarting Programs

This section shows you how to:

- Reset the processor.

- Reset the program counter to the starting address.

- Reset program variables.

To reset the processor

- Select **Execution**→**Reset to Monitor**.

  Or:

- Using the command line, enter:

  Debugger **E**xecution **R**eset_Processor

  And press the **< Return>** key.

  Resetting the processor simulates a microprocessor reset operation by restoring the microprocessor to its initial state.

To reset the program counter to the starting address

- Select **Execution**→**Set PC to Transfer** .

  Or:

- Using the command line, enter:

  Program **P**c_Reset

  And press the **< Return>** key.

  The program counter is reset to the transfer address of your absolute file. The next Program Run or Program Step command entered without a *from* address will restart program execution at the beginning of the program.

To reset program variables

- Reload your program.

  Memory is not reinitialized when you reset the processor or reset the program counter. Therefore, program variables are not reset to their original values. To reset program variables after resetting the processor or program counter, reload your program.

  For faster loading, you can load only the program code. The debugger retains symbol information. You do not have to reload symbol information if symbol addresses have not changed.

  For information on loading programs, refer to the previous "Loading Programs and Symbols" section.

Saving and Loading the CPU State

State files are used to save the current CPU state (memory image, register values, and program symbols) of a debug session. Though state files can only be created from within a debugger/simulator session, you can use them to restore a CPU state in either a debugger/simulator or debugger/emulator session.

This section shows you how to:

- Save the current CPU state.

- Load a saved CPU state.

To save the current CPU state

- Using the command line, enter:

Debugger **E**xecution **S**ave_State

Enter the name of the file in which the CPU state should be saved, and press the **< Return>** key.

The current memory contents and register values are saved to the specified file. If a file name is not specified, the default file name *db68040.sav* (regardless of whether the debugger is started as *db68030* or *db68040*) is used.

Examples

To save the current memory contents and register values in file "session1.sav":

Debugger **E**xecution **S**ave_State session1

To load a saved current CPU state

- Using the command line, enter:

  Debugger **E**xecution **L**oad_State

  Enter the name of the file from which the CPU state should be loaded, and press the **< Return>** key.

  The memory contents and register values saved with the debugger/simulator Debugger Execution Save_State command are restored from the specified state file. If you do not specify a file name, the debugger uses the default file *db68040.sav*.

Examples   To restore memory contents and register values saved in state file "session1":

Debugger **E**xecution **L**oad_State session1

Mapping Memory

This section shows you how to:

- Prevent access to memory locations.

- Prevent writing to memory locations.

- Allow access to memory locations.

- Display current memory map assignments.

To prevent access to memory locations

- Using the command line, enter:

Memory **M**ap **G**uarded

Enter the range of addresses that should not be accessed, and press the
< Return> key.

Examples   To configure memory address range 8000h through 0a000h as guarded
(nonaccessible) memory:

Memory **M**ap **G**uarded 8000h..0a000h

To prevent writing to memory locations

- Using the command line, enter:

Memory **M**ap **R**ead_Only

Enter the range of addresses that should not be written to, and press the
< Return> key.

Examples   To configure memory address range 8000h through 8fffh as read-only (ROM)
memory:

Memory **M**ap **R**ead_Only 8000h..8fffh

To allow access to memory locations

- Using the command line, enter:

Memory **M**ap **W**rite_Read

Enter the range of addresses to which reads and writes are allowed, and press
the **< Return>** key.

Examples   To configure memory address range 2000h through 3fffh as Write_Read
(RAM) memory:

Memory **M**ap **W**rite_Read 2000h..3fffh

To display current memory map assignments

- Using the command line, enter:

Memory **M**ap **S**how

And press the **< Return>** key.

Examples   To display the memory map:

Memory **M**ap **S**how

```
> Memory Map Show
     TYPE    OWNER       ADDRESS           COMMAND LINE
     RAM     SIMU    00000000..0000002F  Load section
     RAM     SIMU    000000BC..000000BF  Load section
     RAM     SIMU    00000100..0000010B  Load section
     RAM     SIMU    00000400..00001044  Load section env
     RAM     SIMU    00001048..00002851  Load section prog
     RAM     SIMU    00002854..0000906B  Multiple load sections
     RAM     SIMU    00040000..00043FFF  Load section stack
     RAM     SIMU    00060000..0006401D  Multiple load sections
```

The command displays memory address ranges mapped as Guarded
(NOMEM), Read_Only (ROM), or Write_Read (RAM) in the journal
window. The display includes a list of sections loaded and their address ranges.

Accessing Input Ports

This section shows you how to:

- Set or alter input port status.

- Delete an input port.

- Rewind the input file associated with an input port.

- Display input port buffer values.

To set or alter input port status

- Using the command line, enter:

Memory **I**nport **A**ssign

Select the size, port address, and input data source; then, press the **< Return>** key.

The Memory Inport Assign command assigns a simulated input port and defines its size, address, and input source. The port address can be any valid address. The source of input data may be the standard I/O screen, the journal window, a file, an expression string, or the input or output port buffers.

Examples   To assign address 0x40C as an I/O port (input) of size byte:

Memory **I**nport **A**ssign **B**yte 0x40C **S**ource_Is **D**ata_String "message"

Read operations from the port will access the string containing the word 'message'.

To delete an input port

- Using the command line, enter:

Memory **I**nport **D**elete

Enter the address of the input port to be disabled, and press the < **Return**> key.

The specified input port address is disabled, allowing the address to behave like a normal memory location.

Examples
To disable the input port at address 400h:

Memory **I**nport **D**elete 400h

To rewind the input file associated with an input port

- Using the command line, enter:

Memory **I**nport **R**ewind

Enter the address of the input port whose associated input file or is to be rewound or whose input string pointer is to be reset, and press the < **Return**> key.

Examples
To rewind the input file or string associated with input port 400h:

Memory **I**nport **R**ewind 0x400h

To display input port buffer values

- Using the command line, enter:

Memory **I**nport **S**how

Enter the address or address range of the input ports whose buffer values are to be displayed, and press the < **Return>** key.

Each input port has a single value buffer associated with it. The buffer contains the last value read from the port. This value can be represented in byte, word, or long format.

Examples   To show all assigned input ports:

Memory **I**nport **S**how

To show input port at address 400h:

Memory **I**nport **S**how 0x400

To show all input ports in the address range 400h through 4ffh:

Memory **I**nport **S**how 0x400..0x4ff

Accessing Output Ports

This section shows you how to:

- Set or alter output port status.

- Delete an output port.

- Rewind the output file associated with an output port.

- Display output port buffer values.

To set or alter output port status

- Using the command line, enter:

Memory **O**utport **A**ssign

Select the size, port address, and output destination; then, press the
< Return> key.

(explanatory text)

The Memory Outport Assign command defines the address, size, and output
destination of a simulated output port. The target program can write output
data the the simulated output port. The port address can be any valid address.

Examples

To assign address 0x408 as an I/O port (output) of size word:

Memory **O**utport **A**ssign **W**ord 0x408 Destination_Is File
"/myproj/cmdout.dat"

Write operations to the port will access file '/myproj/cmdout.dat'. You must
specify the file name in quotation marks.

To assign address 0x40C as an I/O port (output) of size byte:

Memory **O**utport **A**ssign **B**yte 0x40C **D**estination_Is **S**tdio

Write operations to the port will access the stdio window.

To delete an output port

• Using the command line, enter:

Memory **O**utport **D**elete

Enter the address of the output port to be disabled, and press the **< Return>** key.

The specified output port address is disabled, allowing the address to behave like a normal memory location.

Examples   To disable the output port at address 408h:

Memory **O**utport **D**elete 408h

To rewind the output file associated with an output port

• Using the command line, enter:

Memory **O**utport **R**ewind

Enter the address of the output port whose associated output file is to be rewound, and press the **< Return>** key.

Examples   To rewind the output file associated with output port 408h:

Memory **O**utport **R**ewind 0x408

To display output port buffer values

- Using the command line, enter:

Memory **O**utport **S**how

Enter the address or address range of the output ports whose buffer values are to be displayed, and press the **< Return>** key.

Each output port has a one value buffer associated with it that contains the last value written to the port. The buffer value can be displayed in byte, word, or long format.

Examples   To display all assigned output ports:

Memory **O**utport **S**how

To display output port at address 408h:

Memory **O**utport **S**how 0x408

To display all output ports in the address range 400h through 4ffh:

Memory **O**utport **S**how 0x400..0x4ff

Accessing the UNIX Operating System

This section shows you how to:

- Fork a UNIX shell.

- Execute a UNIX command.

To fork a UNIX shell

- Select **File→Term**.

  A terminal emulation window will be created.

  Or:

- Using the command line, enter:

  Debugger **H**ost_Shell

  And press the **< Return>** key.

  The *Debugger Host_Shell* command lets you temporarily leave the debugging environment by forking a UNIX shell. The shell created is whatever the shell variable *SHELL* is expanded to. In this mode, you may enter operating system commands.

  The Debugger Host_Shell command does not end the debugger session; it suspends program operation. To return to the debugger, enter **< Ctrl> -D** or type **exit** at the UNIX prompt, and press the **< Return>** key.

To execute a UNIX command

- Using the command line, enter:

Debugger **H**ost_Shell

Type in the UNIX command, and press the **< Return>** key.

When using the graphical interface, a terminal emulation window will be opened and the UNIX command will be executed in that window (as specified by the "shellCommand" X resource).

When using the standard interface, *stdout* from the command is written to the journal window. *stderr* is not captured. Commands writing to *stderr* will corrupt the display. Interactive UNIX commands **cannot** be used in this mode.

Examples

To display the current working directory, enter:

Debugger **H**ost_Shell pwd

Using simulator and emulator debugger products together

You can continue a debugging session started in the debugger/simulator in the debugger/emulator by following the steps listed below:

1 In the debugger/simulator, use the **Debugger Execution Save_State** command to save the current memory contents and register values.

2 Quit the simulator session using the **Debugger Quit** command.

3 Start the debugger/emulator.

4 Load the state file created with the **Debugger Execution Save_State** command using the **Debugger Execution Load_State** command. This will restore memory and processor registers to the state you saved in the debugger/simulator.

Using the Debugger with the Branch Validator

The Hewlett-Packard Branch Validator (BBA) is an interactive tool that helps you rapidly determine which branches of a program have not been taken. With the missed branches identified, you can modify your regression tests to ensure software reliability.

The branch analysis information is collected by C programs that have been compiled using the **bbacpp** preprocessor.

To unload Branch Validator data from program memory

- Select **File→Store→BBA Data ...**. Then choose a file name from the File Selection dialog box.

  Or:

- Using the command line, enter:

  Memory **U**nload_BBA **A**ll

  And press the **< Return>** key.

  This command unloads branch analysis information associated with all absolute files loaded.

  The default file name is *bbadump.data*.

  The BBA preprocessor (-b option) must be used at compile time in order for this information to exist in program memory.

  Once this information has been unloaded, it can be formatted with the BBA report generator, *bbarep* (see the *HP Branch Validator for AxLS C User's Guide*).

Note   The Unload_BBA command is disabled when the debugger option
Demand_Load is *On*. If Demand_Load is *oFF* but the program was loaded
with Demand_Load On, the Memory Unload_BBA command will generate a
BBA file with incomplete information. See the Debugger Option General
command description in this manual for more information on the
Demand_Load option.

5

Viewing Code and Data

How to find and display source code and memory contents.

Using Symbols

This section shows you how to:

- Add a symbol to the symbol table.

- Display symbols.

- Delete a symbol from the symbol table.

To add a symbol to the symbol table

- Using the command line, enter:

Symbol **A**dd

Enter the symbol data type, the symbol name, and optionally the base address and the initial value; then, press the **< Return>** key.

Two types of symbols can be added:

- Program symbols, which are identical to variables defined in a C or assembly program. These symbols must be given base addresses.

- Debugger symbols, which may be used to aid and control the flow of the debugger. These symbols are specified without a base address, and only debugger commands and C expressions in macros can refer to them. They cannot be referenced by the program in target memory.

Example   To add a program symbol named EOF of type int (default) at target memory address 9ff0h and set the memory location to value -1:

Symbol **A**dd EOF **A**ddress 9ff0h **F**ill_Mem –1

To display symbols

- Select **Display→Symbols→**to display information about the symbol in the entry buffer.

  Or:

- Using the command line, enter:

  Symbol **D**isplay **D**efault

  Enter the symbol, module, or function name; then, press the **< Return>** key.

  Symbols and associated information are displayed in the journal window.

  When displaying a symbol in the current module, the debugger looks for the symbol in the current module. If there is no module qualifier, all symbols with the specified name will be displayed, including global symbols and symbols local to the module.

  The wildcard character * may be placed at the end of a symbol name to represent zero or more characters. If used with no symbol name, * is treated the same as \, that is, all symbols are displayed.

Examples   To display the symbol 'updateSys' in the current module:

Symbol **D**isplay **D**efault updateSys

```
 Symbol Display Default updateSys
     @ecs\\updateSys   : Type is High level module.
                          Code section = 00001436 thru 00001C21
```

To display all symbols in module 'updateSys':

Symbol **D**isplay **D**efault updateSys\

```
> Symbol Display Default updateSys\
  Root is: updateSys

     @ecs\\updateSys   : Type is High level module.
                          Code section = 00001436 thru 00001C21
  updateSys\update_state_of_system
                          : Type is Global Function returning void.
```

```
                                        Address = 00001436 thru 00001513
                update_state_of\refresh
                                        : Type is Local int.
                                        Address = Frame + 8
                update_state_of\interval_complete
                                        : Type is Local int.
                                        Address = Frame + 12
        .
        .
```

To display symbols in all modules

- With "\" in the entry buffer, select **Display→All Symbols ()**.

  Or:

- Using the command line, enter:

  Symbol **D**isplay **D**efault \

To delete a symbol from the symbol table

- Using the command line, enter:

  Symbol **R**emove <symb_name>

  Enter the symbol, module, or function name; then, press the **< Return>** key.

  The specified symbols are removed from the symbol table. Only program symbols and user-defined debugger symbols can be deleted from the symbol table.

Examples   To delete symbol 'current_targets' in function 'alter_settings':

Symbol **R**emove alter_settings\current_targets

138

To delete all symbols in module 'updateSys':

Symbol **R**emove updateSys\

To delete all symbols in all modules:

Symbol **R**emove \

Displaying Screens

A debugger screen is what you see in the display area. Each debugger screen may contain one or more debugger windows. A debugger window is a predefined physical area on the screen containing specific debugger information.

The debugger has three predefined screens. Each predefined screen has a corresponding name and number. The predefined screens and their associated names and numbers are listed below:

Screen Name	Screen Number
High-level screen	1
Assembly-level screen	2
Standard I/O screen	3

This section shows you how to:

- Display the high-level screen.

- Display the assembly level screen.

- Switch between the high-level and assembly screens.

- Display the standard I/O screen.

- Display the next screen (activate a screen).

High-Level Screen

The debugger automatically displays the high-level screen when an executable (.x) file containing the C function main() is loaded from the UNIX command line with the db68030 or db68040 command. This screen has nine windows:

- journal
- code
- monitor
- backtrace
- status
- breakpoint
- error
- help

- view

The high-level screen displays high-level source code and stack backtrace information including the calling sequence of functions and function nesting levels.

Assembly-Level Screen

The debugger automatically displays the assembly-level screen when an executable (.x) file is loaded from within the debugger or the executable file does not contain the C source function main(). This screen has ten windows:

- journal
- code
- monitor
- register
- stack
- status
- breakpoint
- error
- help
- view

The assembly-level window displays assembly-level code and processor register and stack information.

Standard I/O Screen

The debugger displays the standard I/O screen when your program requests interactive input from the standard input device (stdin), or directs output to the standard output device (stdout). It may also be displayed using the *F6* function key. This screen has five windows:

- status
- breakpoint
- error
- help
- view

You can also access the standard I/O screen as a window (window No. 20).

The standard I/O window emulates a dumb terminal. It can be moved about the display, but it can be no larger than 24 rows by 80 columns.

To display the high-level screen

- Select **Settings→High Level Debug**.

  Or:

- Using the command line, enter:

  Window **S**creen_On **H**igh_Level

To display the assembly level screen

- Select **Settings→Assembly Level Debug**.

  Or:

- Using the command line, enter:

  Window **S**creen_On **A**ssembly_Level

To switch between the high-level and assembly screens

- Press the **F3** function key.

  Or:

- Using the command line, enter:

Debugger **L**evel

You can also use the Window New and the Window Active commands to display a different screen.

To display the standard I/O screen

- Press the **F6** function key.

  Or:

- Select **Window→Simio**.

  Or:

- Using the command line, enter:

Window **S**creen_On **S**tdio

The standard I/O screen is displayed when your program requests interactive input from the standard input device (keyboard) or when your program writes information to the standard output device.

To display the next screen (activate a screen)

- Press the **F6** function key.

  Or:

- Using the command line, enter:

  Window **S**creen_On **N**ext

  The next higher-numbered screen will be displayed. Either the high-level or the assembly-level screen will be displayed, not both.

  The debugger screens are numbered as follows:

Screen Name	Screen Number
High-level screen	1
Assembly-level screen	2
Standard I/O screen	3
User-defined screens	4-256

Displaying Windows

This section shows you how to:

- Change the active window.

- Select the alternate view of a window.

- Set the cursor position for a window.

A debugger window is a predefined physical area on the screen. The debugger has 18 predefined windows. Each window displays information specific to its associated name (for example, the breakpoint window displays breakpoint information).

Each of the 18 predefined windows has a corresponding name and number. All windows (except the log file and journal file windows, which are files) also have an associated screen number. The following table lists the predefined windows and their associated names and numbers.

Window Name	Window Number	Screen Number
journal (high–level)	1	1
code (high–level)	2	1
monitor (high–level)	3	1
backtrace	4	1
status (high–level)	5	1
journal (assembly–level)	10	2
code (assembly–level)	11	2
monitor (assembly–level)	12	2
register (assembly–level)	13	2
stack	14	2
status (assembly–level)	15	2
standard I/O	20	3
view	24	1, 2, 3
breakpoint	25	1, 2, 3
error	26	1, 2, 3
help	27	1, 2, 3
log file	28	none
journal file	29	none

The code window displays C source code in high-level mode. The code window displays disassembled machine code in assembly-level mode. The C source code that generated the assembly code can be interleaved with the assembly-level code.

When disassembled code is displayed, the address and machine code of a disassembled instruction are displayed on the left side of the window as hexadecimal values. For instructions over 6 bytes in length, bytes 7 through n are replaced by ellipsis (...).

The stack window displays the stack beginning at the memory location pointed to by the debugger stack pointer @SP. This window is available only within the assembly-level screen.

To change the active window

- Use the *command select* mouse button to click on the border of the window you wish to activate.

  Or:

- Select the window you want to make active from the **Window**→menu.

  Or:

- Use the command line to select a window:

  Window **A**ctive *<window>*

  where *<window>* is the name of the window to be made active, and press the **<Return>** key.

  The debugger uses a highlighted or thick border for the active window. The cursor keys, scroll bar, and function key *F4* (select the alternate display) only operate in the active window.

  If you are using a terminal without graphics capabilities, the active window is indicated by single dashes around the border (other windows all have borders of equals signs).

  The window number is displayed in the upper right border of the window.

Examples   To make the high-level backtrace window active:

Window→**Backtrace**

Or:

Window **A**ctive **H**igh_Level **B**acktrace

To make the breakpoint window active:

Window **A**ctive **B**reakpoint

147

To make user window 57 active:

Window **A**ctive **U**ser_Window 57

To select the alternate view of a window

- Click on the border of the active window with the *command select* mouse button.

  Or:

- Press the **F4** function key.

  Or:

- Using the command line, enter:

  Window **T**oggle_View

  Or:

- Using the command line, enter:

  Window **T**oggle_View *<Window>*

  where *< Window>* is the name of the window whose alternate view is to be displayed, and press the **< Return>** key.

  The typical default alternate view of a window is an enlarged view of the window, letting you view more information. Repeating the command switches between the normal view and the alternate view of the active window.

Example   To display the alternate view of the assembly level code window:

Window **T**oggle_View **A**ssembly **C**ode

To view information in the active window

- Use the scroll bar.

  Or:

- Use the cursor control keys.

  Press the **< Up>** or **< Down>** cursor key to move up or down in the window one line at a time.

  Press the **< Page Down>** (**< Next>**) or **< Page Up>** (**< Prev>**) key to move the window one-half of the window length at a time.

  Press the **< Home>** or **< End>** (**< Shift>** **< Home>**) key to position the window at the beginning or end of the information displayed in the window.

  Type **< Ctrl> -F** or **< Ctrl> -G** to shift the contents of the active window to the right or left.

  The following table describes the functions of the cursor control keys in the active window and the command line window.

Key	Description
→	Move to right in data field of command. Highlight token to the right in status line window.
←	Move to left in data field of command. Highlight token to the left in status line window.
↑	Move up one line in window.
↓	Move down one line in window.
Prev	Move up one half window.
Next	Move down one half window.
Home	Move to the top of the active window (except stack window).
End(Shift Home)	Move to bottom of window (except for stack window).
Insert char	Put keyboard in insert mode for editing data field of command.
Delete char	Delete character within data field of command.
Undo	Back tab.

The Home and End (Shift-Home) keys have additional functions when used with the code and stack windows. The following table describes how the Home and End (Shift-Home) keys work in these active windows.

Active Window	Home Key	End Key
Code	Move to top of module	Move to bottom of module
Stack	Move to current stack pointer (SP)	Move to current frame pointer (FP)

To view information in the "More" lists mode

If the "--More--" prompt is printed at the bottom of a window, the debugger is waiting to display more than one screen of information.

- Press the space bar to display the next screen of information.

- Press the **< Return>** key to display the next line.

- Press "Q" to end the "More" display.

    If you try to enter a command while the debugger is displaying the "--More--" prompt, the command will not be executed until the "More" display has ended.

    You can turn the "More" list mode off or on with the **Settings→Debugger Options** dialog box.

    For more information, see your operating system documentation on the **more** command.

To copy window contents to a file

- Select **File→Copy Window→**.

    Or:

- From the command line, enter the following commands:

```
File User_Fopen Append 99 File <file_name>
Expression Fprintf 99, "%w",<window_number>
File Window_Close 99
```

Displaying C Source Code

This section shows you how to:

- Display the C source code.

- Find first occurrence of a string.

- Find next occurrence of a string.

To display C source code

1 Display the high-level screen (see the instructions in the previous "Displaying Screens" section).

2 Display source code at the location in the entry buffer by selecting **Display→Source ()**. Or click on the **Disp Src ()** action key.

Or, using the command line, enter:

Program **D**isplay_Source

Enter the line number or function name of the code you wish to display, and press the **< Return>** key.

Examples

To display the C source code at line number 1:

Program **D**isplay_Source #1

To display the C source code at function *main*:

Program **D**isplay_Source main

To display C++ source code at overloaded C++ function *cfunc*, you can either give the name of the function and select the definition from a menu, or you can specify the definition by entering the argument type:

```
Program Display_Source cfunc (float)
```

To find first occurrence of a string

1 Display the high-level screen (see the instructions in the previous "Displaying Screens" section).

2 Enter the string in the entry buffer.

3 Select **Display→Source Find Fwd ()** or **Display→Source Find Back ().**

Or, using the command line, enter:

```
Program Find_Source Occurrence <Direction>
```

Select either Forward or Backward as the direction, enter the line number or string you wish to find, and press the **< Return>** key.

Example
To find the first occurrence of the string "main":

```
Program Find_Source Occurrence Forward main
```

To find next occurrence of a string

• Select **Display→Source Find Again**.

Or:

• Using the command line, enter:

```
Program Find_Source Next <Direction>
```

Select either Forward or Backward as the direction, and press the **< Return>** key.

Example   To find the next occurrence of a string:

`Program Find_Source Next Forward`

Displaying Disassembled Assembly Code

To display assembly code

- Select **Settings→Assembly Level Debug**.

  Or:

- Using the command line, enter:

  Window **S**creen_On **A**ssembly_Level

  The Code window will show disassembled insructions.

Displaying Program Context

This section shows you how to:

- Set current module and function scope.

- Display current module and function.

- Display debugger status.

- Display register contents.

- Display the function calling chain (stack backtrace).

- Display all local variables of a function at the specified stack (backtrace) level.

To set current module and function scope

- Select **File→Context→Symbols ...**, enter the module or function name in the dialog box, and click on the OK pushbutton.

  Or:

- Using the command line, enter:

  Program **C**ontext **S**et

  Enter the module or function name, and press the **< Return>** key.

  The module and function scope is used by the debugger to uniquely identify symbols. For example, several functions may have local variables with the same names. When you use that variable name without naming the function, the debugger assumes you mean the variable in the current module or function scope.

Examples   To select module "updateSys" as the current module:

Program **C**ontext **S**et updateSys

To select function "updateSys\paint_display" as the current function:

Program **C**ontext **S**et updateSys\paint_display

To set the program context to the module at which the program counter is pointing:

Program **C**ontext **S**et

To display current module and function

- Select **Display→Context**. Click on the Done pushbutton when you wish to stop displaying the information.

  Or:

- Using the command line, enter:

  Program **C**ontext **D**isplay

  The current module, function, and line number are displayed in the journal window.

To display debugger status

- Select **Window→Status**.

  Or:

- Using the command line, enter:

Debugger **E**xecution **D**isplay_Status

The following information is displayed in the view window (which temporarily overlays the top portion of the screen):

- Product version.
- Current working directory.
- Current log file in use.
- Current journal file in use.
- Startup file used.

To display register contents

- Select **Window→Registers**.

Or:

- Select **Modify→Registers**.

Or:

- Using the command line, enter:

Window **A**ctive **A**ssembly **R**egisters

The register window shows the current values of the microprocessor's registers and several debugger variables. The microprocessor register values are labeled with their standard names. The debugger displays all values in hexadecimal format unless otherwise noted.

If you are running just the debugger the Registers window is available only within the assembly-level screen.

The information displayed in the register window varies with different microprocessors. See the "Reserved Symbols" chapter for a list of other registers and pseudoregisters which you can display using expression commands.

Stack Pointer. A7 is the stack pointer (*SP*) for the current mode (user, supervisor, or interrupt). The stack pointers (*USP*, *ISP*, and *MSP*) that are not current are labeled and displayed separately.

Previous Instruction Pseudoregister @pi. The program counter for the previous instruction is indicated by the pseudoregister *@pi*. This value is useful for finding the instruction that jumped to, called, or returned to the current location.

PC and *@pi* are truncated to show only the address bits valid for the current CPU type.

Cycle Count Pseudoregister @cycles. The number of cycles (*@cycles*) used by instructions since this counter was last cleared (by the command **M**emory **R**egister @cycles = 0, for example) is displayed as the pseudoregister *@cycles*.

Exception Handling Keyword @exc. The value of the pseudoregister *@exc* determines how the debugger handles exceptions. See the chapter titled "CPU Simulation" in this manual a detailed explanation of this keyword.

Access Status Pseudoregister @as. The pseudoregister @as is the access status used by the processor's *CALLM* and *RTM* instructions. It is used to determine the legality of as access level change.

CPU Modes and Registers Some registers in the display are shown only for some CPU modes, as shown in the following table.

68030

Status Register:	TTSM.III...XNZVC
Stack Pointers:	USP, ISP, MSP (only two shown at a time)
Control Registers:	SFC, DFC, VBR, CACR, CAAR
Access Status:	as

68040

Status Register:	TTSM.III...XNZVC
Stack Pointers:	USP, ISP, MSP (only two shown at a time)
Control Registers:	SFC, DFC, VBR, CACR
Access Status:	as

The S and M bits of the status register determine which stack pointer appears in $A7$; the other stack pointers appear separately (as USP, MSP, or ISP).

To display the function calling chain (stack backtrace)

- Select **Window→Backtrace**.

Or:

- Using the command line, enter:

Window **A**ctive **H**igh_Level **B**acktrace

The backtrace window displays the function calling chain, from the compiler startup routine to the current function in high-level mode.

This window displays (from left to right):
- Function nesting level.
- Return address to the calling function.
- Frame status character.

- Module containing the function.
- Function name.

Function Nesting Level. The nesting level of the current function is always 0, the calling function always 1, etc.

You may reference the nesting level when setting a breakpoint. For example, to cause the program to execute until it returns to the second nested function, enter the command:

Program **R**un **U**ntil @2

Another way to execute until a stack level is reached is to choose **Run Until Stack Level** in the Backtrace window pop-up menu.

Return Address. The return address field displays the return address of the calling function.

Frame Status Character. One of several characters immediately precedes a function name in the backtrace window. These frame status characters and their descriptions are listed in the table below.

Character	Description
Space	The debugger is executing within a function.
:	The program counter is at a label. Typically, this is an assembly language function point.
*	The function has been entered, but the function prolog has not been executed. The debugger cannot locate local symbols in the function until the prolog has been executed.
?	The frame is questionable. For example, this is displayed when a function has been stripped of debug information.
!	The frame is not valid.
		The debugger is at the start of an interrupt routine.
+	The debugger is executing an interrupt routine.

Module Name. If the function is in a known module, the backtrace window displays the module name. If the program counter is pointing to an address

that is not contained in a module known to the debugger, the module field in the backtrace window displays a string of question marks (???????).

Function Name. If the return address of a function is inside a known function, the debugger displays the function name. If the address is outside of all known functions, the function field in the backtrace window will display *<unknown>*. This is the case with the compiler startup module crt0, because it is assembly code and contains no debug information.

Backtrace Information. Whenever a break occurs in program execution, the backtrace window is updated. When updating the window, the debugger generates backtrace information as described in the following paragraphs. The backtrace window is displayed only in the high-level screen.

Nesting level 0.

Nesting level 0 information is based solely on the current value of the processor's program counter (PC). The address shown at this level is the value of the PC. The module and function shown at this level are selected because the value of the PC falls within their code spaces.

Nesting level 1.

When program execution breaks on an address that has an associated public label (for example, a function entry point), nesting level 1 information is based on the processor SP. The debugger assumes that the SP is pointing to the return address because the label is assumed to be a function entry point and no stack frame has yet been established. With no stack frame available, the return address of the calling function is at the top of the stack. This return address is the address at level 1. The module and function shown are based on this address, that is, the address falls within their code spaces.

When program execution breaks on an address that has no associated public label, nesting level 1 is based on the processor's frame pointer (register A6). In this case, the stack location four bytes above the location pointed to by register A6 contains the return address of the calling function. This address is the address shown at

level 1; the module and function shown are based on this address.

Nesting levels 2 through n.	Nesting levels 2 through n are always based on existing stack frames. A stack frame is generated for each frame on the stack, based on saved frame pointers. Nesting levels are generated until backtracing of the stack encounters a zero frame pointer. This occurs when the stack frame associated with the compiler startup routines crt0/crt1 is encountered.
Functions with no stack frame.	If a function has no stack frame (due to compiling with the -O option), the function that called it does not appear in the backtrace window at any stack level other than levels 0 or 1.
Assembly language functions.	Assembly language functions that set up stack frames appear in the backtrace window, but the information shown is incomplete. Since high level debug information is not present in such handwritten functions, the stack frame appears as a questionable frame. Additionally, there is no function name associated with the frame, i.e., it is displayed as *<unknown>*.

To display all local variables of a function at the specified stack (backtrace) level

- Select **Disp Vars at Stack Level** from the Backtrace window pop-up menu.

  Or:

- Using the command line, enter:

  Program **C**ontext **E**xpand <@stack_level>

Enter the stack level preceded by an *at* sign (@), and press the **< Return>** key.

The values of the parameters passed to the function and the function's local variables are displayed in the Journal window.

Example   To display local variables at stack level 1, position the cursor over "1." in the Backtrace window, and hold the right mouse button. Move the mouse to **Disp Vars at Stack Level** and release the button.

Or, use the command line to enter:

Program **C**ontext **E**xpand @1

To display the address of the C+ + object invoking a member function

- Display the value of the function's **this** pointer.

If the program has stopped at a function, you can find out the address of the object which invoked the function.

The program counter must be *inside* the function; otherwise you may see a "Local variable not alive" error message.

Example   To see the address of the object that invoked the *cfunc* function in class *C*, enter the following string in the entry buffer:

C::cfunc\this

then select **Display→Var/Expression ().**

Using Expressions

This section shows you how to:

- Calculate the value of a C expression.

- Display the value of an expression or variable.

- Monitor variables.

- Discontinue monitoring specified variables.

- Discontinue monitoring all variables.

- Print formatted output to a window.

- Print formatted output to journal windows.

To calculate the value of a C expression

- Enter the expression in the entry buffer, then select **Display→C Expression ()**.

  Or:

- Using the command line, enter:

  Expression **C_Expression**

  Enter the C expression to be calculated, and press the **< Return>** key.

  The value of the C expression is displayed in the journal window.

  If the C expression is an assignment statement, the Expression C_Expression command sets the value of the C variable.

Examples   To calculate the value of 'time':

```
Expression C_Expression time
Result is: data address 000091DC {time_struct}
```

To calculate the value of member 'hours' of structure 'time':

```
Expression C_Expression time->hours
Result is: 4   0x04
```

To assign the value 1 to 'system_is_running':

```
Expression C_Expression system_is_running = 1
Result is: 1   0x01
```

To display the value of an expression or variable

- Use the mouse to copy the expression or variable into the entry buffer, then select **Display→Var/Expression ()**.

  Or:

- Using the command line, enter:

```
Expression Display_Value
```

Enter the expression or variable whose value is to be displayed, and press the **<Return>** key.

The value of the expression or variable is displayed in the journal window.

The contents of an item, such as an array, are displayed instead of the C value of the item which is its address.

Examples

To display the value of the variable 'system_is_running':

```
Expression Display_Value system_is_running
 01h
```

To display the address of the variable 'system_is_running':

Expression **D**isplay_Value &system_is_running
 000091F0

To display the address of the C structure 'time':

Expression **D**isplay_Value time
000091DC

To display the values of the members of structure 'time':

Expression **D**isplay_Value *time
hours   4
minutes   0
seconds   20

To display the name of the current program module:

Expression **D**isplay_Value @module

To display the name of the current program function:

Expression **D**isplay_Value @function

To display members of a structure

1 Copy the name of the structure into the entry buffer.

2 Add an asterisk (*) in front of the name of the structure.

3 Select **Display→Var/Expression ()**.

If you are using the command line, use the **E**xpression **D**isplay_Value
command.

Example

To display the names of the members of structure *astruct*, use the following expression in the entry buffer:

```
*astruct
```

The * operator tells the debugger to display the members of the structure, rather than the address of the structure.

To display the members of a C++ class

- Using the command line, enter

Symbol **D**isplay **O**ptions **S**earch_all **E**nd_Options
<class_name>\

This will display the type, size, protection, and overloading of each member of *class_name*.

Example

To display the members of class *C*, enter:

Symbol **D**isplay **O**ptions **S**earch_all **E**nd_Options C\

To display the values of all members of a C++ object

- Enter the name of the C++ object in the entry buffer and select
Display→Var/Expression ().

Or:

- Using the command line, enter:

  Expression **D**isplay_Value *<object>*

  Remember, you are displaying the values in an *object*, so you need to run the program to the point where the object is created. To display the members of a class, see "To display the members of a C++ class."

Example   To display the members of object *cobj* in class *C*, enter "cobj" in the entry buffer and select **Display→Var/Expression ()**.

To monitor variables

- Enter the variable to be monitored in the entry buffer and click on the **Monitor ()** action key.

  Or:

- Enter the variable to be monitored in the entry buffer and select **Display→Monitor ()**.

  Or:

- Using the command line, enter:

  Expression **M**onitor **V**alue

  Enter the variable to be monitored, and press the **< Return>** key.

  The monitor window displays monitored variable expressions. This window can be displayed in both the high-level and assembly-level screens.

Variables in the monitor window are updated each time the debugger stops executing the program. (The program is not considered to be "stopped" when a breakpoint with an attached macro is encountered.)

Example   To monitor the value of variable 'current_temp':

Expression **M**onitor **V**alue current_temp

To monitor the value of a register

- Monitor a register just as you would a variable.

Example   To monitor the value of register D2, enter "@D2" in the entry buffer and select **Display**→**Monitor ()**.

Or, using the command line, enter

Expression **M**onitor **V**alue @D2.

To discontinue monitoring specified variables

- Select **Delete Variable** in the Monitor window pop-up menu.

  Or:

- Using the command line, enter:

  Expression **M**onitor **D**elete

  Enter the number of the variable (shown in the monitor window) that should no longer be monitored, and press the **< Return>** key.

The variable is removed from the monitor window.

Example   To stop monitoring variable 2 in the monitor window:

`Expression Monitor Delete 2`

To discontinue monitoring all variables

- Select **Delete All Variables** in the Monitor window pop-up menu.

  Or:

- Using the command line, enter:

  `Expression Monitor Clear_All`

  All variables are removed from the monitor window.

To display C+ + inheritance relationships

- Enter the name of a C+ + class in the entry buffer, then select
  Display→**Symbols**→**Browse C+ + Class ()**.

  Or:

- Using the command line, enter:

  `Symbol Browse`

  Enter the name of the C+ + class to be displayed, and press the **< Return>**
  key.

To print formatted output to a window

- Using the command line, enter:

Expression **F**printf

Enter the number of the user-defined window, the format string (enclosed in quotes), and the arguments; then, press the **< Return>** key.

The formatted output is written to the user-defined window. This command is similar to the C fprintf function.

The debugger associates the log file window (window no. 28) with a log (.com) file so that you can write output to that window using the Expression Fprintf command. This window is not displayed. It is used only for writing to a command file.

The debugger associates the journal file window (window no. 29) with a journal file so that it can write journal window output to the journal (.jou) file. Additional output may be written to the journal file by writing to window 29.

Examples

To print the value of *var* to user window 57 as a single character:

Expression **F**printf 57,"%c",var

To print the string in double quotes to user window 57 followed by the floating point value of 'temperature' with a precision of 2:

Expression **F**printf 57,"The value of 'temperature' is:
%.2f \n",temperature

To print formatted output to journal windows

- Using the command line, enter:

Expression **P**rintf

Enter the format string (enclosed in quotes) and the arguments; then, press
the **< Return>** key.

The formatted output is written to the journal window. This command is
similar to the C printf function.

Examples

To print the value of *var* to the journal window as a single character:

```
Expression Printf "%c",var
```

To print the string in double quotes to the journal window followed by the
floating point value of 'temperature' with a precision of 2:

```
Expression Printf "The value of 'temperature' is: %.2f
\n",temperature
```

Viewing Memory Contents

This sections explains how to to view, compare, and search blocks of memory.

To compare two blocks of memory

- Using the command line, enter:

Memory **B**lock_Operation **M**atch *<Mismatch_Operation>*

Select either Repeat_On_Mismatch or Stop_On_Mismatch to specify what happens when a mismatch is found, enter the address range to be compared and the starting address of the range that it is compared to; then, press the **< Return>** key.

Example

To compare the block of memory starting at address 1000h and ending at address 10ffh with a block of the same size beginning at address 5000h and stop when a difference is found:

Memory **B**lock_Operation **M**atch **S**top_On_Mismatch
1000h..10ffh,5000h

To search a memory block for a value

- Using the command line, enter:

Memory **B**lock_Operation **S**earch *<Size> <Until>*

Select either Byte, Word, or Long as the size of the memory locations, select either Once or Repeatedly to specify when the search should stop, enter the

address range and the value that is to be searched for, and press the
< **Return**> key.

Example

To search for the expression 'gh' in the memory range from address 1000h
through address 10ffh and stop when the expression is found or address 10ffh
is reached:

Memory **B**lock_Operation **S**earch **W**ord **O**nce
1000h..+0xff = 'gh'

To examine a memory area for invalid values

- Using the command line, enter:

Memory **B**lock_Operation **T**est *<Size> <Until>*

Select either Byte, Word, or Long as the size of the memory locations, select
either Once or Repeatedly to specify when the search should stop, enter the
address range and the value that should be found in the range, and press the
< **Return**> key.

Example

To test for the expression 'gh' in the memory range from address 1000h
through address 10ffh and stop when a word not matching the expression is
found:

Memory **B**lock_Operation **T**est **W**ord **O**nce 1000h..+0xff =
'gh'

To display memory contents

- Select **Display→Memory→**.

  Or:

- Using the command line, enter:

  Memory **D**isplay *<Format>*

  Select either Mnemonic, Byte, Word, or Long as the format in which memory contents are to be displayed.

  If you are using the command line, enter the starting address or the address range of the memory whose contents are to be displayed, and press the **< Return>** key.

Examples

To display disassembled memory in the code window starting at the symbol '_emeg_shutdown' (this command works only in assembly-level mode):

Memory **D**isplay **M**nemonic _emeg_shutdown

To display memory in byte format in the journal window starting at the symbol 'current_humid':

Memory **D**isplay **B**yte current_humid

Using Simulated I/O

Simulated I/O (SIMIO) lets programs use the UNIX file system, run UNIX commands, and use the keyboard and display for input and output.

Your programs can use SIMIO by means of the I/O libraries and environment dependent routines provided with the HP B1478 or HP B1463 C Cross Compiler. Your programs use the library functions when they open, close, read, or write to files, etc. These simulated I/O functions are identical in both the debugger/emulator and debugger/simulator to let you write programs that will function correctly in both environments. Refer to the "Environment Dependent Routines" chapter of your compiler manual for information on using the C SIMIO libraries.

Your programs can also use SIMIO by means of user-written assembly code. If you are developing programs that use SIMIO from assembly code, refer to the *Simulated I/O User's Guide* for a complete description of SIMIO protocol.

This chapter shows you how to:

- Enable simulated I/O.

- Disable simulated I/O.

- Set the keyboard I/O mode.

- Redirect I/O.

- Check resource usage.

- Increase file resources.

- Display the simulated I/O system report.

How Simulated I/O Works

Communication between your program running in the simulated system and the SIMIO process takes place through contiguous single-byte length memory locations. The first memory location is called the Control Address (CA). The Control Address and the memory locations that follow it are called the CA buffer.

Control Address buffers are less than or equal to 260 bytes in size. A maximum of 256 bytes of information can be transferred between the debugger and the host system at one time. Some simulated I/O commands require four additional bytes for command parameters.

The debugger supports only one Control Address (CA) for doing SIMIO operations. This buffer is named *systemio_buf* in the HP B1478 or HP B1463 C I/O libraries. Assembly code users who want to use SIMIO with the debugger must label their Control Address as *_systemio_buf* (the compiler prefixes symbols with an underscore).

Communication between a program and the simulated I/O process is a series of requests by the program and responses by the SIMIO process:

The program places a SIMIO command in the CA buffer and then waits for a return code to be placed in the first byte of the CA.

The SIMIO process polls the CA buffer memory. When it finds a command, the SIMIO process executes the command. When the SIMIO process completes the command, the first byte of the CA buffer is changed to the command return code.

In the debugger/simulator, the debugger stops executing the program while a SIMIO command is being processed. This causes no timing related problems because all timing is relative to the system simulator. Once the SIMIO request has finished processing, the program continues executing. This behavior has the effect of producing an I/O system that takes 0 cycles to complete. When simulation stops, the simulated cycle counting also stops. When simulation resumes, the I/O request is complete and the time reference of cycle counting continues.

Simulated I/O Connections

The SIMIO system supports three types of I/O connections. These are:

- Keyboard and display.
- UNIX files.
- UNIX processes.

Display and Keyboard

The debugger provides a window named stdio which functions as the normal display output for target programs. The screen can be opened for output from target programs via SIMIO with the special name */dev/simio/display*. This

name appears to be an UNIX file name. However, it is really a name reserved by the debugger to indicate the internal screen. The keyboard is accessed by the special name *ic/dev/simio/keyboard.*

UNIX Files

UNIX files are accessed by their names from the target program running in the debugger in the same way they are accessed by host software. The file operations of open, close, read, write, and seek are supported by the SIMIO protocol. When opening a stream on an UNIX file, SIMIO supports the same control parameters for file creation and blocking I/O that are available to host programs.

UNIX Processes

UNIX processes can be run as subprocesses to the debugger with their input and output directed to the user program. Subprocesses are controlled from the user program by a Process Identification number (PID). This lets the user program check specific subprocesses, send them signals, or stop them. This subprocess facility allows user programs to take advantage of the powerful software and execution environment of the host UNIX system. Host programs can be used to process data for a debugger user program or to simulate portions of the software that are not available in the user program.

Because simulated I/O lets the debugger execute UNIX commands, the debugger can communicate with other host system I/O devices, such as printers, plotters, modems, etc.

For more information on using UNIX processes, refer to the description of the *exec_cmd( )* function in the "Environment Dependent Routines" chapter of the *68030 C Cross Compiler Reference* or *68040 C Cross Compiler Reference* manual.

Special Simulated I/O Symbols

User Program Symbols

The following symbols are user program symbols that are used by the SIMIO system to process the simulated I/O protocol:

systemio_buf This symbol indicates the start of the Control Address buffer.

_display_message When execution reaches this address, the program has completed and the compiler-library-generated status message is available. The SIMIO system uses this address as a point to stop execution.

Simulated I/O Reserved Symbols

The following names are reserved by the SIMIO system and cannot be used for your file names. The SIMIO system recognizes these names and uses special processing to direct the I/O to the proper location:

stdin This name will be replaced by the name stored in the stdin_name. This name is set via the Stdio_Redirect command.

stdout This name will be replaced by the name stored in the stdout_name. This name is set via the Stdio_Redirect command.

stderr This name will be replaced by the name stored in the stderr_name. This name is set via the Stdio_Redirect command.

/dev/simio/keyboard This name refers to the keyboard while the product is running interactively.

/dev/simio/display This name refers to the stdio display window while the product is running interactively.

To enable simulated I/O

- Using the command line, enter:

```
Debugger Execution IO_System Enable
```

When SIMIO is enabled, polling for simio command begins. In the debugger/simulator, the debugger detects writes to the SIMIO control address. SIMIO behavior in the debugger is identical to that described in the *Simulated I/O User's Guide*.

To disable simulated I/O

• Using the command line, enter:

Debugger **E**xecution **I**O_System **D**isable

To set the keyboard I/O mode to cooked

• Using the command line, enter:

Debugger **E**xecution **I**O_System **M**ode **C**ooked

In the Cooked mode, the keyboard input is processed. This lets you type and
then edit the line to correct errors. When the final line is composed, press the
< Return> key to enter the line. Once the line is entered, it is read by the
target program. Only the characters from the final line and the carriage return
character are passed as input. If program execution is interrupted by entering
< Ctrl> -C before the line is entered, the characters on the input line are lost.

See also   "To set the keyboard I/O mode to raw"

To set the keyboard I/O mode to raw

• Using the command line, enter:

Debugger **E**xecution **I**O_System **M**ode **R**aw

In the Raw mode, each character you type is sent directly to the target
program that is reading from the keyboard. Characters are not echoed as they
are typed. Any input editing, such as backspace, must be handled by the target

program. The only special character that cannot be sent to the target program is < **Ctrl> -C** which is used to interrupt the debugger's execution of the program.

See also   "To set the keyboard I/O mode to cooked"

To control blocking of reads

- Set the *O_NDELAY* flag in the *startup( )* routine.

  The flag *O_NDELAY* is passed to the function *open( )* to control whether or not reads from the keyboard will block *waiting for characters.* This flag can only be set when opening the stream; it may not be changed after the file stream is open. This flag can be set in the compiler-supplied routine *startup( ).* This routine opens streams stdin, stdout, and stderr.

See also   The chapter titled "Environment Dependent Routines" in the *68030 C Cross Compiler Reference* or *68040 C Cross Compiler Reference* manual.

To interpret keyboard reads as EOF

- Using the command line, enter:

  Debugger **E**xecution **I**O_System **K**eyboard_EOF

  This causes the debugger to interpret any further keyboard reads as being at the end of file.

  In cooked mode, pressing < **Ctrl> -D** is equivalent to entering the *Debugger Execution IO_System Keyboard_EOF* command.

To redirect I/O

To redirect the three I/O streams and to reset your program to the startup address, perform the following steps.

1 Redirect the three I/O streams by changing the translation names for the stdio streams. Using the command line, enter:

```
Debugger Execution IO_System Stdio_Redirect
<"stdin_name","stdout_name","stderr_name">
```

Enter the new names for standard input, standard output, and standard error; then, press the **< Return>** key.

2 Reset the program counter to the startup address. Select **Execution→Set PC to Transfer**. Or, using the command line, enter:

```
Program Pc_Reset
```

When the target program starts execution from the normal compiler startup address, the standard C startup libraries open the following three I/O streams:

- stdin
- stdout
- stderr

The debugger uses an internal table to determine where the streams should be opened. Each of the names (stdin, stdout, and stderr) has an associated translation name:

- stdin_name
- stdout_name
- stderr_name

The translation name contains the name of a file to use when the target requests opening of any of these stdio streams. By default, stdin_name contains */dev/simio/keyboard* (the keyboard), and translations stdout_name and stderr_name contain */dev/simio/display* (the standard I/O (stdio) screen).

These translations are used only when opening the streams. They cannot be used to redirect the streams after they have been opened. The target program

must be rerun from the startup address to allow the stdio streams to be
reopened if the translations have been changed.

Examples
To redirect the standard input file to the keyboard, the standard output file to
the display, and the standard error file to file '/users/project/errorfile':

```
Debugger Execution IO_System Stdio_Redirect
"/dev/simio/keyboard","/dev/simio/display",
"/users/project/errorfile"
```

```
Program Pc_Reset
```

To redirect the standard input file to 'temp.dat', the standard output file to
'cmdout.dat', and the standard error file to file 'errorlog.err':

```
Debugger Execution IO_System Stdio_Redirect
"temp.dat","cmdout.dat","errorlog.err"
```

```
Program Pc_Reset
```

To check resource usage

- Using the command line, enter:

```
Debugger Execution IO_System Report
```

The command displays the simulated I/O status, keyboard mode, and the
translation names used for stdin, stdout, and stderr.

The SIMIO system has the following default resource limitations:

- 40 open files
- 4 subprocesses

To increase I/O file resources

1 Change to directory *68030* or *68040* in path */usr/hp64000/include*, as appropriate for your processor, using the **cd** command.

2 Change the value of macro *FOPEN_MAX* from 12 to the new maximum number of open files (the limit is 40) in file *stdio.h* using an editor on your system.

3 Change to the appropriate environment directory under */usr/hp64000/env/*, then change to the *src* subdirectory.

4 Recompile file *startup.c*. Type:

```
cc68030 -Ouc startup.c (for 68030)
```

or

```
cc68040 -Ouc startup.c (for 68040)
```

depending on which processor you are simulating.

5 Add *startup.o* to the environment library using the command:

```
ar68030 -r startup ../env.a
```

or

```
ar68040 -r startup ../env.a
```

You can increase the simulated I/O file limit by modifying the startup code for your compiler. The code must be modified from the UNIX shell. The maximum number of open SIMIO files descriptors can be increased to 40.

Caution   Compiler startup files compiled with the modified *stdio.h* header file will run
                     only in the debugger environment. Emulators which do not have the debugger
                     interface do not support the increased number of open SIMIO file descriptors.
                     Calls to the SIMIO function open() will fail in this environment if 12 file
                     descriptors have already been allocated.

6

Editing Code and Data

How to use the debugger to make permanent or temporary changes to source code, memory contents, and registers.

Editing Files

The graphical interface gives you a number of context-dependent and context-independent editing commands. From several screens, you can bring up the source file that contains the source line or symbol you are viewing in the display.

The interface will choose the "vi" editor as its default editor, unless you specify another editor by setting an X resource. Refer to the chapter "Configuring the Debugger" for more information about setting this resource.

Remember to re-compile

When you use the editor to change a source code file, you will need to re-compile the source file. You can recompile with a click of the mouse if you define the **Make** action key to compile the target program.

To edit source code from the Code window

- Place the mouse pointer over the line you want to edit. Select **Edit source** from the Code window pop-up menu.

The debugger will start the editor in a new X window. The cursor in the editor window will be on the same line of code as the mouse pointer in the Code window.

After editing the file, you quit the edit session by the standard method for the editor used.

You will need to re-compile the source file. You can recompile with a click of the mouse if you define the **Make** action key to compile the target program.

To edit an arbitrary file

1 Select **File→Edit→File**.

2 Using the file selection dialog box, enter the name of the file you wish to edit; then, click on the OK pushbutton.

After editing the file, you quit the edit session by the standard method for the editor used.

To edit a file based on an address in the entry buffer

1 Place an address reference (either absolute or symbolic) in the entry buffer.

2 Select **File→Edit→At () Location**.

The interface determines which source file contains the code generated for the address in the entry buffer and opens an edit session on the file.

To edit a file based on the current program counter

• Select **File→Edit→At PC Location**.

The interface determines which source file generated the address currently in the program counter and opens an edit session on that source file. The interface will issue an error if it cannot find a source file for the address in the PC.

Patching Source Code

When you change source code by editing the C source file, you need to
re-compile.

The debugger provides several ways to patch your program without
re-compiling:

- Change a variable's value using a C expression.

- Apply a patch using a breakpoint macro.

To change a variable using a C expression

1 Enter a C expression in the entry buffer.

A good way to do this is to highlight an expression from your source code
using the left mouse button. When you release the button, the expression will
appear in the entry buffer. Now edit the expression to have the desired value.

2 Click on the **C Expr ()** action key. Or select **Display→C Expression** from the
menu bar.

The value of the variable will be changed until the program modifies it. You
can continuously monitor the variable's value if you display it in the Monitor
window (use the **Monitor ()** action key or the **E**xpression **M**onitor **V**alue
command).

Or:

- Using the command line enter:

```
Expression C_Expression <expression>
```

To patch a line of code using a macro

1 Set a breakpoint at the line you wish to patch.

An easy way to set the breakpoint is to click the right mouse button on the line in the Code window.

2 Attach a macro to the breakpoint.

Choose **Attach Macro ...** from the Code window pop-up menu.

3 Write a macro to patch the code.

In the Macro Operations dialog box, enter the name of a new macro and click on the **Edit** button.

The macro may contain any number of C expressions and debugger commands.

The last two lines of the macro should be:

```
$Modify Register @PC = #next_line$;
return(1)
```

where *next_line* is the number of the line after the breakpoint. Return 0 instead of 1 if you want the debugger to stop after the macro is executed.

Exit the editor as usual, then click on the **Attach** button in the Macro Operations dialog box.

Now whenever the breakpoint line is encountered, the debugger will execute the macro before the patched line is executed. The macro will execute your patch code, then skip to the next line.

To patch C source code by inserting lines

1 Define a macro containing the inserted statements. The macro must provide a
return value of 1 (true) in order for the program to continue after the macro is
executed.

2 Set a breakpoint on the C line following the point where the insertion should
occur and attach the macro to the breakpoint.

3 Start your program.

The program will run until the breakpoint is encountered. The debugger will
then interpret and execute the C statements in the macro, and continue
executing the program.

To patch C source code by deleting lines

1 Write a macro that sets the program counter to point to the first line of code
beyond the lines of code that you want to delete. The macro must provide a
return value of 1 (true) in order for the program to continue after the macro is
executed.

2 Set a breakpoint on the first line to be deleted and specify the macro with that
breakpoint.

3 Start your program.

The program will run until the breakpoint is encountered. The macro will
then set the program counter to the line specified in the macro. Program
execution will then continue, skipping the program lines between the
breakpoint and line specified in the macro.

Example   Consider the following code:

```
25      count = 5;
26      for (i=0; i < MAXNUM; i++)
27      {
28            array[i]=1;
29            count=count+2;
30            k=count*i;
31      }
```

To delete lines 29 and 30, and insert a new line incrementing *count* by one, you could write the following macro:

```
Debugger Macro Add patch_29()
{
    count++;
    $Expression C_Expression @PC = #31$;
    return(1);
}
.
```

To execute the code patch, enter the command:

```
Breakpt Instr #29;patch_29()
```

and run your program.

Editing Memory Contents

This section shows you how to:

- Change memory location values.

- Copy a block of memory.

- Fill a block of memory with values.

- Compare two blocks of memory.

- Change the contents of a register.

- Unload BBA data from program memory.

To change the value of one memory location

1 Select **Modify→Memory**.

Or, using the command line, enter:

Memory **A**ssign *<Size>*

2 Using the command line, select either Byte, Word, or Long as the size of the memory location, and enter the expression that assigns a value to an address, and press the **< Return>** key.

To change the values of a block of memory interactively

1 Select **Modify→Memory**.

Or, using the command line, enter:

Memory **A**ssign *<Size>*

2 Using the command line, select either Byte, Word, or Long as the size of the memory location, enter the address of the beginning of the block, and press the **< Return>** key.

This starts the interactive memory modification mode.

3 Enter the value for the location displayed in the Journal window and press the **< Return>** key.

4 To exit this mode, press the **< Return>** key without entering a value.

Example
To display the contents of memory location 1000h and allow interactive modification of memory contents:

Memory **A**ssign **B**yte 1000h
00001000 = 0x48   72:

To copy a block of memory

1 Using the command line, enter:

Memory **B**lock_Operation **C**opy

2 Enter the address range of the memory to be copied, followed by a comma.

3 Enter the starting address of the destination and press the **< Return>** key.

Example
To copy the block of memory starting at address 1000h and ending at address 10ffh to a block of the same size starting at address 5000h:

Memory **B**lock_Operation **C**opy 1000h..10ffh,5000h

195

To fill a block of memory with values

- Using the command line, enter:

  Memory **B**lock_Operation **F**ill *<Size>*

  Select either Byte, Word, or Long as the size of the memory locations, enter
  the expression that assigns a value to locations in a range of addresses, and
  press the **< Return>** key.

Example

To fill memory locations 1000h through 1007h with the long pattern
61626364, 65666768:

Memory **B**lock_Operation **F**ill **L**ong 0x1000..+7='abcdefgh'

To compare two blocks of memory

- Using the command line, enter:

  Memory **B**lock_Operation **M**atch *<Mismatch_Operation>*

  Select either Repeat_On_Mismatch or Stop_On_Mismatch to specify what
  happens when a mismatch is found, enter the address range to be compared
  and the starting address of the range that it is compared to; then, press the
  < Return> key.

Example

To compare the block of memory starting at address 1000h and ending at
address 10ffh with a block of the same size beginning at address 5000h and
stop when a difference is found:

Memory **B**lock_Operation **M**atch **S**top_On_Mismatch
1000h..10ffh,5000h

To re-initialize all program variables

- Select **File→Load→Program Only ...**, then use the File Selection dialog box to select the absolute file.

  Or:

- Using the command line, enter:

  Program **L**oad **N**ew **C**ode_only **N**o_Pc_Set <absolute_name>

  Enter the name of the absolute file whose code is to be loaded, and press the **< Return>** key.

  The code will be loaded without loading symbols or resetting the PC.

  The debugger does not save the initial values of variables. The only way to restore the initial values is to re-load the program. After re-loading the program, you may need to restore some debugger settings; for example, you might need to re-specify variables for the Monitor window.

To change the contents of a register

- Select **Modify→Register**. This will display the Modify Register dialog box.

  Or:

- Using the command line, enter:

  Memory **R**egister

  On the command line, enter the name of the register and the value to which the register's contents should be changed, and press the **< Return>** key.

  Registers may also be modified by using "@*register*" in a C_expression.

Example

To modify register values interactively:

Memory **R**egister

The program counter (PC) is displayed in the journal window. You can modify the PC by entering a value (10a4h in this example) at the cursor prompt and pressing **< Return>** . The PC will be modified, and the next register will be displayed:

```
@pc     = 0x000010B8     4280: 10a4h
@sp     = 0x00015DB4     89524:
```

Press **< Return>** without entering a value to exit this mode.

To set the value of register @d1 to 44h:

Memory **R**egister @d1=0x44

To interactively change the value of register @d1:

Memory **R**egister @d1

7

Using Macros and Command Files

How to use macros and command files to make debugging easier.

The debugger provides several ways for you to simplify tasks that you do often.

- **Macros** are C-like functions. You can call macros individually, attach them to breakpoints, or automatically execute them with each program step. Macros are especially useful for temporarily patching C code.

- **Command files** contain series of debugger commands. The debugger can read a command file and execute the commands found there as if they were entered directly into the interface command line. Command files are useful for setting up the debugger, for executing a program to a certain point, and for automated testing.

- **Action keys** are shortcut definitions or "hotkeys" which allow you to add new commands to the graphical interface. Action keys are useful for simplifying frequently-used commands, for making the debugger easier to use for co-workers who do not frequently use a debugger, and for making the debugger into a framework for demos and tutorials.

Using Macros

A macro is a C-like function consisting of debugger commands and C statements and expressions.

Macros are most often used to:

- Patch C source code.

  Often, bugs found with the debugger can be temporarily patched with C source statements in macros. You do not have to exit the debugger, edit the source code, recompile and link, and then reenter the debugger. Instead, you can make a temporary patch by using breakpoint macros.

- Return values to expressions.
- Create conditional breakpoints.
- Execute commands after each program step command.
- Execute a set of commands.

Macros can:

- Have input parameters (macro arguments).
- Define macro local variables.
- Contain C statements and expressions.
- Refer to target variables and registers.
- Refer to user-defined variables.
- Have return values.
- Call other macros.
- Can be used in expressions (if they return values).
- Execute most debugger commands.

Macros cannot:

- Define global variables.
- Define static variables.
- Be recursive.
- Define other macros.
- Contain the conditional operator (expression ? expression : expression).

Macros can be called:

- By specifying the macro name in an expression.
- By calling the macro from within another macro.
- With the Debugger Macro Call command.
- With the Breakpt command.
- With the Program Step With_Macro command.

This section shows you how to:

- Define a macro.

- Call a macro.

- Stop a macro.

- Display macro source code.

- Patch C source code by using macros.

- Delete a macro.

Saving and re-using macros

You can define and save macros interactively during a debugger session.

Macro limits

The maximum number of characters that can be entered on a line in a macro definition is 255. When entering macro interactively, the debugger does not respond to more than 78 characters on a line. When reading a command file, the debugger stops recognizing characters after 255 characters have been read on a line.

The maximum number of lines allowed in a macro depends on the complexity of the lines. Macros with too many lines (too complex) will fail. Error 92 "*Not enough memory for expression*" will be displayed.

A maximum of 40 parameters may be specified in a macro definition.

Once you have defined a macro, you can use it any time during the debugging session, whenever that set of commands or statements is needed.

Caution   The pseudoregister @cycles is not implemented in the emulation environment. Macros written for execution in both the simulation and emulation environments must not refer to @*cycles*.

Macro comments

Macros support C comments (introduced by the characters */* and terminated with the characters **/*).

Macro arguments

You can use formal macro arguments throughout the macro definition. They are replaced at execution time by the actual parameters present in the macro call. The actual parameter is coerced to the corresponding formal parameter type. If coercion is not possible, an error occurs.

You must list the macro's arguments (if any), along with their associated types, when you define the macro. For example, the following listing defines arguments for the built-in macro strcpy():

```
Debugger Macro Add int strcpy(target, source)
char    *target;
char    *source;
```

Macro variables

Variables that are local to the macro may be created within the macro. The definition of local variables follows the rules of C, with the exception that you cannot define variables with initializers. Variables may be defined to have a simple type, or they may be of type array or pointer. Derived types (such as structures and unions), enumerated types, and typedefs are not legal within macros.

The macro processor does not recognize the C keywords extern, auto, static, and register. The macro processor reports an error if these C keywords are used. Static variables are not scoped within a macro. However, symbols created with the Symbol Add command (debugger symbols) are globally scoped, and can be accessed from within a macro. Register variables (such as @PC) may also be accessed from within a macro.

Target program symbols can also be accessed from within a macro. Variables which are globally scoped within the target program can be accessed directly. File static, function static, and automatic variables can be accessed directly only if the current context of the debugger is the module or function in which they are scoped. Otherwise, they require a module or function name as a qualifier before they can be accessed. For example, assume the following definition exists in your target program, in a file called init.c:

```
...
static int i;              /* file static */
...
foo(int parm)
{
    static int j;          /* function static */
    auto  int k;           /* function local  */
...
```

```
}
...
```

If a macro is executed while the PC is pointing into the function foo(), variables i, j, and k can be directly accessed. If this is not the case, i must be accessed with a module qualifier, such as init\i. The function static j must be accessed as init\foo\j. The automatic k can be accessed as init\foo\k if the stack frame for foo() is alive.

Macro control flow statements

Macros support the following C control flow statements:

- If-else
- While and For
- Do-while
- Break and Continue in While, For, and Do statements.

However, macros cannot contain conditional expressions of the form:

```
<expression>?<expression>:<expression>
```

Macro return values

Macros support the C "return" statement for returning values.

If a breakpoint macro returns a nonzero value, program execution continues. If it returns a zero value, program execution is halted. If a macro does not return a value, it should be declared as void when it is defined.

Macros containing debugger commands

You can create macros that contain only a sequence of debugger commands. Macros containing only debugger commands are similar to command files. You can use these macros to set up complex initialization conditions.

You cannot use the following commands in macros:

- Program Run
- Program Step
- Program Step Over
- Debugger Host_Shell
- Debugger Macro Add
- Symbol Add
- Symbol Remove

- File Command
- Debugger Quit

To display the Macro Operations dialog box

- Select **Breakpoints**→**Edit/Call Macro** from the menu bar.

  Or:

- Select **Attach Macro** from the Code window pop-up menu.

  The Macro Operations dialog box allows you to call predefined macros, edit or
  call existing user-defined macros, and create new macros.

To define a new macro interactively using the graphical interface

1 Display the Macro Operations dialog box.

2 Move the mouse pointer to the Selected Macro entry area.

3 Type **< Ctrl> -U** to clear the Selected Macro entry area, then type the name of
  the macro you wish to create.

  When you press **< Return>** or click on the **Edit** button, the debugger will
  display an editor window.

  A "skeleton" macro will appear in the editor window.

4 Edit the macro definition.

When you exit the editor, save the macro under the default name. If you save it under a different name, the macro may be lost.

See Also   See "To use an existing macro as a template for a new macro" if you want to use an existing macro as the basis for a new macro.

Example   To create an macro called "test_macro", select **Breakpoints→Edit/Call Macro** and enter "test_macro" in the Selected Macro area. Now press **< Return>** or click on the **Edit** button. Edit the macro in the editor window. If you are using the **vi** editor, exit using the "ZZ" command. The new macro should now appear at the end of the Defined Macros list.

To use an existing macro as a template for a new macro

1 Display the Macro Operations dialog box.

2 In the dialog box, select the macro you wish to use as a template.

3 Click on the **Edit** button.

4 In the editor, change the name of the macro.

Now you may edit the parameters and body of the macro.

When you exit the editor, the macro will be saved under the new name. The original macro will not be changed.

To define a macro interactively using the command line

1 Enter the Debugger Macro Add command followed by an optional return type, and then a macro name. The macro name must be followed by parentheses; the parentheses can optionally enclose macro arguments separated by commas.

```
Debugger Macro Add [<type>] <name> ([parm,parm,...])
[<parm_types>;]
```

2 Enter the text of the macro body.

```
{
    [[<C_expr>|<C_stmt>|$<debugger_cmd>$];...]
}
```

3 End the macro definition with a period as the first and only character on a line. The macro is checked for syntax errors as soon as the period is encountered. If an error is found within a macro, the macro definition is not saved. The macro must be completely reentered.

Your completed macro definition should have the following syntax:

```
Debugger Macro Add [<type>] <name> ([parm,parm,...])
[<parm_types>;]
{
    [[<C_expr>|<C_stmt>|$<debugger_cmd>$];...]
}
.
```

Debugger commands can be embedded in the macro by enclosing the commands between *$* characters. For example,

```
$Expression C_Expression @PC = #31$;
```

No standard C library functions are available from within a macro. However, there are built-in macros available in the debugger that perform similar functions (refer to the "Predefined Macros" chapter).

To define a macro outside the debugger

1 Using a text editor on your host system, define the macro.

2 Save the macro definition in a command file (< filename> .com).

3 Start the debugger.

4 Load the command file into the debugger using the File Command command.

As the macro is loaded into the debugger, the macro processor parses the macro, looking for syntax errors. If the macro definition contains no errors, it is loaded into the debugger's symbol table.

If an error is detected, the macro processor reports the error and quits loading the command file. The macro remains undefined.

The number of macros that you can define is limited only by the available memory on your host computer system.

To edit an existing macro

• If you want to edit a macro attached to a breakpoint, select **Edit Attached Macro** from the Code window pop-up menu.

Or:

1 Display the Macro Operations dialog box.

2 Select the macro you want to edit.

3 Click on the Edit button.

Remember to save the macro under the default file name when you leave the editor (use the "ZZ" or ":wq!" command in **vi**).

To save macros

- Select **File→Store→User-Defined Macros...**.

The File Selection dialog box will be displayed so that you can choose a file in which to save the macros. The debugger will automatically add a *.com* extension to the file name.

The debugger will save all of the your user-defined macros to a file.

The debugger does not provide a way to save only selected macros. If you want to save macros in separate files, you can create the macros using a text editor.

To load macros

- Select **File→Load→User-Defined Macros...**.

Choose the macro file to load from the File Selection dialog box.

If macros do not load

☐ Check that the macros do not directly access local program variables.

When the debugger loads macros which access local program variables, the debugger does not know which local scope to use to define the macro.

If you need to access local program variables in a macro, pass them to the macro as parameters.

To call a macro

- Select **Breakpoints→Edit/Call Macro ...→Call**.

Or:

- Using the command line, enter:

Debugger **M**acro **C**all

Enter the name of the macro to be called, and press the **< Return>** key.

When a macro is called with the Debugger Macro Call command, its return value is ignored. Macros are typically called in this manner for the side effects they generate.

Example

If you have the following macro definition:

```
Debugger Macro Add void stackchk()
{
    /* The symbols 'stack' and 'TopOfStack' exist in the compiler's */
    /* environment library, and are addresses which indicate the    */
    /* bottom and the top of the system stack.  The symbol @sp is a */
    /* debugger reserved symbol which contains the current value of */
    /* the processor's stack pointer.                               */

    $Expression Printf "%d bytes of stack used", TopOfStack - @sp$;
    $Expression Printf "%d bytes of stack available", @sp - stack$;
}
.
```

the command:

Debugger **M**acro **C**all stackchk()

displays, in the journal window, the amount of stack used and the amount of stack left.

To call a macro from within an expression

- Enter a macro call as part of any expression entered on the command line of the debugger.

The debugger will evaluate the macro and use its return value when evaluating the rest of the expression.

Example
If you have the following macro definition:

```
Debugger Macro Add int power(x,y)
int     x;
int     y;
{
    int     i;              /* Loop counter */
    int multiplier;         /* Value x is multiplied by */

    /* Multiply x by itself y -1 times */
    for (i = 1, multiplier = x; i < y;i++)
         x *= multiplier;

    /* Return x ^y */
    return x;
}
.
```

The command:

Expression **D**isplay_Value 33.3 + power(2,3)

will call and evaluate the macro, displaying the value 41.3 in the debugger's journal window.

To call a macro from within a macro

- You can call a macro from within a macro when they are part of an expression.

The following restrictions apply to calling macros from within a macro:

- The macro called must have been previously defined.

- The macro cannot call itself.

Example

If you have the following macro definition:

```
Debugger Macro Add int ten_to_the(y)
int     y;
{
       return power(10,y); }
.
```

the macro will compute 10**y by calling the previously defined macro *power( )*.

To call a macro on execution of a breakpoint

- Select **Attach Macro** from the Code window pop-up menu.

  Or:

- When using the command line to set a breakpoint, add a semicolon (;) and the name of the macro to the command.

  When setting breakpoints, you can attach a macro to the breakpoint. Whenever the breakpoint is encountered, the macro is executed. Depending on the return value of the macro, program execution will either stop or continue.

  - If the macro returns zero, program execution stops at the breakpoint.

  - If the macro returns a nonzero value, program execution continues at the breakpoint.

  Macros attached to breakpoints can test program or user-defined variables before determining whether execution should break or not (by returning zero or nonzero values, respectively).

  Macro control flow statements within a breakpoint macro can alter execution flow in the target environment based on target or debugger variable values. You can also include C expressions in macros. By using control flow statements and C expressions in macros, you can patch your C programs.

Example

The following example shows how return values can be used to conditionally control a breakpoint. The example uses the Debugger Macro Add and Breakpt Write commands to define a breakpoint that occurs only when the target variable days becomes greater than 31.

```
Debugger Macro Add int daycheck()
{
     if (days > 31)
          return 0;
     else
          return 1;
}
.
Breakpt Write &days; daycheck()
```

When the break occurs, the macro is executed. If days is less than or equal to 31, program execution continues. If days is greater than 31, program execution stops.

If you have the following macro definition:

```
Debugger Macro Add int break_when(stopfunction, min, max)
char     *stopfunction;
int      min;
int      max;
{
   /* Debugger symbol @function is a char pointer to the name  */
   /* of the current function.  Compare the current function   */
   /* with the function name passed, using the built-in macro  */
   /* memcmp().                                                 */

     if (!strcmp(@function,stopfunction))
     if ((global_var > min) && (global_var <max))
     {
          $Expression Printf "global_var: %d\n", global_var$;
          return 0;
     }

   /* Not in specified function, return 1 so that program will */
   /* continue executing.                                      */
     return 1;
}
.
```

the command:

```
Breakpt Write &global_var; break_when("foo", 256,512)
```

will set a write breakpoint on the global variable *global_var*. Whenever the program writes to *global_var*, the macro *break_when()* is executed with the parameters *"foo"*, *256*, and *512*. The macro returns the value 1 until the value of *global_var* falls between 256 and 512 because of a write to *global_var* in the function *foo()*. The macro then returns 0, causing the program to halt.

To call a macro when stepping through programs

- Select **Execution→Step→with Macro ...**.

  Or:

- Using the command line, enter:

Program **S**tep **W**ith_Macro

Enter the name of the macro to be called, and press the **< Return>** key.

You can use the Program Step With_Macro command to execute a macro after the step occurs. Calling a macro in this manner is useful in tracking down subtle bugs.

Example

If the function *foo( )* was corrupting automatic variables *index* and *ch* on the stack, the following macro and commands could be used to identify the line where the corruption was occurring:

```
Debugger Macro Add void auto_check()
{
    if ((index < 0 || index > 80) || (ch < 32 || ch > 126))
    {
        $Window Screen_On High_Level$;
        $Expression Printf "Autos corrupted!!!\n"$;
        $Expression Printf "index: %d ch: %c\n", index, ch$;
    }
}
.

Program Run Until foo

Program Step With_Macro auto_check()
```

To stop a macro

- Press < **Ctrl** > **-C**.

Macros can be halted during execution by pressing < **Ctrl** > **-C**.

Caution　　< **Ctrl** > **-C** will stop execution of a macro.　Pressing < **Ctrl** > **-C** may interrupt a code-patching macro before it completes execution.　If this occurs, you cannot restart program execution within the macro where it stopped.

To display macro source code

- Choose **Edit** in the Macro Operations dialog box.

  Or:

- Using the command line, enter:

  Debugger **M**acro **D**isplay <macro_name>

  Enter the name of the macro you want to display, and press the < **Return** > key.

  This command will write the macro source to the journal window. If you want to write the macro source to a user-defined window or to a file, you can specify an optional user window number as the destination.

Example　　To write the source for macro auto_check() to user window 51:

Debugger **M**acro **D**isplay auto_check() ,51

To delete a macro

- Using the command line, enter:

Symbol **R**emove <macro_name>

Enter the name of the macro you want to delete, and press the **< Return>** key.

Use the Breakpt Delete command to remove the breakpoint that called the macro.

Using Command Files

A command file is an ASCII file containing debugger commands.

You can create command files from within the interface by logging commands to a command file as you execute the commands, or you can create or modify command files outside the interface with an ASCII text editor.

The debugger can read a command file and execute the commands found there as if they were entered directly into the interface command line.

Command files can also call other command files and the interface will execute the called file like a subroutine of the calling file.

This section shows you how to:

- Record commands.

- Place comments in a command file.

- Pause the debugger.

- Stop command recording.

- Run a command file.

- Set command file error handling.

- Append commands to a command file.

- Record commands and results to a journal file.

- Stop recording commands and results to a journal file.

- Open a file or device for read or write access.

- Close the file associated with a window number.

- Use the debugger in batch mode.

To record commands

- Use the -l *command_file* option to the db68040 or db68030 command when starting the debugger. (The debugger appends the file extension *.com* to *command_file*.)

  $ **db68040 -l** <command_file> <RETURN>

  Or:

- Select **File→Log→Record Commands**. Using the file selection dialog bog, enter the name of the file to which the commands will be saved, and click on the OK pushbutton.

  Or:

- Using the command line, enter:

  File **L**og **O**n

  Enter the name of the file to which commands will be saved, and press the **< Return>** key.

  All commands, whether they are entered from the menus or the command line, are recorded to the *log file*. If a command causes an error, both the command and the error code are recorded as comments.

Example

To start logging commands to file "cmdfile1.com":

File **L**og **O**n cmdfile1

To place comments in a command file

- Using the command line, enter:

File **L**og **C**omment

Enter the comment that should be placed in the command file, and press the
< Return> key.

In the command file, the comment is prefixed with a semicolon (;).

When editing command files, you can also use C-style comments (introduced
by the characters /* and terminated with the characters */).

Example   To place the comment "Place this comment in a command file." in the
command file:

File **L**og **C**omment Place this comment in the command file.

To pause the debugger

- Using the command line, enter:

Debugger **P**ause

And press the **< Return>** key.

The debugger is paused until you enter the spacebar.

You can also specify that the debugger pause for a number of seconds by using
the Debugger Pause Time command.

The Debugger Pause commands are useful when executing command files.

To stop command recording

- Select **File→Log→Stop Command Recording**.

  Or:

- Using the command line, enter:

  File **L**og o**FF**

  And press the **< Return>** key.

  The command file is closed.

To run a command file

- Use the -c *command_file* option to the db68040 command when starting the debugger. (The *command_file* must end with the *.com* extension.)

  ```
  $ db68040 -c <command_file> <RETURN>
  ```

  Or:

- Select **File→Log→Playback**. Using the file selection dialog box, enter the name of the command file, and click on the OK pushbutton.

  Or:

- Using the command line, enter:

  File **C**ommand

  Enter the name of the command file from which debugger commands will be executed, and press the **< Return>** key.

The debugger will begin executing commands found in the command file as if those commands were entered directly into the interface. The debugger will continue to execute commands until it reaches the end of the file or, perhaps, until an error occurs, depending on the command file error handling mode (see "To set command file error handling").

To interrupt playback of a command file, press the **< Ctrl> -c** key combination. (If the graphical interface is being used, the mouse pointer must be within the interface window.)

Example   To start executing command from the file "cmdfile1.com":

File **C**ommand cmdfile1

See Also   File Startup in the "Debugger Commands" chapter

To set command file error handling

- Using the command line, enter:

File **E**rror_Command *<Handling_Mode>*

Select either Abort_Read, Continue_Read, or Quit_Debugger error handling mode, and press the **< Return>** key.

When an error occurs while executing a command file:

Abort_Read   causes the debugger to stop reading the command file.

Continue_Read   causes the debugger to continue executing the command file with the next command.

Quit_Debugger   causes the debugger session to end.

To append commands to an existing command file

- Using the command line, enter:

File **L**og **A**ppend

Enter the name of the file to which commands will be appended, and press the
< **Return**> key.

Example

To append command to the file "cmdfile1.com":

File **L**og **A**ppend cmdfile1

To record commands and results in a journal file

- Use the -j *journal_file* option to the db68040 command when starting the
debugger. (The debugger appends the file extension *.jou* to *journal_file*.)

$ **db68040 -j** <journal_file> <RETURN>

Or:

- Select **File→Log→Record Journal**. Enter the name of the file to which the
commands and results will be saved, and click on the OK pushbutton.

Or:

- Using the command line, enter:

File **J**ournal **O**n

Enter the name of the file to which commands and results will be saved, and
press the < **Return**> key.

Journal files are similar to command files. They contain debugger commands entered during a debug session. Journal files also contain any output generated by debugger commands. Journal files contain everything that is written to the journal window during a debug session.

Example

To start recording commands and results to file "journal1.jou":

File **J**ournal **O**n journal1

To stop command and result recording to a journal file

- Select **File→Log→Stop Journal Recording**.

  Or:

- Using the command line, enter:

  File **J**ournal o**FF**

  And press the **< Return>** key.

To open a file or device for read or write access

- Using the command line, enter:

  File **U**ser_Fopen

  Select the open option, window number, and file name; then, press the **< Return>** key.

After opening a file using the File User_Fopen Append or File User_Fopen Create command, you can use the Expression Fprintf command to write information to the file. Files opened for reading may be read from the built-in macro fgetc(). See the "Predefined Macros" chapter of this manual for a complete description of this macro.

The window number must be between 50 and 256 inclusive.

Use the Window Delete or the File Window_Close command to close the file.

Example

To open user window 57 and redirect any data written to window 57 to the file 'varTrace.out':

File **U**ser_Fopen **C**reate 57 **F**ile varTrace.out

To close the file associated with a window number

- Using the command line, enter:

File **W**indow_Close

Enter the window number associated with the file when it was opened, and press the **< Return>** key.

Example

To close the file associated with user window number 57:

File **W**indow_Close 57

To use the debugger in batch mode

- Use the -b and -c *command_file* options to the db68040 command when starting the debugger.

When using the debugger in batch mode, stdin, stdout, and stderr are disabled. The -b option must be accompanied by the -c option and a debugger command file. All commands are read from the command file. No user interaction with the debugger is allowed. In batch mode, the debugger can be executed as a background process. This mode is commonly used for automatic testing.

Example

```
$ db68040 -b -c <command_file>
```

8

Configuring the Debugger

How to change the appearance and behavior of the debugger.

Configuring the debugger

These tasks are grouped into the following sections:

- Setting the general debugger options.
- Setting the symbolics options.
- Setting the display options.
- Modifying display area windows.
- Saving and loading the debugger configuration.
- Setting X resources.

Some options can be set using either the Debugger Options dialog box or the command line. Other options can be set only using the command line.

Setting the General Debugger Options

This section describes how to:

- Display the Debugger Options dialog box.

- List the debugger options settings.

- Change debugger options settings.

To display the Debugger Options dialog box

- Select **Settings→Debugger Options** from the menu bar.

  You can change settings in the Debugger Options dialog box by clicking on the appropriate buttons.

To list the debugger options settings

- Select **Settings→Debugger Options ...**

  Or:

- Using the command line, enter:

  Debugger **O**ption **L**ist

  And press the **< Return>** key.

  The following information is displayed:

  ```
  > Debugger Option List
    Processor      = 68040
    Intermixed     = On
    Assem_Symbols  = On
  ```

```
Step_Speed    = 0
Radix         = Decimal_Input, Decimal_Output
Stdio_Window  = Swap
Check_Args    = oFF
Align_Bp      = oFF
Breakpt_Window = Swap
More          = On
Highlight     = Inverse
Frame_Stop    = oFF
Command_Echo  = On
View_Window   = Swap
Demand_Load   = oFF
Amt_Scroll    = 1
Exceptions    = Stop
```

To specify whether command file commands are echoed to the Journal window

- Using the command line, enter:

  Debugger **O**ption **C**ommand_Echo

  Select On or oFF, and press the **< Return>** key.

  On   Command file commands are echoed to the Journal window.

  oFF   Command file commands are not echoed to the Journal
              window.

To set automatic alignment for breakpoints and disassembly

- In the Debugger Options dialog box, click on the Align Breakpoints button to toggle alignment.

  Or:

- Using the command line, enter:

  Debugger **O**ption **G**eneral **A**lign_Bp

  Select On or oFF, and press the **< Return>** key.

  On   Debugger automatically aligns breakpoints or locations to
         be displayed in mnemonic format to the beginning of
         instructions.

  oFF   Breakpoints are not automatically aligned.

To set backtrace display of bad stack frames

- In the Debugger Options dialog box, click on the Frame Stop button to toggle
  display of bad stack frames.

  Or:

- Using the command line, enter:

  Debugger **O**ption **G**eneral **F**rame_Stop

  Select On or oFF, and press the **< Return>** key.

  On   Only consecutive valid stack frames are displayed.

  oFF   All stack frames, including bad frames, are displayed.

To specify demand loading of symbols

- Using the command line, enter:

  Debugger **O**ption **G**eneral **D**emand_Load

  Select On or oFF, and press the **< Return>** key.

  On   Symbol information is loaded on an as-needed basis.

  oFF   All symbol information is loaded.

  The **-doff** command-line option overrides the On setting when the settings are
  saved in a startup file.

To select the microprocessor simulated

- Using the command line, enter:

  Debugger **O**ption **G**eneral **P**rocessor *<processor_id>*

  Enter the identifier of the microprocessor being simulated, and press the
  < Return> key.

  Valid processor identifiers are:

  68030 and 68040.

To select the interpretation of numeric literals (decimal/hexadecimal)

- In the Debugger Options dialog box, hold the *command select* mouse button down on the button for "Input Radix" or "Output Radix". Release the button to select "Decimal" or "Hex".

  Or:

- Using the command line, enter:

  Debugger **O**ption **G**eneral **R**adix

  Select Decimal or Hex, and press the **< Return>** key.

  If you select Hex, input and output values are interpreted as hexadecimal for assembly-level references. Any assembly-level number you want interpreted as decimal must be terminated with a *T* (for example, specify 32 as 32T).

  Even if you select Hex, the following inputs will *not* be interpreted as hexadecimal:

  - Line numbers starting with "# ".

  - Variables in high-level expressions, including **C_Expression** and macro expressions. To cast a high-level expression as hexadecimal, use a leading "0x" or a trailing "h".

  - Debugger variables including:

    - breakpoint numbers,

    - viewport numbers, and

    - data viewport line numbers.

  Binary numbers are not available when you select Hex.

  Floating point and enumeration type values are not affected.

To specify exception processing behavior

- Using the command line, enter:

Debugger **O**ption **G**eneral **E**xceptions

Select Normal, Report, or Stop; then, press the **< Return>** key.

Normal   The debugger handles exceptions as would the processor.
(default)

Report   The debugger reports exceptions to the Journal window and
then handles them as would the processor.

Stop   The debugger reports exceptions to the Journal window and
then halts program execution.

The **D**ebugger **O**ption **G**eneral **E**xceptions command sets the value of @exc.
You can set @exc as you would any program variable. The following table lists
what the values of @exc mean:

@exc value	stop execution	report to journal window	trap coprocessor exceptions
0	yes	yes	yes
1	no	yes	yes
2	no	no	yes
4	yes	yes	no
5	no	yes	no
6	no	no	no

address error exception handling is not simulated. An address error will
terminate processing regardless of the value of the @exc pseudoregister.

To specify step speed

- Using the command line, enter:

  Debugger **O**ption **G**eneral **S**tep_Speed *<numb 0..100>*

  Enter the step speed number (from 0 to 100), and press the **< Return>** key.

  Higher numbers represent slower speeds.

Setting the Symbolics Options

This section shows you how to:

- Display symbols in assembly code.

- Display intermixed C source and assembly code.

- Enable parameter checking in commands and macros.

To display symbols in assembly code

- In the Debugger Options dialog box, click on the Assembly Symbols button to toggle assembly symbol display.

    Or:

- Using the command line, enter:

    Debugger **O**ption **S**ymbolics **A**ssem_symbols

    Select On or oFF, and press the **< Return>** key.

    On   Symbols are displayed instead of addresses wherever
                    possible.

    oFF   Addresses are displayed.

To display intermixed C source and assembly code

- In the Debugger Options dialog box, click on the Intermixed Source/Assembly button to toggle source display.

  Or:

- Using the command line, enter:

  Debugger **O**ption **S**ymbolics **I**ntermixed

  Select On or oFF, and press the **< Return>** key.

  On   Assembly code is intermixed with C source code.

  oFF   Only C source code is displayed.

To enable parameter checking in commands and macros

- In the Debugger Options dialog box, click on the Check Parameters button to toggle parameter checking.

  Or:

- Using the command line, enter:

  Debugger **O**ption **S**ymbolics **C**heck_Args

  Select On or oFF, and press the **< Return>** key.

On   When an assignment is made, the debugger warns you if the
                    assignment contains a C type mismatch.

oFF   The debugger does not perform any argument checking.

Setting the Display Options

This section shows you how to:

- Specify the Breakpoint window display behavior.

- Specify the View window display behavior.

- Display half-bright or inverse video highlights.

- Display information a screen at a time (more).

- Specify the standard I/O window display behavior.

- Specify scroll amount.

To specify the Breakpoint window display behavior

- In the Debugger Options dialog box, hold the *command select* mouse button down on the Breakpoint Window button. Release the button to select On or Swap.

  Or:

- Using the command line, enter:

  Debugger **O**ption **V**iew **B**reakpt_Window

  Select On or Swap, and press the **< Return>** key.

  On   The Breakpoint window is displayed at all times.

  Swap   The Breakpoint window is only displayed when you set or
                delete a breakpoint or when you display breakpoints.

To specify the View window display behavior

- In the Debugger Options dialog box, hold the *command select* mouse button down on the View Window button. Release the button to select On or Swap.

  Or:

- Using the command line, enter:

  Debugger **O**ption **V**iew **V**iew_Window

  Select On or Swap, and press the < **Return**> key.

  On   The View window is displayed at all times.

  Swap   The View window is only displayed when you activate the
                   View window or when you enter the Debugger Execution
                   Display_Status command breakpoints.

To specify the standard I/O window display behavior

- In the Debugger Options dialog box, hold the *command select* mouse button down on the Stdio Window button. Release the button to select On or Swap.

  Or:

- Using the command line, enter:

  Debugger **O**ption **V**iew **S**tdio_Window

  Select On, oFF, or Swap; then, press the < **Return**> key.

  On   The Stdio window is displayed at all times.

oFF   The Stdio window is only displayed when function key **F6** is
                pressed or when the Window Screen_On Stdio command is
                entered.

Swap   The Stdio window is displayed when a program writes to it
                and removed when the program returns to the command
                mode.

To display half-bright or inverse video highlights

- Using the command line, enter:

Debugger **O**ption **V**iew **H**ighlight

Select Half_Bright or Inverse, and press the < **Return**> key.

This setting does not affect the graphical user interface.

To turn display paging on or off (more)

- In the Debugger Options dialog box, hold the *command select* mouse button
  down on the More List Mode button. Release the button to select On or Off.

Or:

- Using the command line, enter:

Debugger **O**ption **V**iew **M**ore

Select On or oFF, and press the < **Return**> key.

On   Information is listed one screen at a time.

oFF   Information is listed all at once.

To specify scroll amount

- Using the command line, enter:

Debugger **O**ption **V**iew **A**mt_Scroll *<numb 0..50>*

Enter the number of lines for information to be scrolled (from 0 to 50), and press the **< Return>** key.

Modifying Display Area Windows

You can reformat display-area screens by modifying their windows. For example, you can reformat the high-level screen by resizing and moving the high-level Code, Monitor, Backtrace, Journal, and Breakpoint windows. You can also resize and move the alternate view of these windows.

This section shows you how to:

- Resize or move the active window.

- Move the Status window (standard interface only).

- Define user screens and windows.

- Display user-defined screens.

- Erase standard I/O and user-defined window contents.

- Remove user-defined screens and windows.

To resize or move the active window

1 Using the command line, enter:

Window **R**esize

And press the **< Return>** key.

2 Type **T** to position the top-left corner, **B** to position the lower-right corner, or **M** to move the window without resizing it; then, use the cursor keys to move the window or window border. When the window is at the desired location, press the **< Return>** key to save the new coordinates.

If you make a mistake while resizing the window, press **CTRL C** or press **Esc** twice to restore the previous coordinates.

The Window Resize command is used to move or alter the size of any existing window, except for the Status window. Use the Window New command to move the Status window in the standard interface.

When you use the Window Resize command on the normal view of a window, the normal dimensions are modified. When you use the command on the alternate view of a window, the alternate dimensions are modified.

You can enter resize commands when any screen is displayed. However, the debugger does not display commands on the standard I/O screen or on any user-defined screen.

To move the Status window (standard interface only)

The Status window cannot be moved in the graphical interface.

1 Using the command line in the standard interface, enter:

Window **N**ew

Specify window number 5 to move the high-level Status window (or window number 15 to move the assembly level Status window), select Tab followed by High_Level (or Assembly), enter the new coordinates for the Status window, and press the **< Return>** key.

The Status window cannot be resized. The difference between the bottom row coordinate and top row coordinate must be 3.

A high-level program must be loaded in order to move the high-level status screen.

Be sure to move any windows that occupy the screen area to which you are moving the Status window. Otherwise, the Status window will be hidden behind these windows.

Examples   To move the high-level Status window to the top of the display (upper left corner at 0,0 and lower right corner at 3,78):

`Window New 5 <tab> High_Level 0,0,3,78`

To move the assembly-level Status window to the bottom of the display:

`Window New 15 <tab> Assembly 19,0,22,78`

To define user screens and windows

- Using the command line, enter:

`Window New`

Enter the window and screen parameters, and press the **< Return>** key.

The debugger lets you define your own screens and windows so that you have flexibility in displaying debugger information.

User-defined windows must be assigned a number greater than or equal to 50, and less than or equal to 256. Numbers below 50 are reserved for predefined debugger screens and windows.

When you make a new window with the Window New command, the normal view and alternate view dimensions are set identically. The debugger allocates a buffer with enough memory to contain the entire window. Therefore, the window with the largest dimensions (normally the alternate view) should be defined first to allocate sufficient memory.

To display a user-defined screen, use the **Window Screen_On** command or press function key **F6**.

Caution

When making a new window on the high-level or assembly-level screens, be careful not to enter coordinates that will result in a window that covers the status line and command line. On a standard 80-column terminal display, a row coordinate may be between 0 and 23. Creating a window with a bottom row coordinate greater than 18 will cause part or all of the status and command lines to be covered.

Examples

To make a user window numbered 57 in user screen 4 with the upper-left corner of the window at coordinates 5,5 and the lower-right corner of the window at coordinates 18,78:

Window **N**ew 57 <tab> **U**ser_Screen 4 <tab> **B**ounds 5,5,18,78

If user screen 4 does not exists, the debugger automatically creates it.

To display user-defined screens

- Using the command line, enter:

Window **S**creen_On **U**ser_Screen *<screen_nmbr>*

Enter the user screen number, and press the **< Return>** key.

Examples

To display user screen 4:

Window **S**creen_On **U**ser_Screen 4

To erase standard I/O and user-defined window contents

- Using the command line, enter:

Window **E**rase *<user_window_nmbr>*

Enter the user window number (the standard I/O window number is 20) whose contents you wish to clear, and press the **< Return>** key.

If you do not specify a window number or if you specify 0, the active user-defined window is cleared. This command is useful in macros.

Examples
To erase the contents of user window 57:

Window **E**rase 57

To remove user-defined screens and windows

- Using the command line, enter:

Window **D**elete *<user_window_nmbr>*

Enter the number of the window to be removed, and press the **< Return>** key.

To remove a user-defined screen, remove all windows associated with that screen.

You cannot remove predefined debugger windows and screens.

Saving and Loading the Debugger Configuration

Information regarding debugger options and screen configurations can be saved in a *startup file*. Startup files can be created only from within the debugger.

This section shows you how to:

- Save the current debugger configuration.

- Load a startup file.

To save the current debugger configuration

- Use the menu select mouse button to choose **File→Store→Startup (.rc) file (as default)**. The information is saved in file "db68040.rc" (regardless of whether the debugger is started as *db68030* or *db68040*) in the current directory.

  Or:

- Use the menu select mouse button to choose **File→Store→Startup (.rc) file**. Using the file selection dialog box, enter the name of the file to which startup information should be saved; then, click on the OK pushbutton.

  Or:

- Using the command line, enter:

  File **S**tartup *<startup_file>*

  Enter the name of the file in which the startup information should be saved, and press the **< Return>** key. If you do not specify the name of the startup file, the default value will be used.

  This command also saves the window and screen settings.

When saving window and screen settings that have been customized for a particular type of terminal, name the startup file the same as the TERM environment variable setting. If no startup file is loaded when starting the debugger, the debugger will automatically search for startup files named "./$TERM.rc" (in the current directory) or "$HOME/.$TERM.rc" (in the home directory). files.

Examples

To save the current debugger state in a file called "my_state.rc":

```
File Startup my_state
```

To save, in you home directory, window and screen settings that have been customized for the 2392, 2392a, 2392A, hp2392, hp2392a, or hp2392A terminal types:

```
File Startup ~/.2392
```

To load a startup file

- Use the -s *startup_file* option to the db68040 command when starting the debugger.

```
$ db68040 -s <startup_file> <RETURN>
```

The debugger's startup options and window specifications are configured as described in *startup_file*.

The *startup_file* must end with the .rc extension and can be created only from within the debugger.

If no startup file is named, the following files are searched for in order. The first one that exists will be used ($HOME and $TERM are UNIX environment variables).

    db68040.rc (regardless of whether the debugger is started as *db68030* or *db68040*) in the current directory
    ./$TERM.rc in the current directory
    $HOME/.$TERM.rc

If no startup file is found, reasonable defaults will be used.

Examples   To start the debugger and load the state saved in the startup file "my_state.rc":

$ **db68040 -s** my_state.rc <RETURN>

Setting X Resources

The debugger's graphical interface is an X Window System application which means it is a *client* in the X Window System client-server model.

The X server is a program that controls all access to input devices (typically a mouse and a keyboard) and all output devices (typically a display screen). It is an interface between application programs you run on your system and the system input and output devices.

An X *resource* controls an element of appearance or behavior in an X application. For example, one resource controls the text in action key pushbuttons as well as the action performed when the pushbutton is clicked.

By modifying resource settings, you can change the appearance or behavior of certain elements in the graphical interface.

Where resources are defined

When the graphical interface starts up, it reads resource specifications from a set of configuration files. Resources specifications in later files override those in earlier files. Files are read in the following order:

1   The application defaults file. For example, /usr/lib/X11/app-defaults/HP64_Debug in HP-UX or /usr/openwin/lib/X11/app-defaults/HP64_Debug in SunOS.

2   The $XAPPLRESDIR/HP64_Debug file. (The XAPPLRESDIR environment variable defines a directory containing system-wide custom application defaults.)

3   The server's RESOURCE_MANAGER property. (The **xrdb** command loads user-defined resource specifications into the RESOURCE_MANAGER property.)

If no RESOURCE_MANAGER property exists, user defined resource settings are read from the $HOME/.Xdefaults file.

4   The file named by the XENVIRONMENT environment variable.

If the XENVIRONMENT variable is not set, the $HOME/.Xdefaults-*host* file (typically containing resource specifications for a specific remote host) is read.

5 Resource specifications included in the command line with the **-xrm** option.

6 System scheme files in directory /usr/hp64000/lib/X11/HP64_schemes.

7 System-wide custom scheme files located in directory $XAPPLRESDIR/HP64_schemes.

8 User-defined scheme files located in directory $HOME/.HP64_schemes (note the dot in the directory name).

Scheme files group resource specifications for different displays, computing environments, and languages.

The HP64_Debug application defaults file is re-created each time debugger's graphical interface software is installed or updated. You can use the UNIX **diff** command to check for differences between the new HP64_Debug application defaults file and the old application defaults file that is saved as /usr/hp64000/lib/X11/HP64_schemes/old/HP64_Debug.

Refer to the "X Resources and the Graphical Interface" chapter for more detailed information about X resources.

To modify the debugger's graphical interface resources

You can customize the appearance of an X Windows application by modifying its X resources. The following tables describe some of the commonly modified application resources.

Application Resources for Schemes		
Resource	**Values**	**Description**
HP64_Debug.platformScheme	HP-UX SunOS (custom)	Names the subdirectory for platform specific schemes. This resource should be set to the platform on which the X server is running (and displaying the debugger's graphical interface) if it is different than the platform where the application is running.
HP64_Debug.colorScheme	BW Color (custom)	Names the color scheme file.
HP64_Debug.sizeScheme	Small Large (custom)	Names the size scheme file which defines the fonts and the spacing used.
HP64_Debug.labelScheme	Label $LANG (custom)	Names to use for labels and button text. The default uses the $LANG environment variable if it is set and if a scheme file named Debug.$LANG exists in one of the directories searched for scheme files; otherwise, the default is Label.
HP64_Debug.inputScheme	Input (custom)	Specifies mouse and keyboard operation.

Commonly Modified Application Resources		
Resource	**Values**	**Description**
HP64_Debug.enableCmdline	True False	Specifies whether the command line area is displayed when you initially enter the debugger's graphical interface.
*editFile	(example) vi % s	Specifies the command used to edit files.
*editFileLine	(example) vi + % d % s	Specifies the command used to edit a file at a certain line number.
*sim68040*actionKeysSub.keyDefs	(paired list of strings)	Specifies the text that should appear on the action key pushbuttons and the commands that should be executed in the command line area when the action key is pushed. Refer to the "To set up custom action keys" section for more information.
*sim68040*dirSelectSub.entries	(list of strings)	Specifies the initial values that are placed in the **File→Context→Directory** pop-up recall buffer. Refer to the "To set initial recall buffer values" section for more information.
*sim68040*recallEntrySub.entries	(list of strings)	Specifies the initial values that are placed in the entry buffer (labeled "():"). Refer to the "To set initial recall buffer values" section for more information.

The following steps show you how to modify the debugger's graphical interface's X resources.

1 Copy part or all of the HP64_Debug application defaults file to a temporary file.

If you are using an HP-UX computer, type:

```
cp /usr/lib/X11/app-defaults/HP64_Debug HP64_Debug.tmp
```

If you are using SunOS, type:

```
cp /usr/openwin/lib/X11/app-defaults/HP64_Debug HP64_Debug.tmp
```

2 Make the temporary file writable:

```
chmod +w HP64_Debug.tmp
```

3 Modify the temporary file.

Modify the resource that defines the behavior or appearance that you wish to change.

For example, to change the number of lines in the main display area to 36, search for the string "HP64_Debug.lines". You should see lines similar to the following.

```
!-----------------------------------------------------------------------------
! The lines and columns set the vertical and horizontal dimensions of the
! main display area in characters, respectively.  Minimum values are 18 lines
! and 80 columns.  These minimums are silently enforced.
!
! Note: The application cannot be resized by using the window manager.

!HP64_Debug.lines:    24
!HP64_Debug.columns:  85
```

Edit the line containing "HP64_Debug.lines" so that it is uncommented and is set to the new value:

```
!-----------------------------------------------------------------------------
! The lines and columns set the vertical and horizontal dimensions of the
! main display area in characters, respectively.  Minimum values are 18 lines
! and 80 columns.  These minimums are silently enforced.
!
! Note: The application cannot be resized by using the window manager.

HP64_Debug.lines:    36
!HP64_Debug.columns:  85
```

If you wish, you may delete any lines which you will not be modifying; any resources you delete will use the default values.

Save your changes and exit the editor.

4 If the RESOURCE_MANAGER property exists (as is the case with HP VUE — if you're not sure, you can check by entering the **xrdb -query** command), use the **xrdb** command to add the resources to the RESOURCE_MANAGER property. For example:

```
xrdb -merge -nocpp HP64_Debug.tmp
```

5 Save the changes where they can be found by the debugger.

One way to do this is to append the temporary file to your $HOME/.Xdefaults file. For example:

```
cat HP64_Debug.tmp >> $HOME/.Xdefaults
```

You can also save the changes in a scheme file (see "To use customized scheme files").

6 Remove the temporary file.

7 Start or restart the debugger's graphical interface.

To use customized scheme files

Scheme files are used to set platform specific resources that deal with color, fonts and sizes, mouse and keyboard operation, and labels and titles. You can create and use customized scheme files by following these steps.

1 Create the $HOME/.HP64_schemes/< platform> directory.

For example:

```
mkdir $HOME/.HP64_schemes
mkdir $HOME/.HP64_schemes/HP-UX
```

2 Copy the scheme file to be modified to the $HOME/.HP64_schemes/< platform> directory.

Label scheme files are not platform specific; therefore, they should be placed in the $HOME/.HP64_schemes directory. All other scheme files should be placed in the $HOME/.HP64_schemes/< platform> directory.

For example:

```
cp /usr/hp64000/lib/X11/HP64_schemes/HP-UX/Debug.Color
    $HOME/.HP64_schemes/HP-UX/Debug.MyColor
```

Note that if your custom scheme file has the same name as the default scheme file, the load order requires resources in the custom file to explicitly override resources in the default file.

3 Modify the $HOME/.HP64_schemes/< platform> /Debug.< scheme> file.

For example, you could modify the "$HOME/.HP64_schemes/HP-UX/Debug.MyColor" file to change the defined foreground and background colors. Also, since the scheme file name is different than the default, you could comment out various resource settings to cause general foreground and background color definitions to apply to the debugger's graphical interface. At least one resource must be defined in your color scheme file for it to be recognized.

4 If your custom scheme file has a different name than the default, modify the scheme resource definitions.

The debugger's graphical interface application defaults file contains resources that specify which scheme files are used. If your custom scheme files are named differently than the default scheme files, you must modify these resource settings so that your customized scheme files are used instead of the default scheme files.

For example, to use the "$HOME/.HP64_schemes/HP-UX/Debug.MyColor" color scheme file you would set the "HP64_Debug.colorScheme" resource to "MyColor":

```
HP64_Debug.colorScheme: MyColor
```

To set up custom action keys

• Modify the "actionKeysSub.keyDefs" resource.

To modify this resource, follow the procedure in "To modify the debugger's graphical interface resources."

The "actionKeysSub.keyDefs" resource defines a list of paired strings. The first string defines the text that should appear on the action key pushbutton. The second string defines the command that should be sent to the command line area and executed when the action key is pushed.

A pair of parentheses (with no spaces, that is "()") can be used in the command definition to indicate that text from the entry buffer should replace the parentheses when the command is executed.

Action keys that use the entry buffer should always include the entry buffer symbol, "( )", in the action key label as a visual cue to remind you to place information in the entry buffer before clicking the action key.

Shell commands can be executed by using the Debugger Host_Shell command.

Also, command files can be executed by using the File Command command.

Finally, an empty action ("") means to repeat the previous operation, whether it came from a pull-down, a dialog, a pop-up, or another action key.

Example
To set up custom action keys, modify the "debug*actionKeysSub.keyDefs" resource:

```
*sim68040*actionKeysSub.keyDefs: \
    "Make"              "D H make      I" \
    "Disp Src ()"       "P C S () ;; P D ()" \
    "Run Until ( )"     "P R U ()" \
    "Step"              "P S"
```

See Also
"To modify debugger's graphical interface resources" in this chapter.

To set initial recall buffer values

- Modify the "entries" resource for the particular recall buffer.

Some of the resources for the pop-up recall buffers are listed in the following table:

Pop-up Recall Buffer Resources	
Recall Pop-up	**Resources**
Entry Buffer ():	*recallEntrySub.entries
File→Context→Directory ...	*dirSelectSub.entries
Modify→Register; Recall Value	*modRegDB*recallSub.entries
Command Line command recall	*recallCmdSub.entries
Macro Operations dialog box; Recall Value	*macroDB_popup*recallSub.entries

Other X resources for the recall buffers are described in the supplied application defaults file.

The window manager resource "*transientDecoration" controls the borders around dialog box windows. The most natural setting for this resource is "title."

Example   To set the initial values for the directory selection dialog box, modify the "debug*dirSelectSub.entries" resource:

```
*sim68040*dirSelectSub.entries: \
    "$HOME" \
    ".." \
    "/users/project1" \
    "/users/project2/code"
```

Refer to the previous "To modify the debugger's graphical interface resources" section in this chapter for more detailed information on modifying resources.

Part 3

Concept Guide

Part 3

9

X Resources and the Graphical Interface

An introduction to X resources.

X Resources and the Graphical Interface

This chapter helps you to understand how to set the X resources that control the appearance and operation of the debugger's graphical interface. This chapter:

- Explains the X Window concepts surrounding resource specification.

- Explains the scheme files used by the debugger's graphical interface.

The debugger's graphical interface is an X Window System application which means it is a *client* in the X Window System client-server model.

The *X server* is a program that controls all access to input devices (typically a mouse and a keyboard) and all output devices (typically a display screen). It is an interface between application programs you run on your system and the system input and output devices.

An X resource is user-definable data

A *resource* is a user-definable piece of data that controls the operation or appearance of an X Windows application. A resource may apply to an application (application-specific resources) or it may apply to the objects called *widgets* from which the application is constructed. That is particularly true of standard widget resources that control the appearance of an application. For example, most widgets have a standard resource that allows the user to specify the font used to display text on objects like buttons, menus, and labels.

An *application-specific resource* is defined by the application developer and may control such things as the mode of operation of an application. For example, you can use an application-specific resource for the debugger's graphical interface to control whether to start the interface with the command line on or the command line off.

A resource specification is a name and a value

Each resource in an application has a name and a value. Because an X Window System application is constructed from widgets, a resource name is closely associated with the names of the widgets that make up the application. Each application begins with a top-level widget that is the parent of all other

widgets in the application. The name of the top-level widget is usually the same as that of the application. This top-level widget may have a number of widgets "beneath" it that are called children of the top-level widget. The names for these widgets are most often chosen for their mnemonic value. These children can also in turn have child widgets. A resource name, then, is simply a name of a piece of data for the lowest-level widget coupled with a string of widget names picked up from each of the widgets along the path starting with the top-level widget and going down to the lowest-level widget.

The data name and widget names within a resource name are separated from each other by dots. The resource name itself is terminated by a colon. A resource value is simply the data value itself. Ignoring the widget names and data name for the moment, a common resource for most widgets is color. A data value for color might be "blue."

To put this all together, a resource string for the foreground color for the "quit" pushbutton displayed on an application called "tracker" might look like the following:

```
tracker.panel.control.quit.foreground: white
```

Don't worry, there are shortcuts

As you might guess, specifying resources for applications with many levels of widgets can be difficult and error-prone. For that reason, you can use a shortened notation. To fully understand how the notation works, however, you must first understand about *instance names* and *class names*.

An *instance name* is a name given to a particular widget by an application developer. You have already seen instance names used. The name "quit" is an instance name for a pushbutton widget used by the developer of the "tracker" application from the last example. An instance name makes the pushbutton widget named "quit" unique from other pushbutton widgets in the "tracker" application.

A *class name* is a general name for all widgets of a particular type. For example, the class name for the OSF/Motif pushbutton widget is XmPushButton. When you refer to a widget in an application by its class name, you are referring to all widgets of that class in the application, and not to just a particular widget.

Instead of specifying the foreground color for the tracker quit button by using a resource name made up of instance names as in the last example, you could instead use a class name, as follows:

```
tracker.panel.control.XmPushbutton.foreground: white
```

Using class names in this way makes it easier to specify resources because it relieves you from having to discover the names of particular widgets in an application. A long string of instance names or class names is still a long string of names, however. Fortunately, a wildcard helps to make the shortcut a true shortcut. The wildcard is an asterisk ("*"). It can be used to replace any number of class or instance names in a resource name. The last example could now be shortened to either of the following:

```
tracker*XmPushButton.foreground: white
```

```
tracker*quit.foreground: white
```

But wait, there is trouble ahead

An X Window System application maintains a complete list of resources, and the application knows the complete instance and class names for each resource. Because you can specify resource values using shortened notation, the application, when starting up, must match specified values to individual resources. Some general rules apply:

- Either a class name or instance name from the request must match each class or instance name in the application's list of resources.

- Entries prefixed by a dot are more specific and therefore have precedence over entries prefixed by an asterisk.

- Instance names are more specific and therefore have precedence over class names.

- Matching is done from left to right. Instance or class names appearing at the beginning of the specification have precedence over those later in the specification.

As you can quickly see, resource matching favors specific resource names over general resource names. General resource names, especially those involving class names, can have unexpected and unintended effects. Consider the last example again. The resource specification

```
tracker*XmPushButton.foreground: white
```

may not only set the foreground color of the quit button on the control panel
of the application to white — it could also set the foreground colors for any
pushbutton anywhere in the application. That is because the combination of
the wildcard and the use of the class name make this resource specification
match a resource request for any pushbutton in the application.

The second of the two specifications in the example does not completely solve
the problem either. Suppose there was another button elsewhere in the
application with the instance name of "quit." (Duplicating instance names is
correct as long as the widget paths to two different widgets of the same name
are different.) The second specification of

```
tracker*quit.foreground: white
```

could match a resource request for that button as well because the wildcard
allows the specification to match a number of different widget paths through
the application.

Resource specification is usually a matter of trial and error. The following
resource is probably specific enough to set just the foreground color for the
quit button on the control panel:

```
tracker*control*quit.foreground: white
```

To view the resources in the debugger's graphical interface, you can choose
Help→X Resource Names and click on the "All names" button.

Class and instance apply to applications as well

Just as there are classes and instances of widgets, there are classes and
instances of X Window applications. Resource specifications can be
constructed in such a way that they apply to a whole class of applications, or
just to an instance of those applications.

The class name for the debugger graphical interface products is *HP64_Debug*.
The instance of the class that this debugger graphical interface falls under is
called *debugsim*. A few examples are in order.

• For a given resource (called < resource>), the following specification
  applies to all debugger interface products for all processors:

```
HP64_Debug*<resource>: <value>
```

• The following specifications apply to all 68030/040 Debugger/Simulator interfaces:

```
HP64_Debug.sim68040*<resource>: <value>
```

```
debugsim.sim68040*<resource>: <value>
```

According to the precedence rules for resource matching, the first specification is the most general and would be overridden by either of the following two.

Resource specifications are found in standard places

X resources are defined in standard places so that applications can find them and use them when starting up.

The app-defaults file

The app-defaults file contains only resources for a specific application. The system directory for application default files is:

HP-UX   */usr/lib/X11/app-defaults*

SunOS   */usr/openwin/lib/X11/app-defaults*

The name of the default file is the same as the class name for the application and is also called the *app-defaults file* (for example, HP64_Debug is the name of the debugger's graphical interface's application defaults file).

These defaults should not be changed by individual users because doing so affects the appearance and behavior of the application for all users of the application.

The .Xdefaults file

The .Xdefaults file in your $HOME directory usually contains user-defined resources for several applications.

Scheme files

X resource specifications can point to *scheme files* in which other X resources are specified.

Loading order resolves conflicts between files

If there are two files, then which resource specification from which file controls the resource in the application? That problem is solved by adhering to a loading order for files. The following is a list of the standard places, in order, that an application looks to find resources:

1 The application default file.

The application default file for the graphical interface is called *HP64_Debug*. This file is created at software installation time and placed in the system application defaults directory.

2 $XAPPLRESDIR/< class>

This environment variable defines an alternative directory path leading to customized class files. Useful for directing the application to system-wide custom files.

3 RESOURCE_MANAGER property. Some X servers have a resource property associated with the root window for the server. Resources are added to the resource property database by using *xrdb*. (HP VUE is an example.) The server can use this property to access those resources.

If no RESOURCE_MANAGER property exists, then $HOME/.Xdefaults is read. The primary and probably best method for creating or adding to this file is by copying part or all of the app-defaults file into the .Xdefaults file.

4 $XENVIRONMENT file. This environment variable defines a file that contains resource specifications.

If the XENVIRONMENT variable is not set, then $HOME/.Xdefaults-*host* is read.

5 Command line options

Resources can be specified on the command line by using the **-xrm** command line option. The application strips these arguments out and

sets these resources before passing the rest of the command line on to the application.

Remember, load order specifies the precedence for resource overrides. A resource found later in the load order overrides a resource found earlier in the load order if the resource specifications match each other.

The app-defaults file documents the resources you can set

The *HP64_Debug* file is complete, well-commented, and a good source of reference for graphical interface resources. The HP64_Debug file should be your primary source of information about setting graphical interface resources. This file can be easily viewed from the help topic menu by choosing **Help→General Topic** and selecting the "X Resources: Setting" topic.

To further assist you with setting X resources, there is also another topic on the help menu pull-down that you should use. Choose **Help→X Resource Names** to display the class and instance name for the graphical interface in a dialog box. From the dialog box, you can also display all widget class and instance names for all widgets that make up the debugger's graphical interface. In most cases, you will not need to delve that far into the widget tree, but it is there if you need it.

In addition to the app-defaults file, the graphical interface uses *scheme files*. Resources are not duplicated between scheme files and the HP64_Debug file. You may wish to set resources found in the scheme files as well, so you need to understand how scheme files relate to the interface and to the other X resource files.

Scheme files augment other X resource files

Hewlett-Packard realizes that the debugger's graphical interface will be run in environments made up of workstations with different display capabilities and even in environments with different types of computers (platforms) running the X Window System. The debugger's graphical interface, unlike many other X applications, makes determinations about display hardware as to the platform type, the resolution of the display, and whether the display is color or monochrome. The interface then loads the appropriate scheme files to allow the interface to come up in a reasonable way based on the hardware.

There are six scheme files. Their names and a brief description of the resources they contain follows:

Debug.Label   Defines the labels for the fixed text in the interface.
                       Such things as menu item labels and similar text are in
                       this file. If the $LANG environment variable is set, the
                       scheme file "Debug.$LANG" is loaded if it exists;
                       otherwise, the file "Debug.Label" is loaded.

Debug.BW   Defines the color scheme for black and white displays.
                       This file is chosen if the display cannot produce at least
                       16 colors.

Debug.Color   Defines the color scheme for color displays. This file is
                       chosen if the display can produce 16 or more colors.

Debug.Input   Defines the button and key bindings for the mouse and
                       keyboard.

Debug.Large   Defines the window dimensions and fonts for high
                       resolution display (1000 pixels or more vertically).

Debug.Small   Defines the window dimensions and fonts for low
                       resolution displays (less than 1000 pixels vertically).

Debug.Label (or Debug.$LANG) resides in the directory
/usr/hp64000/lib/X11/HP64_schemes. This directory is the upper level directory
for scheme files. The other five files are in subdirectories below this one
named by platform (or operating system). For example, the HP 9000 scheme
files are in the subdirectory */usr/hp64000/lib/X11/HP64_schemes/HP-UX*.

Like the app-defaults file, these scheme files are system files and should not be
modified directly.

You can create your own scheme files, if you choose

The debugger's graphical interface supports user-defined scheme files. The
interface searches two places for user-defined scheme files and loads any it
finds after loading the system scheme files. Refer to any of the scheme files
mentioned for information about where to place your own scheme files.

Scheme files continue the load sequence for X resources

Scheme files extend the load order for finding X resources. System scheme file resources override all other resources gathered so far, and user-defined scheme files, in turn, override the system scheme files. Continuing from the load order list previously, the scheme files follow, in the order

1 /usr/hp64000/lib/X11/HP64_schemes/Debug.Label
/usr/hp64000/lib/X11/HP64_schemes/< platform> /Debug.< scheme>

2 $XAPPLRESDIR/HP64_schemes/Debug.Label
$XAPPLRESDIR/HP64_schemes/< platform> /Debug.< scheme>

Just as $XAPPLRESDIR can point to a system-wide app-defaults file, so can it point to a set of system-wide scheme files.

3 $HOME/.HP64_schemes/Debug.Label
$HOME/.HP64_schemes/< platform> /Debug.< scheme>

Please note the dot (.) in the ".HP64_schemes" directory name.

You can force the debugger's graphical interface to use certain schemes

Five application-specific resources allow you to force the interface to use certain schemes. The resources and what they control are as follows:

HP64_Debug.platformScheme:

Controls the platform scheme chosen by the interface. This resource is particularly useful in mixed-platform environments where you might be executing the interface remotely on an HP 9000 computer, but displaying the interface on a Sun SPARCsystem computer. In this situation, you may wish to set the resource to use the SunOS scheme so that you can use the same key and mouse button bindings as other Sun OpenWindows applications.

The value of this resource is actually the name of a subdirectory under /usr/hp64000/lib/X11/HP64_schemes or one of the alternative directories for scheme files. You can create your own file and subdirectory under /usr/hp64000/lib/X11/HP64_schemes (or alternative) and then set this resource to choose that subdirectory instead of the standard platform subdirectory.

Values can be: HP-UX, SunOS, or the name of a sub-directory containing custom scheme files.

HP64_Debug.colorScheme:

Chooses the black and white or color scheme.

Values can be: Color, BW, or the name of a custom scheme file.

HP64_Debug.inputScheme:

Chooses the keyboard and mouse bindings.

Values can be: Input or the name of a custom scheme file.

HP64_Debug.sizeScheme:

Chooses the large or small scheme for fonts and sizes.

Values can be: Large, Small, or the name of a custom scheme file.

HP64_Debug.labelScheme:

Chooses a different label scheme for fixed text. Again, this resource is affected by the $LANG variable.

Values can be: Label, $LANG (if this environment variable is set and there is a Debug.$LANG scheme file), or the name of a custom scheme file.

These resources are in the app-defaults file. To override these resources, set them in your *.Xdefaults* file.

Again, setting X resources is a trial and error process. The scheme files used by the debugger's graphical interface simplify the process by collecting related resources in specific files.

To review the organization:

- The app-defaults file contains resources that control the operation of the interface. To override a resource in this file, copy the resource to your .Xdefaults file and change it there.

- Resources that control the appearance of the display and keyboard and mouse button bindings for your platform are in the scheme files. Copy the scheme files to an appropriate place and modify the resources found in them to change the look of the interface.

If you would rather place these resources in your .Xdefaults file, remember the load order. Make the resource name in the .Xdefaults file more specific or it will be overridden by the one in the scheme file.

The app-defaults file and the scheme files are your best sources of reference for help with modifying individual resources.

Resource setting - general procedure

Application specific resources

If you plan to modify an application-specific resource, you should look in the HP64_Debug file for information about that resource.

If the RESOURCE_MANAGER property exists (as is the case with HP VUE), copy the complete HP64_Debug file, or just the part you are interested in, to a temporary file. Modify the resource in your temporary file and save the file. Then, merge the temporary file into the RESOURCE_MANAGER property with the **xrdb -merge < filename>** command.

If the RESOURCE_MANAGER property does not exist, copy the complete HP64_Debug file, or just the part you are interested in, to your *.Xdefaults* file. Modify the resource in your .Xdefaults file and save the file.

Finally, if the debugger's graphical interface is currently executing, you must exit and restart the interface for the change to have any effect.

General resources

If you plan to modify a general resource that could not be found in the HP64_Debug file, look to the scheme files for information about that resource. A general discussion of the kinds of information found in the scheme files can be found in the previous "Scheme files augment other resources" section.

Copy the appropriate scheme file to one of the alternative directories and make the modifications there. (If you are using $XAPPLRESDIR, make sure the variable is set and exported.) Save the file. If the debugger's graphical interface is currently executing, you must exit the application and restart it to see the results of your change.

Part 4

Reference

Part 4

10
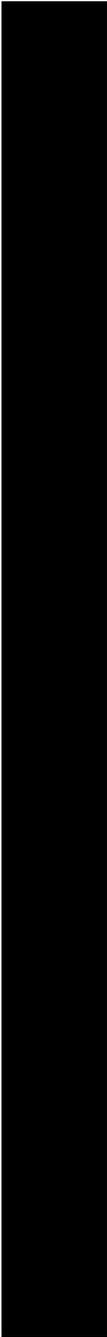
Debugger Commands

Detailed descriptions of command line commands.

How Pulldown Menus Map to the Command Line

Pulldown	Command Line
File→Context→Directory	Debugger Directory Change_Working
File→Context→Symbols	Program Context Set
File→Load→Executable	Program Load Default
File→Load→Program Only	Program Load New Code_Only No_Pc_Set
File→Load→Symbols Only	Program Load New Symbols_Only No_Pc_Set
File→Load→User-Defined Macros	File Command < filename>
File→Store→Startup (.rc) file (as default)	File Startup
File→Store→Startup (.rc) file	File Startup < filename>
File→Store→User-Defined Macros	N/A
File→Store→BBA Data	Memory Unload_BBA < filename>
File→Copy Window→	File User_Fopen Append < win> File < filename>
File→Log→Playback	File Command
File→Log→Record Commands	File Log On < filename>
File→Log→Stop Command Recording	File Log oFF
File→Log→Record Journal	File Journal On
File→Log→Stop Journal Recording	File Journal oFF
File→Edit→File	Debugger Host_Shell < editor>
File→Edit→At () Location	Debugger Host_Shell < editor_at_line>
File→Edit→At PC Location	Debugger Host_Shell < editor_at_line>
File→Term	Debugger Host_Shell
File→Exit→Window	Debugger Quit Yes
Display→Context	N/A

Pulldown	Command Line
Display→Memory→Mnemonic ()	Memory Display Mnemonic
Display→Memory→byte	Memory Display Byte
Display→Memory→word	Memory Display Word
Display→Memory→long	Memory Display Long
Display→Source ()	Program Display_Source
Display→Source at PC	Program Context Set
Display→Source Find Fwd ()	Program Find_Source Occurrence Forward
Display→Source Find Back ()	Program Find_Source Occurrence Backward
Display→Source Find Again	Program Find_Source Next
Display→C Expression ()	Expression C_Expression
Display→Var/Expression ()	Expression Display_Value
Display→Monitor Variable	Expression Monitor Value
Display→All symbols ()	Symbol Display Default
Display→Symbols→Data & Macros ()	Symbol Display Options Data&macros End_Options
Display→Symbols→Functions & Labels ()	Symbol Display Options Functions&labels End_Options
Display→Symbols→Modules ()	Symbol Display Options Modules End_Options
Display→Symbols→Browse C++ Class ()	Symbol Browse
Display→Error Log	N/A
Modify→C Expression ()	Expression C_Expression < variable> =
Modify→Memory	Memory Assign
Modify→Memory at ()	Memory Assign Long
Modify→Register	Expression C_Expression @reg

Pulldown	Command Line
Execution→Run→from PC	Program Run
Execution→Run→from ()	Program Run From
Execution→Run→from Transfer Address	Program PC_Reset - Program Run
Execution→Run→until ()	Program Run Until
Execution→Step Over→	Program Step Over
Execution→Step→	Program Step
Execution→Reset to Monitor	Debugger Execution Reset_Processor
Execution→Set PC to Transfer	Program Pc_Reset
Breakpoints→Set→Instruction ()	Breakpt Instr
Breakpoints→Set→Read ()	Breakpt Read
Breakpoints→Set→Write ()	Breakpt Write
Breakpoints→Set→Read/Write ()	Breakpt Access
Breakpoints→Delete ()	Breakpt Delete
Breakpoints→Delete All	Breakpt Clear_All
Breakpoints→Edit/Call Macro	N/A
Window→	Window Active < window name>
Settings→High Level Debug	Debugger Level High_Level
Settings→Assembly Level Debug	Debugger Level Assembly
Settings→Debugger Options	N/A
Settings→Command Line	N/A

How Popup Menus Map to the Command Line

Code window pop-pup	Command Line
Set/Delete Breakpoint	Breakpt Instr or Breakpt Delete
Attach Macro	N/A
Edit Attached Macro	N/A
Edit source	N/A
Run until	Program Run Until # < line_number>

Default window pop-up	Command Line
Highlight/Toggle Window	Window Active/ < window_name>
	Window Toggle_View
Remove Window	N/A

Breakpoint window pop-up	Command Line
Delete Breakpoint	Breakpt Delete < brkpt_nmbr>
Delete All Breakpoints	Breakpt Clear_All

Monitor window pop-pup	Command Line
Delete Variable	Expression Monitor Delete < number>
Delete All Variables	Expression Monitor Clear_All

Backtrace window pop-up	Command Line
Disp Source at Stack Level	Program Context Set @< level>
Disp Vars at Stack Level	Program Context Expand @< level>
Run Until Stack Level	Program Run Until @< level>

Status Line Popup	Command Line
Command Line On/Off	N/A
Remove Temporary Message	N/A

Command Line Popup	Command Line
Forward Tab	< Tab>
Backward Tab	< Shift> -< Tab>
Execute Command	< Return>
Clear to End of Line	< Ctrl> -E
Clear Entire Line	< Ctrl> -U
Command Line On/Off	N/A

Command Summary

Breakpoint Commands

Breakpoint commands control execution of a program.

Command	Definition
Breakpt Access	Set a breakpoint on access (read/write) of an address
Breakpt Clear_All	Clear all breakpoints
Breakpt Delete	Delete specified breakpoints
Breakpt Instr	Set an instruction breakpoint
Breakpt Read	Set a breakpoint on a read from an address
Breakpt Write	Set a breakpoint on a write to an address

Session Control Commands

The session control commands select debugger operating modes, set debugger session options, define and display macros, allow access to the host operating system, and end debugger sessions.

Command	Definition
Debugger ?	Access debugger on-line help
Debugger Directory	Display or change present working directory
Debugger Execution Display_Status	Display current directory and files in use
Debugger Execution IO_System	Control debugger simulated I/O
Debugger Execution Load_State	Restore previously saved debugger session
Debugger Execution Reset_Processor	Simulate microprocessor reset
Debugger Execution Save_State	Save current debugger session
Debugger Host_Shell	Enter HP-UX operating system environment
Debugger Level	Select debugger mode (high-level or assembly)
Debugger Macro Add	Create a macro
Debugger Macro Call	Call a macro
Debugger Macro Display	Display macro source code
Debugger Option	Set or list debugger options for this session
Debugger Pause	Pause debugger session
Debugger Quit	Terminate a debugging session

Expression Commands

Expression commands calculate expression values, print formatted output to a window, and monitor variables.

Command	Definition
Expression C_Expression	Calculate the value of a C expression
Expression Display_Value	Display the value of an expression or variable
Expression Fprintf	Print formatted output to a window
Expression Monitor Clear_All	Discontinue monitoring all variables
Expression Monitor Delete	Discontinue monitoring specified variables
Expression Monitor Value	Monitor variables
Expression Printf	Print formatted output to Journal window

File Commands

File commands read and process command files, open files or devices for writing, log debugger commands to a file, and save debugger startup parameters.

Command	Definition
File Command	Read in and process a command file
File Error_Command	Set command file error handling
File Journal	Copy Journal window output to a journal file
File Log	Record debugger commands/errors in a file
File Startup	Save the default startup options
File User_Fopen	Open a file or device for read or write access
File Window_Close	Close the file associated with a window number

Memory Commands

Memory commands do operations on the target microprocessor's memory.

Command	Definition
Memory Assign	Change the values of memory locations
Memory Block_Operation Copy	Copy a memory block
Memory Block_Operation Fill	Fill a memory block with values
Memory Block_Operation Match	Compare two blocks of memory
Memory Block_Operation Search	Search a memory block for a value
Memory Block_Operation Test	Examine memory area for invalid values
Memory Display	Display memory contents
Memory Inport Assign	Set or alter input port status
Memory Inport Delete	Delete an input port
Memory Inport Rewind	Rewind input file associated with input port
Memory Inport Show	Display input port buffer values
Memory Map Guarded	Prevent access to memory locations
Memory Map Read_Only	Prevent writing to memory locations
Memory Map Show	Display current memory map assignments
Memory Map Write_Read	Allow access to memory locations
Memory Outport Assign	Set or alter output port status
Memory Outport Delete	Delete an output port
Memory Outport Rewind	Rewind output file associated with output port
Memory Outport Show	Display output port buffer values
Memory Register	Change the contents of a register
Memory Unload_BBA	Unload BBA data from program memory

Program Commands

Program commands load and execute programs, control program execution, display source code and program variables, and set or cancel program interrupts.

Command	Definition
Program Context Set	Specify current module and function scope
Program Context Display	Display all local variables of a function
Program Context Expand	Display all local variables of a function at the specified stack (backtrace) level
Program Display_Source	Display C source code
Program Find_Source Occurrence	Find first occurrence of a string
Program Find_Source Next	Find next occurrence of a string
Program Interrupt Add	Simulate an interrupt
Program Interrupt Remove	Cancel all pending interrupts
Program Load	Load an absolute file for debugging
Program Pc_Reset	Reset the program starting address
Program Run	Start or continue program execution
Program Step	Execute a number of instructions or lines
Program Step With_Macro	Execute macro after each instruction step

Symbol Commands

Symbol commands add, remove, and display symbols.

Command	Definition
Symbol Add	Add a symbol to the symbol table
Symbol Browse	Browse C++ class
Symbol Display	Display symbol, type, and address
Symbol Remove	Delete a symbol from the symbol table

Window Commands

Window commands do operations on the debugger windows.

Command	Definition
Window Active	Activate a window
Window Cursor	Set the cursor position for a window
Window Delete	Remove a user-defined window or screen
Window Erase	Clear data from a window
Window New	Make a new screen or window
Window Resize	Change the size of a window
Window Screen_On	Activate a screen
Window Toggle_View	Select the alternate display of a window

Breakpt Access

```
 ( Breakpt )──→( Access )─┐
                          │
 ┌────────────────────────┘
 │
 └→┌────────┐──────────────────────────────────────────┌──────────┐
   │ <addr> │                                           │ <Return> │
   └────────┘──┐                         ┌──            └──────────┘
              ┌──────────┐          ┌──────────────┐
              │ ..<addr> │          │ ;<macro_call>│
              └──────────┘          └──────────────┘
```
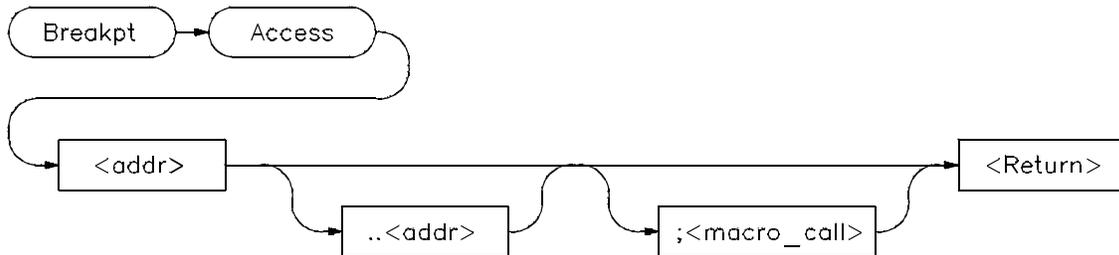
The Breakpt Access command sets an access breakpoint at the specified memory location (< addr>) or range (< addr> ..< addr>). The access breakpoint halts program execution each time the target program attempts to read from or write to the specified memory location or range. Memory locations may contain code or data.

You can attach a macro to a breakpoint using the optional < macro_call> parameter. See the chapter titled "Using Macros and Command files".

Each time the debugger detects an access of the address or range, it does the following:

1   Suspends program execution.

Execution will stop immediately following the current instruction.

2   Executes a macro (if you attached one to the breakpoint). Depending on the macro return value, the debugger does one of the following actions:

– If the macro return value is true (nonzero), the debugger resumes execution with the next instruction after the instruction that caused the read or write to the memory location. No breakpoint information is displayed.

– If the macro return value is false (zero), the debugger returns to command mode and displays breakpoint information.

3   Returns to command mode if no macro was attached and displays breakpoint information.

See Also

Breakpt Clear_All	Breakpt Write
Breakpt Delete	Program Run
Breakpt Instr	Program Step
Breakpt Read	

Examples

To set a breakpoint on accesses of addresses 'assign_vectors' through 'assign_vectors' + 16:

Breakpt **A**ccess &assign_vectors..+16

To set a breakpoint on access of the address of the variable 'current_temp':

Breakpt **A**ccess &current_temp

To stop program execution when the value of variable system_running is set or read as TRUE:

Breakpt **A**ccess &system_running; when (system_running==1)

The predefined macro 'when' is executed when the breakpoint is encountered.

Breakpt Clear_All

| Breakpt | ——→ | Clear_All | ——→ | <Return> |

The Breakpt Clear_All command clears (removes) all defined breakpoints.

See Also

Breakpt Access	Breakpt Write
Breakpt Delete	Program Run
Breakpt Instr	Program Step
Breakpt Read	

Examples To remove all defined breakpoints:

Breakpt **C**lear_all

Breakpt Delete

```
( Breakpt ) → ( Delete ) ──┐
                            │
  ┌─────────────────────────┘
  │
  └→ <brkpt_nmbr> ──┬──────────────────────→ <Return>
                    │                      │
                    └→ ..<brkpt_nmbr> ─────┘
```
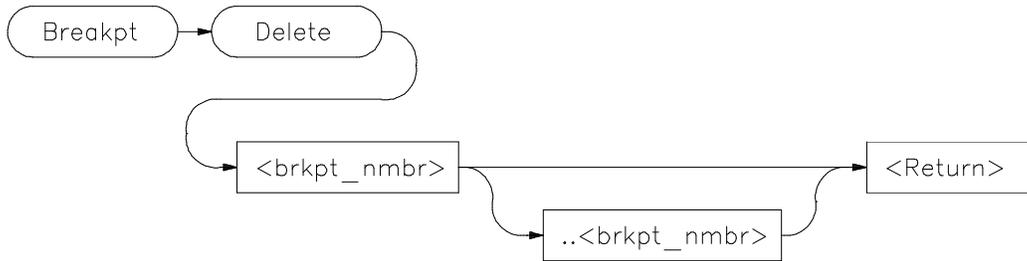
The Breakpt Delete command deletes (removes) one or more previously set breakpoints. When you set a breakpoint, the debugger assigns it a breakpoint number. Use this breakpoint number (< brkpt_nmbr>) to remove a specific breakpoint. You can delete a group of breakpoints by specifying a range of breakpoint numbers (< brkpt_nmbr> ..< brkpt_nmbr>). The debugger displays the breakpoint numbers in the Breakpoint window.

When you remove a breakpoint, the Breakpoint window displays the remaining breakpoints. Any breakpoints following the one removed are renumbered.

See Also

Breakpt Access   Breakpt Write
Breakpt Clear_All   Program Run
Breakpt Instr   Program Step
Breakpt Read

Examples   To delete breakpoint number 2:

Breakpt **D**elete 2

To delete breakpoint numbers 3 through 5:

Breakpt **D**elete 3..5

Breakpt Instr

```
  ( Breakpt ) → ( Instr )
          ┌──────────────────────────────────────────────────────┐
          └→ [ <addr> ] ──┬─ [ ..<addr> ] ─┬─┬─ [ ;<macro_call> ] ─┬─→ [ <Return> ]
                          └────────────────┘ └────────────────────┘
```
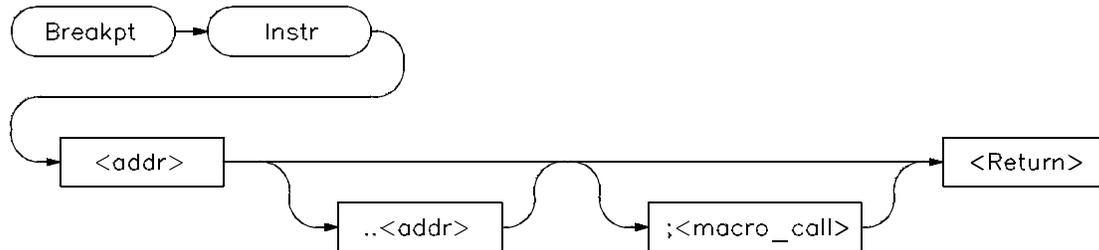
The Breakpt Instr command sets an instruction breakpoint at a specified
memory location (< addr>) or range (< addr> ..< addr>). The instruction
breakpoint halts program execution each time the target program attempts to
execute an instruction at the specified memory location(s). If you specify a
range, the debugger sets breakpoints on the first byte of each instruction
within the specified range or (in high-level mode) the first instruction of each
line within the range.

If you set a breakpoint for an overloaded C+ + function, the debugger will ask
you to choose which definition of the function to use. You can also specify the
argument type of the function definition in parentheses after the function
name in the Breakpt Instr command.

You can attach a macro to a breakpoint using the optional < macro_call>
parameter. See the "Using Macros and Command Files" chapter.

The debugger performs the following actions when it encounters an
instruction breakpoint:

1   Suspends program execution before the program executes the instruction
at the breakpoint address.

2   Executes a macro (if you attached one when you set the breakpoint).
Depending on the macro return value, the debugger does one of the
following actions:

–   If the macro return value is true (nonzero), the debugger resumes
execution starting at the instruction where the break occurred. No
breakpoint information is displayed.

– If the macro return value is false (zero), the debugger returns to command mode without executing the instruction where the break occurred and displays breakpoint information.

3   Returns to command mode without executing the instruction where the break occurred if no macro was attached and displays breakpoint information.

See Also

Breakpt Access	Breakpt Write
Breakpt Clear_All	Program Run
Breakpt Delete	Program Step
Breakpt Read	

Examples   To set an instruction breakpoint at line 82 of the current module:

Breakpt **I**nstr #82

To set an instruction breakpoint at line 83 of the current module only when the system is running (using the predefined macro 'when'):

Breakpt **I**nstr #83; when (system_running)

To set an instruction breakpoint starting at address 10deh and ending at address 10e4h:

Breakpt **I**nstr 10deh..10e4h

To set instruction breakpoints beginning on lines 15 through 25 of module 'initSystem':

Breakpt **I**nstr initSystem\#15..#25

Breakpt Read

```
┌──────────┐   ┌──────┐
│ Breakpt  │──▶│ Read │──┐
└──────────┘   └──────┘  │
                         │
┌────────────────────────┘
│
│  ┌────────┐                                              ┌──────────┐
└─▶│ <addr> │──────────────────────────────────────────┬─▶│ <Return> │
   └────────┘                                           │  └──────────┘
           │   ┌───────────┐        ┌──────────────────┐│
           └──▶│ ..<addr>  │───┬───▶│ ;<macro_call>    ├┘
               └───────────┘   │    └──────────────────┘
                               │
```
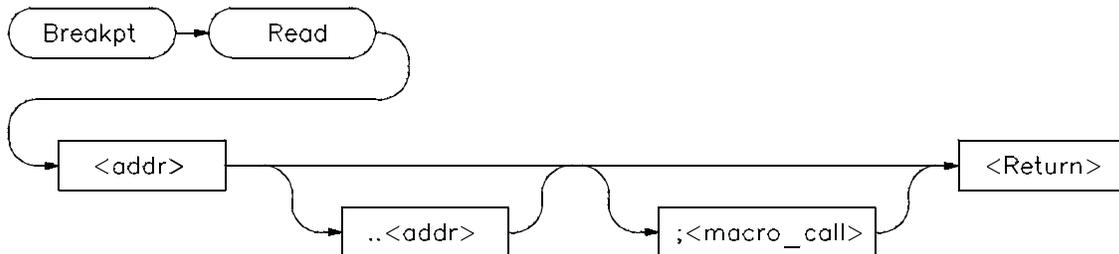
The Breakpt Read command sets a read breakpoint. The read breakpoint halts program execution each time the target program attempts to read data from the specified memory location (< addr>) or range (< addr> ..< addr>).

The Breakpt Read command behaves just like the Breakpt Access command.

See Also   Breakpt Access

Examples   To set a breakpoint on reads from variable 'system_running':

Breakpt **R**ead &system_running

To set a read breakpoint starting at the address of variable 'current_temp' and ending 8 bytes after the address of 'current_temp':

Breakpt **R**ead &current_temp..+8

To stop program execution when the value of variable system_running is read as TRUE:

Breakpt **R**ead &system_running; when (system_running==1)

Breakpt Write
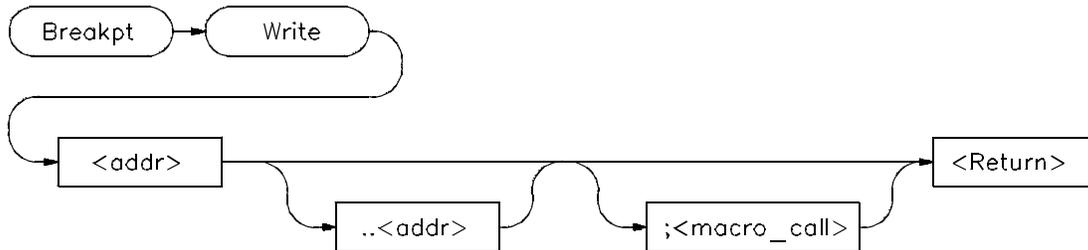
The Breakpt Write command sets a write breakpoint. The write breakpoint halts program execution each time the target memory attempts to write data to the specified memory location (< addr>) or range (< addr> ..< addr>).

The Breakpt Read command behaves just like the Breakpt Access command.

See Also   Breakpt Access

Examples   To set a breakpoint to occur when the program writes a false value to variable 'system_is_running':

`Breakpt Write &system_running; when (system_running==00)`

To set a write breakpoint starting at the address of global variable 'current_temp' and ending 8 bytes after the address of 'current_temp':

`Breakpt Write &current_temp..+8`

Debugger Directory
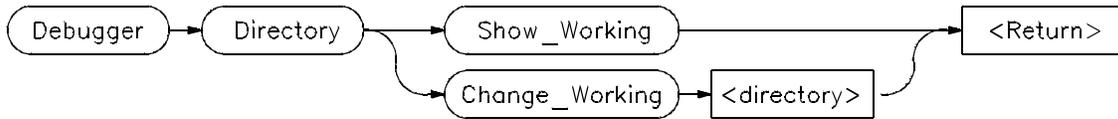
The Debugger Directory command displays or changes the current working directory. When you specify the *Show_Working* parameter, the debugger displays the current working directory in the journal window. When you specify the *Change_Working* parameter with a directory name, the debugger makes that directory the current working directory.

Examples
To display the current working directory:

Debugger **D**irectory **S**how_Working

To change the current working directory to /users/project/sources:

Debugger **D**irectory **C**hange_Working /users/project/sources

Debugger Execution Display_Status

Debugger → Execution → Display_Status → <Return>
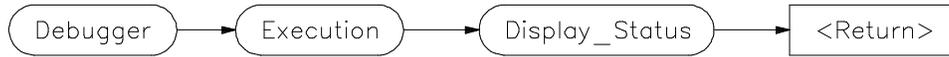
The Debugger Execution Display_Status command activates the debugger View window and displays the following status information:

Version of debugger
Current working directory
Current log file
Current journal file
Startup file used in current debug session
Loaded absolute files

If no files have been loaded, the absolute file will be missing from the display. If multiple executable files have been loaded using the Program Load Append command, they will be displayed in the View window. You may need to toggle the window (click on the window border) to see all of the files.

Example

To display product version, current working directory, and current log, journal, startup, and absolute files in the View window:

Debugger **E**xecution **D**isplay_Status

Debugger Execution IO_System

deidmn
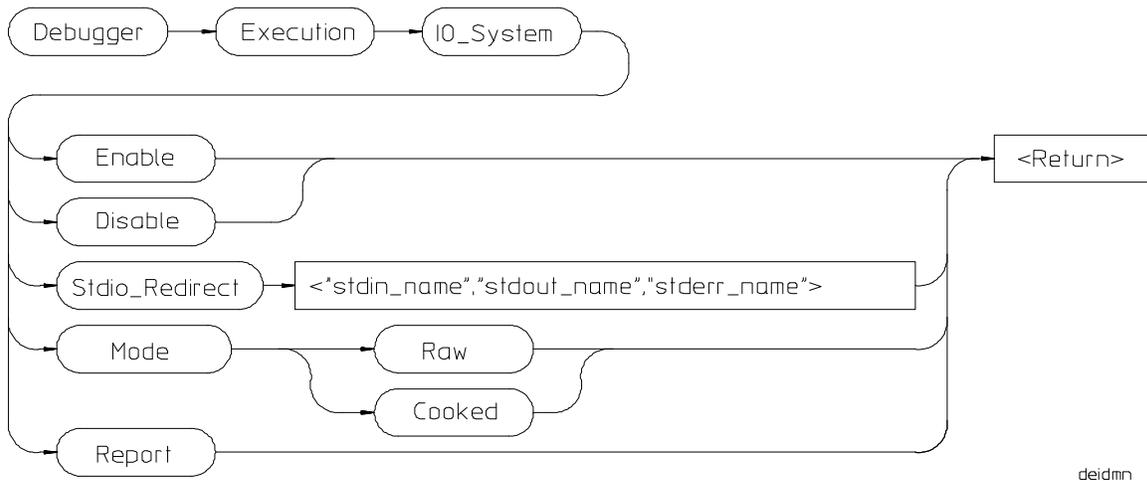
The Debugger Execution IO_System command enables you to configure the
simulated I/O system to use the host system keyboard, display, and file system
to simulate I/O devices for your target program.

Debugger Execution IO_System Enable

The Debugger Execution IO_System Enable command enables the debugger
simulated I/O system.

Debugger Execution IO_System Disable

The Debugger Execution IO_System Disable command disables the debugger
simulated I/O system.

Debugger Execution IO_System Stdio_Redirect

The Debugger Execution IO_System Stdio_Redirect command allows you to
define the standard I/O input (< stdin_name>), output (< stdout_name>),
and error (< stderr_name>) files/devices. These are file/device names in the
host computer file system. Two special filenames allow you to access the

system keyboard (/dev/simio/keyboard) and the system display
(/dev/simio/display).

Debugger Execution IO_System Mode

The Debugger Execution IO_System Mode command selects how keyboard
I/O input is processed. Keyboard I/O may be either cooked or raw.

Cooked Mode. In cooked mode, the target program gets input from the
keyboard in the form of lines. Editing operations, such as backspace, line kill,
etc., on input is done by the debugger. When **Return** or **CTRL D** is entered,
the line is passed to the target program by the simulated I/O system. The
keyboard input is echoed to the screen during the editing operation. If
program execution is interrupted by entering < **Ctrl> -C** before the line is
entered, the characters on the input line are lost.

Raw Mode. In raw mode, each keystroke is passed from the keyboard to the
simulated I/O system with no processing. No carriage return is needed to enter
characters and no editing operations are available. In the raw mode any
character is valid, including *CTRL D*. No characters are echoed to the screen
upon entry. The only special character that cannot be sent to the target
program is < **Ctrl> -C** which is used to interrupt the debugger's execution of
the program.

Debugger Execution IO_System Report

The Debugger Execution IO_System Report command displays the status of
the simulated I/O system.

See Also   The "Using Simulated I/O" section in the "Viewing Code and Data" chapter.
The "Environment-Dependent Routines" chapter in the *68030 C Cross
Compiler Reference* manual.

Examples   To enable simulated I/O:

Debugger **E**xecution **I**O_System **E**nable

To disable simulated I/O:

Debugger **E**xecution **I**O_System **D**isable

To redirect the standard input file to the keyboard, the standard output file to the display, and the standard error file to file '/users/project/errorfile':

```
Debugger Execution IO_System Stdio_Redirect
"/dev/simio/keyboard","/dev/simio/display",
"/users/project/errorfile"
```

To redirect the standard input file to 'temp.dat', the standard output file to 'cmdout.dat', and the standard error file to file 'errorlog.err':

```
Debugger Execution IO_System Stdio_Redirect
"temp.dat","cmdout.dat","errorlog.err"
```

To set data input mode to cooked:

```
Debugger Execution IO_System Mode Cooked
```

Debugger Execution Load_State

```
( Debugger )──▶( Execution )──▶( Load_State )──────────────────────────────┌──────────┐
                                              │                             │ <Return> │
                                              │       ┌──────────────┐      │          │
                                              └──────▶│ <save_file>  │──────┘
                                                      └──────────────┘
```
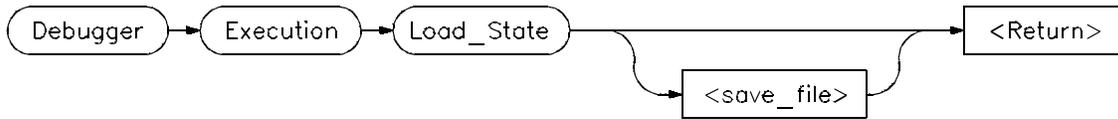
The Debugger Execution Load_State command restores the memory contents
and register values saved with the debugger/simulator Debugger Execution
Save_State command. If you do not specify a file name (< save_file>), the
debugger uses the default file *db68040.sav*.

Example   To restore memory contents and register values saved in save file "session1":

Debugger **E**xecution **L**oad_State session1

Debugger Execution Reset_Processor

```
( Debugger ) ──▶ ( Execution ) ──▶ ( Reset_Processor ) ──▶ [ <Return> ]
```

This command simulates a microprocessor reset. It does the following:

1   The program counter is loaded from exception vector 1 at location 4 in memory.

2   The interrupt stack pointer is loaded from exception vector 0 at location 0 in memory.

3   The status register is reset as follows;
   – the trace bits are cleared,
   – the supervisor bit is set to 1,
   – the master bit is set to 0,
   – the interrupt priority mask is set to level 7.

4   All other bits in the status register are set to 0.

5   The vector base register is set to 0.

6   The cache control register is set to 0.

7   The cycle count (*@cycles*) is set to zero.

8   Any pending interrupt or exception is cleared.

9   Registers A0-A6 and D0-D7 are set to 0.

Note   This command does not re-initialize memory. Use the **Program Load New Code_Only** command to reset C variables.

See Also   Program Pc_Reset

Example   To reset the microprocessor:

Debugger **E**xecution **R**eset_Processor

Debugger Execution Save_State

```
( Debugger ) → ( Execution ) → ( Save_State ) ────────────→ [ <Return> ]
                                      └→ [ <save_file> ] →┘
```
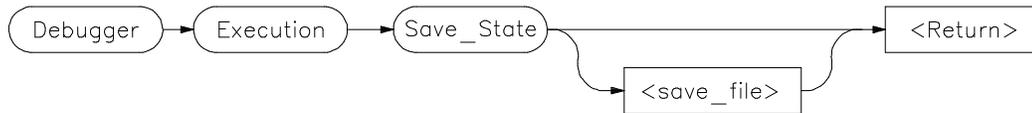
The Debugger Execution Save_State command saves the current memory contents and register values. These values can be restored at a later time by executing the Debugger Execution Load_State command. If a file name (< save_file>) is not specified, the default file name *db68040.sav* is used.

See Also   Debugger Execution Load_State

Example   To save the current memory contents and register values in file "session1.sav":

Debugger **E**xecution **S**ave_State session1

Debugger Host_Shell
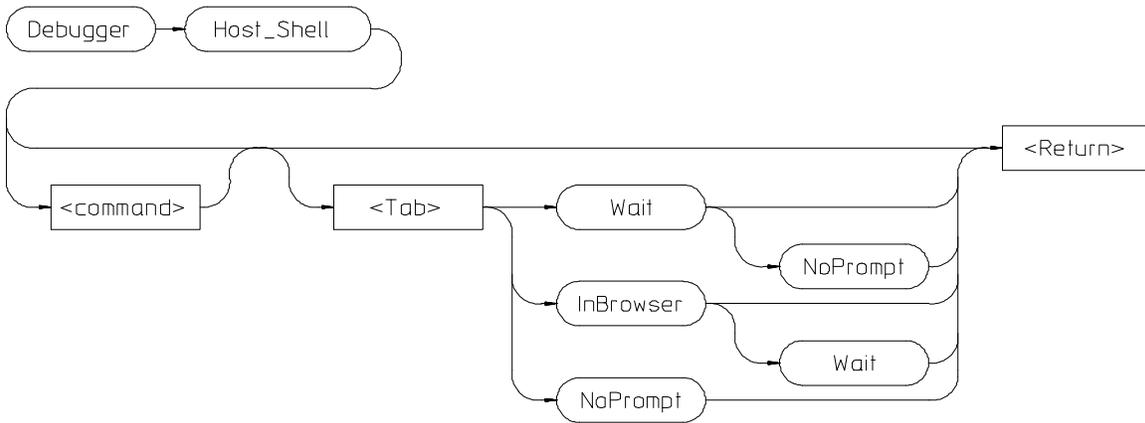
The Debugger Host_Shell command enables you to temporarily leave the debugging environment by forking an operating system shell or to execute a single UNIX operating system command from within the debugger. The type of shell forked is based on the shell variable SHELL. In this mode, you may enter operating-system commands. To return to the debugger, enter **CTRL D** or type **exit** and press the **Return** key.

You can execute operating system commands from within the debugger by entering a single operating system command with the debugger *Debugger Host_Shell* command. If you are using the graphical interface, the operating system command is executed in a "cmdscript" window. Press **< Return>** to close the window. If you are using the standard interface, *stdout* from the command is written to the Journal window and *stderr* is not captured. Commands writing to *stderr* will corrupt the display. Interactive commands **cannot** be used in this mode.

The following options are available only in the graphical user interface:

InBrowser

Directs stderr and stdout of the command into text browser windows.

Wait

Suspends the interface until the command completes.

NoPrompt

When the command completes, the "cmdscript" window is closed immediatelly.

See Also   Debugger Quit

Examples   To temporarily exit the debugger to the UNIX operating system command mode:

Debugger **H**ost_Shell

To write the current working directory to the journal window:

Debugger **H**ost_Shell pwd

Debugger Help

| Debugger | ——————————→ | ?(Help) | ——————————————→ | \<Return\> |

This command displays the on-line help screen. The debugger provides on-line help for all debugger commands, debugger command arguments, and debugger function keys. You can access on-line help by entering the command **Debugger ?** or by pressing the **F5** function key.

If you are using the graphical interface, a Help dialog box will be displayed. If you are using the standard interface, a menu will appear in the display area.

If you enter the command *Debugger ?* in the standard interface, the debugger puts the cursor at the top of the topic list in the help menu. If you press the **F5** function key, the debugger puts the cursor at the entry for the command displayed on the command line (if one is displayed). Otherwise, the cursor is positioned at the top of the topic list. You can select topics from the help menu in two ways:

- Use the cursor keys to move to the desired topic and press the **Return** key.

- Type the first letter of the desired topic. This positions the cursor at that topic. Then press the **Return** key.

Use the **Return** key to see more topics in alphabetical order.

To exit help in the standard interface, press the **Esc** (escape) key twice or press function key **F5**.

Example   To display the debugger help screen:

```
Debugger ?
```

Debugger Level

```
( Debugger ) → ( Level ) ─────────────────────→ | <Return> |
                        → ( Assembly ) →
                        → ( High_Level ) →
```
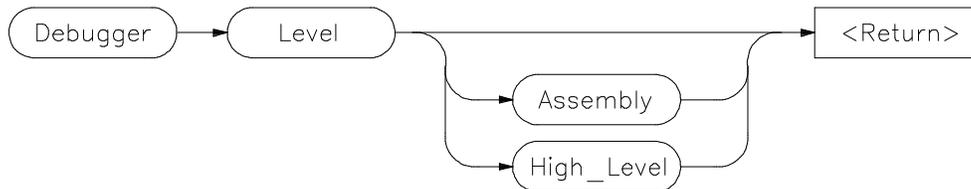
The Debugger Level command selects either high-level mode or
assembly-level mode for debugging. When debugging programs containing C
modules, you can switch back and forth between the two modes. If the
program contains no high-level modules accessible to the debugger, the
debugger displays an error message if you attempt to select high-level mode.

If no parameters are specified with this command, the mode is switched back
and forth between the two modes, performing the same function as the *F3*
function key.

Examples   To select the assembly-level debug mode:

Debugger **L**evel **A**ssembly

To select the high-level debug mode:

Debugger **L**evel **H**igh_Level

To switch to the alternate debug mode:

Debugger **L**evel

Debugger Macro Add
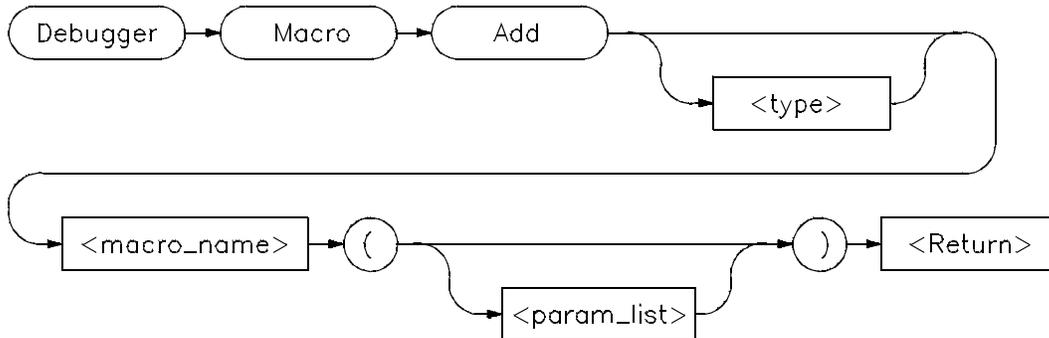
The Debugger Macro Add command defines a macro.

The name of the macro is specified by *<macro_name>*. The result type of the macro is specified by *<type>*. If a type is not specified, it defaults to type int. A parenthesized list of parameters *(<param_list>)* may optionally follow the macro name. Parameter names must be composed of alphanumeric characters. A maximum of 40 parameters is allowed.

When you enter the Debugger Macro Add command, the Journal window is automatically enlarged, and the debugger displays the macro text prompt character (>) indicating that you can enter the macro body.

Note   If the stdio screen or a user-defined screen is active when the Debugger Macro Add command is issued, the Journal window will not become active. Keyboard input at this point will be interpreted by the debugger as the macro definition.

To terminate the macro definition, a period (.) must be entered as the first and only character on a line.

The macro definition consists of all lines entered after the macro name and before the terminating period. The macro definition consists of the source lines of the macro (the macro body) and optional formal arguments. The syntax for the macro body is:

```
{macro_statement; [macro_statement;]...}
```

The curly braces ({ }) are required punctuation. Formal arguments can be used throughout the macro definition, and are later replaced by the actual arguments in the macro call.

The maximum number of characters that can be entered on a line in a macro definition is 255. When entering macros interactively, the debugger does not respond to more than 78 characters on a line. When reading a command file, the debugger stops recognizing characters after 255 characters have been read on a line.

The maximum number of lines allowed in a macro depends on the complexity of the lines. Macros with too many lines (too complex) will fail. Error 92 "*Not enough memory for expression*" will be displayed.

A macro is similar to a C function. The body can contain any legal C statement (except the SWITCH and GOTO statements). The statements IF, ELSE, DO, WHILE, FOR, RETURN, BREAK, and CONTINUE can be used to control program flow within a macro, just as in C. Macros have return types and can be used in expressions.

Note   Debugger commands may be used in macro definitions; they are indicated by placing a dollar sign ($) at the beginning and the end of a command sequence. For example, the following command sequences are legal in macro definitions:

```
$Program Find_Source Occurrence Forward system$;
```

   or

```
$
Memory Assign Long &time=12
Program Find_Source Occurrence Forward system
$;
```

Macros can be executed by specifying the macro name on the command line in a Debugger Macro Call command, in an expression, or with a breakpoint command.

Macros can be removed using the command:

```
Symbol Remove <macro_name>
```

See Also   Breakpt Access
Breakpt Instr
Breakpt Read
Breakpt Write

Debugger Macro Call
Debugger Macro Display
Program Run
Symbol Remove
The "Using Macros and Command Files" chapter
The "Predefined Macros" chapter in this manual.

Example

```
Debugger Macro Add int power(x, y)
int    x;
int     y;
{
  int     i;                /* Loop counter */
  int multiplier;           /* Value x is multiplied by */

  /* Multiply x by itself y -1 times */
  for (i = 1, multiplier = x; i < y; i++)
    x *= multiplier;

  /* Return x ^y */
  return x;
}
.


Debugger Macro Add void stackchk()
{
  /* The symbols 'stack' and 'TopOfStack' exist in the compiler's */
  /* environment library, and are addresses which indicate the    */
  /* bottom and the top of the system stack. The symbol @sp is a */
  /* debugger reserved symbol which contains the current value of */
  /* the processor's stack pointer.                              */

  $Expression Printf "%d bytes of stack used", TopOfStack - @sp$;
  $Expression Printf "%d bytes of stack available", @sp - stack$;
}
.
```

Debugger Macro Call

```
( Debugger ) → ( Macro ) → ( Call ) → [ <macro_call> ] → [ <Return> ]
```

The Debugger Macro Call command calls a macro previously defined by the
Debugger Macro Add command or a macro built into the debugger.

See Also   Debugger Macro Add
Debugger Macro Display
Symbol Remove

Example   To call the previously defined macro 'stackchk()':

Debugger **M**acro **C**all stackchk()

Debugger Macro Display

```
   ( Debugger )  →  ( Macro )  →  ( Display )
```
```
   →  [ <macro_name> ]  →  [ <Return> ]
              →  [ ,<user_window_nmbr> ]
```
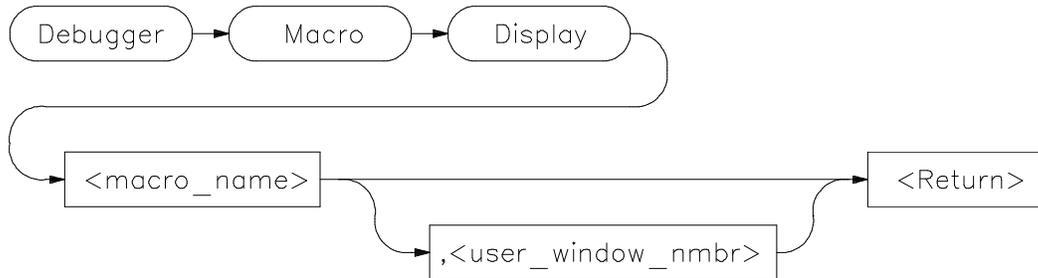
The Debugger Macro Display command displays the source code for the
named macro. If a window number is specified (< user_window_nmbr>), the
macro source is written to the file or user-defined window associated with the
number. If you do not specify a window number, the macro source is written to
the Journal window.

Macro source for built-in macros cannot be displayed.

See Also   Debugger Macro Add
                    File Command
                    Symbol Display

Examples   To display the source for macro 'stackchk' in user-defined window 57:

Debugger **M**acro **D**isplay stackchk,57

To display the source for macro 'stackchk' in the Journal window:

Debugger **M**acro **D**isplay stackchk

Debugger Option Command_Echo

```
Debugger → Option → Command_Echo →┬→ On ──┬→ <Return>
                                   └→ oFF ─┘
```
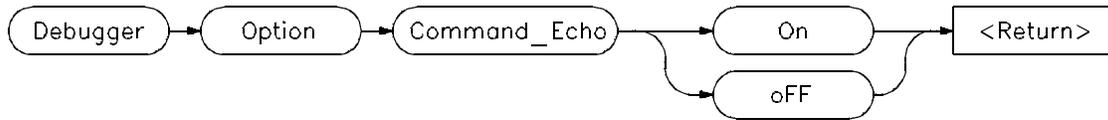
The Debugger Option Command_Echo command controls whether or not commands executed from a command file are echoed (copied) to the Journal window. If the *oFF* parameter is specified, only the results (if any) of a command are copied to the Journal window. If the *On* parameter is specified, both the command and its results (if any) are echoed to the Journal window. The default setting is *On*.

Examples

To turn OFF echo to the Journal window of commands executed from a command file:

Debugger **O**ption **C**ommand_Echo oFF

To turn ON echo to the Journal window of commands executed from a command file:

Debugger **O**ption **C**ommand_Echo On

Debugger Option General
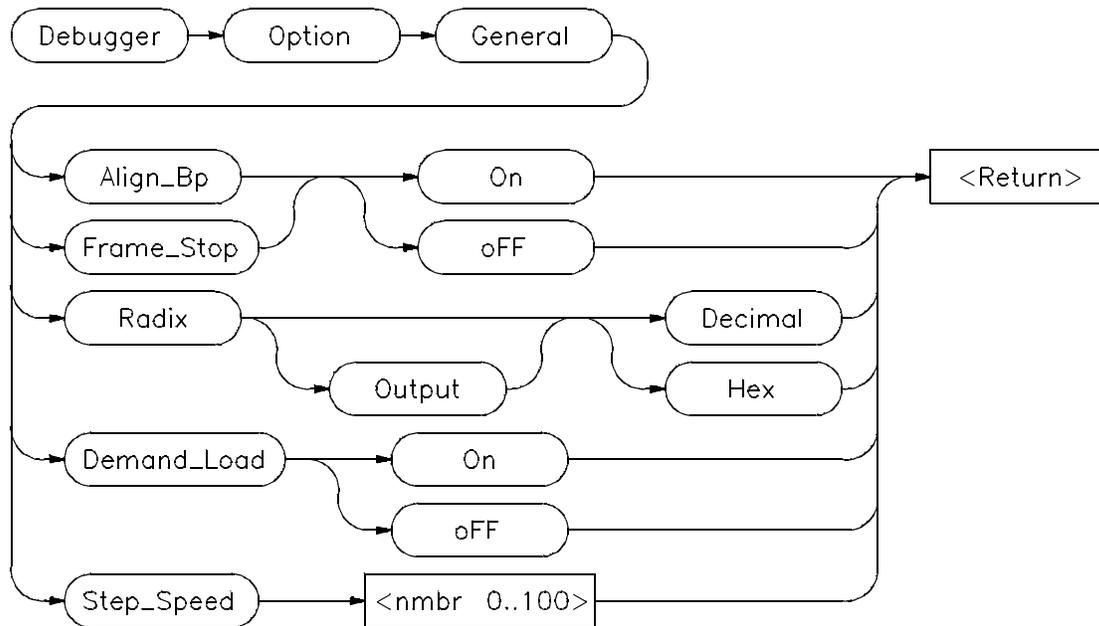
The Debugger Option General command changes the default values for the following debugger startup options for the current debugging session:

Align_Bp   Aligns breakpoints with processor instruction start

Frame_Stop   Controls stack walking

Demand_Load   Enables/disables demand loading of symbols

Processor   Sets the processor type

Radix   Interprets assembly-level numbers as decimal or hex

Exceptions   Controls behavior of exception processing

Step_Speed   Specifies the stepping speed

Use the Debugger Option List command to display the current option values.

To permanently change any option default values, first use the Debugger Option command to change the value(s) and then use the File Startup command to save the new default values in a startup file. See the File Startup command for more information.

Align_Bp

The Align_Bp option controls automatic alignment of low-level breakpoints and automatic alignment of disassembly. If the Align_Bp option is set to *On*, the debugger locates what it interprets as the starting address of all instructions in a module (by disassembling code from the beginning of the module). If you try to set the breakpoint at an address other than the start of an instruction, the debugger moves the breakpoint to the beginning of the next instruction and displays a warning. If you try to display memory mnemonically from an address other than the start of an instruction, the debugger moves the disassembly address to the beginning of an instruction. No Warning is displayed. If the Align_Bp option is set to *oFF*, the debugger lets you set the breakpoint at any address. The default setting is *oFF*.

Frame_Stop

When you set the Frame_Stop option to *On*, if the debugger encounters a bad stack frame, it displays only the valid stack frames below the bad frame in the Backtrace window. When you set the Frame_Stop option to *oFF*, the debugger displays all frames, including the bad frame. The default setting is *oFF*.

Demand_Load

When the Demand_Load option is set to *On*, the debugger loads some symbol information on an as-needed, demand basis rather than during the initial loading of the executable (.x) file. Symbol information for global symbols, local symbols in the source module containing main, and local symbols in assembly modules are loaded during the initial load of the executable file. Local symbols in C source modules other than that module which contains main are loaded when the debugger explicitly references the module or when the program is stopped with the program counter set to an address in the module. Demand loading lets you load and debug programs that you could not otherwise load because of very large amounts of symbol information. The default setting for Demand_Load is *oFF*.

There are several side effects of demand loading. The debugger command Memory Unload_BBA is disabled. Type mismatch errors may not be detected during the initial load of the executable (.x) file. Global symbols may have leading underscores stripped, depending on whether they were defined or referenced in a C or assembly source module.

Processor

The processor option selects a specific microprocessor for simulation. The microprocessor selected for simulation is displayed on the status line. The processor selections are:

68030 or 68040.

The default processor setting is 68040.

Radix

The radix option causes the debugger to interpret numeric literals, including integers and addresses, as either decimal or hexadecimal values. By default, numeric literals are interpreted as decimal values.

If you set *Radix* to hexadecimal, any number you want interpreted as decimal must be terminated with a *T* (for example, specify 32 as 32T).

Even if you select Hex, the following inputs will *not* be interpreted as hexadecimal: line numbers starting with "# ", variables in high-level expressions, and debugger variables including breakpoint numbers, viewport numbers, and data viewport line numbers. To cast a high-level expression as hexadecimal, use a leading "0x" or a trailing "h".

Binary numbers are not available when *Radix* is set to hexadecimal. Floating point and enumeration type values are not affected by the radix option.

The *Output* parameter lets you specify whether the output of the Expression Display_Value, Expression Monitor Value, and Program Context Expand command is displayed in decimal or hexadecimal format.

Exceptions

The exceptions option controls the behavior of exception processing by the debugger. In *Normal* mode, the debugger internal variable *@exc* is set to 2. This causes the debugger to allow exceptions to be handled as they would be by the processor. In *Report* mode, the debugger reports the exception type,

and where it occurred to the Journal window, and then handles the exception as the processor would. In the *Stop* mode, the debugger reports the exception to the Journal window and halts program execution. This is the default mode of operation.

Step Speed

The Step_Speed option specifies the stepping speed. The stepping speed can be in the range of 0 to 100 units. Higher numbers represent slower speeds. This option affects the Program Step command. The default value is 0.

See Also   File Startup
Debugger Option List

Example   To align assembly-level breakpoints at the beginning of an instruction:

Debugger **O**ption **G**eneral **A**lign_Bp **O**n

Debugger Option List

The Debugger Option List command lists the current debugger option values in the Journal window. The list will be similar to the sample list shown in the example.

See Also   Page 229 for an example of the list
Debugger Option Command_Echo
Debugger Option General
Debugger Option Symbolics
Debugger Option View
Settings→Debugger Options ...

Examples   To list the current debugger option settings in the Journal window:

Debugger **O**ption **L**ist

Debugger Option Symbolics
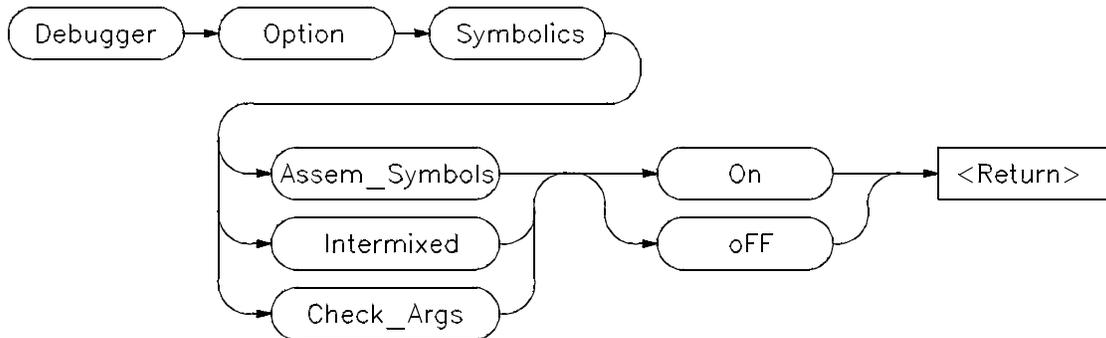
The Debugger Option Symbolics command changes the default values for the following debugger symbol options and C source line display options for the current debugging session:

Assem_Symbols   Displays symbols in assembly code

Intermixed   Intermixes C source with assembly code

Check_Args   Enables parameter checking in commands and macros

Use the Debugger Option List command to display the current option values.

To permanently change any option default values, first use the Debugger Option command to change the value(s) and then use the File Startup command to save the new default values in a startup file. See the File Startup command for more information.

Assem_Symbols

The Assem_Symbols option causes symbols instead of memory addresses to be displayed in the disassembled code whenever possible. Symbol names are placed to the right of the disassembled code for immediate values. This is done because there is no sure way of telling if the immediate value was represented by the symbol at assembly time. This option is set to *On* by default.

Intermixed

The Intermixed option intermixes C source code with the assembly code generated for each respective C statement. This option is off by default.

Check_Args

The Check_Args option controls parameter checking in commands and macros. If *oFF* is selected, the debugger does not do any argument checking. If *On* is selected, the debugger warns you when an assignment is made which contains a C type mismatch. This option is off by default.

See Also   File Startup

Examples   To display symbol names instead of address values in disassembled code:

Debugger **O**ption **S**ymbolics **A**ssem_Symbols **O**n

To turn OFF display of C source lines in assembly-level Code window:

Debugger **O**ption **S**ymbolics **I**ntermixed o**FF**

To enable debugger expression parameter checking:

Debugger **O**ption **S**ymbolics **C**heck_Args **O**n

Debugger Option View
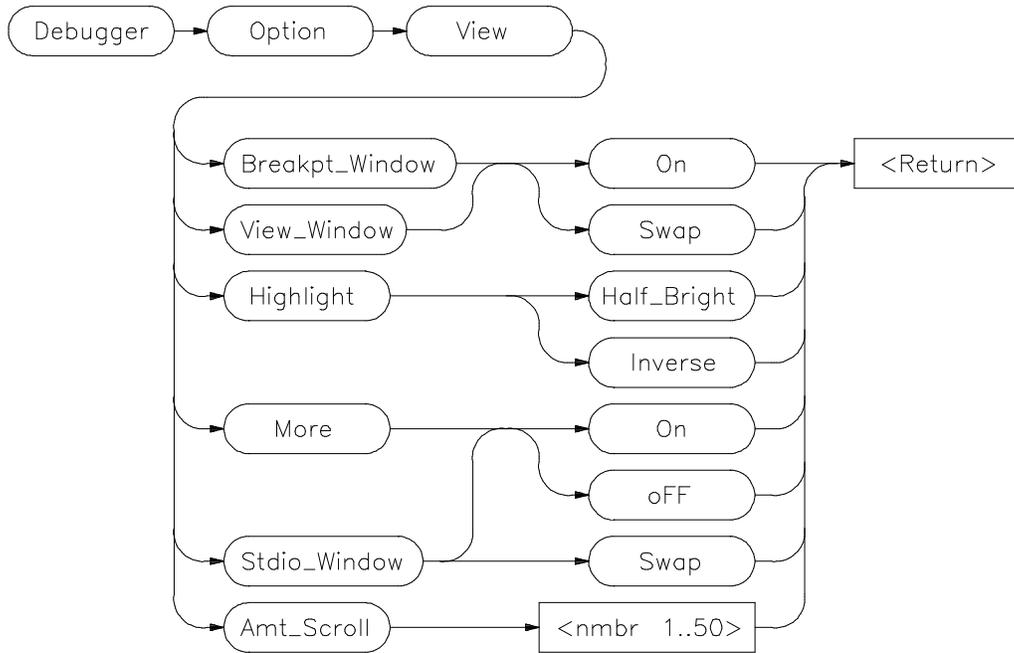
The Debugger Option View command changes the default values for the
following debugger display options for the current debugging session:

    Breakpt_Window
    View_Window
    Highlight
    More
    Stdio_Window
    Amt_Scroll

Use the Debugger Option List command to display the current option values.

To permanently change any of the default values, first use the appropriate
Debugger Option command to change the value(s) and then use the File
Startup command to save the new default values in a startup file. See the File
Startup command for more information.

Breakpt_Window

The Breakpt_Window option controls the display of the breakpoint window.

The *On* setting causes the Breakpoint window to be displayed at all times. The window may be hidden by other windows but will be displayed whenever a breakpoint is set or deleted.

If you specify the *Swap* setting, the window is not automatically displayed. You must set or delete a breakpoint or enter the Window Active Breakpoint command to display the window. The default setting is *Swap*.

View_Window

The View_Window option controls the display of the view window.

The *On* setting causes the View window to be displayed at all times. The window may be hidden by other windows but will be displayed whenever a Debugger Execution Display_Status command is executed.

If you specify the *Swap* setting, the window is not automatically displayed. You must enter the Debugger Execution Display_Status command or the Window Active View command to display the window. The default setting is *Swap*.

Highlight

The Highlight option determines whether highlighted information in debugger windows is displayed in half-bright video or inverse video. The default is Inverse.

More

The More option controls how information resulting from a debugger command is listed to the Journal window.

If the More option is *On*, information is listed one screen at a time in the Journal window, in the same way as the more command in the Unix operating system works.

If the More option is *OFF*, all information resulting from a debugger command is written to the display at once, making it difficult to view information greater than the number of lines available in the Journal window. The default setting is *On*.

Stdio_Window

The Stdio_Window option controls the display of the Stdio window.

The *Swap* setting causes the Stdio window to be displayed when a program writes to it and to be removed when the program returns to the command mode.

The *On* setting causes the Stdio window to be displayed at all times. The window may be hidden by other windows but will be displayed when a program is writing to it.

If the *oFF* setting is selected, the window is not automatically displayed. You must press function key **F6** or enter the command **Window Screen_On Stdio** to display the window.

The default setting is *Swap*.

Amt_Scroll

The Amt_Scroll option controls the amount that the Journal and Stdio windows are scrolled when written to. When the output reaches the bottom of the window, the data scrolls up one line by default. You can specify a number of lines from one to 50.

Examples

To set the Swap option so that the Breakpoint window is displayed only when the Window Active Breakpoint command is executed:

Debugger **O**ption **V**iew **B**reakpt_Window **S**wap

To set the View_Window option so that the view window is always displayed:

Debugger **O**ption **V**iew **V**iew_Window **O**n

Debugger Pause
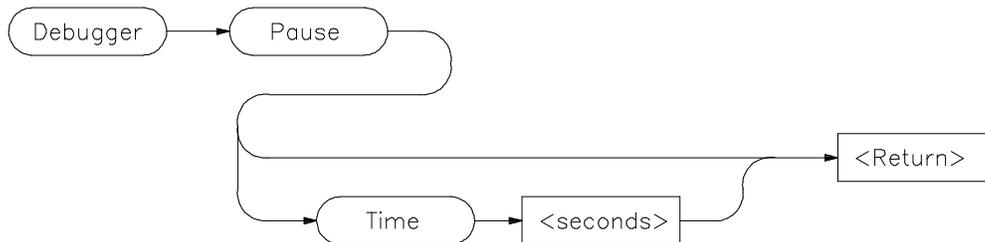
The Debugger Pause Time command pauses the debugger for the specified number of seconds or (if you enter the Debugger Pause command without the Time parameter) pauses the debugger until you press the space bar, **CTRL C**, or the escape key (**Esc**) twice.The Debugger Pause command is useful when executing command files.

See Also   File Command

Examples   To pause the debugger for ten seconds:

Debugger **P**ause **T**ime 10

To pause the debugger until the space bar, CTRL C, or Esc-Esc is pressed:

Debugger **P**ause

Debugger Quit

```
( Debugger ) ──→ ( Quit ) ──────────────────────→ [ <Return> ]
                            │                    │
                            └──→ ( Yes ) ──┘
```
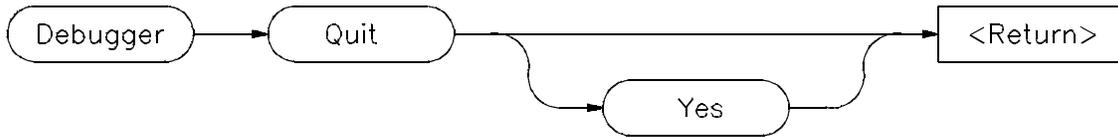
The Debugger Quit command ends a debugging session without saving the session. If you enter the command *Debugger Quit Yes*, the debugging session is immediately ended. If you enter the command *Debugger Quit* without an option, the debugger asks the question "*Are you sure?*". If you reply *yes*, the debugging session is ended. Otherwise the debugging session continues.

The Debugger Quit command does not save the debugging session. Use the File Startup command to save the current set of debugger startup options and window parameters in a startup file.

Note If you want to save the debugging session, use the Debugger Execution Save_State command to save the current memory contents and register values. This command is not available with the debugger/emulator products.

See Also Debugger Host_Shell

Examples To end the debugger/simulator session:

Debugger **Q**uit

The debugger will prompt you with the question "Are you sure?" before ending the session.

To terminate the debugging session immediately:

Debugger **Q**uit **Y**es

Expression C_Expression

```
( Expression ) ──────→ ( C_Expression ) ──────→ [ <C_expr> ] ──────→ [ <Return> ]
```
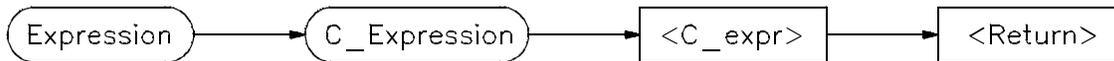
The Expression C_Expression command calculates the value of most valid C expressions or assigns a value to a variable. The result is displayed in floating point or in decimal, hexadecimal, and ASCII formats.

The Expression C_Expression command can be used to set C variables by specifying a C assignment statement. This command recognizes variable types, and the assignment expressions specified behave according to the rules of C.

Note

The Expression C_Expression command cannot evaluate conditionals of the form:

```
<expression>?<expression>:<expression>
```

Examples

To calculate the value of 'time' and display the result "data address 000091DC {time_struct}":

Expression **C**_Expression time

To calculate the value of member 'hours' of structure 'time' and display the result "4 0x04":

Expression **C**_Expression time->hours

To assign the value 1 to 'system_is_running' and display the result "1 0x01":

Expression **C**_Expression system_is_running = 1

Expression Display_Value

```
  ┌────────────┐      ┌──────────────┐
  │ Expression │ ───▶ │ Display_Value│ ──┐
  └────────────┘      └──────────────┘   │
                                         │
      ┌──────────────────────────────────┘
      │
      │      ┌──────────────┐                          ┌──────────┐
      └────▶ │    <expr>    │ ──────────────────┬────▶ │ <Return> │
             └──────────────┘                   │      └──────────┘
                      │     ┌──────────────┐    │
                      └───▶ │   ..<expr>   │ ───┘
                            └──────────────┘
```
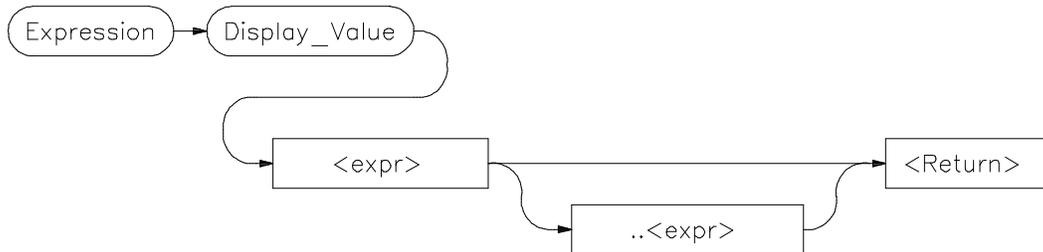
The Expression Display_Value command displays expressions and their values
in the Journal window. All expressions displayed with this command are
displayed according to their type as shown in the following list:

Type	Display Format
Ints	32-bit signed decimal numbers
Longs	32-bit signed decimal numbers
Shorts	16-bit signed decimal numbers
Chars	8-bit characters (unsigned hexadecimal numbers if not printable)
Pointers	32-bit unsigned numbers
Enums	Name of Enumerator constant (enumerator value if name not defined)
Arrays	All elements
Structures	All members
Quoted String	All characters as typed, in by double quotes (" ")
Hex Byte	8-bit hexadecimal
Hex Word	16-bit hexadecimal
Hex Double Word	32-bit hexadecimal
Float	32-bit floating point
Double	64-bit floating point

Note

The contents of an item such as an array is displayed instead of the C value of
the item, which is its address.

If an expression range is displayed, each value within the range is displayed according to the base type (if one exists). For example, if the variable *flags* is a character array, the following command results in elements *flags[10]* through *flags[30]* being displayed:

Expression **D**isplay_Value flags+10..+30

Note that the command first evaluates *flags[10]* to a character, and uses this as the base of the address range. *Flags[30]* is also evaluated to a character. It is used as the end of the address range.

Any expression can be type cast to display it in a different format. All values that make up a complex type are printed. For example, if the variable *count* is a long, the following statement displays it as a four-character array:

Expression **D**isplay_Value (char[4])&count

To display the contents of a character array as a string, cast the variable using the quoted string cast, as shown in the following example:

Expression **D**isplay_Value (Q S)buf

If the type of the expression is unknown, it defaults to type byte. See the "Expressions and Symbols in Debugger Commands" chapter for more information about type casting.

See Also

Expression Fprintf
Expression Monitor Value
Expression Printf
Memory Display

Examples

To display the value of the variable 'system_is_running': 01h

Expression **D**isplay_Value system_is_running

To display the address of the variable 'system_is_running': 000091F0

Expression **D**isplay_Value &system_is_running

To display the address of the C structure 'time': 000091DC

Expression **D**isplay_Value time

To display the values of the members of structure 'time':
hours   4
minutes   0
seconds   20

Expression **D**isplay_Value *time

To display the name of the current program module:

Expression **D**isplay_Value @module

To display the name of the current program function:

Expression **D**isplay_Value @function

Expression Fprintf
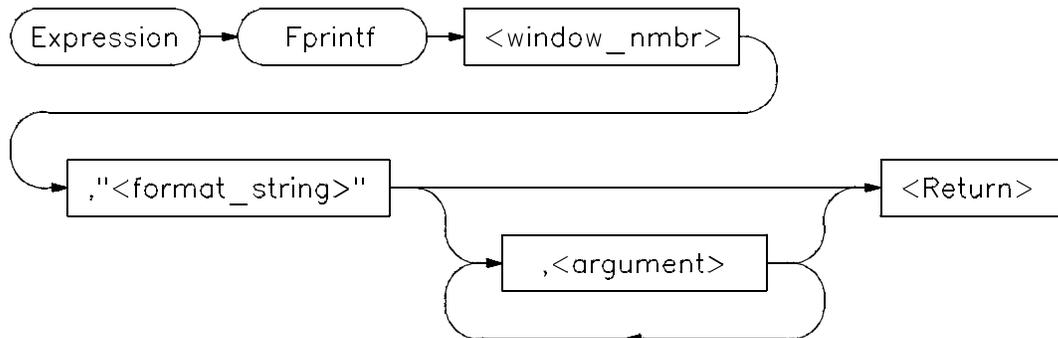
The Expression Fprintf command prints formatted output to the specified user-defined window. Formatted output may be written to a file that has been opened by the File User_Fopen command. The Expression Fprintf command is similar to the C fprintf function.

This command allows type conversions, scaling, and positioning of output in a file or in a window. The window number must have been previously assigned by a File User_Fopen or Window New command or the window number must be the log file number (28) or journal file number (29), if opened.

The command requires a format string as the second parameter. The format string may contain both text and argument conversion specifications. Whenever a conversion specification is encountered, the next argument is converted according to the specification, and the result is copied to the output window.

The conversion specifiers are similar to those in C and have the following format:

```
%[-] [digits] [.[digits]] [l] conversion_char
```

where:

%   indicates the start of a conversion specification.

–   indicates that the result of conversion is to be left-justified
                     within the field.

digits   is a string of one or more decimal characters. The first *digits*
                     is a minimum field width. The field will be at least this many
                     characters wide, padded if necessary. The padding is
                     normally on the left. When '–' is used, padding is on the
                     right. The field is padded with blanks unless the first digit in
                     digits is a *0*; then the field is padded with zeros.

.   separates two digit strings and must be specified if a second
                     digit string is used.

digits   (second occurrence) is the maximum field width. For
                     strings, it is the maximum number of characters to print; for
                     f and e notations, it is the maximum number of fractional
                     decimal places to print. For g notation, it is the number of
                     significant digits to be printed.

l   indicates that a conversion character (d, x, or u)
                     corresponds to a long argument.

Conversion Characters

Conversion characters are listed in the following table with a detailed
description of each character.

Char	Description
c	The argument is converted to character format.
d	The argument is converted to decimal format.
e, E	The float or double argument is converted to the format `[-]d.ddde+dd`, where the number of digits after the decimal point is equal to the precision. If precision is zero, no decimal point is printed. The default precision is 6. The E conversion character produces a number with E instead of e introducing the exponent. The exponent always contains at least two digits.

f   The double argument is converted to decimal notation in the format [–]ddd.ddd, where the number of digits after the decimal point is equal to the precision specification. If the precision is not specified, it is 6 by default; if the precision is explicitly zero, no decimal point appears. If there is a decimal point, at least one digit appears before it.

g, G   The double argument is printed in f or e notation, or in F or E notation when G is used. The precision specifies the number of significant digits. The notation used depends on the value converted; e or E notation will be used only if the exponent resulting from the conversion is less than –3 or greater than or equal to the precision. Trailing zeros are removed from the result; a decimal point appears only if it is followed by a digit.

h   The argument is either the debugger internal variable @HLPC, or a high level line number preceded by the # character. Source lines are formatted as strings according to %s rules. (Note: See @HLPC in the "Reserved Symbols" chapter of this manual.)

m   The argument is an instruction address. The disassembled instruction is treated as a string.

s   The argument is a string. The characters from the string are copied to the output until a NULL character is encountered or the maximum number of characters specified have been printed.

u   The argument is converted to unsigned decimal format.

v   The argument is displayed according to its type.

w   The argument is is a window number. The current contents of the window are written to the specified window.

X   The argument is converted to hexadecimal. Letters are displayed in upper case. *0x* is not printed before the value.

x   The argument is converted to hexadecimal. Letters are displayed in lower case.

%   The character % is substituted for the field. Any other non-conversion character following a % is printed. %%% is used to generate % in the output as a literal character.

Conversion characters are case-sensitive. Values printed in E notation have the following format:

```
[-]d.d...E{+|-}dd
```

Each *d* represents a decimal digit. The number is first scaled so that one digit appears to the left of the decimal point. The number of digits in the fractional part is six by default, or the maximum field width if specified. The sign of the mantissa is printed only if the number is negative. The sign of the exponent is always printed.

Values printed in F notation have the following format:

```
[-] d... . d...
```

Each *d* represents a decimal digit. The number of digits in the fractional part is six by default or the maximum field width if specified. The number of digits printed depends on the number of significant digits in the number.

Because floating point values are passed as parameters, they are converted to double precision. Parameters must be promoted to double precision values as a requirement of the C language. Other values passed as parameters may also be converted.

The Expression Fprintf command uses the format string to decide how many arguments to print. The number of conversion specifications must equal the number of arguments. If there are too many arguments, some of them will not be printed. If there are too few arguments, the value printed cannot be determined.

If the argument type does not correspond to its conversion field specification, arguments may be converted incorrectly.

See the Expression Printf command for details about conversion specifiers.

See Also   Expression Printf
File Journal
File Log
File User_Fopen
Window New

Examples   To print value of 'var' to user window 57 as a single character:

Expression **F**printf 57,"%c",var

To print the string in double quotes to user window 57 followed by the floating point value of 'temperature' with a precision of 2:

Expression **F**printf 57,"The value of 'temperature' is: %.2f \n",temperature

To print source line 24 to user window 55:

Expression **F**printf 55,"%h",#24

To print the contents of the assembly-level stack window to user window 256:

Expression **F**printf 256,"%w",14

Expression Monitor Clear_All

$$\boxed{\text{Expression}} \longrightarrow \boxed{\text{Monitor}} \longrightarrow \boxed{\text{Clear\_All}} \longrightarrow \boxed{\text{<Return>}}$$

The Expression Monitor Clear_All command stops monitoring of all expressions being monitored with the Expression Monitor Value command and removes all expressions from the Monitor window.

See Also   Expression Fprintf
Expression Monitor Delete
Expression Monitor Value
Expression Printf
Memory Display

Examples   To stop monitoring all expressions:

Expression **M**onitor **C**lear_All

Expression Monitor Delete

```
( Expression ) → ( Monitor ) → ( Delete )
                                                        
  → [ <display_nmbr> ] ──────────────────────→ [ <Return> ]
              → [ ..<display_nmbr> ] →
```
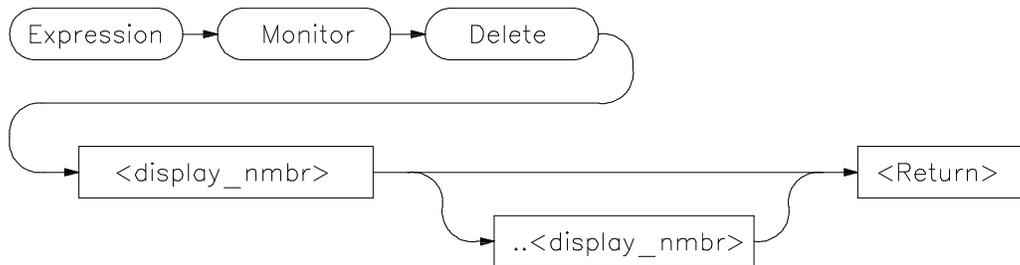
The Expression Monitor Delete command stops monitoring of specified expressions being monitored with the Expression Monitor Value command and removes those expressions from the Monitor window.

When an expression is monitored using the Expression Monitor Value command, it is assigned a line number, which is displayed in the Monitor window. These assigned line numbers are used to specify the expression or group of expressions to be deleted (removed). All monitored expressions can be deleted with the Expression Monitor Clear_All command.

See Also   Expression Fprintf
Expression Monitor Clear_All
Expression Monitor Value
Expression Printf
Memory Display

Examples   To stop monitoring expression 2 in the Monitor window:

Expression **M**onitor **D**elete 2

To stop monitoring expressions 3 through 6 in the Monitor window:

Expression **M**onitor **D**elete 3..6

Expression Monitor Value

```
(Expression) → (Monitor) → (Value) → [<expr>]
                                        [..<expr>]
                                        (,)

[;<display_nmbr>] → <Return>
    [..<display_nmbr>]
```
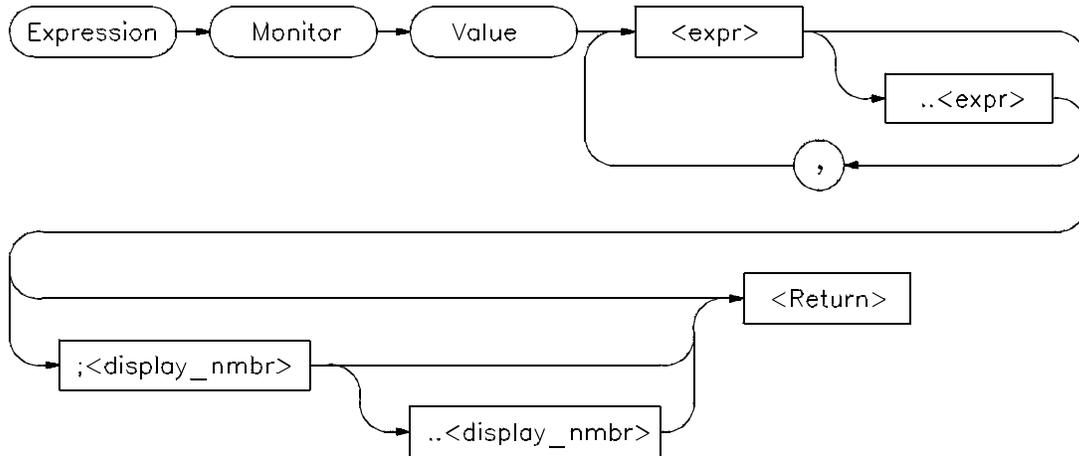
The Expression Monitor Value command monitors the specified expressions as the target program is executing. Expressions are updated and displayed in the Monitor window each time the debugger stops executing the program.

Up to seventeen lines, selected by the display line range parameter (;< display_nmbr> ..< display_nmbr>), can be displayed in the Monitor window.

Variables located in registers are shown with a *?* between their names and values.

All expressions monitored with this command are displayed according to their type as follows:

Type	Display Format
Ints	32-bit signed decimal numbers
Longs	32-bit signed decimal numbers
Shorts	16-bit signed decimal numbers
Chars	8 bit characters (unsigned hexadecimal numbers if not printable)
Pointers	32-bit unsigned numbers
Enums	Name of Enumerator constant (enumerator value if name not defined)
Arrays	All elements if enough lines, else first element
Structures	All members if enough lines, else first element
Quoted String	Characters surrounded by double quotes (" ")
Hex Byte	8-bit hexadecimal
Hex Word	16-bit hexadecimal
Hex Double Word	32-bit hexadecimal
Float	32-bit floating point
Double	64-bit floating point

If an expression range is displayed, each value within the range is displayed according to the base type (if one exists). For example, if the variable *flags* is a character array, the following command displays 20 characters.

Expression **M**onitor **V**alue flags+10..+29

Any expression can be type cast to display its value in a different format. For example, if the variable count is a long value, the following statement causes count to be displayed as a four character array:

Expression **M**onitor **V**alue (char[4])&count

If the type of the expression is unknown, it defaults to type byte.

Only 17 lines can be displayed in the data window. By default, a single line is used to display monitored expressions. If an array is monitored, only the elements that will fit on one line will be displayed. If a structure is monitored, only the first member will be displayed. To display an entire array or structure, a display line range may have to be specified. If all lines in the data window are

filled, you must use the Expression Monitor Delete command to delete an expression before monitoring another one.

If you do not specify a display line range, the next available line in the data window is selected to display the monitored variable. If you specify one line, the expression is displayed on that line. If you specify a range of lines, the amount of data that will fit on those lines is displayed.

See Also   Expression Monitor Clear_All
Expression Monitor Delete
Symbol Display

Examples   To monitor the value of variable 'current_temp':

Expression **M**onitor **V**alue current_temp

To monitor the value of the three members in structure 'time' and display them on Monitor window lines 4 through 6:

Expression **M**onitor **V**alue *time;4..6

To monitor the contents of string buf:

Expression **M**onitor **V**alue (Q S)buf

Expression Printf
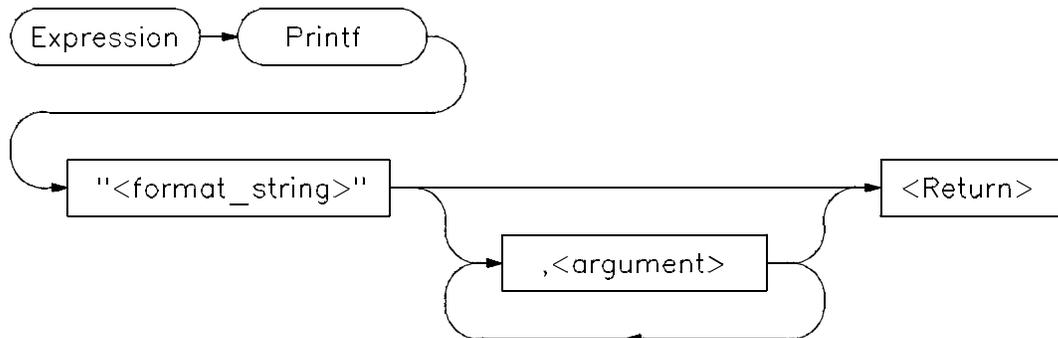
The Expression Printf command prints formatted output to the Journal window.

See the Expression Fprintf command for a detailed description.

See Also   Expression Fprintf
                  File User_Fopen

Examples   To print the string in double quotes to the journal window followed by the floating point value of 'temperature' with a precision of 2:

Expression **P**rintf "The value of 'temperature' is: %.2f \n",temperature

To print source line 24 to the Journal window:

Expression **P**rintf "%h",#24

To print the name of the current module to the Journal window:

Expression **P**rintf "%s",@module

To print the disassembled instruction at address 2030h to the Journal window as a string:

Expression **P**rintf "%m", 2030h

```
00002030 2040        MOVEA.L D0,A0
```

To print the contents of the assembly-level stack window to the Journal window:

Expression **P**rintf "%w",14

```
> Expression Printf "%w",14
    00043FC8=00000690
FP->00043FC4=00043FF0
    00043FC0=000604AC
    00043FBC=00000001
SP->00043FB8=00000001
```

File Command
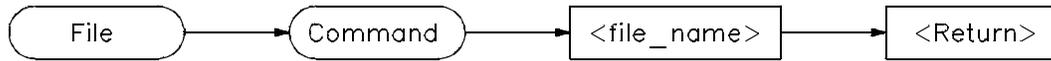
File → Command → <file_name> → <Return>

The File Command command reads the file specified by < file_name> and executes the commands contained in the file as though they were entered from the keyboard. Commands in the file are executed until the end of the file is reached. Input then continues from the previous source. The previous source can be the keyboard or another command file.

This command is commonly used to read macro definitions from a file, to set up I/O ports, or to change window displays.

File Command commands may be nested up to 16 levels deep.

If the filename consists of alphanumeric characters, a period, or a backslash, double quotation marks are optional. Otherwise, quotation marks must enclose the file name. If a filename extension is not specified, the debugger automatically appends a default extension, *.com*.

Command files can be executed at debugger startup using the -c option, from the command line during a debugging session, or from a startup file.

See the File Startup command description for information about how to automatically execute a command file when the debugger is started.

See Also   File Log
File Startup
The "Using Macros and Command Files" chapter.

Example   To execute command file 'varTrace.com':

File **C**ommand varTrace

File Error_Command

```
┌──────────┐    ┌───────────────┐    ┌─────────────┐    ┌──────────┐
│   File   ├───►│ Error_Command ├─┬─►│  Abort_Read  ├─┬─►│ <Return> │
└──────────┘    └───────────────┘ │  └─────────────┘ │  └──────────┘
                                  │  ┌──────────────┐ │
                                  ├─►│ Continue_Read ├─┤
                                  │  └──────────────┘ │
                                  │  ┌──────────────┐ │
                                  └─►│ Quit_Debugger ├─┘
                                     └──────────────┘
```
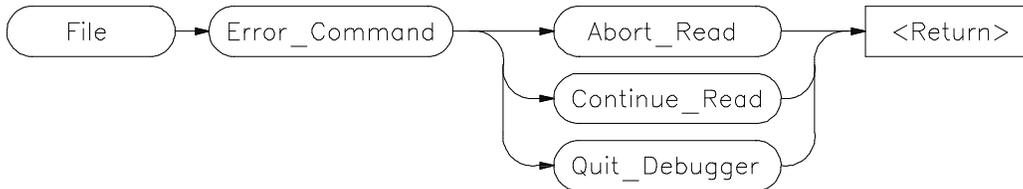
The File Error_Command command sets the command file error handling
mode. The command specifies what action the debugger takes when an error
occurs while reading a command file. *Abort_Read* causes the debugger to
return to the command line after an error and wait for keyboard input. This is
the default action. *Continue_Read* causes the debugger to continue to the next
command in the command file after an error. *Quit_Debugger* causes the
debugger to end the debugging session when an error occurs (as if you typed
Debugger Quit Yes).

See Also   File Command
                 File Log

Examples   To return to the command line after an error and wait for keyboard input:

File **E**rror_Command **A**bort_Read

To continue to the next command in the command file after an error:

File **E**rror_Command **C**ontinue_Read

To exit the debugger when an error occurs:

File **E**rror_Command **Q**uit_Debugger

File Journal

```
┌──────┐   ┌─────────┐     ┌──────┐    ┌─────────────┐    ┌──────────┐
│ File │──▶│ Journal │──┬─▶│  On  │──┬▶│ <file_name> │──┬▶│ <Return> │
└──────┘   └─────────┘  │  └──────┘  │ └─────────────┘  │ └──────────┘
                        │  ┌─────────┐│                  │
                        ├─▶│ Append  │┘                  │
                        │  └─────────┘                   │
                        │  ┌──────┐                       │
                        └─▶│ oFF  │───────────────────────┘
                           └──────┘
```
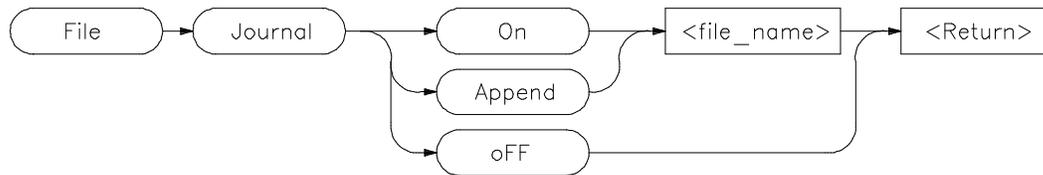
The File Journal command copies the information written to the Journal
window output into a journal file specified by < file_name> . The default
journal filename extension *.jou* will be appended to < filename> . The journal
file provides a history of your debugging session.

File Journal On opens a journal file for writing. If a file already exists
with the specified file name, new information is appended to the end of the
existing file.

File Journal Append opens an existing file. New information is
appended to the end of the existing file.

File Journal oFF closes the journal file.

A window number (29) is assigned to the journal file so that output can be
written to that file using the Expression Fprintf command.

See Also   Expression Fprintf

Examples   To make and open journal file 'debug1.jou' for writing:

File **J**ournal **O**n debug1

To close the currently open journal file:

File **J**ournal o**FF**

To open existing journal file 'debug1.jou' for writing and append new
information at the end of the file:

File **J**ournal **A**ppend debug1

File Log
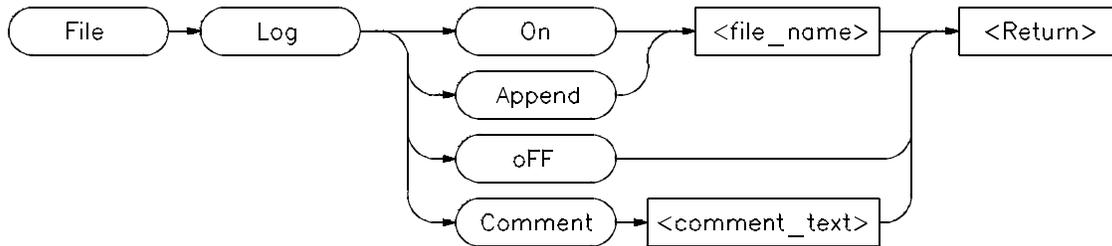
The File Log command records user input in a command file, specified by
< file_name> . The default filename extension *.com* will be appended to
< filename> . The File Log command allows an interactive debugger session to
be logged as a command file which can be rerun at a later time.

File Log On opens a file for writing. If the specified file already exists, the
file is overwritten by the new data.

File Log Append reopens a logging file to allow new information to be
added to the end of the file.

File Log oFF terminates logging to the file.

File Log Comment places a string of text in the file as a comment. If a log
file is not open, File Log Comment commands are ignored by the debugger.

All successful commands are written to the log file so the file can later be used
as a command file.

Commands which are entered but not successfully completed, are written to
the .com file as comments along with their error codes.

User input is recorded in the log file until the Log oFF command is executed.

A window number (28) is assigned to the log file so that output can be written
to that file using the Expression Fprintf command.

See Also   Expression Fprintf
                    File Error_Command

Examples

To make and open log file 'log1.com' for writing:

File **L**og **O**n log1

To close the currently open log file:

File **L**og o**FF**

To open existing log file 'log1.com' for writing and append new information at the end of the file:

File **L**og **A**ppend log1

To place the comment 'This is a comment string' in the log file:

File **L**og **C**omment This is a comment string.

If a log file is not open, this command is ignored.

File Startup

```
┌─────────┐    ┌──────────┐
│  File   │───▶│ Startup  │
└─────────┘    └──────────┘

                                                        ┌──────────┐
                                                        │ <Return> │
                                                        └──────────┘

  ┌───────────────────┐         ┌────────────────────────┐
  │  <startup_file>   │         │  ,<command_file>       │
  └───────────────────┘         └────────────────────────┘
```
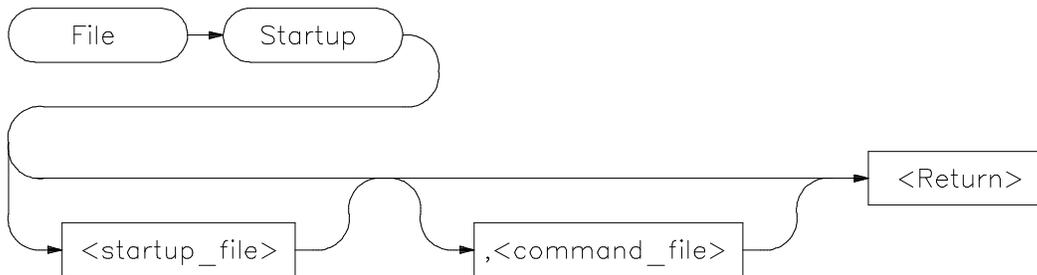
The File Startup command saves the current debugger option settings and window parameters in a startup file specified by < startup_file> . When you start a debugging session and specify the startup file with the -s option of the db68040 command, the startup options and window parameters you saved will be the default parameters in that debugging session.

A startup file has an extension of *.rc* appended to the end of it. If you do not specify a startup file name, the startup options are saved in a file named *db68040.rc*.

You can modify default debugger startup option values with the Debugger Option command and window parameters with the Window commands.

Remember that you can also specify a command file to be executed when the debugger starts.

See Also   Debugger Option
File Command
Window New
Window Resize
the "Using Macros and Command Files" chapter

Examples   To save the current set of debugger startup options and window parameters in startup file 'my_start_file.rc':

File **S**tartup my_start_file

To save the current set of debugger startup options and window parameters in startup file 'my_start_file.rc' and execute the command file 'initDemo.com' whenever the debugger is started using 'my_start_file.rc':

File **S**tartup my_start_file , initDemo

File User_Fopen

```
  ┌─────────┐    ┌──────────────┐      ┌──────────┐    ┌──────────────────┐
──┤  File   ├───►┤  User_Fopen  ├──┬──►┤  Append  ├─┬─►┤  <window_nmbr>   ├─┐
  └─────────┘    └──────────────┘  │   └──────────┘ │  └──────────────────┘ │
                                   │   ┌──────────┐ │                       │
                                   ├──►┤  Create  ├─┤                       │
                                   │   └──────────┘ │                       │
                                   │   ┌──────────┐ │                       │
                                   └──►┤   Read   ├─┘                       │
                                       └──────────┘                         │
  ┌────────────────────────────────────────────────────────────────────────┘
  │   ┌──────────┐    ┌──────────┐    ┌──────────────┐    ┌──────────┐
  └──►┤  <Tab>   ├───►┤   File   ├───►┤ <file_name>  ├───►┤ <Return> │
      └──────────┘    └──────────┘    └──────────────┘    └──────────┘
```
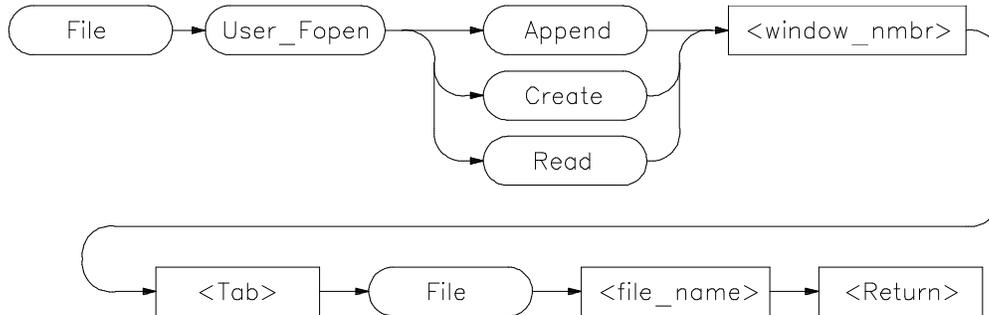
The File User_Fopen command opens the file specified by < file_name> for reading or writing and assigns a window number to it.

The *File User_Fopen Append* command opens an existing file for writing, adding new information at the end of the file.

The *File User_Fopen Create* command creates a new file for writing.

The *File User_Fopen Read* command opens an existing file for reading.

After opening a file using the File User_Fopen Append or File User_Fopen Create command, you can use the Expression Fprintf command to write information to the file. Files opened for reading may be read from the built-in macro fgetc(). See the "Predefined Macros" chapter of this manual for a complete description of this macro.

The window number must be between 50 and 256 inclusive.

Use the Window Delete or the File Window_Close command to close the file.

See Also
Expression Fprintf
File Window_Close
Window Delete
Window New

Examples   To open user window 57 and redirect any data written to window 57 to the file 'varTrace.out':

File **U**ser_Fopen **C**reate 57 **F**ile varTrace.out

To open user window 57 and append any data written to window 57 to the existing file 'varTrace.out':

File **U**ser_Fopen **A**ppend 57 **F**ile varTrace.out

To open file 'temp.dat' for reading, accessing the file as user window 52:

File **U**ser_Fopen **R**ead 52 **F**ile temp.dat

File Window_Close

```
( File ) ──▶ ( Window_Close ) ──▶ [ <file_window_num> ] ──▶ [ <Return> ]
```
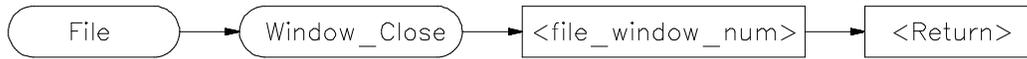
The File Window_Close command closes a device or file which was previously
opened with the File User_Fopen command. The Window Delete command
may also be used for this purpose.

See Also   File User_Fopen
Window Delete

Example   To close file associated with user window number 57:

File **W**indow_Close 57

Memory Assign

```
Memory → Assign → Byte → <addr>
                  Word
                  Long

       =<expr>        → <Return>
       =<expr_string>
```
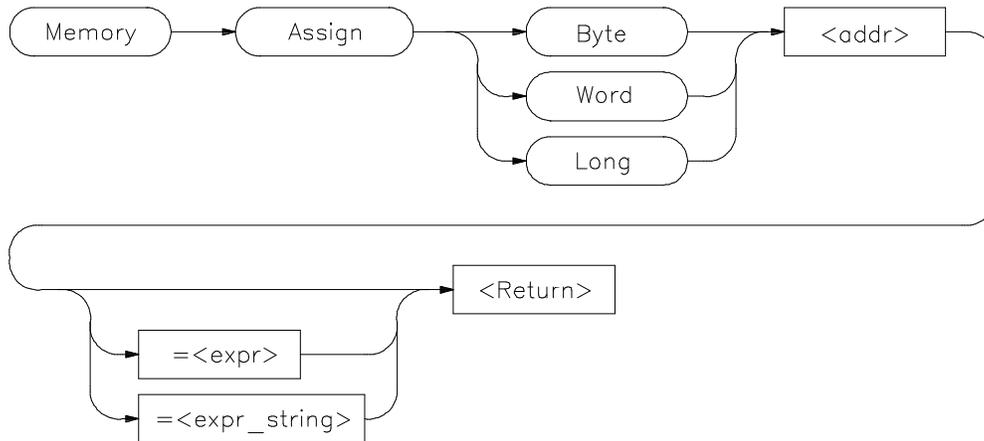
The Memory Assign command changes the contents of the memory location specified by $<addr>$ to the value or values defined by the expression $<expr>$ or expression string $<expr\_string>$. The size of the memory elements to be modified is specified by one of the size qualifiers (Byte, Word, or Long).

Expression strings are specified as ASCII characters enclosed in quotation marks and/or as a list of values separated by commas. Expressions and expression string elements will be truncated or padded as required, based on the size qualifier.

Memory values can be entered interactively if you do not define a value on the command line. When a value is not specified, the contents of the specified memory locations are displayed in hexadecimal and decimal. You can change the existing value by entering any legal expression followed by a carriage return. The next memory location and its contents are then displayed. The return key entered without a value will cause the command to terminate.

The Memory Assign command does not recognize variable typing. It is intended to be used as an assembly-level memory setting routine. For example, assume that the variable *count* is a long integer. If you want to set the value of count equal to 5, the command

Memory **A**ssign **L**ong count=5

will not work. The command will set the memory location referenced by the value of count equal to 5, not the contents of the variable. To set the value of count equal to 5, use the following command:

Memory **A**ssign **L**ong &count=5

The Expression C_Expression command should be used to set C variables. This command recognizes variable types and the specified expressions behave according to the rules of C. The command:

Expression **C_Expression** count=5

will set count equal to 5.

See Also   Expression C_Expression
Memory Register

Examples   To display the contents of memory location 1000h and allow interactive modification of memory contents:
00001000 = 0x48 72:

Memory **A**ssign **B**yte 1000h

To change the contents of memory locations 2000h through 2005h to 00, 41, 00, 42, 00, 43, and change the contents of locations 2006h/2007h to the value of 'system_is_running':

Memory **A**ssign **W**ord 2000h=41h,42h,43h,system_is_running

Memory Block_Operation Copy
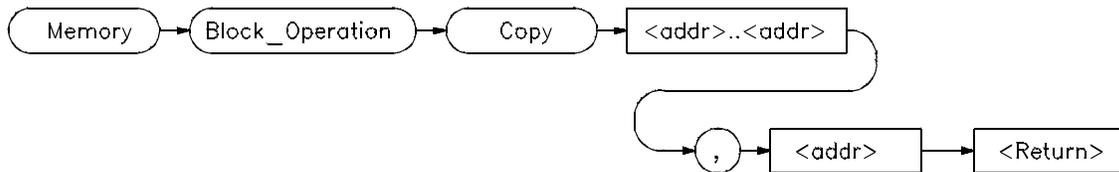
The Memory Block_Operation Copy command copies the contents of the memory range specified by < *addr*> ..< *addr*> to a block of the same size starting at the memory location specified by < *addr*> .

See Also   Memory Assign
Memory Block_Operation Fill
Memory Block_Operation Match
Memory Block_Operation Search
Memory Block_Operation Test

Examples   To copy the block of memory starting at address 1000h and ending at address 10ffh to a block of the same size starting at address 5000h:

Memory **B**lock_Operation **C**opy 1000h..10ffh,5000h

To copy the block of memory starting at the address of the structure 'current_targets' and ending 15 bytes after this address to a block of memory starting at the address of the structure 'default_targets':

Memory **B**lock_Operation **C**opy &current_targets..+0xf,
&default_targets

Memory Block_Operation Fill
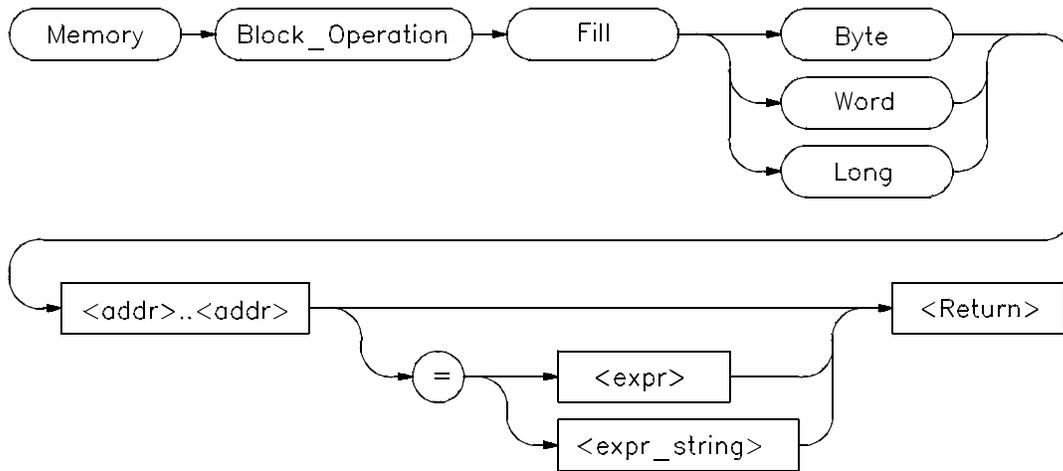
The Memory Block_Operation Fill command fills the range of memory locations specified by the address range *<addr>..<addr>* with the value or values specified by an expression *<expr>* or an expression string *<expr_string>*. If no expression is given, the debugger fills the specified memory locations with zeros. The specified size qualifier (Byte, Word, or Long) determines the size of the value.

If you specify a single expression value, the debugger fills the memory area with that value. If you enter an expression string, the debugger fills the memory area with the specified string pattern.

An expression string is a list of values separated by commas and can include ASCII characters enclosed in quotation marks. All expressions in an expression string are padded or truncated to the size specified by the size qualifiers if they do not fit the specified size evenly.

If the number of values in an expression string is less than the number of bytes in the specified address range, the debugger repeatedly places the list of values in memory until all designated memory locations are filled. If you specify more values than can be contained in the specified address range, the debugger ignores the excess values.

See Also   Memory Assign
                      Memory Block_Operation Copy
                      Memory Block_Operation Match
                      Memory Block_Operation Search
                      Memory Block_Operation Test
                      Memory Register

Examples   To fill memory locations 1000h through 1007h with the long pattern
                      61626364, 65666768:

                      Memory **B**lock_Operation **F**ill **L**ong 0x1000..+7='abcdefgh'

                      To fill the memory area starting at location 1000h and ending at location 10ffh
                      with zeros:

                      Memory **B**lock_Operation **F**ill **B**yte 0x1000..0x10ff

Memory Block_Operation Match
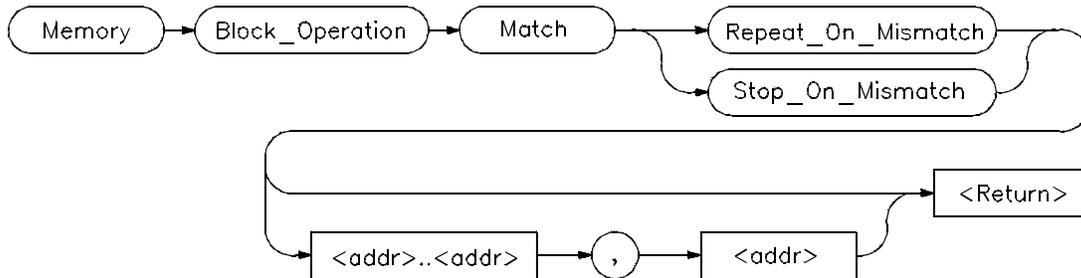
The Memory Block_Operation Match command compares the contents of two blocks of memory to determine their similarities or differences. The command compares the block of memory specified by the address range *<addr>..<addr>* with the same size block starting at *<addr>*.

The debugger displays differences between the two blocks of memory, mismatched values and addresses, in the Journal window. If the contents of the two blocks of memory are the same, the debugger displays the message *Memory blocks are the same.*

The Memory Block_Operation Match Stop_On_Mismatch command halts when a mismatch is found. If the Memory Block_Operation Match Repeat_On_Mismatch command is selected, the comparison continues until the end of the block.

When you execute the Memory Block_Operation Match Stop_On_Mismatch/Repeat_On_Mismatch command without specifying an address range, the debugger continues comparing the address range specified in the previous Memory Block_Operation Match Stop_On_Mismatch command starting from where it found the last mismatch.

See Also   Memory Block_Operation Copy
Memory Block_Operation Fill
Memory Block_Operation Search
Memory Block_Operation Test

Examples   To compare the block of memory starting at address 1000h and ending at
address 10ffh with a block of the same size beginning at address 5000h and
stop when a difference is found:

Memory **B**lock_Operation **M**atch **S**top_On_Mismatch
1000h..10ffh,5000h

To execute the previous Memory Block_Operation Match
Stop_On_Mismatch command starting from where it found the last mismatch:

Memory **B**lock_Operation **M**atch **S**top_On_Mismatch

To compare the block of memory starting at address 1000h and ending at
address 10ffh with a block of the same size beginning at address 5000h and
stop at the end of the memory block:

Memory **B**lock_Operation **M**atch **R**epeat_On_Mismatch
1000h..10ffh,5000h

Memory Block_Operation Search
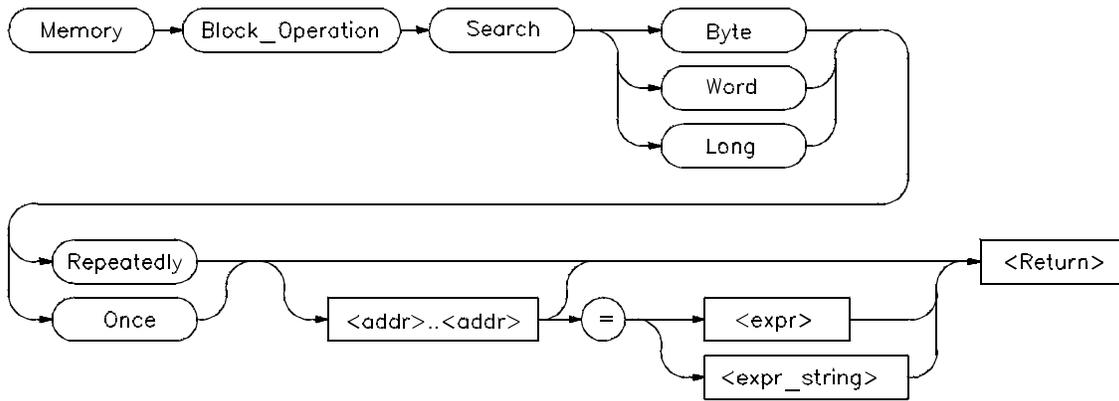
The Memory Block_Operation Search command searches the block of memory specified by *<addr>..<addr>* for the specified expression *<expr>* or expression string*<expr_string>* . The size qualifier (Byte, Word, or Long) specifies the size of an expression or each expression in an expression string. A Memory Block_Operation Search command given without parameters continues the search of a previous Memory Search command given with the Once qualifier. The Repeatedly qualifier causes the search to repeat.

You can specify expression strings as ASCII characters enclosed in quotation marks and/or as a list of values separated by commas. If the strings do not fit the specified size evenly, all expressions in an expression string will be padded or truncated to the size specified by the size qualifiers.

If you specify the Once qualifier, the search stops when the expression is found. If you specify the Repeatedly qualifier, the debugger repeatedly searches for the specified expression, displaying each match until it reaches the end of the block or until you press **CTRL C**.

When you execute the Memory Block_Operation Search command with the Once qualifier, subsequent Memory Block_Operation Search commands that are executed without expression parameters cause the debugger to continue searching through the originally specified address range starting from where it found the last match. If the expression or expression string is not found in the specified block, the debugger displays the message *Not found.*

See Also   Memory Display
                       Memory Block_Operation Copy
                       Memory Block_Operation Fill
                       Memory Block_Operation Match
                       Memory Block_Operation Test
                       Program Find First
                       Program Find Next

Examples   To search for the expression 'gh' in the memory range from address 1000h
                       through address 10ffh and stop when the expression is found or address 10ffh
                       is reached:

                       Memory **B**lock_Operation **S**earch **W**ord **O**nce
                       1000h..+0xff = 'gh'

                       To execute the previous Memory Block_Operation Search command starting
                       from where it found the last match:

                       Memory **B**lock_Operation **S**earch **W**ord **O**nce

                       To search for the hexadecimal value '65666768' in long format in the address
                       range 1000h through 10ffh and stop at the end of the address range:

                       Memory **B**lock_Operation **S**earch **L**ong **R**epeatedly
                       0x1000..0x10ff=0x65666768

Memory Block_Operation Test
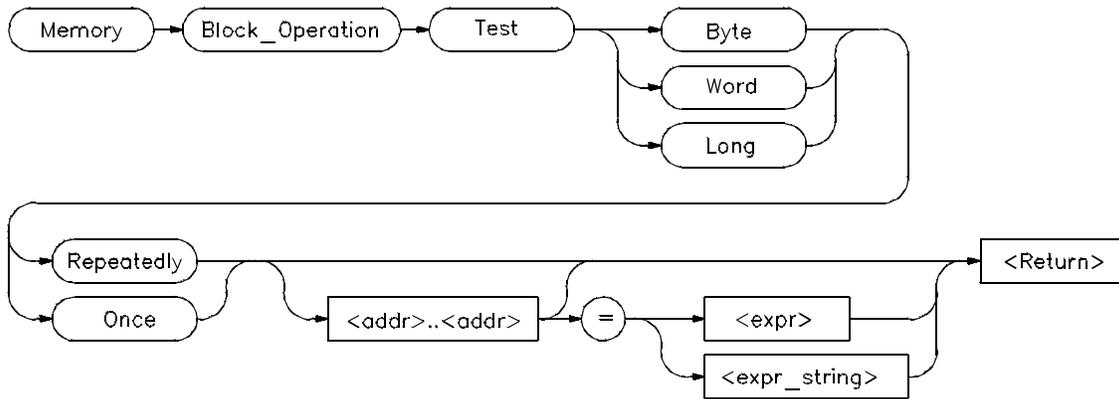
The Memory Block_Operation Test command examines the specified memory locations specified by <*addr..addr*> to verify that the value(s) defined by <*expr*> or <*expr_string*> exist throughout the specified memory area. When the debugger finds a mismatch, it displays the mismatched address and value. The size qualifier (Byte, Word, or Long) specifies the size of an expression or expression in a string.

If you enter a single expression value, the debugger tests the memory area for that value. If you specify an expression string, the debugger tests the memory area to verify that it is filled with the values found in the expression string.

You can specify expression strings either as ASCII characters enclosed in quotation marks or as a list of values separated by commas. If they do not evenly fit the specified size, all expressions in an expression string will be padded with zero-valued bytes to the size specified by the size qualifier.

Once Qualifier

If you specify the Once qualifier, the test stops when a mismatch is found. If you execute the Memory Block_Operation Test command with the Once qualifier specified, subsequent *Memory Block_Operation Test . . . Once* commands that are specified without parameters will continue testing through the address range originally specified, beginning with the last address tested. A Memory Block_Operation Test command given without parameters continues

the test of a previous Memory Block_Operation test command given with the Once qualifier, beginning with the last address tested.

Repeatedly Qualifier

If you specify the Repeatedly qualifier, the debugger continues testing the specified value(s) for mismatches until the end of the block is reached, or until you enter **CTRL C**.

Examples

To test for the expression 'gh' in the memory range from address 1000h through address 10ffh and stop when a word not matching the expression is found:

Memory **B**lock_Operation **T**est **W**ord **O**nce 1000h..+0xff = 'gh'

To execute the previous Memory Block_Operation Test command starting from where it found the last mismatch:

Memory **B**lock_Operation **T**est **W**ord **O**nce

To test for the hexadecimal value '65666768' in long format in the address range 1000h through 10ffh and stop at the end of the address range:

Memory **B**lock_Operation **T**est **L**ong **R**epeatedly 0x1000..0x10ff=0x65666768

Mismatched values are displayed in the Journal window.

Memory Display
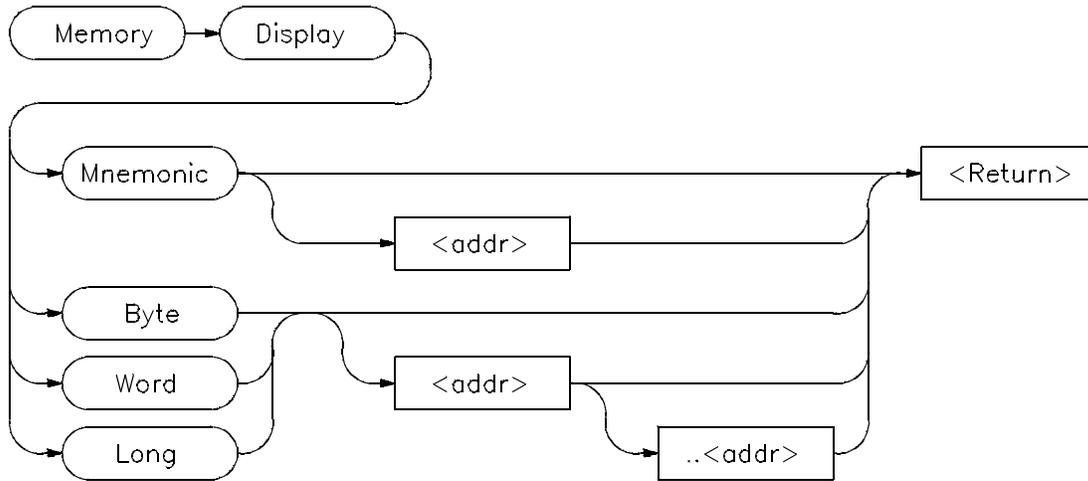
The Memory Display displays the contents of the specified memory locations.

Mnemonic Option

The *Mnemonic* option displays memory in assembly language mnemonics starting at the memory location specified by *<addr>*. If you do not specify an address, the debugger displays memory beginning with the address pointed to by the program counter. This command functions only in the assembly-level mode.

If you have executed the Debugger Options Symbolics Intermixed On command, C source code lines will be intermixed with the assembly language code (when applicable). If you have executed the Debugger Options Symbolics Assem_Symbols On command, symbol references will be displayed with the assembly language code.

The *Prev*, *Next*, *Up*, and *Down* keys may be used when the Code window is active to display instructions with higher or lower addresses. Note that the Prev and Up keys do not function when disassembling addresses outside of the target program.

Note   If the Align_bp option is set to *On*, the address of the first instruction in the assembly Code window may be incorrect after executing the Memory Display Mnemonic command.

Byte, Word, and Long Options

The byte, word, or long qualifier option displays the contents of memory locations specified by *< addr> ..< addr>* in the Journal window in both hexadecimal and ASCII formats. The debugger represents nonprintable ASCII characters by a period (.). The debugger displays memory contents in the size specified by the size qualifier (Byte, Word, or Long).

If you specify an address range, the debugger displays all memory locations in that range.

If you specify a single address, the debugger displays two lines of data.

If you do not specify any parameters, the debugger displays the next 80 bytes (five lines) of data after the previously displayed address range.

The memory contents are displayed in the Journal window.

See Also   Expression Display_Value
Symbol Display

Examples   To display disassembled memory in the Code window starting at the symbol '_emeg_shutdown' (this command works only in assembly-level mode):

Memory **D**isplay **M**nemonic _emeg_shutdown

To display memory in word format in the Journal window starting at the symbol 'time' and ending 15 bytes after 'time':

Memory **D**isplay **W**ord time..+0xf

Memory Inport Assign
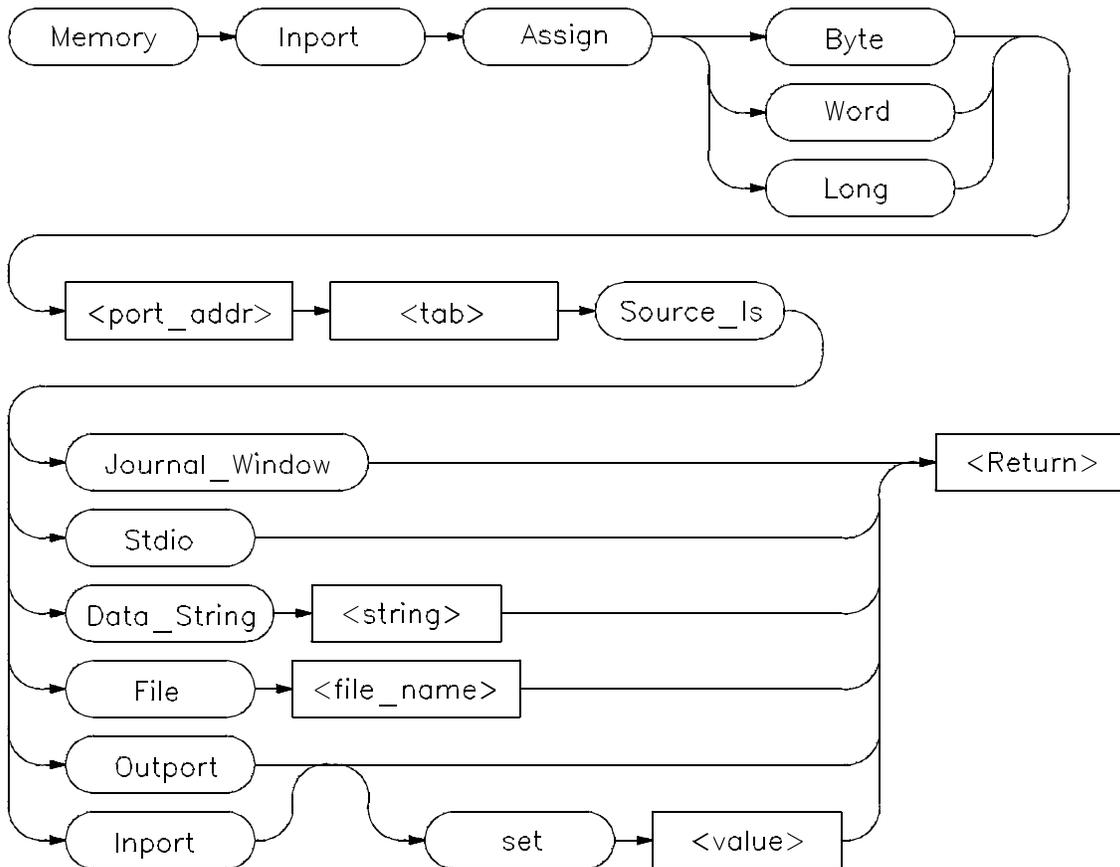
The Memory Inport Assign command assigns a simulated input port and defines its size, address, and input source. The port address can be any valid address. The source of input data may be the standard I/O screen, the Journal window, a file, an expression string, or the input or output port buffers.

Inport Port Size

You can specify the size of an input port by using one of the size qualifiers (Byte, Word, or Long).

Input Data Source

Specify the input data source (Source_Is) by entering one of the following associated keywords:

Journal_Window	Journal window
File < file_name>	Specified file
Stdio	Standard I/O device
Data_String	Specified string
Outport	Output port buffer
Inport	Input port buffer

Journal_Window. If you specify Journal_Window, the input port reads data from the Journal window that you enter interactively. The debugger displays the port address in hexadecimal. At this point, you can enter an input value, which can be a decimal, hexadecimal, or binary number, or a string.

File. If you enter File < file_name> , the input port reads data from the specified file < file_name> . When the end of a file is reached, use the Memory Inport Rewind command to rewind the file.

Stdio. If you specify Stdio, the debugger reads data from the standard I/O screen. The debugger automatically displays the standard I/O screen when an input port is read. The cursor is positioned at the last point where data was written. The standard I/O screen border flashes and the debugger displays a message indicating the port from which it is reading data. Keyboard entry is echoed to the screen only if the target software echoes input characters to the standard output port.

Data_String. If you specify Data_String < string> , the input port reads data from the specified string (< string>). When the end of the string is reached, the debugger automatically resets the string so that the string can be read again if the input port is accessed again. The Memory Inport Rewind command may be used to reset the string pointer at any time.

Outport. The input port uses the value in the output port buffer at the same address. If you enter a value, the debugger sets the output port buffer to that

value. If you do not enter a value and an output port has not been declared at that address, the debugger places a value of zero in the buffer.

Inport. If you specify Inport < port_addr> , the input port uses the value in the input port buffer at the same address. If you enter a value, the debugger sets the input port buffer to that value. If you do not specify a value, a value of zero is placed in the buffer.

To abort input from the console or standard I/O screen, press **CTRL C**. This returns the debugger to command mode.

See Also   Memory Inport Rewind
Memory Inport Show
Memory Outport Show

Examples   To assign address 0x400 as an I/O port (input) of size word:

Memory **I**nport **A**ssign **W**ord 0x400 **S**ource_Is **F**ile
"/myproj/ecs.dat"

Read operations from the port will access file '/myproj/ecs.dat'. You must specify the file name in quotation marks.

To assign address 0x40C as an I/O port (input) of size byte:

Memory **I**nport **A**ssign **B**yte 0x40C **S**ource_Is **D**ata_String
"message"

Read operations from the port will access the string containing the word 'message'.

To assign address 0x40C as an I/O port (input) of size byte:

Memory **I**nport **A**ssign **B**yte 0x40C **S**ource_Is **S**tdio

Read operations from the port will access the stdio window or device.

Memory Inport Delete

Memory → Inport → Delete → `<port_addr>` → `<Return>`

The Memory Inport Delete command disables the specified input port address, allowing the address to behave like a normal memory location.

See Also Memory Inport Assign
Memory Outport Assign

Example To disable the input port at address 400h:

Memory **I**nport **D**elete 400h

Memory Inport Rewind

```
( Memory )  →  ( Inport )  →  ( Rewind )  →  [ <port_addr> ]  →  [ <Return> ]
```

The Memory Inport Rewind command rewinds an input file or resets the pointer to the input string associated with the input port. The input port is specified by the port address (< port_addr>).

See Also   Memory Inport Assign
Memory Inport Delete
Memory Inport Show

Example   To rewind the input file or string associated with input port 400h:

Memory **I**nport **R**ewind 0x400h

369

Memory Inport Show

```
   ( Memory ) → ( Inport )
        
   ( Show ) ───────────────────────────── [ <Return> ]
            [ <port_addr> ]
                     [ ..<port_addr> ]
```
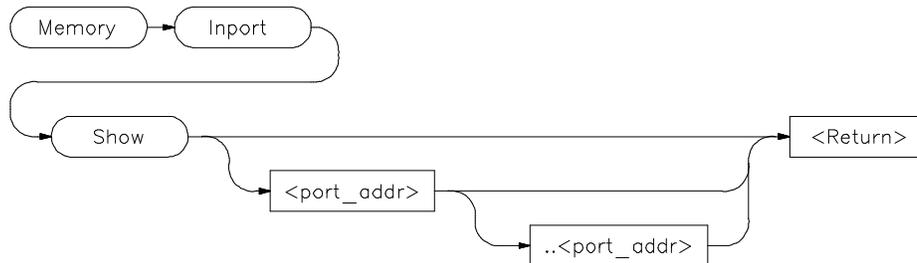
The Memory Inport Show command displays the value stored in the buffer of the specified input port or ports. Each input port has a single value buffer associated with it. The buffer contains the last value read from the port. This value can be represented in byte, word, or long format.

You can display an input port buffer value by specifying either a port address (< port_addr>) or a port address range (< port_addr> ..< port_addr>). If you specify a range, the debugger displays all buffer values in the range.

If you do not specify any parameters, the debugger displays all declared input ports, with their port address, size, value, and data source.

See Also

Memory Inport Assign
Memory Inport Delete
Memory Inport Rewind

Examples

To show all assigned input ports:

Memory **I**nport **S**how

To show all input ports in the address range 400h through 4ffh:

Memory **I**nport **S**how 0x400..0x4ff

Memory Map Guarded
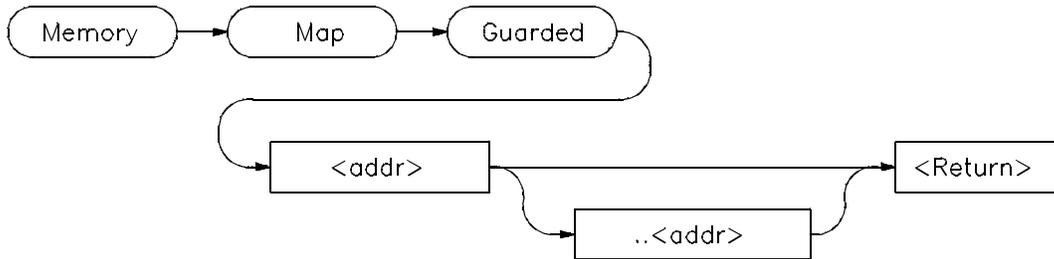
The Memory Map Guarded command prevents access to a specified memory location or range of memory locations. These locations cannot be accessed during execution of the target program. The Memory Map Guarded command overrides any Memory Map Read_Only and Memory Map Write_Read commands previously executed.

The size of the address range specified in a Memory Map Guarded command and the size of the address space that is mapped by combining all Memory Map commands is limited only by the amount of virtual memory available to the debugger. You can map a maximum of 60 banks of memory.

See Also   Memory Map Read_Only
Memory Map Show
Memory Map Write_Read

Example   To configure memory address range 8000h through 0a000h as guarded (nonaccessible) memory:

Memory **M**ap **G**uarded 8000h..0a000h

Memory Map Read_Only

```
  ( Memory )  →  ( Map )  →  ( Read_Only ) ─┐
                                             │
  ┌──────────────────────────────────────────┘
  │
  └→  [ <addr> ] ──────────────────────┬→  [ <Return> ]
              │                         │
              └→  [ ..<addr> ] ─────────┘
```
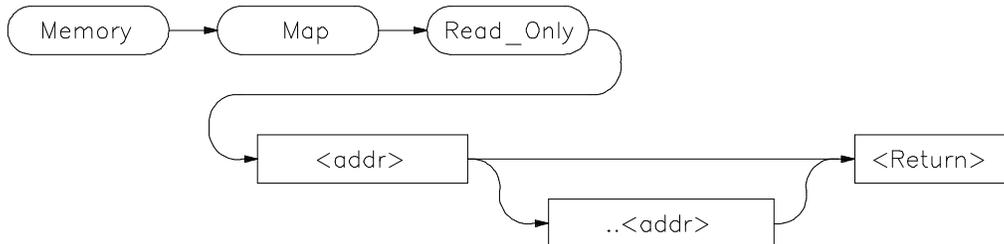
The Memory Map Read_Only command prevents a specified memory location or range of memory locations from being written to during execution of the target program. *Read_Only* protects the target program memory so that specified code and/or data can only be read. The Memory Map Read_Only command overrides any Memory Map Guarded and Memory Map Write_Read commands previously executed.

The size of the address range specified in a Memory Map Guarded command and the size of the address space that is mapped by combining all Memory Map commands is limited only by the amount of virtual memory available to the debugger. You can map a maximum of 60 banks of memory.

See Also   Memory Map Guarded
Memory Map Show
Memory Map Write_Read

Example   To configure memory address range 8000h through 8fffh as read-only (ROM) memory:

Memory **M**ap **R**ead_Only 8000h..8fffh

Memory Map Show
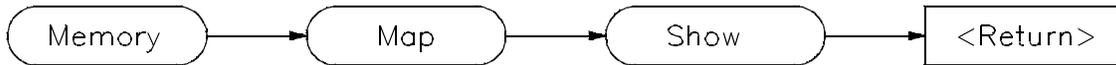
| Memory | → | Map | → | Show | → | \<Return\> |

The Memory Map Show command displays a map of the memory location assignments (Guarded, Read_Only, Write_Read).

See Also   Memory Map Guarded
Memory Map Read_Only
Memory Map Write_Read

Example   To display the memory map:

Memory **M**ap **S**how

```
> Memory Map Show
     TYPE    OWNER       ADDRESS            COMMAND LINE
     RAM     SIMU    00000000..0000002F  Load section
     RAM     SIMU    000000BC..000000BF  Load section
     RAM     SIMU    00000100..0000010B  Load section
     RAM     SIMU    00000400..00001044  Load section env
     RAM     SIMU    00001048..00002851  Load section prog
     RAM     SIMU    00002854..0000906B  Multiple load sections
     RAM     SIMU    00040000..00043FFF  Load section stack
     RAM     SIMU    00060000..0006401D  Multiple load sections
```

The command displays memory address ranges mapped as Guarded (NOMEM), Read_Only (ROM), or Write_Read (RAM) in the Journal window. The display includes a list of sections loaded and their address ranges.

Memory Map Write_Read
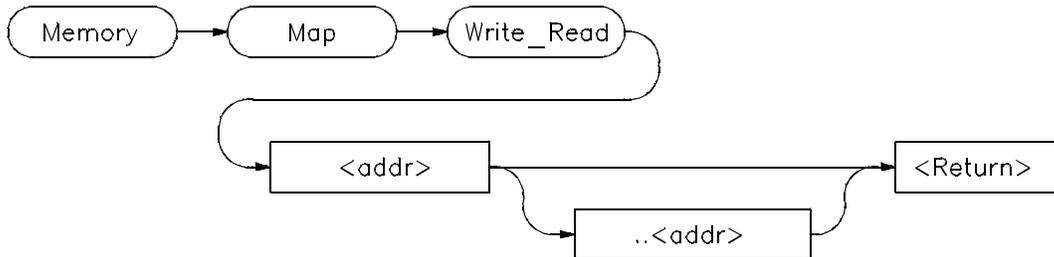
The Memory Map Write_Read command enables read/write access to a specified memory location or a range of memory locations by the target program. Memory Map Write_Read allows the program to read from or write to specified code or data areas. The Memory Map Write_Read command overrides any Memory Map Guarded and Memory Map Read_Only commands previously executed.

The size of the address range specified in a Memory Map Guarded command and the size of the address space that is mapped by combining all Memory Map commands is limited only by the amount of virtual memory available to the debugger. You can map a maximum of 60 banks of memory.

See Also   Memory Map Guarded
Memory Map Read_Only
Memory Map Show

Example   To configure memory address range 2000h through 3fffh as Write_Read (RAM) memory:

Memory **M**ap **W**rite_Read 2000h..3fffh

Memory Outport Assign
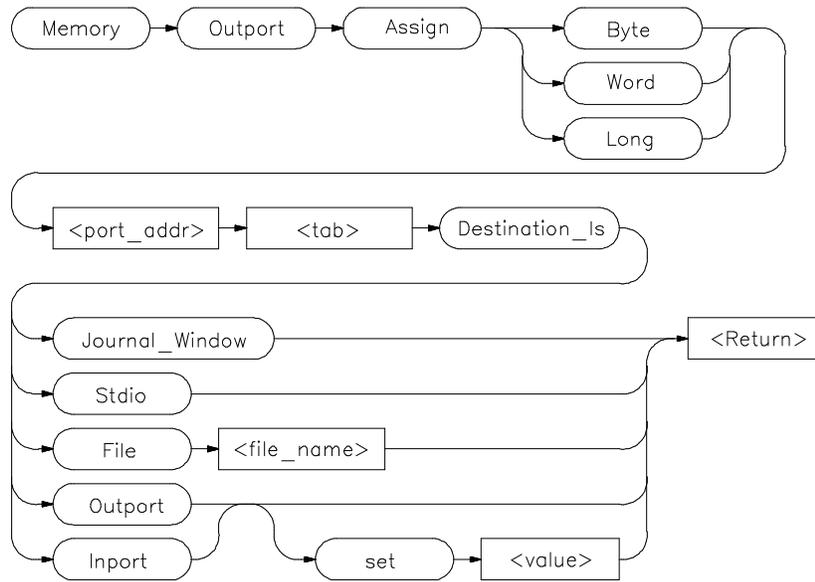
The Memory Outport Assign command defines the address, size, and output destination of a simulated output port. The target program can write output data to the simulated output port. The port address can be any valid address.

Outport Port Size

You can specify the size of an output port buffer by entering one of the size qualifiers (Byte, Word, or Long).

Output Data Destination

You can direct the output port data to be written to one of the following destinations by entering the associated keyword for the output destination.

Journal_Window	Journal window
File < file_name>	Specified file
Stdio	Standard I/O device
Outport	Output port buffer
Inport	Input port buffer

Journal_Window. If you specify Journal_Window, the output port writes output data to the Journal window. The debugger displays the current output value and port address in hexadecimal.

Stdio. If you specify Stdio, data is written to the standard I/O screen. The debugger automatically displays the standard I/O screen and positions the cursor at the last point where data was written. When simulation stops and control is returned to command mode, the debugger displays the previously displayed screen again.

File. If you specify File < file_name> , output port data is written to the specified file (< file_name>). The Memory Outport Rewind command may be used to rewind the file.

Outport. If you specify Outport Set < value> , output port data is written to the output port buffer. Previous values written to the buffer are lost. This buffer is only one level deep. The last value written to a port can be viewed by issuing the Memory Display Outport command. The output port buffer may be initialized by using the optional value parameter. If no value is specified, a zero is placed in the buffer.

Inport. If you specify Inport Set < value> , output port data is written to the input port buffer at the same port address. This buffer is only one level deep. Previous values written to the buffer are lost. The last value written to a port may be viewed by issuing the Memory Display Outport command. If no input port exists at this address, one will be created.

To abort output to the console or standard I/O screen, press **CTRL C**. This returns the debugger to command mode.

See Also   Memory Outport Delete
Memory Outport Rewind
Memory Outport Show

Examples   To assign address 0x408 as an I/O port (output) of size word:

Memory **O**utport **A**ssign **W**ord 0x408 Destination_Is File
"/myproj/cmdout.dat"

Write operations to the port will access file '/myproj/cmdout.dat'. You must
specify the file name in quotation marks.

To assign address 0x40C as an I/O port (output) of size byte:

Memory **O**utport **A**ssign **B**yte 0x40C **D**estination_Is **S**tdio

Write operations to the port will access the Stdio window.

Memory Outport Delete

Memory → Outport → Delete → \<port_addr\> → \<Return\>

The Memory Outport Delete command disables the specified output port address, allowing the address to behave like a normal memory location.

See Also   Memory Outport Assign
Memory Outport Show

Example   To disable the output port at address 408h:

Memory **O**utport **D**elete 408h

Memory Outport Rewind

```
( Memory )  →  ( Outport )  →  ( Rewind )  →  [ <port_addr> ]  →  [ <Return> ]
```

The Memory Outport Rewind command rewinds an output file associated with an output port specified by the port address (< port_addr>). Subsequent output to that port starts at the beginning of the file.

See Also   Memory Outport Assign
Memory Outport Delete
Memory Outport Show

Example   To rewind the output file associated with output port 408h:

Memory **O**utport **R**ewind 0x408

Memory Outport Show

```
  ( Memory ) → ( Outport )
       │
       └──→ ( Show ) ─────────────────────────→ [ <Return> ]
                  │                          │
                  └─→ [ <port_addr> ] ───────┤
                              │              │
                              └─→ [ ..<port_addr> ]
```
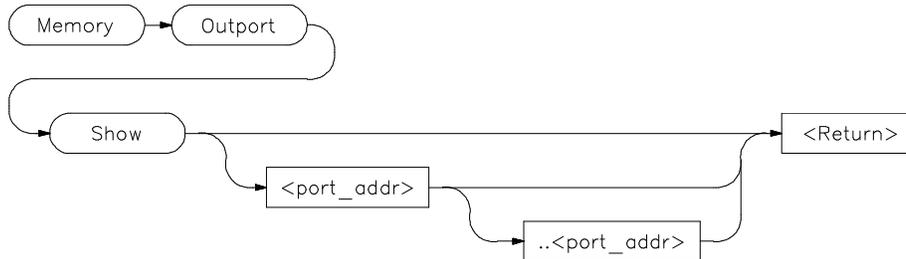
The Memory Outport Show command displays the value stored in the buffer of the specified outport port or ports. Each output port has a one-value buffer associated with it that contains the last value written to the port. The buffer value can be displayed in byte, word, or long format.

You can display an output port buffer value by specifying either a port address (< *port_addr* >) or port address range (< *port_addr* > ..< *port_addr* >). If you specify a range, the debugger displays all buffer values in that range.

If you do not specify any parameters, the debugger displays all declared output ports with their port addresses, sizes, values, and data destinations.

See Also

Memory Outport Assign
Memory Outport Delete
Memory Outport Rewind

Examples

To display all assigned output ports:

Memory **O**utport **S**how

To display all output ports in the address range 400h through 4ffh:

Memory **O**utport **S**how 0x400..0x4ff

Memory Register
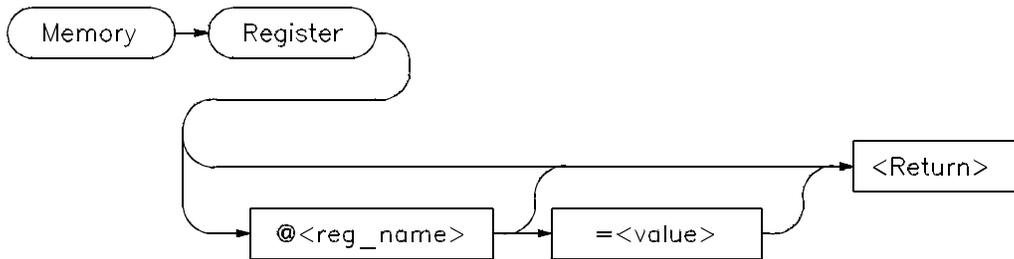
The Memory Register command changes the contents of a register, status flag, or other processor variables such as cycle count. The new contents are defined by < value> .

The PC (program counter) is displayed or changed if you do not specify a register name.

If you do not specify a value in the command, values are entered interactively. You can enter multiple register values interactively. The debugger displays contents of the specified register in binary, hexadecimal, or decimal, as appropriate for the register. You can change the existing value by entering any legal expression and pressing the **Return** key.

Pressing the **Return** key without specifying a register value terminates the command.

All register names are preceded with an @ sign. A complete list of the reserved words that you can use with this command is given in the "Reserved Symbols" chapter of this manual.

See Also   Memory Assign

Examples   To modify register values interactively:

Memory **R**egister

The program counter (PC) is displayed in the Journal window. You can
modify the PC by entering a value (10a4h in this example) at the cursor
prompt and pressing Return. The PC will be modified, and the next register
will be displayed:

```
@pc      = 0x000010B8      4280: 10a4h
@sp      = 0x00015DB4      89524:
```

To set the value of register @d1 to 44h:

Memory **R**egister @d1=0x44

To interactively change the value of register @d1:

Memory **R**egister @d1

Memory Unload_BBA
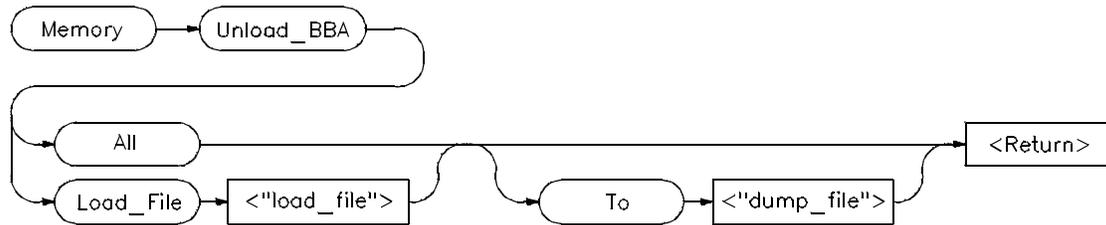
Note   You must have the HP Branch Validator product for the processor you are
debugging code for installed on your system in order to use this command.

If you do not have the HP Branch Validator for your processor, the debugger
will display the following error message when you attempt to execute this
command:

```
error code = 141
```

```
No valid BBA spec file for <processor> processor
```

The Memory Unload_BBA command unloads basis branch analysis (BBA)
information from program memory. The BBA preprocessor (-b option) must
be used at compile time in order for this information to exist in program
memory. The file name *bbadump.data* is used as the default name of all dump
files if none is specified in the command.

Once this information has been unloaded, it can be formatted with the BBA
report generator, *bbarep* (see the *HP Branch Validator for AxLS C User's
Guide*).

Note

The Unload_BBA command is disabled when the debugger option Demand_Load is *On*. If Demand_Load is *oFF* but the program was loaded with Demand_Load On, the Memory Unload_BBA command will generate a BBA file with incomplete information. See the Debugger Option General command description in this manual for more information on the Demand_Load option.

Memory Unload_BBA All

The Memory Unload_BBA All command unloads branch analysis information associated with all absolute files loaded into the file *bbadump.data*.

This command lets you run *bbarep* without specifying a file name. The file name *bbadump.data* is used as the default name of all dump files.

Memory Unload_BBA All To < "dump_file">

The Memory Unload_BBA All To < "dump_file"> command unloads branch analysis information associated with all absolute files loaded into < "dump_file"> .

Memory Unload_BBA Load_File < "load_file">

The Memory Unload_BBA Load_File command unloads only basis branch information associated with the specified absolute file (< "load_file">) into the file *bbadump.data*.

This command lets you run *bbarep* without specifying a file name. The file name *bbadump.data* is used as the default name of all dump files.

Memory Unload_BBA Load_File < "load_file"> To < "dump_file">

The Memory Unload_BBA Load_File < "load_file"> To < "dump_file"> command unloads only basis branch information associated with the specified absolute file (< "load_file">) into the file < "dump_file"> .

Examples

To unload all branch analysis information into file "bbadump.data":

Memory **U**nload_BBA **A**ll

To unload all branch analysis information into file "mydata":

Memory **U**nload_BBA **A**ll **T**o "mydata"

To unload branch analysis information associated with absolute file a.out.x into file "bbadump.data":

Memory **U**nload_BBA **L**oad_file "a.out"

To unload branch analysis information associated with absolute file a.out.x into file "mydata":

Memory **U**nload_BBA **L**oad_file "a.out" **T**o "mydata"

Program Context Display

```
( Program )  →  ( Context )  →  ( Display )  →  [ <Return> ]
```
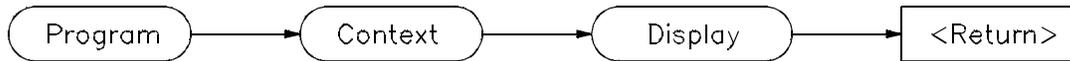
The Program Context Display command displays the current module, function, and line number in the Journal window. The current module is the one pointed to by the program counter.

This command will display both the view context, as set by a Program Context Set command, and the context of the current program counter, if the two are different.

Example

To display the current module, function, and line number:

Program **C**ontext **D**isplay

Current context is: @ecs\\main\main On line 81

See "Expression Elements" section of the "Expressions and Symbols in Debugger Commands" chapter for a description of debugger operators.

Note

If the PC does not point to a valid module, an alternate context is displayed. The alternate context is the name of the executable file that has been loaded into the debugger.

Program Context Expand

Program → Context → Expand → @<stack_level> → <Return>

The Program Context Expand command displays values of the parameters passed to a function, and the local variables in a function. The values are displayed in the Journal window.

To display a function's calling parameters and local variables, specify the function's stack level preceded by an at sign (@). The Backtrace window in high-level mode displays the function calling chain from the main program to the current function. The debugger displays the function stack (nesting) level beside each function name. The current function is level 0, the caller is always 1, etc.

You can use the Program Context Expand command to display the local variables and parameters of any function shown in the backtrace window. The calling parameters and local variables are accessible on the C run-time stack for functions in a directly-called chain from the main program to the current function.

See Also   Expression Display_Value
Expression Monitor
Symbol Display

Example   To display local variables and calling parameters of the function at stack level 2:

Program **C**ontext **E**xpand @2

Program Context Set

```
( Program )─→( Context )─→( Set )─────────────────────┬─→ <Return>
                                  │                     │
                                  ├─→ <module_name> ────┤
                                  │                     │
                                  └─→ <function_name> ──┘
```
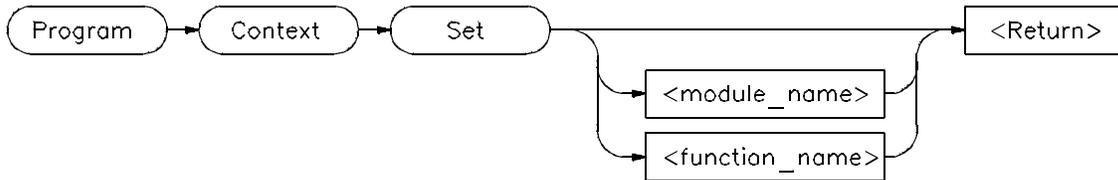
The Program Context Set command changes the default module and function (context). The current module (the one to which the program counter is pointing) is the default when functions are referenced without a module or function qualifier.

The default module reverts to the current module when you invoke any command that causes program execution, or if you execute the Program Context Set command without a parameter.

Example

To select module 'updateSys' as the current module:

Program **C**ontext **S**et updateSys

Program Display_Source

```
( Program )──→( Display_Source )──────────────────────────→│ <Return> │
                              │                          │
                              └──→│ #<line_nmbr> │───────┤
                              │                          │
                              └──→│ <function_name> │────┘
```
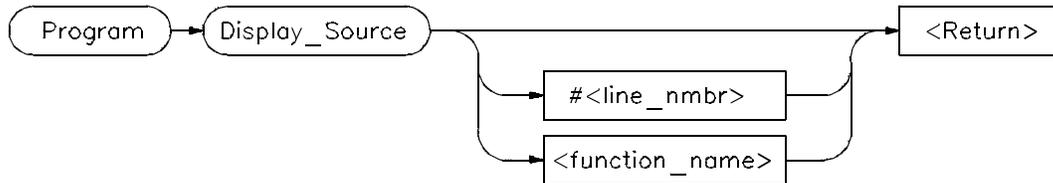
The Program Display_Source command displays C source code in the Code window beginning at the specified line or function. This command works in high-level mode only. If you do not specify a line number or function name, the debugger displays the line pointed to by the program counter.

You can display lines or functions in other modules by preceding them with a module name. The *Next Page*, *Prev Page*, *Up* arrow, and *Down* arrow keys may be used when the Code window is active to display code at higher or lower line numbers.

This command does not change the current program context.

See Also   Memory Display Mnemonic
Program Context Set
Program Find_Source

Examples   To display line 82 of the current module in the Code window:

Program **D**isplay_Source #82

To display the source code for function 'update_state_of_system' in the Code window:

Program **D**isplay_Source update_state_of_system

To display line 25 of module updateSys:

Program **D**isplay_Source updateSys\#25

Program Find_Source Next
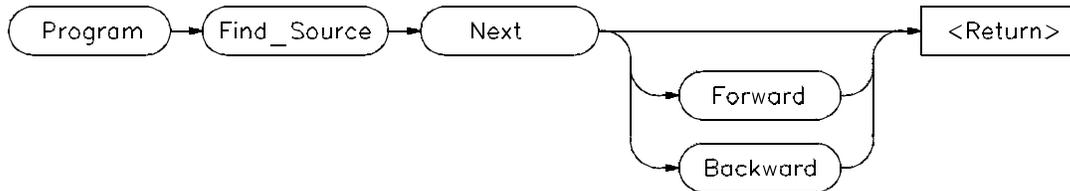
The Program Find_Source Next command searches a high-level source program for the next occurrence of the string specified in the last Program Find_Source Occurrence command. When the debugger finds the string, it displays the line containing the string at the top of the Code window.

If you specify *Forward*, the debugger searches forward through the file for the string.

If you specify *Backward*, the debugger searches backward through the file for the string.

If neither Forward nor Backward is specified, the debugger searches forward through the file for the string.

If the debugger cannot find the specified string, it displays the message "*string not found*". The screen remains unchanged.

See Also   Program Find_Source Occurrence

Example   To find the next forward occurrence of the string specified in the last Program Find_Source Occurrence command:

Program **F**ind_Source **N**ext

Program Find_Source Occurrence

```
( Program )──►( Find_Source )──►( Occurrence )──┬──►( Forward )──┐
                                                 └──►( Backward )─┤
   ┌─────────────────────────────────────────────────────────────┘
   └──►[ <string> ]──┬────────────────────────►[ <Return> ]
                     └──►[ ,#<line_nmbr> ]──┘
```
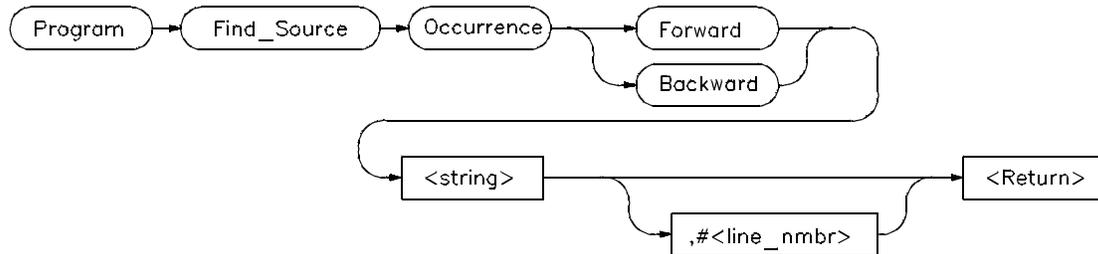
The Program Find_Source Occurrence command searches a high-level source
file for the first occurrence of the specified string. If you provide a line
number, the debugger searches for the string starting at the given line number.
If you do not specify a line number, the string search starts at the top of the
Code window.

If you specify *Forward*, the debugger searches forward through the file for the
string.

If you specify *Backward*, the debugger searches backward through the file for
the string.

You must enclose strings containing nonalphanumeric characters in quotation
marks. Quotation marks are not required if the string consists of only
alphanumeric characters.

If the debugger finds an occurrence of the string, it displays the line containing
the string at the top of the Code window. If the string does not exist or the
debugger cannot find it, the debugger displays the message "*string not found*".
The screen remains unchanged.

You can use the Program Find_Source Next command to search for the next
occurrence of the specified string.

If you specify a line number with a module reference, the debugger displays the
source code for that module in the Code window.

See Also　　　Program Display_Source
Program Find_Source Next

Examples   To search forward through the current module for the string 'time':

Program **F**ind_Source **O**ccurrence **F**orward 'time'

To search backward through the current module for the string 'time', starting at line 237:

Program **F**ind_Source **O**ccurrence **B**ackward 'time',#237

To search forward through the module 'main', for the string system_is_running, beginning at line 1:

Program **F**ind_Source **O**ccurrence **F**orward
"system_is_running", main\\#1

Program Load
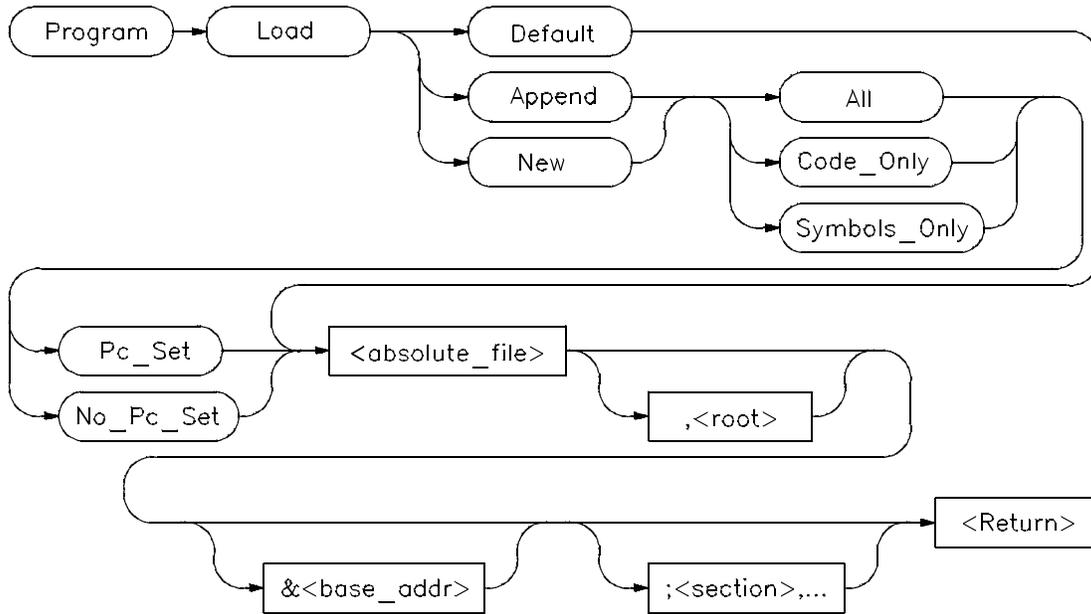
The Program Load command loads the specified executable module into the debugger.

Default Parameter

When you specify the Default parameter, the debugger:

- removes all previous program symbols
- removes all previously set breakpoints
- resets the program counter (PC)
- loads the full symbol set
- loads the executable module

New/Append Parameters

The *New* parameter loads a new program, removing any old program that may have been loaded. The *New* parameter optionally allows you to load the

program image, the program symbols, or both. The program counter can be set from the transfer address in the load file or ignored.

The *Append* parameter loads another program without deleting the existing program.

If you enter the Program Load command with the *New* or *Append* parameter, the following qualifiers are available:

All	Both the program image and program symbols to be loaded.
Code_Only	Only the program image is loaded.
Symbols_Only	Only the program symbols are loaded.
Pc_Set	The program counter is set from the transfer address in the load file.
No_Pc_Set	The program counter is not reset.

Using the All or Symbols_Only qualifiers along with the Pc_Set qualifier resets static variables for a complete restart.

The optional root parameter (,< root>) allows you to specify an alternate name for the root of the symbol tree.

The base address (&< base_addr>) allows PC relative code to be shifted upon loading.

The section list (;< section>) enables partial loading of absolute file sections, i.e., prog, data, const, etc. The symbols for all sections will be reloaded.

Resetting Program Variables

To reset static and global program variables after entering a Debugger Execution Reset_Processor or Program Pc_Reset command, you must reload your program by using the Program Load command. For faster loading, specify Program Load New Code_Only. The debugger retains symbol information. You do not have to reload symbol information if symbol addresses have not changed.

The address where the object module will be loaded is specified at link time. However, the address can be changed by specifying a new base address.

See Also   Debugger Execution Reset_Processor
                      Program Pc_Reset
                      Debugger Option General Demand_Load

Examples   To load absolute file 'ecs', remove all existing program symbols, reset the
                      program counter, and load the full symbol set:

                      Program **L**oad **D**efault ecs

                      To load only the program image of the prog section of absolute file 'ecs'
                      without resetting the program counter:

                      Program **L**oad **N**ew **C**ode_Only **N**o_Pc_Set ecs;prog

Program Pc_Reset

Program ──────────► Pc_Reset ──────────► <Return>

The Program Pc_Reset command resets the program counter to the transfer address from the absolute file. This causes the next Program Run or Program Step command to restart execution at the beginning of the program. The command does not clear breakpoints. All declared I/O ports still exist as originally specified.

See Also   Debugger Execution Reset_Processor
                    Program Load
                    Program Run

Example   To reset the program counter to the transfer address from the absolute file:

Program **P**c_Reset

Program Run

```
[Program]→[Run]
[From]→<start_addr>
  ,
[Until]→<break_addr>
          %%<pass_count>
          ,
                ;<macro_call>
          <Return>
```
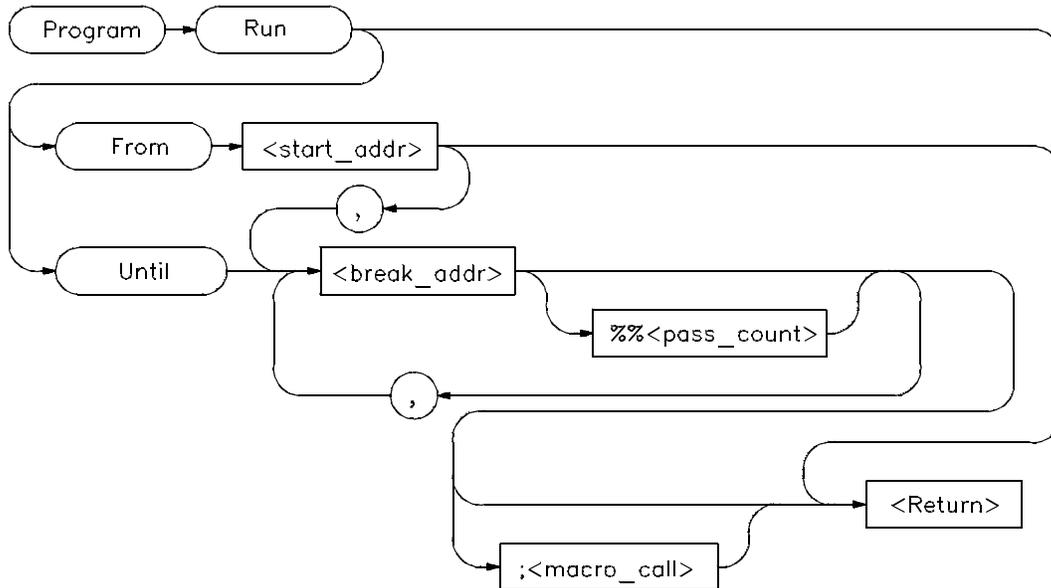
The Program Run command starts or continues target program execution. The program runs until it encounters a permanent or temporary breakpoint, an error, or a stop instruction, or until you press **CTRL C**.

The Program Run command may be used to resume execution after program execution has been suspended.

Program Run From

The Program Run From command begins program execution at the specified start address < start_addr> .

Using the Program Run From command to specify a starting address in high-level mode may cause unpredictable results if the compiler startup module is bypassed.

Program Run Until

The Program Run Until command begins program execution at the current program counter address and breaks at the specified address.

Break Address

The break address (< break_addr>) acts as a temporary instruction breakpoint. It is automatically cleared when program execution is halted. Multiple break addresses are ORed. For example, the command

Program **R**un **U**ntil #20,#90 **Return**

causes the program to run until either line 20 or line 90 is encountered, whichever occurs first.

Pass Count

The pass count (< pass_count>) specifies the number of times the break address is executed before the program is halted. For example, a pass count of three will cause the program to break on the fourth execution of the break address.

Macro Calls

If specified, a macro (< macro_name>) is invoked when the temporary break occurs.

See Also

Breakpt Access
Breakpt Clear_All
Breakpt Delete
Breakpt Instr
Breakpt Read
Breakpt Write
Program Pc_Set
Program Step

Examples

To execute the target program starting at address 'main':

Program **R**un **F**rom main

To begin program execution at the current program counter address and run until line 110 of the current module:

Program **R**un **U**ntil #110

To begin program execution at the current program counter address, run until the program returns to the calling function of the current function, and then execute the macro 'read_val':

Program **R**un **U**ntil @1;read_val()

Program Step
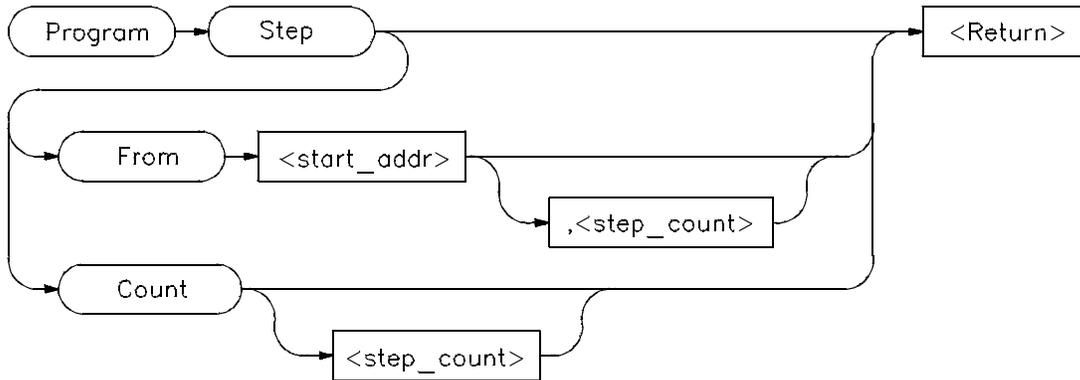
The Program Step command executes the specified number of instructions or lines, beginning with the location identified with < start_addr> . In high-level mode, single-stepping is done one C source line at a time. In assembly-level mode, single-stepping is done one machine instruction at a time. When the program calls a function, stepping continues in the called function.

If you do not specify a starting address, single-stepping begins at the address contained in the program counter.

If you do not specify a step count (< step_count>), the debugger will either step one C source line or one machine instruction.

Note   If the debugger steps into an HP library routine, you can then use the Program Run Until @1 (stack level 1) command to run to the end of the library routine.

Program Step From

The Program Step From command executes one instruction or line, beginning with the location specified by < start_addr> . If you do not specify the optional step count (< step_count>), the debugger executes one line or one instruction.

Program Step Count

The Program Step Count command executes the specified number of either instructions or lines, starting at the location pointed to by the program counter.

The debugger updates the screen after each instruction or line is executed. If a breakpoint is encountered, single-stepping is halted.

You can also use function key *F7* to single-step.

See Also   Breakpt Instr
Program Run
Program Step Over
Program Step With_Macro

Examples   To step four source lines, starting at line 39:

Program **S**tep **F**rom #39,4

To step ten source lines (high-level mode) or ten processor instructions (assembly-level mode), starting at the program counter address:

Program **S**tep **C**ount 10

To step one source line (high-level mode) or one processor instruction (assembly-level mode), starting at the program counter address:

Program **S**tep

Program Step Over

```
  ┌─────────┐    ┌──────┐    ┌──────┐                              ┌──────────┐
──┤ Program ├───▶┤ Step ├───▶┤ Over ├──────────────────────┬──────┤ <Return> ├──
  └─────────┘    └──────┘    └──────┘                       │      └──────────┘
                                                            │
     ┌───────┐         ┌──────────────┐                     │
   ┌▶┤ From  ├────────▶┤ <start_addr> ├────────────────┬───▶┤
   │ └───────┘         └──────────────┘                │    │
   │                         ┌────────────────┐        │    │
   │                       ┌▶┤ ,<step_count>  ├────────┘    │
   │                       │ └────────────────┘             │
   │ ┌───────┐             │                                │
   ┼▶┤ Count ├─────────┬───────────────────────────────────┘
     └───────┘         │
                ┌──────────────┐
              ┌▶┤ <step_count> ├──┐
              │ └──────────────┘  │
```
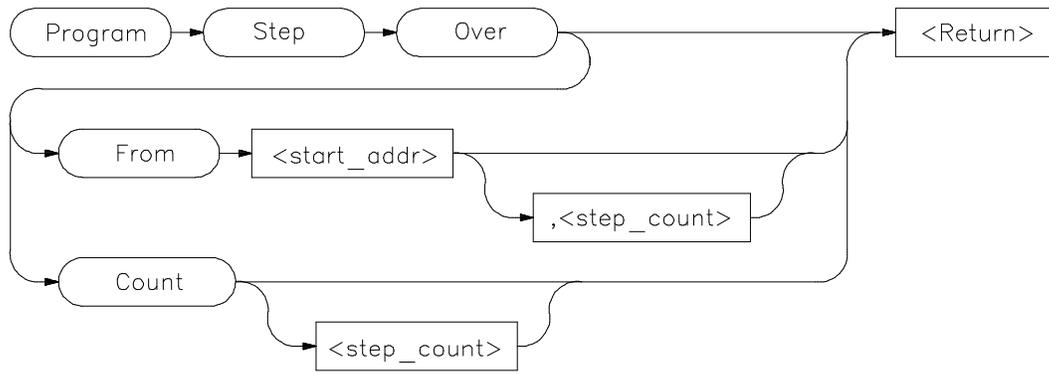
The Program Step Over command executes the number of instructions or lines specified, but executes through function calls, that is, the called function is executed without stepping through it. Execution begins at the specified starting address.

When the debugger encounters a C function or assembly-level JSR instruction, it stops stepping, executes the function or JSR, and then continues stepping when the called subroutine returns.

In high-level mode, the debugger executes one C source line at a time. In assembly-level mode, the debugger executes one microprocessor instruction at a time.

If you do not specify a starting address, single-stepping begins at the address contained in the program counter.

If you do not specify a step count (< step_count>), the debugger will either step one C source line or one machine instruction.

Program Step Over From

The Program Step Over From command executes one instruction or line, beginning with the location specified by < start_addr> . If you do not specify the optional step count (< step_count>), the debugger executes one line or one instruction.

Program Step Over Count

The Program Step Over Count command executes the specified number of either instructions or lines, starting at the location pointed to by the program counter. The debugger updates the screen after each instruction or line is executed. If the debugger encounters a breakpoint, it halts single-stepping.

You can also use function key *F8* to single-step over functions.

See Also

Breakpt Instr
Program Run
Program Step Count
Program Step From
Program Step With_Macro

Examples

To step four source lines, starting at line 39, and execute through any function calls:

Program **S**tep **O**ver **F**rom #39,4

To step ten source lines (high-level mode) or ten processor instructions (assembly-level mode), starting at the program counter address, and execute through any function calls:

Program **S**tep **O**ver **C**ount 10

To step one source line (high-level mode) or one processor instruction (assembly-level mode), starting at the program counter address, and execute through any function calls:

Program **S**tep **O**ver

Program Step With_Macro

```
( Program )──▶( Step )──▶( With_Macro )──▶│ <macro_call> │──▶│ <Return> │
```

The Program Step With_Macro command single steps through the program and executes the specified macro (<*macro_call*>) after each instruction or high-level line. Program execution continues if the macro returns a nonzero value.

Single-stepping is done by C source line in high-level mode and by microprocessor instruction in assembly-level mode.

See Also

Program Run
Program Step From
Program Step Over

Example

To step through the program one source line (high-level mode) or one processor instruction (assembly-level mode) at a time, executing the macro read_var after each step:

Program **S**tep **W**ith_Macro read_var()

Symbol Add
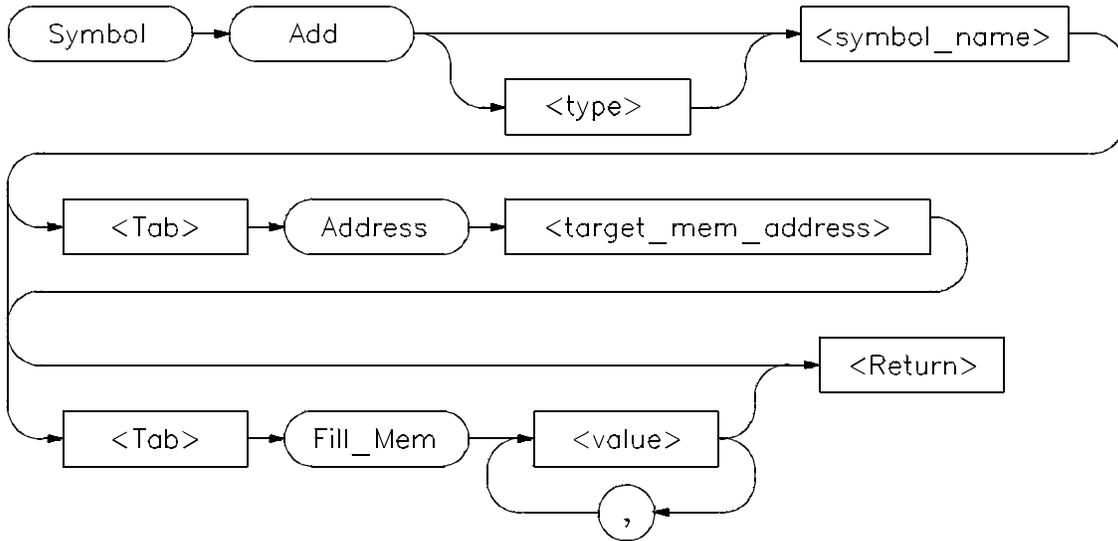
The Symbol Add command creates a symbol and adds it to the debugger symbol table. When defining a symbol, you must declare the symbol's name. It may be any name not previously used.

Type

You can optionally assign any valid C data type < type> to the symbol. If you do not assign a data type, the symbol type defaults to type int.

If the symbol type is a pointer, the initial value must be a data address. If the type is an array, the initial value must be a string of values separated by commas and/or enclosed in quotation marks. If fewer values are given than will fill the array, the pattern is repeated until the entire array is filled.

When initializing symbols, the symbol type is not used. Only the size is used. If a char array is defined, it is filled with the specified pattern in the same way as with the Memory Block_Operation Fill command. A zero is not appended to char arrays. The size is not determined by the string as in C. Complex values such as floating point representation are not recognized.

Program Symbols

Program symbols are specified with a base address (Address
< target_memory_address>). The base address references an address in target
memory. Program symbols are identical to variables defined in a C or assembly
language program. The value of a program symbol is placed in target memory.
If an initial value is specified for the program symbol, the value is loaded in the
memory location referenced by the symbol. If an initial value is not specified,
the memory location referenced by the symbol is not changed.

Debugger Symbols

Debugger symbols are specified without a base address and are not associated
with a target memory address. Debugger symbols may be used to aid and
control the flow of the debugger. They are located at a fixed location in
debugger memory. Only debugger commands and C expressions in macros can
refer to debugger symbols. They cannot be referenced by the program in target
memory.

If an initial value is specified for the debugger symbol, the value is loaded in
the memory location referenced by the symbol. If an initial value is not
specified, the memory location referenced by the symbol is set to zero.

See Also   Debugger Macro Add
Symbol Display
Symbol Remove

Examples   To add a program symbol of type int (default) at target memory address 9ff0h
and set the memory location to value -1:

Symbol **A**dd EOF **A**ddress 9ff0h **F**ill_Mem -1

To add a debugger symbol named str1 of type char referencing an
eight-character array and fill the array with string 'abcdefgh':

Symbol **A**dd char str1[8] **F**ill_Mem 'abcdefgh'

To add a debugger symbol of type short named s1 and fill the memory location
with value 0x10203:

Symbol **A**dd short s1 **F**ill_Mem 0x10203

In this example, we assigned a value to the symbol that is too large for the specified type. In this case, the debugger fills the memory location with the lower bytes of the specified value. Executing the command:

Expression **P**rintf "%x",s1

shows that the value is 203, the lower two bytes of the specified value.

Symbol Browse

```
( Symbol ) ━ ( Browse ) ━ <class_name> ━ <RETURN>
```
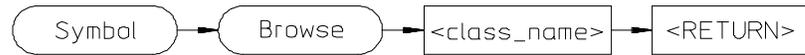
The Symbol Browse command displays the parents and children of a C+ +
class. The inheritance relationship is displayed in the Journal window.

Example
To display the parents and children of the C+ + class *fruit*, type:

Symbol **B**rowse fruit

Symbol Display
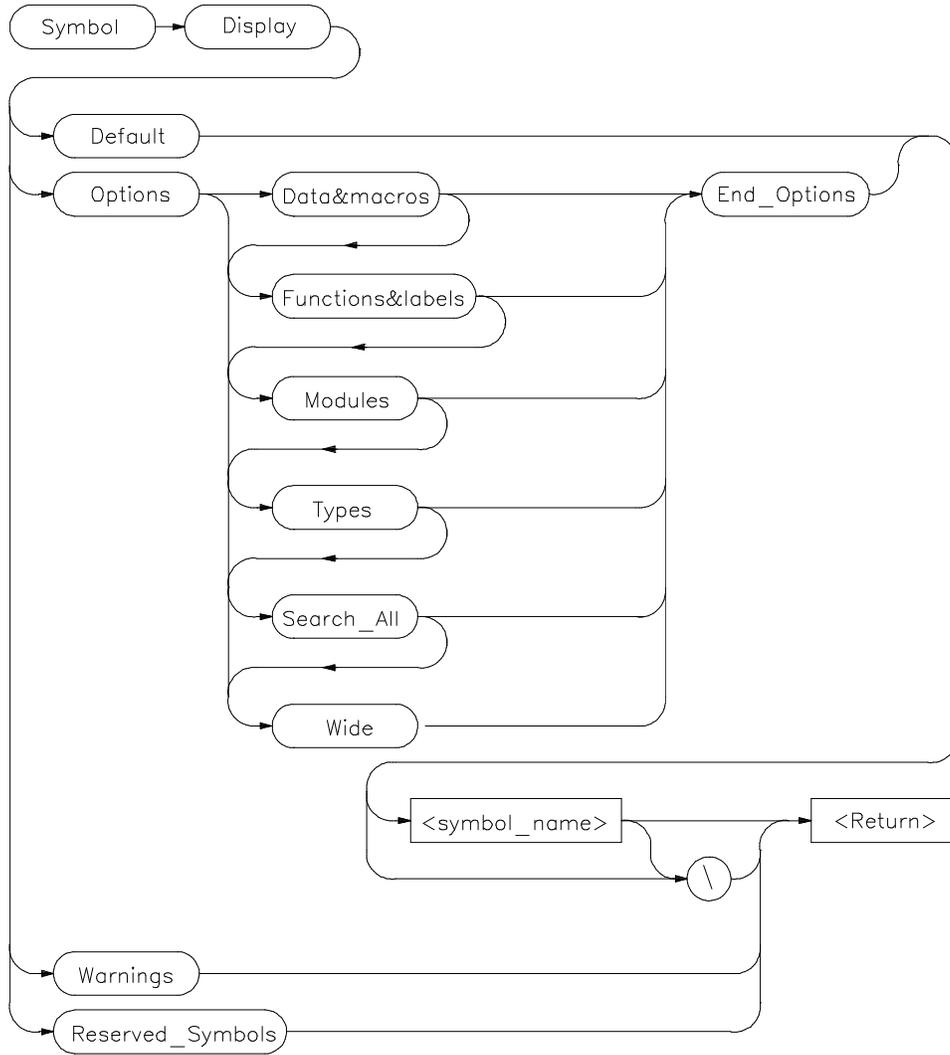
The Symbol Display command displays symbols and associated information in
the Journal window.

To display symbols in all modules, specify a backslash as the command argument.

Symbol **D**isplay **D**efault \

To displays all symbols in a specified module or function, enter a module name or function name followed by a backslash.

Symbol **D**isplay **D**efault memset\

The wildcard character * may be placed at the end of a symbol name with any option. The * can be used to represent zero or more characters. If used with no symbol name, * is treated the same as \, that is, all symbols are displayed.

If you enter a symbol name without a module specification, the debugger looks for the symbol in the current module. If there is no module qualifier, all symbols with the specified name will be displayed, including global symbols and symbols local to the module. Global symbols are not attached to a module.

Symbol **D**isplay **D**efault dest

If you specify a structure name using the Types option, the debugger shows all members in the structure and their types.

Default

If you specify Default, the debugger displays all types of symbols.

Options

The following options may be specified to display subsets of symbols.

Data&macros	displays symbol name, storage class, data type, and addresses of data and macro symbols.
Functions&labels	displays symbol name, storage class, data type, return type, and addresses of functions and labels.
Modules	displays names, module type (high-level, assembly-level, or non-loaded), and section addresses of modules.
Types	displays all symbol types.

Search_All   displays symbols of all types in all roots (contexts).

Wide   shows symbol names only in multicolumn (compressed) format.

If you do not specify any options, the debugger displays all symbols.

Warnings

When you execute the Symbol Display Warnings command, the debugger displays type mismatches. Mismatches occur when global variables are declared with different types in different modules or global functions are declared with different return types or argument counts in different modules. The command displays all mismatches and the names of the modules in which the symbols are declared.

Reserved_Symbols

If you specify Reserved_Symbols, the debugger displays processor reserved symbols, registers, and internal debugger variables.

See Also   Symbol Add
Symbol Remove

Examples   To display the symbol 'updateSys' in the current module:

Symbol **D**isplay **D**efault updateSys

```
 Symbol Display Default updateSys
     @ecs\\updateSys   : Type is High level module.
                          Code section = 00001436 thru 00001C21
```

To display all symbols in module 'updateSys':

Symbol **D**isplay **D**efault updateSys\

```
> Symbol Display Default updateSys\
  Root is: updateSys

     @ecs\\updateSys    : Type is High level module.
                           Code section = 00001436 thru 00001C21
  updateSys\update_state_of_system
                         : Type is Global Function returning void.
                           Address = 00001436 thru 00001513
```

```
    update_state_of\refresh
                          : Type is Local int.
                            Address = Frame + 8
    update_state_of\interval_complete
                          : Type is Local int.
                            Address = Frame + 12
 .
 .
```

To display all modules in the current symbol tree:

Symbol **D**isplay **O**ptions **M**odules **E**nd_Options \

```
 Symbol Display Options Modules End_Options \
  Root is: @ecs
     31 source and 23 assembler modules, 28 source procedures.
     Filename = ecs.x

     @ecs\\main         : Type is High level module.
                          Code section = 00001050 thru 00001121
                          Code section = 00000100 thru 0000010B
     @ecs\\initSystem   : Type is NON-LOADED module.
                          Code section = 00001122 thru 00001435
     .
     .
```

To display all function and labels in module 'main':

Symbol **D**isplay **O**ptions **F**unction&labels **E**nd_Options main\

To display all reserved symbols:

Symbol **D**isplay **R**eserved_Symbols

To display all symbols in module systemInt in compressed format (symbol names only):

Symbol **D**isplay **O**ptions **W**ide **E**nd_Options systemInt\

```
 Symbol Display Options Wide End_Options systemInt\
  Root is: systemInt

  systemInt\              system_interrupt function
  struct_system_clock     hours                   minutes
  seconds                 struct_system_clock *
  tick_clock function     argument_1              system_interrupt
  tick_clock              time                    reg_param1
  increment
```

To display all data and macros found within any symbol tree (that is, search \\, @a.out\\, @file1\\, etc.):

Symbol **D**isplay **O**ptions **D**ata&Macros **S**earch_All
End_Options \

```
Symbol Display Options Data&macros Search_All End_Options \
  Root is: @ecs
     31 source and 23 assembler modules, 30 source procedures.
     Filename = ecs.x

  update_state_of\refresh
                        : Type is Local int.
                          Address = Frame + 8
  update_state_of\interval_complete
                        : Type is Local int.
                          Address = Frame + 12
```

To display data type struct_temp_settings:

Symbol **D**isplay **O**ptions **T**ypes **E**nd_Options
struct_temp_settings\

```
Symbol Display Options Types End_Options struct_temp_settings\
  Root is: struct_temp_settings

     @ecs\\struct_temp_settings
                        : Type is Type definition of Structure, size
= 8.
  struct_temp_set\temp  : Type is Member of type float.
                          Offset = 0
  struct_temp_set\humid : Type is Member of type int.
                          Offset = 4
```

To display any type mismatches detected in the user program during program
loading, along with the address of the symbol that has the mismatch:

Symbol **D**isplay **W**arnings

```
2 mismatches.
getOpInput\system_off: Type is Global Function returning void.
                          Address = 000024AE to 000025ED
                          ***1 modules mismatched ***
                          Modules:'main'
updateSys\write_output_command
                        : Type is Global Function returning void.
                          Address = 00001944 to 0000197B
                          *** 1 modules mismatched ***
                          Modules: 'getOpInput'
```

Symbol Remove

The Symbol Remove command removes the specified symbol from the symbol table. Only program symbols and user-defined debugger symbols can be deleted from the symbol table.

To delete all symbols within a named module or function, append a backslash (\\) to the module or function name (< symbol_name>).

Symbol **R**emove updateSys\\

Entering a backslash without a module or function name deletes all symbols in all modules.

Symbol **R**emove \\

If you specify a symbol name without a module specification, the debugger looks for the symbol in the current module.

If you specify more than one symbol to be deleted or if the specified symbol has local symbols (for example, when a macro is deleted), the debugger requests confirmation. Entering *,y* after the symbol name provides automatic confirmation of the request. This option is useful in command files.

The debugger lets you add a debugger symbol with the same name as a target module's local symbol or a predefined macro's local symbol. If you do add a debugger symbol with same name as a local symbol, you must specify the entire symbol name with the Symbol Remove command in order to remove it. For example, if you added the debugger symbol *alter_settings* when running the demonstration program, you must enter *\\\\alter_settings* instead of *alter_settings* to delete the symbol because there is a local symbol alter_settings in target module updateSys. Otherwise the error message *error # 152, Cannot delete: more than one symbol with this name* is displayed.

See Also   Symbol Add
                      Symbol Display

Examples   To delete symbol 'current_targets' in function 'alter_settings':

                      Symbol **R**emove alter_settings\current_targets

                      To delete all symbols in module 'updateSys':

                      Symbol **R**emove updateSys\

                      To delete symbol 'alter_settings' in module 'updateSys':

                      Symbol **R**emove updateSys\alter_settings

                      In this example, the symbol being removed is a function which contains other
                      symbols. The debugger prompts you with the message 'This symbol has a
                      sub-tree. Delete with sub-tree? (Y/N)'. Enter 'Y' to delete the symbol and its
                      sub-tree. If you respond with 'N', the command is canceled.

                      To delete all symbols in all modules:

                      Symbol **R**emove \

Window Active
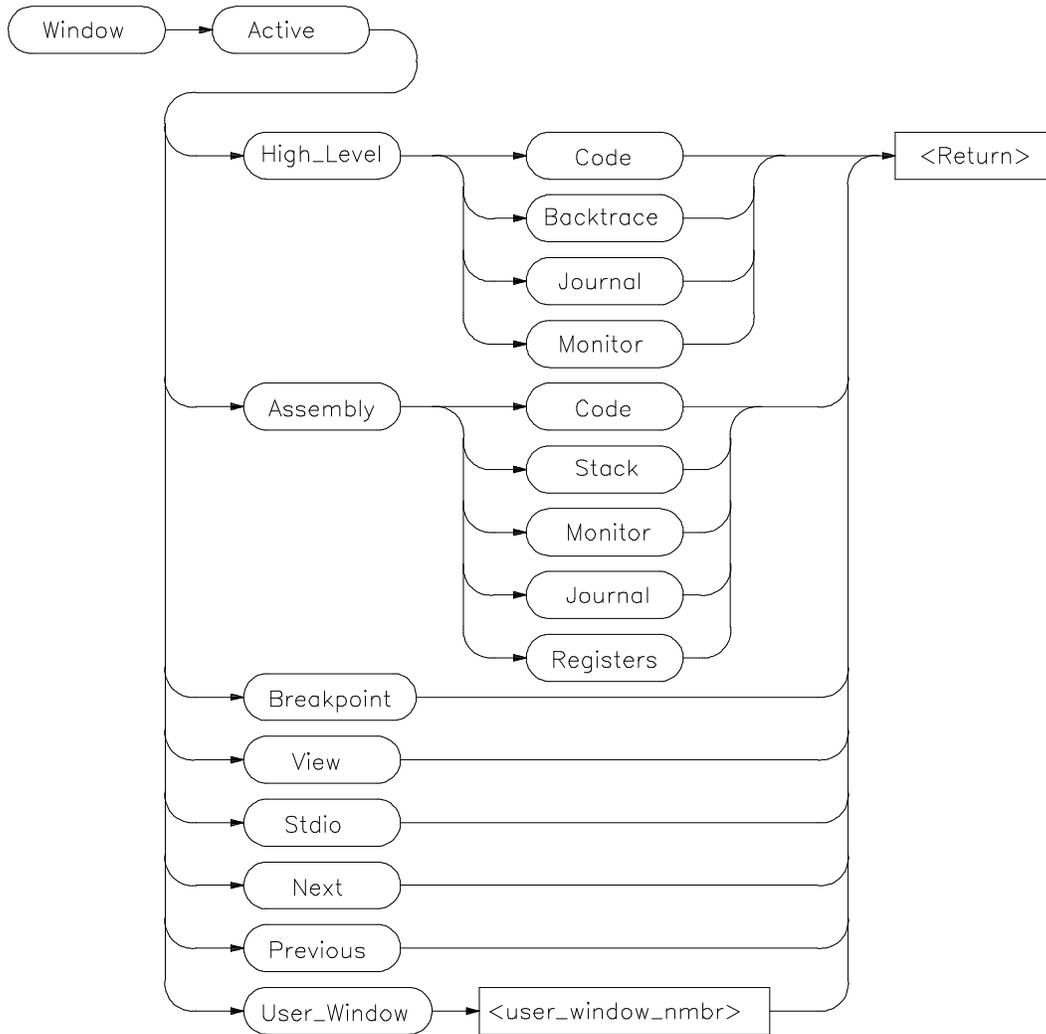
The Window Active command activates the specified window. The border of the active window is highlighted. The Code window is active by default within the high level and low level screens.

The Next and Previous parameters specify the next higher-numbered or lower-numbered window relative to the active window.

The cursor keys and the F4 function key only operate in the active window.

The Error, Help, and Status windows cannot be made active.

See Also   Window Cursor
Window Delete
Window Erase
Window New
Window Resize
Window Screen_On
Window Toggle_View

Examples   To make the high-level Backtrace window active:

Window **A**ctive **H**igh_Level **B**acktrace

To make the assembly Code window active:

Window **A**ctive **A**ssembly **C**ode

To make user window 57 active:

Window **A**ctive **U**ser_Window 57

Window Cursor

```
( Window ) → ( Cursor ) → [ <user_window_nmbr> ] → [ <tab> ] ⌐
                                                               │
  ⌐────────────────────────────────────────────────────────────┘
  └→ ( Position ) → [ <line,col> ] → [ <Return> ]
```
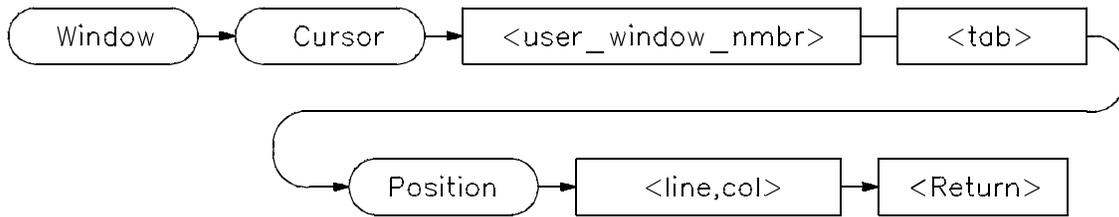
The Window Cursor command sets the cursor position in the window specified by < user_window_nmbr> . The top left corner of the window is represented by coordinates 0,0.

Subsequent output to the window begins at the cursor position.

Only user-defined windows and the standard I/O window (window No. 20) may be specified with this command.

See Also

Window Active
Window Delete
Window Erase
Window New
Window Resize
Window Screen_On
Window Toggle_View

Examples

To move the cursor to line 5, column 22 in the Stdio window:

Window **C**ursor 20 **P**osition 5,22

To move the cursor to line 3, column 0 in user window 57:

Window **C**ursor 57 **P**osition 3,0

Window Delete

```
( Window )──►──( Delete )──────────────────────────►──┌─<Return>─┐
                      └──►──┌─<user_window_nmbr>─┐──┘
```
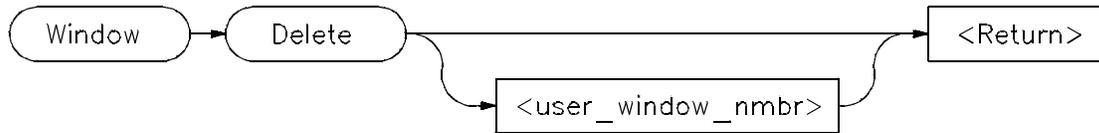
The Window Delete command removes a window (possibly a screen) defined previously with the Window New command. Remove a window by entering the window's associated window number. If you do not specify a window number or if you specify 0, the active window is removed.

Remove screens by removing all windows associated with that screen. For example, if a user-defined screen has three windows and you delete all three windows, the screen will be deleted as well. See the "Displaying Screens" and "Displaying Windows" sections of the "Viewing Code and Data" chapter for more information about window and screen numbers. Predefined debugger windows and screens cannot be removed.

Files opened with the File User_Fopen command may also be closed with this command.

See Also   File User_Fopen
File Window_Close
Window Active
Window Cursor
Window Erase
Window Open
Window Resize
Window Screen_On
Window Toggle_View

Example   To delete user window 57:

Window **D**elete 57

Window Erase
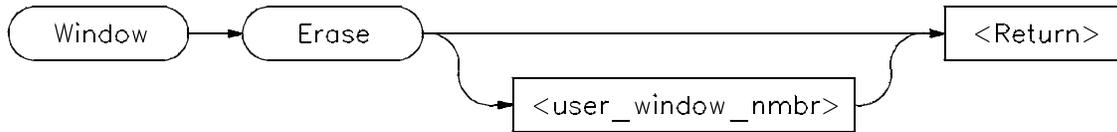
The Window Erase command clears all displayed information in the specified window. It then places the cursor in the specified window to the 0,0 position. If you do not specify a window number or if you specify 0, the active user-defined window is cleared. Only user-defined windows and the standard I/O screen (window No. 20) can be cleared. This command is primarily for use within macros.

See Also

Window Active
Window Cursor
Window Delete
Window New
Window Resize
Window Screen_On
Window Toggle_View

Examples

To clear all displayed information in the Stdio window:

Window **E**rase 20

To clear all displayed information in user window 57:

Window **E**rase 57

Window New
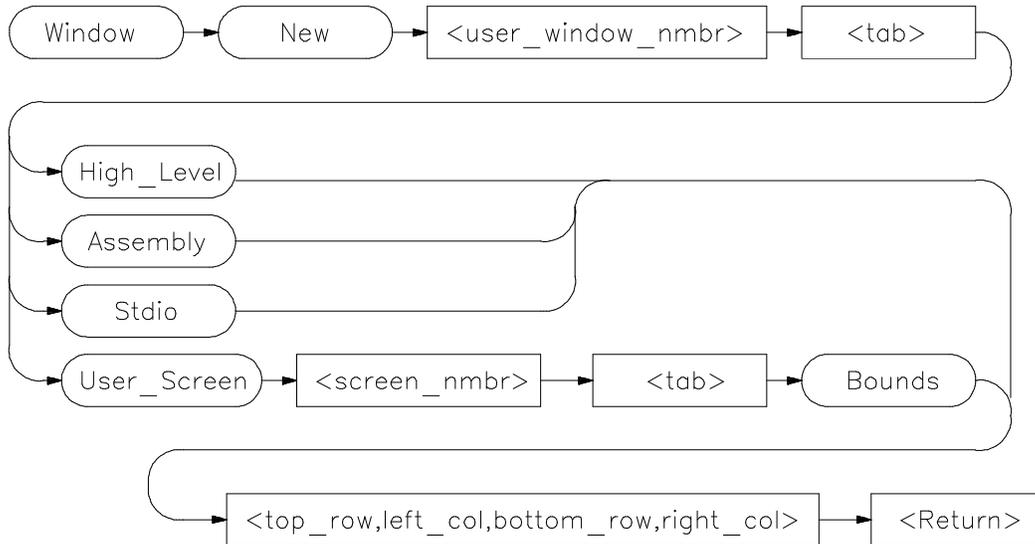
The Window New command makes (creates) new windows and screens. It may also be used to move existing windows to a new location within a screen. Windows must be assigned a number between 50 and 256 inclusive. Numbers 1 through 49 are reserved for predefined debugger windows. The bounds parameter specifies both the window size and location on the screen.

Window coordinates 0,0 correspond with the upper-left corner of the screen.

Note

When making new window, be careful not to enter coordinates that will result in a window that will cover the status line and command line.

On a standard 80-column by 24-row terminal display, a row coordinate may be between 0 and 23. However, creating a window whose bottom row coordinate is greater than 18 will cause part or all of the status line to be covered.

Command Parameters

Definition of the Window New command parameters are as follows:

Parameter	Definition	Range
< user_window_nmbr>	Window number	50 to 256 inclusive
< user_screen_nmbr>	User_Screen	4 to 256 inclusive
< top row>	Upper row coordinate	0 to N-1 inclusive
< left col>	Left column coordinate	0 to N-1 inclusive
< bottom row>	Lower row coordinate	0 to N-1 inclusive
< right col>	Right column coordinate	0 to N-1 inclusive

N is the number of rows or columns on your display. The value of N is dependent on display type.

Note

The Window New command will fail if row or column coordinates are greater than the screen boundary. For example, the command *Window New 15 Assembly 36,1,39,80* will fail if you have an 80 column by 40 row screen. The command *Window New 15 Assembly 36,0,39,79* will work.

Alternate Window Views

To create alternate views of a user-defined window, follow the procedure outlined below.

1 Execute the **Window New** command to define a window with specific size parameters.

2 Execute the **Window Toggle_View** command, or press function key **F4**.

3 Execute the **Window Resize** command to redefine the previously defined window with new size parameters. The new size parameters must be smaller than the previously assigned parameters.

See Also

Expression Fprintf
File User_Fopen
Window Active
Window Cursor
Window Delete
Window Erase
Window Resize

Window Screen_On
Window Toggle_View

Examples

To make a new user window, number it 57, and display it in user screen 4 with upper-left corner at coordinates 5,5 and the lower right corner at coordinates 18,78:

Window **N**ew 57 **U**ser_Screen 4 **B**ounds 5,5,18,78

To make a new user window, number it 55, and display it in the high-level screen with upper-left corner at coordinates 5,5 and the lower right corner at coordinates 10,20:

Window **N**ew 55 **H**igh_Level 5,5,10,20

To move the high level status line window to the top of the display in the standard interface:

Window **N**ew 5 **H**igh_Level 0,0,3,78

For this command to execute, the high-level window must be displayed and the difference between the bottom row coordinate and top row coordinate $(3 - 0)$ must equal three (3). You cannot move the status line if you are using the graphical interface.

Window Resize

```
( Window ) ──────────────▶ ( Resize ) ──────────────▶ [ <Return> ]
```

The Window Resize command lets you change the size and position of the active window interactively. The cursor keys (left, right, up, and down arrows) move either the top left corner, or the bottom right corner of the window.

To reposition the top left corner, press **T** and position the top left corner of the window using the cursor control keys.

To reposition the lower right corner of the window, press **B** and use the cursor control keys to position the lower right corner.

To move the window without resizing it press **M** and use the cursor control keys to move the window on the screen.

Press the **Return** key to save the new coordinates.

Press **CTRL C** or **Esc Esc** to restore the previous coordinates.

If an alternate window view is selected, the size alterations are made to the alternate view.

Note The Window Resize command can be used to alter the size of any existing window, including the predefined debugger windows, with the exception of the Status Line or View window. In the standard interface (but not in the graphical interface), the Status Line window can be moved or resized using the Window New command.

See Also Expression Fprintf
File User_Fopen
other Window commands

Window Screen_On
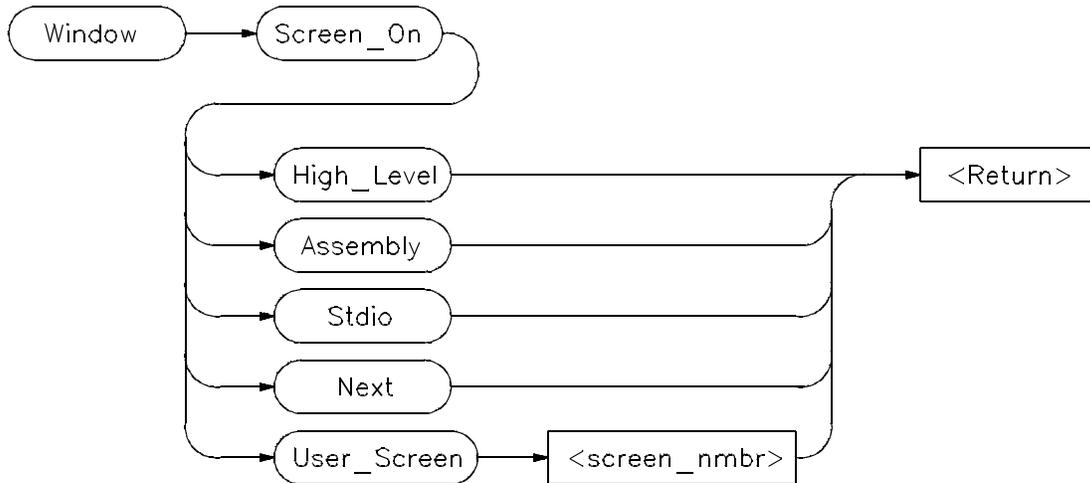
The Window Screen_On command displays the selected screen. You can also
use function key *F6* to display a screen.

If the high level screen is displayed, the debugger is placed in the high level
mode. Likewise, when you display the assembly level screen, the debugger is
placed in the low level mode.

See Also

Window Active	Window New
Window Cursor	Window Resize
Window Delete	Window Toggle_View
Window Erase	

Example
To activate the Assembly-level screen and place the debugger in low level
mode:

Window **S**creen_On **A**ssembly

Window Toggle_View
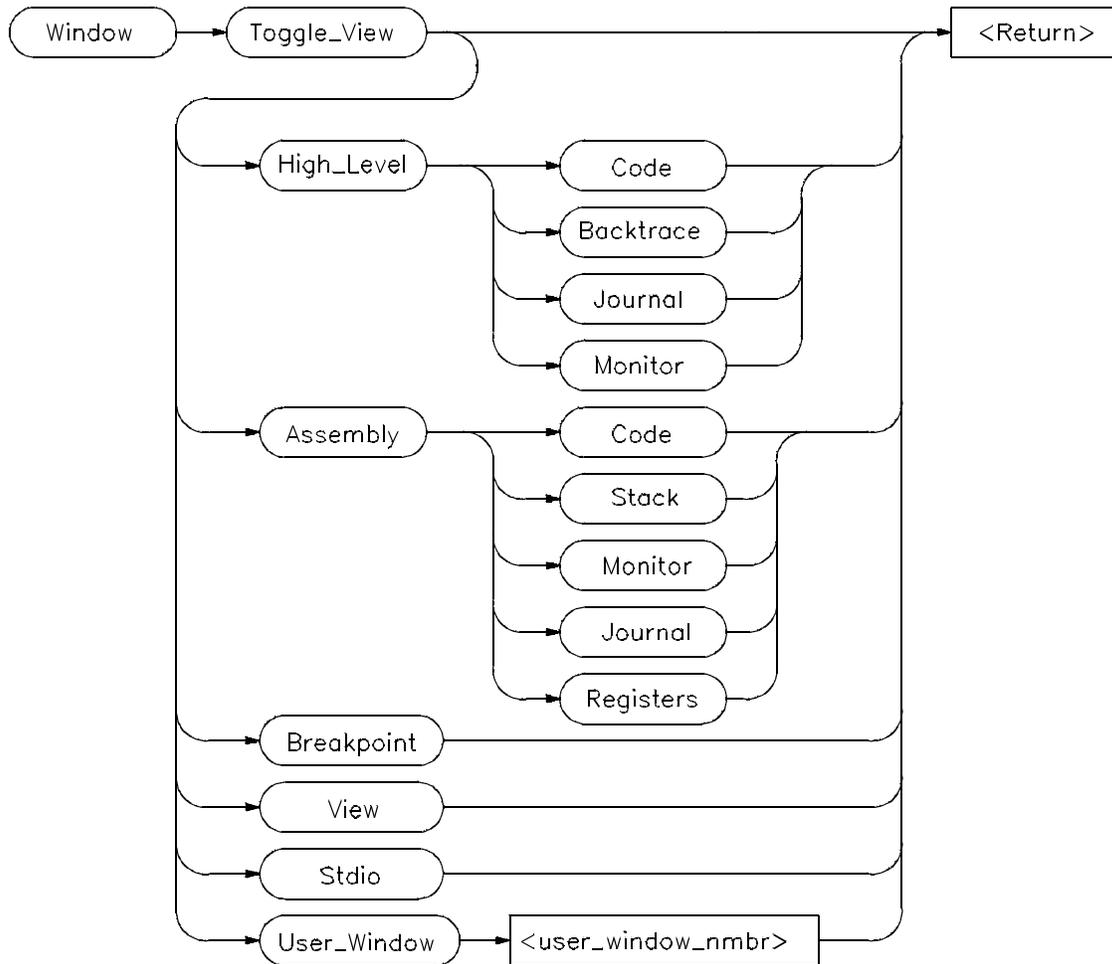
The Window Toggle_View command selects the alternate view of a window.
Typically, this is an enlarged view of the window. If you do not specify a
window number or if you specify 0, the active window is the default.

When you execute the Window Toggle_View command, the display alternates
between the two views of the window.

You can also use the *F4* function key to alternate views of the active window.

To create alternate views of a user-defined window, follow the procedure outlined in the Window New command description.

See Also   Window Active
Window Cursor
Window Delete
Window Erase
Window New
Window Resize
Window Screen_On

Examples   To display the alternate view of the active window:

Window **T**oggle_View

To display the alternate view of the high-level Code window:

Window **T**oggle_View **H**igh_Level **C**ode

To display the alternate view of user window 57:

Window **T**oggle_View **U**ser_Window 57

11

Expressions and Symbols in Debugger Commands
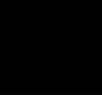
A description of the expressions and symbols you can use in debugger commands.

Expressions and Symbols in Debugger Commands

This chapter discusses the following language elements used in debugger commands:

- Expression elements.

- Formatting expressions.

- Symbolic referencing.

Debugger commands use standard C operators and syntax. This chapter describes the elements of C expressions and how expressions are structured. It also discusses memory and variable referencing.

Expression Elements

Most debugger commands require simple C expressions that evaluate to a scalar value. Simple C expressions are the same as standard algebraic expressions. These expressions evaluate to a single scalar value. Expressions consist of the following elements:

- operators
- constants
- program symbols
- debugger symbols
- built-in symbols
- macros
- keywords
- registers
- addresses
- address ranges
- line numbers

Debugger commands allow any legal C expression. The following paragraphs describe elements of C expressions used in debugger commands.

Operators

The debugger supports most standard C language operators and special debugger operators.

C Operators

C operators include arithmetic operators, relational operators, assignment operators, and structure, union, and array operators. The following table lists these operators in order of precedence (first line of the table is the highest precedence).

<div align="center">

Supported C Operators

</div>

Operators	Order of Association
( ) [ ] -> .	Left to right
~ ! ++ -- sizeof (type) - * &	Right to left
* / %	Left to right
+ -	Left to right
<< >>	Left to right
< <= > >=	Left to right
== !=	Left to right
&	Left to right
^	Left to right
|	Left to right
&&	Left to right
||	Left to right
= += -= *= /= %= &= ^= |= <<= >>=	Right to left
,,	Left to right

C++ Operators

The debugger also supports C++ operators: ::, ., ->, and &.

Debugger Operators

The debugger uses some characters as special debugger operators. These debugger operators and their descriptions are listed in the following table:

Debugger Operators

Operator	Description
[ ]	References the contents of a memory location. For example:

```
Expression Display_Value [0x20b0]
```

| # | Identifies a line number. For example: |

```
Program Run Until #82
```

| @ | Identifies a stack level, reserved symbols, or symbol tree root. For example: |

```
Program Display_Source @2
  (stack level)
Expression Display_Value @module
  (reserved symbol)
Symbol Display Default @ecs\\
  (symbol tree root)
```

' '	Identifies a character constant.
" "	Identifies a character string constant.
\	Qualifies a symbol reference. For example:

```
Program Run Until updateSys\#20
```

| \\ | Specifies an executable file as the root of a symbol tree. The specified file must be loaded into the debugger. For example: |

```
Program Context Set @ecs\\main
```

Constants

A constant is a fixed quantity. Constants may be integers, floating point values, or character string constants.

Integer Constants.

An integer constant may be defined as a sequence of numeric characters optionally preceded by a plus or minus sign. If unsigned, the debugger assumes the value is positive.

Positive integer constants may range between 0 and 2**31-1. When a constant is negative, its two's complement representation is generated. Negative integer constants may range to –2**31.

Constants can be specified as binary, decimal, or hexadecimal values. This is done by placing a prefix or suffix descriptor before or after the constant. The following table lists the legal prefixes or suffixes that may be specified with integer constants to denote a specific base.

Integer Constant Prefixes and Suffixes

Constant Type	Prefix Descriptor	Suffix Descriptor	Base	Digit
Binary		b, B	2	0-1
Decimal		t, T	10	0-9
Hexadecimal	0x,0X	h, H	16	0-9, A-F, a-f

Hexadecimal constants starting with the letters A through F (or a through f) must be prefixed with a zero. Otherwise, the debugger attempts to interpret the value as a symbol name.

By default, the debugger interprets integer constants as decimal values. The "Configuring the Debugger" chapter describes how to change the default radix for assembly-level values.

Note   You cannot use binary numbers when the radix is hexadecimal.

The debugger truncates values larger than that which can be contained in an element of an expression or command. The debugger extends values less than that allowed in the element. The truncation and extension are both implemented according to the rules of C.

The examples given in the following table show the use of prefix and suffix descriptors.

Prefix and Suffix Descriptor Examples

Constant	Decimal Mode	Hexadecimal Mode
73T	Decimal	Decimal
0EFF1h	Hexadecimal	Hexadecimal
10b	Binary	Hexadecimal
0x2214	Hexadecimal	Hexadecimal
23C3	Illegal	Hexadecimal
123	Decimal	Hexadecimal

Floating Point Constants

The debugger represents floating point constants internally in standard IEEE binary format. All floating point calculations follow the rules of C. The debugger treats all floating point constants as double precision values internally.

Floating point constants specified on the debugger command line must have the following syntax:

[sign] integer_part.[fractional_part] [exponent]

> where *sign* is an optional plus (+) or minus (–) sign.
>
> *integer_part* consists of one or more decimal digits.
>
> . is a decimal point.
>
> *fractional_part* may be zero or more decimal digits.
>
> *exponent* is an optional exponent, which is letter E (or e) followed by an integer part.

When specifying a floating point constant, the debugger uses a more restrictive syntax than the C language. The debugger always requires an integer part and a decimal point.
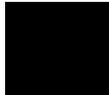
Examples: 76.3e-1   76.3   -0.3e1
                        76.3E+0   76.e5   0.3
                        76.3E2   76.   0.

Character Strings and Character Constants

Character Strings. A character string is a sequence of one or more ASCII characters enclosed in double quotation marks or two or more characters enclosed in single quotes. If the string has more than one character, subsequent ASCII characters are stored in consecutive bytes.

When a character string is referenced in a C expression, the debugger substitutes an address pointer to the string in the expression.

Character Constants.   A character constant is a single character enclosed in single quotation marks.

When a character constant is referenced in a C expression, the debugger substitutes the actual ASCII character value in the expression, not the address of the character.

Non-printable characters. Some non-printable characters may be embedded in both character strings and character constants enclosed in double quotation marks (") by using the escape sequences listed in the table which follows. Escape sequences are indicated by a backslash (\).

The backslash is interpreted as a character in character strings enclosed in single quotation marks (').

Any characters other than those listed in the following table are interpreted literally if preceded by a backslash. For example, to have literal double quotation marks in a string, enclose the string in double quotation marks and use the escape sequence for double quotes shown above. For example:

```
"This is a \"string\" using embedded double quotation
marks"
```

To have literal single quotation marks in a character string, enclose the string in double quotation marks. For example:

```
"This is a string that's using a single embedded
quotation mark"
```

Non-Printable Character Escape Sequences

Sequence	ASCII Name	Hex Value	Description
\b	BS	08	Back Space
\f	FF	0C	Form Feed
\n	NL	0A	New Line
\r	CR	0D	Carriage Return
\t	HT	09	Horizontal Tab
\"	"	22	Double Quote
\\	\	5C	Backslash
\xnumber[*]	—	xnumber	Hex Character Value

[*] \xnumber must be entered in the format \xnn where nn is a two digit hexadecimal value. For example: `\x0f`, not `\xf`

Note

The debugger automatically terminates character strings enclosed in quotation marks with a null character. However, when you use a character string with a Memory Assign or Memory Block_Operation (Fill, Search, or Test) command, the debugger uses only the characters within the quotation marks (null characters are not added).

Symbols

A symbol (also called an identifier) is a name that identifies a location in memory. It consists of a sequence of characters that identify program and debugger variables, macros, keywords,registers, memory addresses, and line numbers.

Symbols may be up to 40 characters in length. The first character in a symbol must be alphabetic, an underscore (_), or an at sign (@). The characters allowed in a symbol include upper and lower case alphabetic characters, numeric characters, dollar signs (*$*), at signs (@), or underscores (_). No other characters may be used in symbols. The debugger differentiates between upper case and lower case characters in a symbol.

The following sections describe the different categories of symbols used by the debugger.

Program Symbols

Program symbols are identifiers associated with a source program. They consist of symbolic variable data names and function names that the programmer defined when writing the source program. All symbols that were defined in the source program can be passed to the debugger and referenced during a debugging session. Note that preprocessor names are not symbols.

The compiler includes all program symbol information in the resulting output object module file by default. When you load an executable file for debugging, the debugger places all program symbols into the debugger symbol table by default. The debugger preserves symbol types and treats the symbols according to their type.

The debugger may be instructed to load only global symbols at load time, loading local symbols as they are referenced. This behavior is known as *symbols on demand.* Refer to the description of the Debugger Option General Demand_Load command in the "Debugger Commands" chapter for more information on *symbols on demand.*

Normally, the compiler prefixes a leading underscore to all global program symbols. This is done to distinguish program symbols from reserved assembler names. If the debugger has loaded all symbols, two symbols will be available; the high-level symbol (for example, *main*), and its low-level

counterpart(*_main*). However, with symbols on demand, only the high-level symbol is available (*main*).

Debugger Symbols

Debugger symbols can be added during a debugging session using the Symbol Add command. The debugger treats debugger symbols as global symbols. When you create a debugger symbol, you must assign it a name. You may optionally assign it a type. An initial value may also be given to a debugger symbol. If you do not specify an initial value, the initial value defaults to zero.

Debugger symbols are stored in the debugger's memory and are not associated with the processor target memory.

Macro Symbols

You can use macros to:

- Create complex user commands.
- Patch your source code temporarily.
- Display information in user-defined windows.

A macro is similar to a C function. It has a name, return type, optional arguments, optional macro local symbols, and a sequence of statements.

There are two types of macro symbols:

- Macro names.
- Macro local symbols.

Macro Names

Macro names identify a macro. You assign macro names with the Debugger Macro Add command.

Macro Local Symbols

Macro local symbols are local variables and parameters defined within macros. They are declared when you create a debugger macro with the Debugger Macro Add command. A macro local symbol can be accessed only by the macro in which it is defined. It is created when the macro is executed. The macro local symbol has an undefined initial value.

Reserved Symbols

Reserved symbols are reserved words that represent processor registers, status bits, and debugger control variables. These symbols are always recognized by the debugger. You can use reserved symbols any time during a debugging session. Reserved symbols have special meanings within the debugger command language. They cannot be defined and used for other purposes. To avoid conflict with other symbols, the names of all reserved symbols begin with a commercial at sign (@).

See the "Reserved Symbols" chapter for a complete list of reserved symbols and their descriptions.

Line Numbers

Line numbers can be used to refer to lines of code in your original source program. The compiler generates line numbers by default.

Line number references must be preceded by a pound sign (#). For example:

Program **R**un **U**ntil #82

When you refer to a source line number, the debugger translates it to the address of the first instruction generated by the compiler for that C statement. If a C source line did not generate executable code, a reference to that line number actually refers to the next line that did generate executable code.

To reference a line number that is in a module other than the current one, precede the line number with a module name. For example:

Breakpt **I**nstr updateSys\#332

If supported by your compiler, you can debug multiple statements on one line. A dot qualifier (**.**) identifies the sequence of a statement on the source line. A colon qualifier (**:**) identifies a column number within the source line. Hewlett-Packard cross assemblers do not support multi-statement debugging.

Addresses

An address may be represented by any C expression that evaluates to a single value. The C expression can contain symbols, constants, line numbers, and operators.

Code Addresses

Code addresses refer to the executable portion of a program. In high level mode, expressions that evaluate to a code address cannot contain numeric constants or operators.

Data and Assembly Level Code Addresses

Data addresses refer to the data portion of a program. Data address and assembly level code address expressions may be represented by most legal C expressions. There are no restrictions on constants or operators.

Address Ranges

An address range is a range of memory bounded by two addresses. You specify an address range with a starting address, two periods (..), and an ending address. These addresses can be actual memory locations, line numbers, symbols, or expressions that evaluate to addresses in memory.

You can also specify a byte offset as the ending address parameter. If you specify a byte offset, the debugger adds the specified number of bytes to the starting address and uses the resulting address as the ending address. You must precede a byte offset with a plus sign (+).

You may specify module names before symbols and line numbers to override the default module.

The following examples show how to specify address ranges.

To set instruction breakpoints starting at line number 80 and ending at line number 90:

```
Breakpt Instr #80..#90
```

To display code as bytes starting at line number 82 and ending at address 10d0 (hex):

```
Memory Display Byte #82..0x10d0
```

To display code as bytes, starting at memory location *tick_clock* and ending at 20 bytes past *tick_clock*:

```
Memory Display Byte tick_clock..+20
```

To map memory to RAM, starting at memory address 3000h and ending 0fffh bytes after address 3000h:

```
Memory Map Write_Read 3000h..+0fffh
```

Keywords

Keywords are macro conditional statements that can be used in a macro definition. These keywords are very similar to the C language conditional statements. You cannot redefine keywords or use them in any other context. The debugger keywords are listed below.

IF
ELSE
FOR
WHILE
DO
BREAK
CONTINUE
RETURN

Forming Expressions

The debugger groups expressions into two classes:

- Assembly language expressions used in assembly level mode.

- Source language expressions used in either assembly level mode or high level mode.

When you use a source language expression to express a code address in high level mode, it can consist only of a single symbol or a single line number. Source language expressions cannot contain numeric constants or operators. This restriction reduces confusion when entering high level expressions. There are no restrictions on source language expressions that evaluate to data addresses or on assembly language expressions.

Examples of legal and illegal source language code expressions in high level mode are shown below.

Legal	# 80
	main
Illegal	# 80+ 3
	main+ 10

With several commands, the size of an expression can be specified by size qualifiers. The size qualifiers are explained in the "Debugger Commands" chapter.

You may use C+ + classes in expressions.

Floating point calculations follow the rules of C. Single precision numbers are converted to double precision, the specified operation is done, and the result is translated back to single precision.

Note   Any value can be treated as an address. For example, a character value (byte) can be treated as an address. You should be careful when using values as addresses.

Examples of valid expressions are shown in the following table.

Valid Expressions

Expression	Meaning
# 7	Line number reference (code address)
i	Symbol reference (value or address)
x+ (y*5)	Arithmetic operation (value or address)
default_targets[2]	Array reference (value or address)
assign_vectors	Function name reference (code address)

Expression Strings

An expression string is a list of values separated by commas. The expression string can contain expressions and ASCII character strings enclosed in quotation marks. For several commands, each value in an expression string can be changed to the size specified by the size qualifiers. If you change the size, the debugger pads elements that do not fit evenly. Examples of expression strings are shown in the following table.

Expression String Examples

String	Results
1,2,"abc"	Values 1 and 2, and ASCII values of abc.
3+ 4, time, mac1()	Value 7, value of time, results of calling the macro 'mac1'.
'1xyz123'	ASCII values.

Symbolic Referencing

The debugger references symbols in a different manner than the standard C language definition. Therefore, understanding how variables are allocated and stored in memory is important. The following sections describe symbol storage classes and data types. These sections are followed by a discussion on:

- Referencing symbols with root, module, and function names.

- Making stack references.

In the following paragraphs, the notion of a 'module' is synonymous with a file in C. In fact, the module name is simply the basename of the source file with no suffix.

Storage Classes

All variables and functions in a C source program have a storage class that defines how the variable or function is created and accessed. The storage classes are:

- extern (global)
- static
- automatic
- register

C preprocessor symbols are not available to the debugger. The following paragraphs describe each storage class used in a C source program.

Extern (global)

Global variables in a C program are declared outside of a function and are accessible to all functions. Storage for these variables is allocated only once. Thereafter, references are made to the previously allocated space.

Global functions can be called from any other function.

Static

Static variables in a C program are allocated permanent storage and can be local to a module or local to a function.

In C, static variables local to a module can only be accessed by functions in that module. In the debugger, static variables local to a module can be accessed either when a function is active in that module or when the variable is qualified by the module name in which it is defined. A static variable that is local to a function can only be accessed by the function in which it was declared, unless it is qualified by the module and function in which it is defined.

Static functions can only be accessed when the function is in the current module, unless the function is qualified by the module in which it is defined.

Automatic

Automatic variables are declared inside a function and are accessible only to that function. Storage for these variables is allocated on the stack when the function is called and released when the function returns. Automatic variables do not have an initial value (their values are not retained between function calls).

You can access an automatic (local) variable when it is local to the current function, or when its function is on the stack. Use the stack-level prefix @<*stack_level*> to access an automatic variable in a function on the stack.

Register

Register variables are also declared inside a function and are accessible only to that function. Storage for these variables is allocated in a specific hardware register when the function is called and released when the function returns. Register variables do not have an initial value (their values are not retained between function calls).

A register variable is accessible when it is local to the current function, or when its function is on the stack.

Note　　Breakpoints cannot be set on accesses to register variables. If you need to set breakpoints on a variable, make sure that it is allocated on the stack by declaring its type as automatic.

Data Types

All symbols and expressions have an associated data type. Assembly language modules may contain variables with the types BYTE, WORD, or LONG. The

debugger treats these types as unsigned char, unsigned short int, and unsigned long, respectively. A segment attribute indicates whether a variable was defined in a code segment or a data segment.

Source language modules may contain any valid C language data type. The data types for each type of module are listed in the following tables. The ranges of values are decimal representations.

Assembly Level Data Types

Type	Size	Range
BYTE (unsigned char)	8 bits, unsigned	0 to 255
WORD (unsigned short int)	16 bits, unsigned	0 to 65535
LONG (unsigned long)	32 bits, unsigned	0 to 4294967295

High Level Scalar Data Types

Type	Size	Range
char	8 bits, signed	–128 to 127
unsigned char	8 bits, unsigned	0 to 255
short int	16 bits, signed	–32768 to 32767
unsigned short int	16 bits, unsigned	0 to 65535
int	32 bits, signed	–2147483648 to 2147483647
unsigned int	32 bits, unsigned	0 to 4294967295
long	32 bits, signed	–2147483648 to 2147483647
unsigned long	32 bits, unsigned	0 to 4294967295
enum	8-32 bits, unsigned	0 to 4294967295
pointer	32 bits, unsigned	0 to 4294967295
float	32 bits	1.18×10^{-38} to $3.4 \times 10^{+38}$
double	64 bits	9.46×10^{-308} to $1.79 \times 10^{+308}$

High Level Complex Data Types	
Type	**Size**
struct	Combined size of members (plus possible padding)
union	Size of largest member
array	Combined size of elements

Type Conversion

The debugger does data type conversions under the following conditions:

- When two or more operands of different types appear in an expression, the debugger does data type conversion according to the rules of C.

- When arguments are passed to a macro function, the debugger converts the types of the macro's arguments to the types defined in the macro.

- When the data type of an operand is forced by type casting, the debugger converts the data type.

- When a specific type is required by a command, the value is converted by the debugger according to the rules of C.

Type Casting

Type casting forces the conversion of a debugger symbol or expression to a specified data type. The debugger converts the resulting value of the expression to the specified data type, as if the expression was assigned to a variable of that type. The debugger does not alter the contents of the variable.

You can cast debugger symbols and expressions into different types using the following syntax:

(typename) expression

For example, the following symbol is cast to type char:

(char) prime

The following example casts the variable expression ptr__char to type int:

(int) ptr__char

Unlike C, the debugger allows casting to an array. The following example casts the address of the symbol int_value to an array of four chars:

(char[4]) &int_value

This type of casting to an array can be used with both the Expression Display_Value and Expression Monitor_Value commands.

Special Casting

In addition to the standard C type casts, the following assembly level casts are also recognized by the debugger's expression handler.

(Q S)

This type cast coerces an expression into a quoted string. For example, assuming the symbol has a value of 0x61626364,

Expression **D**isplay_Value (Q S)

causes to be displayed as "abcd". Note that the expression evaluates to an address because the (Q S) type cast is semantically synonymous with the C type cast (char *).

(I A)

This type cast coerces an expression into an instruction address. For example, assuming the symbol int_val has a value of 0x400,

Breakpt **I**nstr (I A) int_val

sets an instruction breakpoint at address

(H D)

This type cast coerces an expression into a long word (4 bytes) and displays the value in hexadecimal format. For example, assuming the symbol char_val has a value of 0x3F,

Expression **D**isplay_Value (H D) char_val

will cause char_val to be displayed as 0x0000003F.

(H W)

This type cast coerces an expression into a word (2 bytes). For example, assuming the symbol has the value 0x12345678,

Expression **D**isplay_Value (H W)

will cause to be displayed as 0x5678.

(H B)

This type cast coerces an expression into a byte. For example, assuming the symbol has a value of 0x12345678,

Expression **D**isplay_Value (H B)

will cause to be displayed as 0x78.

Scoping Rules

References to symbols follow the standard scoping rules of C. For example, if the symbol 'x' is referenced, the debugger searches its symbol table for 'x' using the following priority:

- A variable local to the current macro (if any).
- A variable local to the current function (if any).
- A variable static to the current module (if any).
- A global variable or debugger symbol.

Referencing Symbols

Symbols are qualified (and therefore referenced) according to their context. Context in the debugger is defined by a symbol tree and, if applicable, by a module and function name.

Root Names

Within the debugger, the symbol table is represented as a hierarchical tree, with each level representing a scoping level. There are two types of symbol trees which exist within the debugger:

- non-program symbol tree
- program symbol tree

Non-program symbol tree. This tree is composed of non-program symbols. Only one non-program symbol tree exists. This tree is made up of:

- debugger symbols (@ pc, @ sp, etc.)
- macros
- user-defined debugger symbols

The root name of this tree is \\.

Program symbol tree. The second type of symbol tree is the program symbol tree. The debugger allows up to 30 program trees. This tree is made up of symbols which exist in the target program. Since there may be multiple program trees within the debugger, the root of a program tree is specified as @ *absfile*\\, where absfile is the name of the executable file with its suffix stripped. For example, the root name of the program tree associated with the executable file a.out.x would be @ a_out\\.

Note	Any embedded '.' characters in a file name are converted to underscores. This prevents conflicts with the '.' structure operator. For example, the module name of source file myfile.bar.c would be myfile_bar.

There is no method for generating a list of multiple program trees.

If two or more executable files with the same name are loaded, the debugger appends an underscore and number to one of the files to make the root names unambiguous. For example, loading two a.out.x files would result in the creation of two program trees, with root names a_out and a_out_1.

Whenever the is pointing to the code space of a program, the root name of the program's symbol tree is the *current* root. A shorthand notation for specifying the current root is the symbol \. For example, if the debugger is invoked without loading an executable file, the current root would be \\, which would be synonymous with \. However, once an executable file (a.out.x) is loaded with the set to an address within the executable's code space, the current root becomes @ *a_out*\\, which would be synonymous with \.

The reserved symbol "@ root" points to a character string representing the name of the current root, and the symbol "@ file" points to the name of the file containing the current PC. These may be empty strings ("") if the PC is outside of any defined symbol database.

Module Names

The C language does not contain the concept of a module. Within the context of the debugger, a module is a scoping level which is identical to the scoping level of a file in C. Module names (which are generated by the compiler), are derived from source file names by removing the suffix of the source file. For example, the module name associated with the source file myfile.c would be myfile. Module names are used to qualify symbol references within the program symbol tree. When used as such, they are separated from any following function name by a \.

Note

If files in two directories have the same name, they will have identical module names. Since the debugger cannot distinguish between the two modules, all references will resolve to the last loaded module.

Assembly level modules with multiple code sections. If assembly language modules have more than one code section, the debugger breaks the module down into sub-modules. For example, if the source file myfile.s had three code sections, the modules myfile, myfile_2, and myfile_3 would appear in the program's symbol tree. This module separation only affects the address ranges of the module, not the scoping, i.e. all symbols scoped under the file myfile.s would be scoped under module myfile.

Context. Some symbol references are dependent on the current context. See the examples in the following tables. The current context is based on the PC and consists of the current root, current module, and current function. To display the current context, execute the command:

Program **C**ontext **D**isplay **Return**

Symbolic Referencing With Explicit Roots

Example	Comment
`Symbol Display Default \\`	Display symbols scoped under the non-program root.

Symbolic Referencing With Explicit Roots

Example	Comment
`Symbol Display Default @a_out\\`	Display symbols scoped under the program root *a_out*.
`Symbol Display Default \`	Display symbols scoped under the current root.
`Symbol Display Default @a_out\\mod1`	Display symbol information for module *mod1* scoped under program root *a_out*.
`Symbol Display Default \mod1`	Display symbol information for module *mod1* scoped under the current root.
`Symbol Display Default @a_out\\mod1\`	Display symbols scoped under module *mod1* in program root *a_out*.
`Symbol Display Default \mod1\`	Display symbols scoped under module *mod1* in the current root.
`Breakpt Instr @a_out\\mod1\func1`	Set a breakpoint at the entry point to function *func1* in module *mod1* in program root *a_out*.
`Breakpt Instr \mod1\func1`	Set a breakpoint at the entry point to function *func1* in module *mod1* in the current root.
`Symbol Display Default @a_out\\mod1\func1\`	Display symbols scoped under function *func1* in module *mod1* in program root *a_out*.
`Symbol Display Default \mod1\func1\`	Display symbols scoped under function *func1* in module *mod1* in the current root.
`Breakpt Access @a_out\\mod1\func1\j`	Set a breakpoint on accesses of variable *j* scoped under function *func1* in module *mod1* in program root *a_out*.
`Breakpt Access \mod1\func1\j`	Set a breakpoint on accesses of variable *j* scoped under function *func1* in module *mod1* in the current root.

Symbolic Referencing With Explicit Roots

Example	Comment

Notes:

The variable *mod1* must be a module name.
The variable *func1* must be a function name.
The example pairs are equivalent if the current root is *a_out*.

Symbolic Referencing Without Explicit Roots

Example	Comment
`Symbol Display Default x`	Display symbol information for all symbols named *x* at any scoping level in any root.
`Breakpt Access x`	Set a breakpoint at the *x* found using the scoping rules described in this chapter.
`Symbol Display Default x\`	Display symbol information for global symbol *x* in the current root and all symbols scoped under *x*. *x* may be a variable, function, or module name.
`Breakpt Instr x\#18`	Set a breakpoint at line 18 of module *x*.
`Symbol Display Default x\y`	Display symbol information for local variable *y* in function *x* (or function *y* in module *x*) in the current context.
`Symbol Display Default x\y\`	Display symbol information for local variable *y* in function *x* (or function *y* in module *x*) in the current context and for all symbols scoped under *x\y*.
`Breakpt Access x\y\j`	Set a breakpoint at local variable *j* in function *y* in module *x* in the current root.

Evaluating Symbols

The debugger evaluates symbols in expressions using the rules of the C
language as follows:

- Function names and labels evaluate to addresses.

- Variables generally evaluate to the contents of the memory location at the address of the variable (the exception is unsubscripted array names which evaluate to addresses.)

The examples in the following table show the differences in evaluation of these symbol types.

Symbol Evaluation Examples

Example	Comment
Breakpt Instr foo	The symbol *foo* is a function name. The breakpoint is set at the address of *foo*.
Breakpt Access &i	*i* is a variable. Therefore, the debugger evaluates the symbol as the value of *i* rather than the address of *i*. The & operator causes the breakpoint to be set on the address of *i*.
Breakpt Access a	*a* is an array. The breakpoint is set at the address of the first element of the array.
Breakpt Access a[3]	A breakpoint is set at the address specified in a[3], not the address of a[3].
Breakpt Access &a[3]	A breakpoint is set at the address of a[3].

Stack References

When a function is invoked in C, space is allocated on the stack for local variables. If one function calls another function, all information is saved on the stack to continue execution when the called function returns. The caller function is now nested.

You can reference variables and functions on the stack implicitly or explicitly.

Implicit Stack References

The default compiler setting allocates storage for all local variables in a C program in registers, if possible. Variables that cannot be stored in registers are allocated storage on the stack. With the debugger, you can implicitly reference variables on the stack as follows:

- To refer to variables on the stack in the current function, specify the name of the variable. For example: *x*.

- To refer to a local variable in a nested function, specify the function name followed by a backslash and then the name of the local variable, for example, *main\i*.

Explicit Stack References

A function is allocated storage on the stack when it is executing, or when it has called another function. To refer to functions and variables on the stack explicitly, you must specify the function's nesting level preceded by a commercial at sign (@). The backtrace window in high-level mode displays nesting level information (for example, if the current function is @0, its calling function is @1, etc.). You may reference functions on the stack as follows:

- To refer to the address that the function will continue to execute from, specify the function nesting level preceded by an at sign (@). For example, the command *Program Run Until @1* executes the program until the current function returns to its caller.

- To refer explicitly to a local variable in a nested function, specify the function nesting level followed by a backslash and then the name of the variable. For example, the command *Expression Display_Value @3\str* references the local variable 'str' of the function at nesting level 3.

- To reference a function itself, enter the command Program Context Expand followed by a space and then the function nesting level. For example, the command *Program Context Expand @7* displays all information about the function at the specified level for that particular invocation. This information includes the name of the function, the current line number, and all local variables in the function and their values. See the Program Context Expand command syntax description in the "Debugger Commands" chapter for more information.
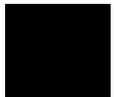
12

Reserved Symbols

The symbols listed in this chapter are predefined, reserved symbols for the 68030/040 debugger/simulator. Reserved symbols cannot be deleted by the user.

Note that reserved symbols may be entered in either upper or lower case characters.

Reserved Symbols		
Symbol	**Meaning**	**Maximum Value**
@A0-A7	Address Registers	0xFFFFFFFF
@AS	Access Level Status	0xFF
@C	Carry Flag	1
@CAAR	Cache Address Register	0xFFFFFFFF
@CACR	Cache Control Register	0xF
@CCR	Condition Codes Register	0x1F
@CHIP	Processor Chip Id. Number	
@CYCLES	Cycle Count	2**32-1
@D0-D7	Data Registers	0xFFFFFFFF
@DFC	Destination Function Code	7
@ENTRY	Address of first executable statement in function	
@EXC	Exception Handling Flag	1
@FILE	Name of file containing current PC	
@FUNCTION	Pointer to current function name	
@HLPC	High-Level Program Counter (line number)	32767
@I	Interrupt Mask	7
@ISP	Interrupt Stack Pointer	0xFFFFFFFF
@M	Master/Interrupt Flag	1
@MODULE	Pointer to current module name	
@MSP	Master Stack Pointer	0xFFFFFFFF
@N	Negative Flag	1
@PC	Program Counter	0xFFFFFFFF
@PI	Previous Instruction Address	
@PISIZE	Size of Previous Instruction	
@ROOT	Name of root of symbol tree represented by PC	
@S	Supervisor State Flag	1
@SFC	Source Function Code	7
@SP	Stack Pointer	0xFFFFFFFF
@SR	Status Register	0xFFFF
@SSP	Supervisor Stack Pointer	0xFFFFFFFF
@T0	Trace 0 Flag	1

Reserved Symbols		
Symbol	**Meaning**	**Maximum Value**
@T1	Trace 1 Flag	1
@USP	User Stack Pointer	0xFFFFFFFF
@V	Overflow Flag	1
@VBR	Vector Base Address	0xFFFFFFFF
@WAIT_STATE	Number of Wait States	0xFF
@X	Extend Flag	1
@Z	Zero Flag	1

13

Predefined Macros

Predefined Macros

Predefined macros are provided with the debugger. These predefined macros provide commonly used functions to help in debugging your program. The predefined macros available for your use are listed in the "Predefined Debugger Macros" table and are described on the following pages.

The following predefined debugger macros provide services to the SIMIO system and internal debugger functions. They are not designed for use by the debugger user. These names will be displayed if you check the debugger's predefined macro list using the Symbol Display command:

    bbaunload
    hpsimio
    hp_redirect
    hpnosimio
    hpioctl
    hpeofkbd
    hpioreport
    hpsimlock

Predefined Debugger Macros

Macro	Description
break_info	Display information about a breakpoint
byte	Return a byte value at the specified address
call	Call target function (not implemented in this product)
close	Close a UNIX file
dword	Return a long value at the specified address
error	Display error message
fgetc	Reads character from file
fopen	Open a file and associate it with a user window
inport	Advance the input data from its source
key_get	Get (read) a key from the keyboard
key_stat	Check keyboard for availability of key
memchr	Search for character in memory
memclr	Clear memory bytes
memcpy	Copy characters from memory
memset	Set the value of characters in memory
open	Open a UNIX file for reading and/or writing
outport	Write a value to the simulated memory-mapped outport port
read	Read from a system file
reg_str	Get the register value using the register name in the string
showversion	Show the software version number for the debugger product
strcat	Concatenate two strings
strchr	Locate first occurrence of a character in a string
strcmp	Compare two strings
strcpy	Copy a string
stricmp	Comparison of two strings without case distinction
strlen	String length
strncmp	Limited comparison of two strings
until	Run until expression is true
when	Break when expression is true
word	Return a word value at the specified address
write	Write to a system file

break_info

Function

Return information about a breakpoint

Synopsis

```
int break_info (addr)
unsigned long *addr;
```

Description

The break_info macro returns the address and type of a breakpoint if it is called when a breakpoint is encountered. The macro returns the 32-bit representation of the breakpoint address used by the debugger and the following values for breakpoint type:

-1	The cause of the breakpoint is unknown.
0	A breakpoint did not cause this macro call.
1	The breakpoint was caused by a read from the address.
2	The breakpoint was caused by a write to the address.
3	The breakpoint was caused by an access (read/write status unknown) of the address.
4	The breakpoint was caused by an instruction breakpoint.

Diagnostics

None.

Example

If you have the following code segment:

```
main()
{
    auto i,j,k;
    i = 1;
    j = 3;
    k = i + j;
}
```

and you execute the following command file:

```
Debugger Macro Add int print_info()
{
unsigned long address;
int reason;

reason = break_info(&address);
$Expression Printf "Breakpoint at %8x. Reason: %d\n",
address,reason$;
return(1);
}
.

Program Run Until main
Program Step
Breakpt Read &i;print_info()
Breakpt Write &k;print_info()
Breakpt Access &j;print_info()
Program Run
```

the debugger will display the breakpoint address and type value in the journal window.

byte

Function

Return a byte value at the specified address

Synopsis

```
unsigned char byte (addr)
void *addr;
```

Description

The byte macro returns a byte value of the memory contents at the specified address. The value of the expression *addr* is computed and used as the address.

Diagnostics

The byte value of the memory contents at the specified address is returned.

close

Function

Close a UNIX file

Synopsis

```
int close(fildes)
int    fildes;
```

Description

The close macro closes a UNIX file. This macro is an interface to the UNIX
system call *close(2)*. Refer to the *HP-UX Reference Manual* for detailed
information.

Diagnostics

If the system call to close(2) is successful, 0 is returned. Otherwise, *-1* is
returned and a system generated error message is written to the journal
window of the debugger.

Example

The following command file segment defines two global debugger symbols and
includes the definition of a user-defined macro that uses close().

```
Symbol Add int infile
Symbol Add int outfile

Debugger Macro Add int close_files(infile, outfile)
int     infile;      /* file descriptor to close */
int     outfile;     /* file descriptor to close */
{
    /* close input file */
    infile = close(infile);
    if (infile == -1)
        return 0;     /* close failed */

    /* close output file */
    outfile = close(outfile);
    if (outfile == -1)
        return 0;     /* close failed */

    return 1;     /* both files were closed successfully */
}
```

dword

Function

Return a long value at the specified address

Synopsis

```
unsigned long dword (addr)
void *addr;
```

Description

The dword macro returns a LONG (4-byte) value of the memory contents at the specified address. The value of the expression *addr* is computed and used as the address.

Diagnostics

The LONG value of the memory at that address is returned.

error

Function

Display error message

Synopsis

```
void error(level, text, parm)
int    level;
char  *text;
long   parm;
```

Description

The error() macro is used to display error messages due to errors generated within macros. *level* must have a value of 1, 2, or 3. *text* is a string which can contain one %d format character, where *parm* is the associated integer value.

level can be used to indicate the severity of the error by its value. The following explains the values available for *level*, and the associated action taken by error().

1 *text* is displayed in the journal window.

2 *text* is displayed in the journal window and the macro halts program execution.

3 An error box pops up, *text* is displayed within the box, and the macro halts program execution.

fgetc

Function

Reads character from file

Synopsis

```
int fgetc(vp_num)
int vp_num;
```

Description

The macro fgetc() returns the next character in the file associated with the window number vp_num. The window number must be a result of the File User_Fopen command. The value -1 is returned on end of file.

fopen

Function

Open a file and associate it with a user window

Synopsis

```
int fopen(vp_num, filename, mode)
int   vp_num;
char *filename;
char *mode;
```

Description

The macro fopen() opens a file and associates it with a user-defined window. This macro is equivalent to the File User_Fopen debugger command. *filename* is the name of the file to be opened. *mode* is a string that specifies the mode in which the file is opened. Valid modes are:

"r"   Open file for reading only

"w"   Open file for reading and/or writing (existing file contents
                are erased)

"a"   Open file for appending

Diagnostics

If successful, a window number is returned. The error code -27 indicates that the window is already open or that the window number is out of range. The error code -101 is returned for other errors; for example, if the file to be read does not exist.
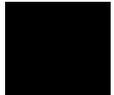
inport

Function

Advance the input data from its source

Synopsis

```
unsigned long inport (addr, size)
void *addr;
unsigned size;
```

Description

The inport macro moves *size* bytes of data into the port address specified by *addr* when the macro is executed. The value of *size* can be 1, 2, or 4. This macro allows inport buffers to receive new data from the defined source. The action is equivalent to the target program reading from the inport which will advance the input source.

If no inport exists at the port address, *value* is read from the the journal window.

If a port exists at the port address, *value* is read from the source specified when the inport was created.

Diagnostics

If an error occurs, inport returns a 0; if inport completes normally, it returns a 1.

Example

Memory **I**nport **A**ssign **B**yte 3 **S**ource_Is **D**ata_String "Hello"

Debugger **M**acro **C**all inport(3, 1)

Memory **I**nport **S**how 3

Note how *H* has been moved into the inport.

key_get

Function

Get a key from the keyboard

Synopsis

```
unsigned short key_get()
```

Description

The macro key_get() reads a key from the keyboard. It returns only after a key is available. The return value is the value of the key.

key_stat

Function

Check keyboard for availability of key

Synopsis

```
unsigned short key_stat()
```

Description

The key_stat() macro checks the keyboard to see if a key is available to read. It returns 0 if no key is available. The first pending key is returned if any keys are available.

Diagnostics

The value -1 is returned if the macro fails.

memchr

Function

Search for character in memory

Synopsis

```
char *memchr (str1, byte_value, count)
char   *str1;
char   byte_value;
unsigned count;
```

Description

The memchr macro locates the character *byte_value* in the first *count* bytes of memory area *str1*.

Diagnostics

The memchr macro returns a pointer to the first occurrence of character *byte_value* in the first *count* characters in memory area *str1*. If *byte_value* does not occur, memchr returns a NULL pointer. For debugger variables, -1 (0xFFFFFFFF) is returned if *byte_value* does not occur.
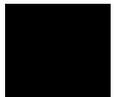
memclr

Function

Clear memory bytes

Synopsis

```
char *memclr (dest, count)
char    *dest;
unsigned count;
```

Description

The memclr macro sets the first *count* bytes in memory area *dest* to zero.

Diagnostics

The memclr macro returns *dest*.

memcpy

Function

Copy characters from memory

Synopsis

```
char *memcpy (dest, src, count)
char   *dest,
char   *src
unsigned count;
```

Description

The memcpy macro copies *count* characters from memory area *src* to *dest*.

Diagnostics

The memcpy macro returns *dest*.

memset

Function

Set the value of characters in memory

Synopsis

```
char *memset (dest, byte_value, count)
char   *dest;
char   byte_value;
unsigned count;
```

Description

The memset macro sets the first *count* characters in memory area *dest* to the value of character *byte_value*.

Diagnostics

The memset macro returns *dest*.

open

Function

Open a UNIX file for reading and/or writing

Synopsis

```
int open(path,oflag)
char  *path;
int   oflag;
```

Description

The *open( )* macro opens a UNIX file, returning an UNIX file descriptor. *path*
is the name of the file to be opened. *oflag* is the mode in which the file will be
opened. The possible modes may be found in the header file
/usr/include/fcntl.h. Some useful modes are:

read only	0
write only	1
read/write	2
no delay	4
append	8
create	256 (HP-UX) or 512 (SunOS)
truncate	512 (HP-UX) or 1024 (SunOS)

These modes may be combined be adding the appropriate values together.

This macro is an interface to the UNIX system call *open(2)*. Refer to the
HP-UX Reference Manual for detailed information.

Diagnostics

If the system call to open(2) is successful, the system file descriptor is
returned. Otherwise, *-1* is returned and a system generated error message is
written to the journal window of the debugger.

Example

The following command file segment defines two global debugger symbols and includes the definition of a user-defined macro that uses open().

```
Symbol Add int infile
Symbol Add int outfile

Debugger Macro Add int open_files(infile, outfile)
char    *infile;       /* file to read from */
char    *outfile;      /* file to write to  */
{
    /* open input file in read only mode */
    infile = open(infile, 0);
    if (infile == -1)
         return 0;     /* open failed */

    /* create output file in read/write mode */
    outfile = open(outfile, 258);
    if (outfile == -1)
         return 0;      /* open failed */

    return 1;      /* both files were opened successfully */
}
```

outport

Function

Write a value to the simulated memory-mapped outport port

Synopsis

```
char   outport (addr,  size, value)
void   *addr;
unsigned size;
long   value;
```

Description

The outport macro moves *size* bytes of the data *value* from the port specified by *addr. size* can be 1, 2, or 4. The action of this macro is equivalent to the target program writing to the outport destination.

If no outport exists at the port address, *value* is written to the journal window.

If a port exists at the port address, *value* is written to the destination specified when the outport was created.

Diagnostics

If an error occurs, outport returns a 0; if the macro ends normally, it returns a 1.

Examples

Memory **O**utport **A**ssign **B**yte 3 **D**estination_Is **F**ile
"/tmp/junk"

Debugger **M**acro **C**all outport(3,1,'h')

Memory **O**utport **S**how 3 /* Note the outport contains 'h'
*/

Debugger **H**ost_Shell cat /tmp/junk /* Note file
contains 'h' */

read

Function

Read from a system file

Synopsis

```
int read(fildes, buf, nbyte)
int     fildes;
char    *buf;
unsigned nbyte;
```

Description

The read macro reads from a system file. This macro is an interface to the
UNIX system call *read(2)*. Refer to the *HP-UX Reference Manual* for detailed
information.

Diagnostics

If the system call to read(2) is successful, the number of bytes read is returned.
Otherwise, *-1* is returned and a system generated error message is written to
the journal window of the debugger.

Example

The following command file segment defines two global debugger symbols and
includes the definition of a user-defined macro that uses read().

```
Symbol Add int infile
Symbol Add int outfile

Debugger Macro Add int foo(infile, outfile)
int     infile;     /* file descriptor to read from */
int     outfile;    /* file descriptor to write to  */
{
    char  buf[80];

    while (!read(infile, buf, 80))
          write(outfile, buf, 80);
}
```

reg_str

Function

Get register value

Synopsis

```
unsigned long reg_str(str1)
char *str1;
```

Description

The reg_str macro gets the contents of a register using a string variable
representation of its name. This is not possible using standard debugger
commands. The register value is returned by the macro.

Diagnostics

If the string does not contain a valid register name, an unknown value will be
returned and the debugger will display an error message in the debugger error
window.

Examples

To display the value of register D0:

```
Symbol Add char reg_name[10]
Debugger Macro Call strcpy(reg_name,"@D0")
Expression Display_Value reg_str(reg_name)
```

To display the value of register D1:

```
Expression Display_Value reg_str("@D1")
```

To display the value of register D1:

```
Expression C_Expression reg_str("@D1")
```

showversion

Function

Show the software version number for the debugger product

Synopsis

```
void showversion ()
```

Description

The showversion macro lists the software version number for your debugger product.

strcat

Function

Concatenate two strings

Synopsis

```
char *strcat (dest, src)
char *dest, *src;
```

Description

The strcat macro appends a string to the end of another string. The string in *src* is appended to the string in *dest* and a pointer to *dest* is returned.

Diagnostics

No checking is done on the size of *dest*.

strchr

Function

Locate first occurrence of a character in a string

Synopsis

```
char *strchr (str1, byte_value)
char *str1;
char byte_value;
```

Description

The strchr macro returns a pointer to the first occurrence of the character
byte_value in the string *str1*, if *byte_value* occurs in *str1*.

Diagnostics

If the character *byte_value* is not found, strchr returns a NULL pointer. For
debugger variables, -1 (0xFFFFFFFF) is returned if *byte_value* does not occur.

strcmp

Function

Compare two strings

Synopsis

```
unsigned long strcmp (str1, str2)
char *str1,
char *str2;
```

Description

The strcmp macro compares strings in lexicographic order. Lexicographic order means that characters are compared based on their internal machine representation. For example, because an ASCII 'A' is 41 hexadecimal and an ASCII 'B' is 42 hexadecimal, 'A' is less than 'B'.

The strings *str1* and *str2* are compared and a result is returned according to the following relations:

relation	result
s1 < s2	negative integer
s1 = s2	zero
s1 > s2	positive integer

Diagnostics

Strings are assumed to be NULL terminated or to be within the array boundaries. The comparison is always signed, regardless of how the string is declared.

strcpy

Function

Copy a string

Synopsis

```
char *strcpy (dest, src)
char *dest,
char *src;
```

Description

The strcpy macro copies *src* to *dest* until the NULL character is moved.
(Copying from the right parameter to the left resembles an assignment
statement.) A pointer to *dest* is returned.

Diagnostics

No checking is done on the size of *dest*.

stricmp

Function

Comparison of two strings without case distinction

Synopsis

```
unsigned long stricmp (str1, str2,)
char *str1;
char *str2;
```

Description

The stricmp macro compares *str1* with *str2* without case distinction. This means that the strings "ABC" and "abc" are considered to be identical.

The strings *str1* and *str2* are compared and a result is returned according to the following relations:

relation	result
s1 < s2	negative integer
s1 = s2	zero
s1 > s2	positive integer

Diagnostics

Strings are assumed to be NULL terminated or to be within the array boundaries because the comparison is limited to the number of stated characters. The comparison is always signed, regardless of how the string is declared.

strlen

Function

String length

Synopsis

```
unsigned long strlen (str1)
char *str1;
```

Description

The strlen macro returns the length of a string. It returns the length of *str1*, excluding the NULL character.

Diagnostics

If *str1* is not properly terminated by a NULL character, the length returned is invalid.

strncmp

Function

Limited comparison of two strings

Synopsis

```
unsigned long strncmp (str1, str2, count)
char *str1;
char *str2;
unsigned count;
```

Description

The strncmp macro compares strings in lexicographic order. Lexicographic order means that characters are compared based on their internal machine representation. For example, because an ASCII 'A' is 41 hexadecimal and an ASCII 'B' is 42 hexadecimal, 'A' is less than 'B'.

The *count* in the synopsis above specifies the maximum number of characters to be compared.

The strings *str1* and *str2* are compared and a result returned according to the following relations:

relation	result
s1 < s2	negative integer
s1 = s2	zero
s1 > s2	positive integer

Diagnostics

Strings are not required to be NULL terminated or to fit within the array boundaries because the comparison is limited to the number of stated characters. Less than *count* characters will be compared if the strings are smaller than *count* characters. The comparison is always signed, regardless of how the string is declared.

until

Function

Run until expression is true

Synopsis

```
char until (boolean)
int boolean;
```

Description

The until macro returns a zero when *boolean* is nonzero. The Until macro is used with the Program Run and Program Step With_Macro commands. It halts execution when the expression passed is true, and continues when the expression passed is false. Any C expression resulting in a value may be used.

Example

Program **R**un **U**ntil #3 ,#17 ,printf ;until (i==3 || x < y)

The command above sets temporary breakpoints at line numbers 3 and 17 in the current module and at entry to the function *printf*. When any one of these locations is encountered by the executing program, the debugger will stop and check the *until* conditional statements. If the variable *i* is equal to 3, or the variable *x* is less than *y*, a break will occur. Otherwise, program execution continues.

when

Function

Break when expression is true

Synopsis

```
char when (boolean)
int boolean;
```

Description

The when macro returns a zero when *boolean* is nonzero; it returns a one when *boolean* is zero. This macro is used with the Breakpt Instr command. When used with this command, program execution will halt when the stated expression is true, and will continue when the stated expression is false. Any C expression resulting in a value may be used.

Example

```
Breakpt Instr strcpy;when(*str==0)
```

This command sets a breakpoint at the entry point of the routine *strcpy*. Each time the breakpoint occurs, the when macro is executed. The macro causes program execution to stop when the byte pointed to by *str* is zero.

word

Function

Return a word value at the specified address

Synopsis

```
unsigned short int word (addr)
void *addr;
```

Description

The word macro returns a WORD (2-byte) value of the memory at the specified address. The value of the expression *addr* is computed and used as the address.

Diagnostics

The WORD value of the memory at that address is returned.

write

Function

Write to a system file

Synopsis

```
int write(fildes, buf, nbyte)
int    fildes;
char   *buf;
unsigned nbyte;
```

Description

The write macro writes to a system file. This macro is an interface to the UNIX system call *write(2)*. Refer to the *HP-UX Reference Manual* for detailed information.

Diagnostics

If the system call to write(2) is successful, the number of bytes written is returned. Otherwise, *-1* is returned and a system generated error message is written to the journal window of the debugger.

Example

The following command file segment defines two global debugger symbols and includes the definition of a user-defined macro that uses write().

```
Symbol Add int infile
Symbol Add int outfile

Debugger Macro Add int foo(infile, outfile)
int     infile;      /* file descriptor to read from */
int     outfile;     /* file descriptor to write to  */
{
    char  buf[80];

    while (!read(infile, buf, 80))
          write(outfile, buf, 80);
}
```

14

Debugger Error Messages

A list of the error messages generated by the debugger.

The debugger displays the error window whenever it detects a command error. The debugger displays an error message and a pointer to the location where it detected the error.

This chapter lists and describes the error messages and warnings issued by the debugger. These errors are listed numerically with possible error solutions.

4   **Invalid characters follow command.**

A command was entered with incorrect characters or with more characters than were expected. Check the command name and re-enter the command.

5   **This command is not implemented yet.**

The command specified is currently not supported, but will be implemented in a later release.

6   **Unknown switch.**

An attempt was made to specify a switch that does not exist. Check the command syntax for the switches supported.

7   **Argument missing.**

A command was entered without an argument that is required to execute the command. Check the syntax description for the command and enter the command again with the correct argument specification.

8   **Invalid argument.**

The argument specified is not valid for this command. Check command syntax and re-enter the command with a valid argument.

9   **Unexpected separator encountered.**

The argument separator is not valid in this context. Check the syntax and enter the correct separator.

10   **Unknown expression character.**

The specified expression character is not recognized by the debugger. Check the syntax and enter the correct expression character.

11   **Missing ') ', '] ', or ' } ' in expression.**

The matching right parentheses, right bracket, or right curly brace in the specified expression is missing. Check the expression and add the appropriate right delimiter.

12   **Missing ' (', ' [', or ' { ' in expression.**

The matching left parentheses, left bracket, or left curly brace in the specified expression is missing. Check the expression and add the appropriate left delimiter.

13   **Missing end quote.**

The second quotation mark for a character string is missing at the end of the line. Terminate the character string with an ending quotation mark.

14   **Invalid expression element.**

An expression element was specified incorrectly. The error window will display the expression specified and place a pointer at the position where the invalid element is located. Check the syntax description and re-enter the command. Possible errors include: invalid value, missing operand, missing operator, and unknown operand combination.

15   **Invalid filename.**

The filename specified could not be created. Valid filenames are dependent upon your host computer system.

16   **Invalid line number.**

The line number specified is not valid. Line numbers must be preceded with a pound sign (#), and must be in a valid range. This error will occur if you enter a pound sign followed by zero or if you enter a pound sign without a number.

17   **Invalid address value.**

This error indicates that a value was used for an address that cannot be interpreted as an address (for instance, a floating point number).

18   **Invalid structure member.**

A member name was given that is not a member of the specified structure. Member names must be members of the specified structure.

19   **Invalid instruction address.**

This error occurs mainly in high-level mode. In high-level mode, this error will occur if the instruction address is not a function name or line number. Code addresses in high-level mode may not be numeric or expressions. In assembly-level mode, most instruction address values are legal.

20   **Invalid port value.**

The specified port does not exist, or the port value was not specified with the Memory Inport Assign command. Port values must be specified with the Memory Inport Assign command.

21   **The values are not correct for this expression.**

An attempt was made to use an operand type that is not allowed for this operator. Operators must match operands according to the C language specifications.

22   **Upper bound less than lower bound.**

An attempt was made to specify a lower bound that is greater than the upper bound. The upper bound must be greater than the lower bound.

23   **Upper bound missing.**

An attempt was made to specify a lower bound without an upper bound. The upper bound must be specified.

24   **Function symbol ranges not allowed.**

An attempt was made to specify a range from one function to another in high-level mode. Function to line number is allowed.

25   **Range not of addresses.**

A print command was entered, but the specified range contained a value instead of an address. Place an ampersand (**&**) before the symbol name in the range.

26   **Invalid screen specification.**

The command entered contains a screen specification that does not correspond to the screen where the specified window is located, or the specified screen does not exist. The screen number should be verified.

27   **Invalid window specification.**

You tried to create or alter the size of a window, but the screen number, window number, or size coordinates were illegal. See the Window Open command for valid window specifications.

28   **Invalid cast. Must use format '(type)'.**

This error indicates that type casting was attempted outside of an expression, or without being enclosed in parentheses. Types can only be used in expressions as casts, and must be enclosed in parentheses.

29   **Unknown special key.**

A key was pressed that the debugger does not recognize.

30   **Start line invalid.**

The starting line for the Program Find_Source command may be omitted, or may be any valid line optionally within a module.

31   **Invalid exception vector.**

You tried to specify an exception vector that is invalid. In a Program Interrupt Add command, the optional exception vector must be in the range of 0 to 255.

32   **Invalid trace speed.**

An attempt was made to specify a step speed with the debugger Option General Step_Speed command that is not in the valid range. Tracing speed ranges from 0 to 100.

33   **Must be ON or OFF.**

An attempt was made to specify an invalid argument with an option. Options can be switched to ON or OFF.

34   **Cannot divide by zero.**

An attempt was made to divide by zero within an expression of Expression
Display_Value or Expression C_Expression.

35   **This feature not available in this version.**

51   **This command cannot be used in this mode.**

A command that is not supported in the current mode was issued. The
Program Display_Source command is only supported in high-level mode, and
the Memory Display Mnemonic command is only supported in assembly-level
mode.

52   **Switches cannot be used together.**

Two switches of the same group were given. Only one switch per group may be
specified.

53   **Invalid switch given for this command.**

The specified qualifier is not associated with the specified command. Check
the command syntax and re-enter the command.

54   **Value too large.**

A value that is out of range was specified. Values must be in the valid range for
the command.

55   **Instruction expressions are invalid in this mode.**

An expression was used for a code address in high-level mode. Only a single
line number or function symbol may be used in high-level mode.

56   **Module not found.**

The specified module name does not exist. Specify a valid module name.

57   **Line number not found.**

The line number specified does not exist in the current module. If the line number exists in a different module, the module name must be specified.

58   **Symbol not found.**

The symbol name was entered incorrectly, or the symbol does not exist. The symbol name may have been mistyped.

59   **Macro not found.**

The specified macro has not been defined, or an invalid macro name was entered. Check the macro name, or define the macro and re-enter the macro name.

60   **File not found.**

The specified file does not exist in the current directory, or in the search directories. Check the current directory for the filename that was specified. A typing error may have occurred.

61   **Structure member not found.**

The specified structure member does not exist in the specified structure. Check the structure definition for the member that was specified. A typing error may have occurred.

62   **Numeric addresses not allowed in this mode.**

An attempt was made to specify an invalid address value.

63   **Line numbers from different modules.**

Line numbers from different modules were specified. Only one module specification may be given.

65   **Port input does not come from file or string.**

You cannot rewind an input port that does not get its input from a file or a string.

66 **Port output does not go to a file.**

Only port output directed to a file may be rewound with the Memory Port Rewind Output command.

67 **This breakpoint is already set.**

An attempt was made to set a breakpoint that already exists. The current breakpoint must be deleted before it can be reset.

68 **Port value not found.**

A port was specified that has not been created with the Memory Inport Assign or Memory Outport Assign command.

69 **Address in range already specified as Read_Only or Guarded.**

An address that was previously specified with a Memory Map Read_Only or Memory Map Guarded command was specified. Memory Map Read_Only and Memory Map Guarded commands can only act on Write_Read areas.

70 **Arguments do not match any Read_Only or Guarded area.**

The arguments specified with a Memory Map Write_Read command do not match the corresponding Memory Map Read_Only or Memory Map Guarded command. The arguments must match exactly. Entering a Memory Map Show command gives a map of Read_Only and Guarded areas.

71 **Address range contains unacceptable breakpoints.**

An illegal breakpoint was specified.

72 **Bad size specification for window.**

An illegal size specification was given for a window. See the Window New command for the correct size specifications.

73 **Cannot repeat a cycle count of zero.**

A Program Interrupt Add command qualifier cannot request that an interrupt occur every zero cycles; this would cause an infinite loop.

74   **Invalid level number. Must be 1 to 7.**

The Program Interrupt Add command, as well as the 68030 family of microprocessors, permit 7 levels of interrupts.

75   **Attempt to delete nonexistent breakpoint(s).**

You tried to clear a breakpoint that was not previously set. Check that the breakpoint was set, or not already cleared.

76   **Symbol not available from this scope unreferenced.**

You must reference the symbol with a qualified function or module name.

77   **Symbol with this name already exists.**

You tried to define a symbol that was previously defined. Another name should be used.

78   **Cannot create this symbol.**

An error occurred when trying to create the symbol. Check that it is valid as a symbol name.

79   **Symbol is not a module.**

An attempt was made to enter a symbol when a module was expected.

80   **Invalid stack level.**

This error indicates that a stack level was specified that is greater than the current stack nesting.

81   **Not a source function.**

An attempt was made to enter an illegal function with the Program Context Set command. The Program Context Set command requires either a module name or a source procedure name.

82   **Cannot delete this symbol.**

Registers and predefined symbols cannot be deleted.

83   **Invalid processor name.**

This error indicates that you specified a processor other than one supported by your debugger. See your user's guide for a list of supported microprocessors.

84   **Breakpoint limit exceeded.**

The number of breakpoints allowed has been exceeded. This breakpoint has not been set.

91   **Internal command/expression processor error.**

An internal memory error has occurred.

92   **Not enough memory for expression.**

The expression specified requires more memory than there is available. Try clearing breakpoints or deleting macros to obtain more memory.

93   **Invalid memory/register address.**

An attempt was made to read or write to inaccessible target memory. Target memory that is protected cannot be read from or written to.

94   **Source is not available for this module.**

An attempt was made to access source code in an assembly language module. Use the Debugger Level command or the **F3** function key to switch to assembly-level mode to display this module.

95   **Cannot build source table.**

There is not sufficient memory available to build the source table for source display.

96   **Cannot read absolute file.**

An attempt was made to load a file that is not an absolute object module. The code may need to be compiled, assembled, or linked.

| 97 | **Cannot build disassembly table.** |

There is not sufficient memory available to build the disassembly table for up-arrow and page-up support in the disassembler.

| 98 | **Cannot split monitor lines.** |

An attempt was made to monitor different elements on the same line. Only one element per line may be monitored.

| 99 | **No empty lines available.** |

An attempt was made to specify a line number with the Expression Monitor Value command, but the entire window is already filled. The number of lines in the data window is limited to 17. Use the Expression Monitor Delete command to delete some of the lines.

| 100 | **No available windows.** |

This error indicates that the numbers allocated for user-defined windows have all been used. Some windows must be deleted before creating another user-defined window.

| 101 | **Cannot open file.** |

An attempt was made to open a file that does not exist.

| 102 | **Local variable not alive.** |

A local variable was specified, but the function containing the variable is not active (current or nested).

| 103 | **No source level information available.** |

The source file for the specified source module cannot be found.

| 104 | **A log or journal file is already open.** |

An attempt was made to open a new log file when one is already in use. Close the existing log file with the File Log Off command before opening a new log file.

106   **Not enough memory.**

This error indicates that not enough memory was available for the specified command.

107   **Terminated when processing absolute file.**

This error indicates that an invalid control value was encountered in loading the ".x" file.

108   **At start of function, no local variables yet.**

This error indicates that arguments and local variables are not available to the debugger at this time. They are available when the prolog to the function has been executed.

109   **Local already defined.**

This error indicates that a local variable has been defined twice in a macro definition. One definition of the variable must be deleted.

110   **This argument not defined.**

This error indicates that an argument was declared that was not defined on the command line with the Debugger Macro Add command.

111   **This macro is in use already.**

Macros cannot be called recursively.

112   **This is not allowed outside of a macro.**

Keywords are allowed in macros only.

113   **Cannot begin execution from a macro.**

Program Run, Program Step With_Macro, Program Step, and Program Step Over are not allowed from within macros. The PC may be altered with the Memory Register @PC= command.

114 **This command not allowed from a macro.**

Some commands are not allowed from a macro, such as Debugger Host_Shell and Debugger Macro Add.

115 **Invalid float expression, results in NAN.**

A floating point expression resulted in a non-number.

116 **Cannot convert float value.**

Float value is too large to convert to an integer.

117 **Help file unavailable.**

This error indicates that the help file, "db68040.hlp", was not found.

118 **Unsupported float type.**

A floating point type other than 32 or 64 bit has been defined.

119 **Cannot get address of register or constant.**

An attempt was made to find the address of a register or constant. One example is: Expression Display_Value &&@a1.

121 **Cannot open command file for reading.**

This error indicates that the command file specified cannot be found.

122 **Include file name too long.**

This error indicates that the filename specified (including its pathname) is too long to be handled by the debugger's internal buffers. Limit the number of characters in the filename specification, or move the file to the default directory.

123 **Could not read source line.**

This error indicates that there was an error reading the C source file.

124   **Cannot create file for logging.**

This error indicates that there was an error when trying to create the specified log file or that the current directory does not have write permission.

125   **Write error occurred while writing to a file.**

This error indicates that the disk is probably full.

126   **Cannot open startup file < startupfile> .**

This error indicates that the debugger could not open the specified setup file. The filename may have been misspelled, or the filename does not exist.

127   **Invalid number of arguments for macro.**

This error indicates that an incorrect number of arguments was specified in the call or too many parameters were used in the macro definition.

128   **Cannot show built-in macros.**

This error indicates that predefined macros cannot be shown with the Debugger Macro Display command. They have no text.

129   **Runtime error in macro.**

This error indicates that an error occurred when executing a macro.

130   **Command not implemented in simulator version.**

This error indicates that the command entered will not work in this version of the debugger.

131   **'option chip''not implemented in this version.**

This error indicates that "option chip" will not work in this version of the debugger.

132   **Breakpoint adjusted.**

This error indicates that the breakpoint has been moved to an address at the start of an instruction. See the Debugger Option General Align_Bp command syntax description in the "Debugger Commands" chapter.

133   **Error return from child process.**

This error indicates that an error was returned when interacting with the host system through the Debugger Host_Shell command.

134   **This command cannot be executed from batch mode.**

This error indicates that the command entered will not work in batch mode.

135   **No search string available.**

The command Program Find_Source Next was entered without previously entering the Program Find_Source Occurrence command.

136   **Cannot open file for logging; file in use for commands.**

The file specified for logging is currently open and being used to read commands from. Choose another name for the log file.

137   **Cannot open file for logging; file in use for logging.**

The file specified to read commands from is open and being used as a log file. Turn off logging with the File Log OFF command or choose another name for the command file.

141   **Miscellaneous error.**

All available error information is displayed on the screen. Any one of a number of error messages may be displayed on your screen.

One possible error message is:

No valid BBA spec file for <*processor*> **processor**

You must have the HP Branch Validator product for your processor installed on your system in order to use the Memory Unload_BBA command.

142   **Miscellaneous warning.**

All available warning information is displayed on the screen. Any one of a number of warning messages may be displayed on your screen.

143   **Miscellaneous note.**

All available information is displayed on the screen. Any one of a number of notice messages may be displayed on your screen.

145   **Too many interrupts pending.**

Too many Program Interrupt commands have been given without a sufficient number of interrupts being processed. The current limit on pending interrupts is 16.

146   **Voids have no value.**

This error message is returned when certain commands are attempted on voids.

147   **Invalid suboption.**

This suboption does not work with this command. Refer to the "Debugger Commands" chapter of this manual for valid suboptions for various commands.

148   **Invalid option.**

This option does not work with this command. Refer to the "Debugger Commands" chapter of this manual for valid options for various commands.

149   **No temporary breakpoints for the macro.**

The command Program Run From < addr> ;< macro> will return this error because a temporary breakpoint has not been specified.

150   **Invalid type for this argument, expecting a target address.**

The command was expecting an address. Re-enter the command with a target memory address.

151   **Invalid type for this argument, expecting a number.**

The command was expecting a number. Re-enter the command with a number.

152   **Cannot delete: more than one symbol with this name.**

Multiple symbols with the same name exist. More fully qualify the symbol to make it unique and then retry the command.

153	**Cannot save into this address (not 'lvalue').**

The expression used is not an address. This command can only save at an address which is an 'lvalue'. Check the address and then retry the command.

154	**Invalid type for macro argument.**

This is an invalid type for the macro argument. Refer to the chapter on macros for more information on valid types for macro arguments.

155	**Stopped by user.**

The execution of this command was halted by the user.

156	**Not a logical expression (= = , != , < , > , < = , > = , !).**

The expression entered is not a logical expression. Refer to the "Expressions and Symbols in Debugger Commands" chapter for more information on logical expressions and then re-enter the command.

157	**Cannot create log file.**

Unable to open the specified file as a journal file.

159	**Interrupted during I/O.**

Keyboard I/O was in cooked mode and a read from the keyboard was interrupted.

161	**Bad command for current context (No root, start, etc.)**

162	**Ambiguous member name, must qualify with more local class.**

The referenced C+ + member function may be one of several function which have the same name. Use a class name to be more specific.

163	**Cannot currently access via virtual base class.**

15
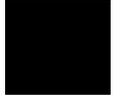
Debugger Versions

Information about how this version of the debugger differs from previous versions.

Version C.05.10

Larger Symbol Table

The debugger can load up to 16 million symbols. The previous limit was 64K symbols.

Each symbol uses 128 bytes of memory. If so many symbols are loaded that your host operating system runs out of swap space, the practical limit may be less than 16 million symbols.

More Global Symbols

The maximum number of global sysmbols that can be read from an HP-MRI IEEE-695 file has been increased from 8000 to 64K symbols.

Radix Option Side Effects

Input and output values are interpreted as hexadecimal only for assembly-level references.

To cast a high-level expression as hexadecimal, use a leading "0x" or a trailing "h".

When the radix option is set to **hex**, the following inputs will *not* be interpreted as hexadecimal:

- line numbers starting with "# "

- variables in high-level expressions, including **C_Expression** and macro expressions.

- debugger variables including:

    – breakpoint numbers

    – viewport numbers

    – data viewport line numbers

New Demand Loading Option

The **-doff** command line option turns off demand loading, overriding the option in the startup file.

Graphical User Interface

The debugger now has a graphical user interface. Some of the many features of the graphical interface include:

- pull-down and pop-up menus
- user-definable action keys
- a mouse-driven command line
- improved on-line help
- powerful macro editing

The debugger's old standard interface may still be used.

New Product Number

The old product number of this debugger was HP 64368 for HP 9000 Series 300 computers. The new number is HP B1467.

New Reserved Symbols

@ENTRY is the address of the first executable statement in a function. For example, func1\@ENTRY is the first executable statement of *func1*. If you set a breakpoint at func1\@ENTRY rather than at func1, the local variables in *func1* will be active.

@ROOT is the name of the root of the symbol tree represented by the program counter.

@FILE is the name of the file containing the current program counter (if any).

Environment Variable Expansion

Operating system environment variables will now be expanded when they appear in a debugger command.

For example, "Debugger Directory Change_working $HOME/test" will now work as expected.

Target Program Function Calls

You may now reference target program functions in C expressions.

Target and debugger variables may be passed by value, and target variables may be passed by reference.

C++ Support

The debugger now supports C++ name mangling/de-mangling and object/instance breakpoints for the Microtec Research Inc. C++ compiler.

Simulated I/O Changes

The debugger's simulated I/O features are now compatible with the emulation interface's simulated I/O.

Simulated I/O in the debugger/emulator now requires the setting up of simulated I/O polling and addresses in the emulator configuration.

The I/O Report no longer reports on processes used.

The keyboard EOF function is no longer supported since it is not supported by the emulation interface's I/O.

Part 5

Installation Guide

Part 5

16

Installation

How to install the debugger software on your computer.

Installation at a Glance

The debugger/simulator is a tool for debugging C programs for 68030/40 series microprocessors in a simulated execution environment.

Follow these steps to install the debugger:

1 Install the software on your computer.

2 Set up your software environment to run the debugger.

3 Verify the software installation.

Supplied interfaces

When an X Window System that supports OSF/Motif interfaces is running on the host computer, the debugger has a *graphical interface* that provides pull-down and pop-up menus, point and click setting of breakpoints, cut and paste, on-line help, customizable action keys and pop-up recall buffers, etc.

The debugger also has a *standard interface* for several types of terminals, terminal emulators, and bitmapped displays. When using the standard interface, commands are entered from the keyboard.

The installation procedure described in this chapter shows you how to install both debugger interfaces and verify the installation.

Supplied filesets

As you install the software, you will see a list of the filesets on the tape. The filesets are identified by their HP product number.

The tape may contain several products. Usually, you will want to install all of the products on the tape.

However, to save disk space, or for other reasons, you can choose to install selected filesets. For example, if you plan on using the debugger's standard interface instead of the graphical interface, you can save about 3.5 megabytes of disk space by not installing the XUI suffixed filesets. (Also, if you choose not to install the graphical interface, you will not have to use the "-t" command line option to start the standard interface.)

C Compiler Installation

Some older versions of HP C Cross Compilers will overwrite the file $HP64000/bin/db68040, making the graphical interface unavailable. If you encounter this problem, install the C compiler *before* you install the debugger software.

To install software on an HP 9000 system

Required Hardware and Software

To install and use the debugger's graphical interface, you need:

- HP 9000 Series 300/400 computer running HP-UX version 7.03 or later, or HP 9000 Series 700 computer running HP-UX version 8.01 or later.

  To check the HP-UX operating system version, enter the **uname -a** command at the HP-UX prompt. If the version number of the HP-UX operating system is less than 7.03, you must update the operating system to version 7.03 or higher before you can use the debugger. (Refer to the "Updating HP-UX" chapter of the *HP-UX System Administration Tasks* manual for detailed information concerning updating your system.

  Motif/OSF. For HP 9000 Series 700 workstations, you must also have the Motif 1.1 dynamic link libraries installed. They are installed by default, so you do not have to install them specifically for this product, but you should consult your HP-UX documentation for confirmation and more information.

  Hardware and Memory. The debugger's graphical interface requires workstations to have a minimum of 16 megabytes of memory. Series 300 workstations should have a minimum performance equivalent to that of a HP 9000/350. A color display is also highly recommended.

- Approximately 3 Mbytes of disk space.

- HP B1467 debugger/simulator software.

Step 1. Install the software

During the install process, you have some choices about how much you load from the product media. As a general rule, you should load everything from the media. However, to save disk space, or for other reasons, you can choose to install selected filesets.

There are several reasons why you might want to install selected filesets:

- You were shipped the HP B1471 64000-UX Operating Environment instead of the HP B1471 64700 Operating Environment, you were shipped files that are not necessary to the operation of the debugger/simulator. Excluding these files will save you about 4.5 megabytes of disk space.

- You will not be using the debugger's graphical interface. In this case, you should exclude the filesets that contain the graphical interface because:

    - You will save about 3.5 megabytes of disk space.

    - If you are using X Windows, the graphical interface is the default interface. If you load the graphical interface, but do not use it, you will have to use a special command line option each time you start the standard interface.

The following sub-steps assume that you may want to exclude partitions or filesets. Perform the following sub-steps to load the software on your system:

1 Become the root user on the system you want to update.

2 Make sure the tape's write-protect screw points to SAFE.

3 Put the product media into the tape drive that will be the *source device* for the update process.

4 Confirm that the tape drive BUSY and PROTECT lights are on.

If the PROTECT light is not on, remove the tape and make sure the tape's write-protect screw points to SAFE. If the BUSY light is not on, check that the tape is installed correctly in the drive and that the drive is operating correctly.

5 When the BUSY light goes off and stays off, start the update program by entering

```
/etc/update
```

at the HP-UX prompt.

6 When the HP-UX update utility main screen appears, confirm that the source and destination devices are correct for your system. Refer to your HP-UX System Administration documentation if you need to modify these values.

7 Select the choice on the update menu that allows you to view the product partitions.

Except for the debugger/simulator partition and the 64700 Operating Environment partition, mark all other partitions with "y" to confirm that you want these partitions loaded. (Do not mark B1471 with "y" if you marked it with "n" in the last step.)

The debugger/simulator partition will be named something like "68030/40 debugger/simulator".

8 If you plan to install and use the debugger's graphical interface, do the following:

- Mark the debugger/simulator partition and the 64700 Operating Environment partition with "y" to confirm the installation of these partitions.

9 *If you do not want to install the debugger's graphical interface*, do the following:

- View the filesets for the 64700 Operating Environment.

- Mark the **XUI** fileset with "n" to exclude it from installation.

- Mark all other filesets in the partition with "y" to confirm installation.

- Return to the partition screen.

- View the filesets in the emulator-specific partition (named something like 68030/40 Series debugger/simulator).

- Mark the **XUI** fileset with "n" to exclude it from installation.

- Mark all other filesets in the partition with "y" to confirm installation.

- Return to the partition screen.

10 From the partition screen, choose the update utility softkey that starts the installation process.

To install the software on a Sun SPARCsystem™

Required Hardware and Software

To install and use the debugger/simulator's graphical interface, you need:

- Sun SPARCsystem computer running SunOS version 4.1 or 4.1.1 or greater. The tape uses the QIC-24 data format.

  To check the SunOS operating system version, enter the **uname -a** command at the UNIX prompt. If the version number of the SunOS operating system is less than 4.1, you must update the operating system to version 4.1 or higher before you can use the debugger. For instructions on updating your system, see the Sun *Installing SunOS* manual.

- System V software. To find out whether the System V environment is already installed on your system, check that the directory /usr/5bin exists. For instructions on installing System V, see the Sun *Installing SunOS* manual.

- System V IPC facilities (semaphores). To find out whether the IPC facilities are installed on your system, type **ipcs**. For instructions on installing the IPC facilities, see the Sun *System and Network Administration* manual.

- At least 16 megabytes of memory (for the graphical user interface).

- Color display (optional, but recommended for the graphical user interface).

- Approximately 3 Mbytes of disk space.

- HP B1467 debugger/simulator software.

Step 1: Install the software

For instructions on how to install software on your SPARCsystem, refer to the
HP 64000-UX for SPARCsystems—Software Installation Guide.

Normally you should install all of the filesets on the tape. If you do not wish to
install the graphical interface, install everything except the XUI suffixed
filesets. This will save about 3.5 megabytes of disk space.

Step 2: Map your function keys

If you are using the character-based Standard Interface, map your function
keys by following the steps below:

1 Copy the function key definitions by typing:

```
cp $HP64000/etc/ttyswrc ~/.ttyswrc
```

This creates key mappings in the .ttyswrc file in your $HOME directory.

2 Remove or comment out the following line from your .xinitrc file:

xmodmap -e 'keysym F1 = Help'

If any of the other keys F1-F8 are remapped using xmodmap, comment out
those lines also.

3 Add the following to your .profile or .login file:

```
stty erase ^H
setenv KEYMAP sun
```

The erase character needs to be set to backspace so that the Delete key can be
used for "delete character."

If you want to continue using the F1 key for HELP, you can use use F2-F9 for the Softkey Interface. All you have to do is set the KEYMAP variable. If you use OpenWindows, type:

```
setenv KEYMAP sun.2-9
```

If you use xterm windows (the xterm window program is located in the directory /usr/openwin/demo), type:

```
setenv KEYMAP xterm.2-9
```

Reminder: If you are using OpenWindows, add /usr/openwin/bin to the end of the $PATH definition, and add the following line to your .profile:

```
setenv OPENWINHOME /usr/openwin
```

To set up your software environment

Follow these steps to prepare your computer to run the debugger:

1   Start the X server.

2   Set the necessary environment variables.
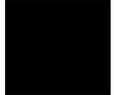
To start the X server

If you are not already running the X server and a window manager, do so now.
The X server is required to use the Graphical User Interface because it is an X
Windows application. A window manager is not required to execute the
interface, but, as a practical matter, you must use some sort of window
manager with the X server.

- If you are using an HP workstation, start the X server and the Motif window
  manager by entering:

```
x11start
```

- If you are using a Sun workstation, enter:

```
/usr/openwin/bin/openwin
```

Consult the X Window documentation supplied with the operating system
documentation if you do not know about using X Windows and the X server.
The chapter "Using X Resources" in this book also discusses X Windows and
the X server.

To start HP VUE

If you will be using the X server under HP VUE and have not started
HP VUE, do so now.

HP VUE differs slightly from other window managers in that it does not read
your .Xdefaults file to find resources you may want to customize. Instead, it
uses resources from the X resource database. In order to customize resources
for the Graphical User Interface under HP VUE therefore, you must either
merge a file of customized resources with the X resource database, or set an
environment variable that causes the X resource manager to read a file of
customized resources. For ease of use, choose the *.Xdefaults* file as your merge
file.

- To merge the file *.Xdefaults* with the X resource database, enter

```
xrdb -merge .Xdefaults
```

at the HP-UX prompt.

Customized resources will be merged with the X resource database and will be
available for retrieval by the Graphical User Interface.

- To enable the graphical interface to find the *.Xdefaults* file directly, enter the
following commands:

```
XENVIRONMENT=$HOME/.Xdefaults
```

```
export XENVIRONMENT
```

The graphical interface will be able to find and read the file in order to retrieve
customized resources.

To set environment variables

The following instructions show you how to set these variables at the UNIX prompt. Modify your ".profile", ".login", or ".vueprofile" file if you wish these environment variables to be set when you log in.

- Set the DISPLAY environment variable.

- Set the HP64000 environment variable.

- Set the PATH environment variable to include the **usr/hp64000/bin** directory.

  For the ksh login shell (most HP systems), set a variable by entering

  ```
  export <variable>=<value>
  ```

  For the csh login shell (most Sun systems), set a variable by entering

  ```
  setenv <variable> <value>
  ```

  The DISPLAY environment variable must be set before the debugger's graphical interface will start. Consult the X Window documentation supplied with the UNIX system documentation for an explanation of the DISPLAY environment variable.

  Set the HP64000 environment variable if you installed the software in a directory other than "/usr/hp64000" (that is, if you told the installation script to use a path other than "/").

  Modify the PATH environment variable to include the "$HP64000/bin" directory environment variable to specify search paths. Including **usr/hp64000/bin** in your PATH relieves you from prefixing HP 64700 executables with the directory path.

See Also   For information on setting the location of C source files, see page 93.

Examples   These examples use ksh syntax. If you are using csh as your login shell, then use the **setenv** style instead.

If your system is named "myhost," set the display variable by typing:

```
export DISPLAY=myhost:0.0
```

If you installed the HP 64000 software in the root directory, "/", enter:

```
export HP64000=/usr/hp64000
```

```
export PATH=$PATH:$HP64000/bin
```

If you installed the software in the directory /users/team, enter:

```
export HP64000=/users/team/usr/hp64000
```

To verify the software installation

A number of new filesets were installed on your system during the software installation process. This step assumes that you chose to load the filesets for the debugger/simulator's graphical interface.

You can use this step to further verify that the filesets necessary to successfully start the graphical interface have been loaded and that customize scripts have run correctly. Of course, the update process gives you mechanisms for verifying installation, but these checks can help to double-check the install process.

1 Verify the existence of the **HP64_Debug** file in the **/usr/lib/X11/app-defaults** subdirectory by entering

```
ls /usr/lib/X11/app-defaults/HP64_Debug
```

at the HP-UX prompt.

Finding this file verifies that you loaded the correct fileset and also verifies that the customize scripts executed because this file is created from other files during the customize process.

2 Examine **/usr/lib/X11/app-defaults/HP64_Debug** near the end of the file to confirm that there are resources specific to your microprocessor.

Near the end of the file, there will be resource strings that contain references to specific microprocessors. For example, if you installed the debugger graphical interface for the 68030/40 series microprocessors, resource name strings will have "debug*m68030" embedded in them.

To remove software

1 Find the fileset name of the product you wish to remove.

To see a list of the fileset names which you can remove, on an HP-UX system type:

```
ls /etc/filesets
```

Or, on a Sun system, type:

```
ls $HP64000/etc/filesets
```

Each file in this directory contains a list of files which were installed for that fileset.

2 Log in as root.

3 At the operating system prompt, type:

```
sysrm <product_number>
```

Sometimes you may wish to remove all of the files which were installed for a certain product. For example, maybe you have finished a software development project and you need to make more free space on your hard disk.

To make removing software easier, the installation script creates two files. The first file is a list of all of the files installed for a certain product. The list is stored by product number in the /etc/filesets directory. The second file is the sysrm script, which is stored in the $HP64000/usr/hp64000/bin directory. The sysrm script removes the files listed in /etc/filesets for the specified product.

Example

If you wish to remove all of the files which were installed for the B1460 product, type:

```
sysrm B1460
```

Configuring Terminals for Use with the Debugger

If you are using the debugger's graphical interface, you do not need to read this section.

This section shows you how to:

- Configure HP terminals or bit-mapped displays.

- Configure DEC VT100 terminals.

- Configure DEC VT220 terminals.

- Set the TERM environment variable.

- Set up control sequences.

- Resize X terminal emulation windows running the standard interface.

Supported Terminals

The following table lists the terminals and bit-mapped display devices supported by the debugger. The environment variable TERM is the variable which most UNIX applications (such as the debugger) use to customize the application for a particular terminal, display or window. In /bin/sh and /bin/ksh, the TERM variables setting may be queried by typing:

```
echo $TERM
```

The term variable is usually set by the login process (usually by /etc/profile). See the *HP-UX System Administration Task Manual* for more information on adding Peripheral Devices (terminals and modems).

The following table shows the supported configurations for:

- RS-232 terminals.

- bit-mapped displays.

- X-Window support.

Supported Terminals TERM Settings[1]			
Terminal/Driver Card	**RS-232 Terminal**	**Bit Mapped Display**	**X-Windows (using client hpterm)**
HP 2392	TERM= 2392		
HP 700/92	TERM= 70092		
HP 700/94	TERM= 70094		
VT100	TERM= vt100a		
VT220	TERM= vt220		
HP 700/22	TERM= vt220		
HP 64020A[4]	TERM= 2392		
HP 98544[AB][4]		TERM= 300h[2]	TERM= X-hpterm[3]
HP 98547A[4]		TERM= 300h[2]	TERM= X-hpterm[3]
HP 98548A[4]		TERM= 98548	TERM= X-hpterm[3]
HP 98550A[4]		TERM= 98550	TERM= X-hpterm[3]

Notes:

[1] TERMINFO= /usr/hp64000/lib/terminfo.

[2] On Bit Mapped Displays, you may need to set the function keys to *user* mode by holding down the **shift** key and then pressing the **user** key.

[3] TERM may be set to X-hpterm, hpterm, hp2622, or 2392 (see note following this table).

[4] Supported only on HP 9000 Series 300 computers.

RS-232 Terminals. RS-232 terminals interface to HP-UX by means of HP 98626, HP 98642, or HP 98629 RS-232 interface cards.

RS-232 terminals and bit mapped displays should be added to HP-UX according to the instructions in the *HP-UX System Administration Task Manual* for adding Peripheral Devices (terminals and modems).

Note

If TERM is set to 2392, the debugger attempts to draw solid lines using an alternate character set. For HP terminals, the debugger assumes this character set is the HP line drawing character set. The X11 command "hpterm" has the capability to shift in and out of an alternate character set. However, it defaults this character set to a font other than the HP line drawing font unless the line drawing font is explicitly specified as the alternate character set when the window is started. The alternate character set can be specified using an hpterm command line option or an X11 resource specification. See your HP-UX X11 documentation for more information.

Availability of fonts depend on the X11 server, not the X client program (hpterm). Only one HP line drawing font is available with HP-UX X11 servers. This font is "line.8x16". It must be used with the primary font "hp8.8x16" because the primary and alternate fonts must have the same dimensions. Refer to the "hpterm" manual page for more information.

Example: The following command starts an hpterm window with a primary font of hp8.8x16 and an alternate font of line.8x16.

```
hpterm -fn hp8.8x16 -fb line.8x16
```

Bit-Mapped Displays. Bit-mapped displays are listed by the boot up display as ITE (Internal Terminal emulator). They are high resolution display tops running without a window manager such as HP Windows or X Windows. Bit-mapped displays are supported only on HP 9000 computers.

X-Windows. X Windows refers to applications running on a bit-mapped display with X Windows managing the display top. Within X Windows, *hpterm* is a good general purpose HP terminal emulator for applications which are not specially customized for the X environment.

DEC Terminal ANSI Keypad Functions

The following figures show the ANSI keypad functions and the mapping of the top row of function softkeys.

PF1 Softkey F1	PF2 Softkey F2	PF3 Softkey F3	PF4 Softkey F4,
7 Softkey F5	8 Softkey F6	9 Softkey F7	– Softkey F8,
4 Roll Up	5 Next Page	6 Insert Char	,, Delete Char,
1 Roll Down	2 Prev Page	3 Unused	Enter CR,
0 Command Recall		Clear Line	^ ,

DEC Terminal Top Row Function Keys

F6	F7	F8	F9	F10		F11 (ESC)	F12 (BS)	F13 (LF)	F14
Softkey F1	Softkey F2	Softkey F3	Softkey F4	Softkey F4		Softkey F5	Softkey F6	Softkey F7	Softkey F8

To configure HP terminals or bit-mapped displays

1 Press the **System** key. (New function labels appear.)

2 Press the **Config Keys** key. (New function labels appear.)

3 Press the **Terminal Config** key.

4 Use the **Tab** key to select fields.

5 Choose the appropriate options for:

LocalEcho	OFF
CapsLock	OFF
SPOW(B)	NO
InhEolWrp(C)	NO
ReturnDef	^{c}r

6 Save the configuration by pressing the **SAVE CONFIG** key.

All other terminal configuration options are user-definable.

Note   With bit-mapped displays, you may need to set the function keys to *user* mode by holding down the **shift** key and then pressing the **user** key.

To configure the DEC VT100 terminal

You must configure two menus in order for the VT100 display to work
properly: SET-UP A and SET-UP B.

SET-UP A

1 Press **SET-UP** to enter the setup menu.

2 Press **80/132 COLUMNS** to select 80 columns per line. The columns per line
must be set to 80.

3 Press **SET-UP** to exit SET-UP A.
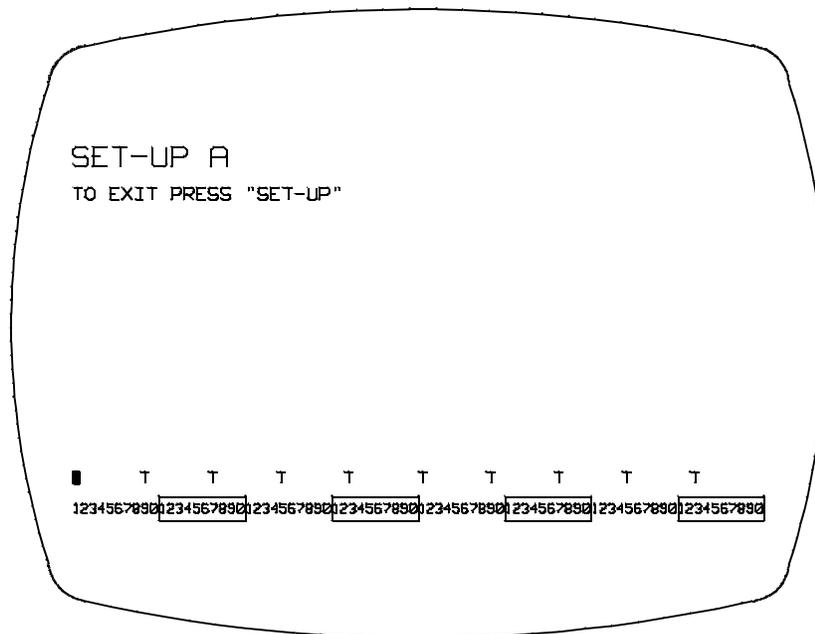
SET-UP B

4 Press **SET-UP** to enter the setup menu, then press **SETUP A/B**.

5 Using the arrow keys, position the cursor at the SET-UP feature to be changed.

6 Press **TOGGLE 1/0** to select the feature. Do this for the four features shown below:

7 Press **SET-UP** to exit SET-UP B.

```
1  XXXX      2  XX1X      3  X10X      4  XX0X


ANSI/VT52=ANSI ─┐
AUTO  WRAP=ON ──┘
LINE  FEED/NEW  LINE=OFF──┘
LOCAL  ECHO=OFF ──────┘
```
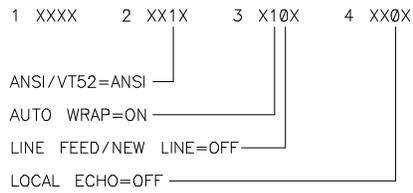
An "X" indicates a user-definable
terminal/computer dependency option.

```
SET-UP B
TO EXIT PRESS "SET-UP"


█
1  1101  2  0111  3  0101  4  1000      P=7S  T=9600  R=9600
```
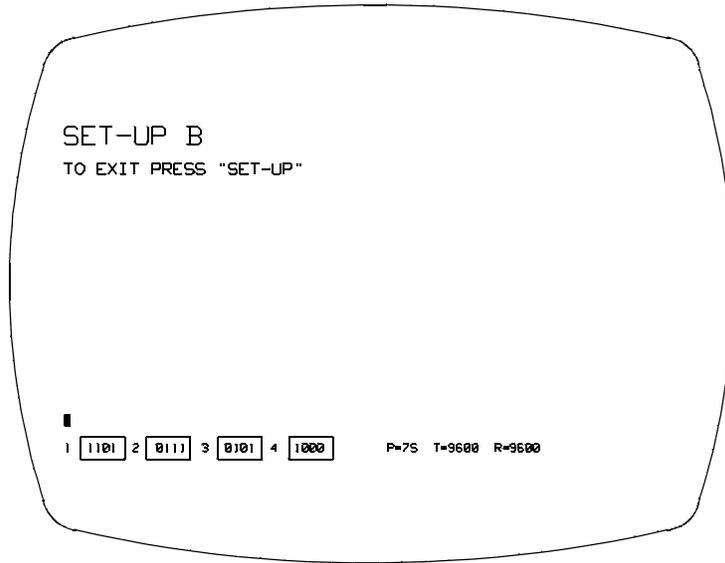
To configure the VT220 terminal

There are three menus that you must set up in order for the VT220 display to work properly:

- Display

- General

- Comm

From the power-on condition, press the **SET-UP** key to move into the initial SET-UP directory.

Note   To change fields within the menus, position the cursor at the desired field and press the **Enter** key.

Follow these steps to set up all three menus:

1 Press the **SET-UP** key to enter the setup directories.

2 With the cursor at the *Display* field, press the **Enter** key.

Use the arrow keys to move the cursor to the *Columns* field. This must be set for 80 columns. If 80 columns is not the selection displayed, press the **Enter** key.

Position the cursor at the *Auto Wrap* field. This field must be set to *Auto Wrap*.

Position the cursor at the *To Directory* field. Press **Enter**.

3 Position the cursor at the *General SET-UP* field. Press **Enter**.

Position the cursor at the *VT200 Mode* field. This field must be set to *VT200 Mode, 7-Bit Controls*.

Position the cursor at the *New Line* field. This field must be set to *No New Line*.

Position the cursor at the *To Directory* field. Press **Enter**.

4 Position the cursor at the *Comm SET-UP* field. Press **Enter**.

Position the cursor at the *Local Echo* field. This field must be set to *No Local Echo*.

5 Position the cursor at the *To Directory* field. Press **Enter**. At this point, if you want to permanently save the selections you have made, position the cursor to the *Save* field and press **Enter**. Otherwise, press **SET-UP** to exit the menus and return to the normal operating mode.
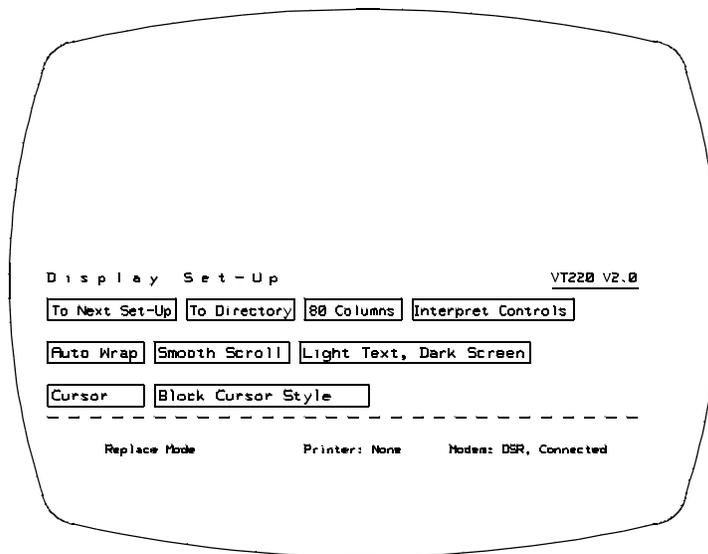
```
Display  Set-Up                      VT220 V2.0

To Next Set-Up  To Directory  80 Columns  Interpret Controls

Auto Wrap  Smooth Scroll  Light Text, Dark Screen

Cursor     Block Cursor Style
- - - - - - - - - - - - - - - - - - - - - - - -

      Replace Mode          Printer: None     Modem: DSR, Connected
```

G e n e r a l   S e t - U p   VT220 V2.0

| To Next Set-Up | To Directory | VT200 Mode, 7 Bit Controls | VT220 ID

| User Defined Keys Unlocked | User Features Unlocked | Multinational |

| Numeric Keypad | Normal Cursor Keys | No New Line |

Replace Mode   Printer: None   Modem: DSR, Connected

C o m m u n i c a t i o n s   S e t - U p   VT220 V2.0

| To Next Set-Up | To Directory | Transmit=4800 Receive=Transmit |

| XOFF at 64 | 7 Bits, Space Parity | | 1 Stop Bit | No Local Echo |

| EIA Port, Data Leads Only | Disconnect, 2 s Delay | Limited Transmit |

Replace Mode   Printer: None

To set the TERM environment variable

- If you are using the *ksh* shell, use the **export** command to set the TERM environment variable.

  Or:

- If you are using the *csh* shell, use the **setenv** command to set the TERM environment variable.

  Find the appropriate setting for the TERM environment variable in the previous table.

Examples

To set the TERM environment variable for a color bit-mapped display top using the HP 98547A driver card:

```
export TERM=300h
```

To set the TERM environment variable for a HP 2392 terminal:

```
export TERM=2392
```

To define and export the TERM environment variable for the VT220 terminal:

```
export TERM=vt220
export TERMINFO=/usr/hp64000/lib/terminfo
```

To define and export the TERM environment variable for the VT100 terminal:

```
export TERM=vt100a
export TERMINFO=/usr/hp64000/lib/terminfo
```

To set up control sequences

• Configure the stty settings in either /etc/profile or in $HOME/.profile.

The control characters for most HP applications should be set as follows. If the control characters get changed, you can reset them by typing the commands shown below after you have logged in to the system.

stty intr   \^c   (sets intr to CTRL c)

stty kill   \^u   (sets clear inputline to CTRL u)

stty quit   \^ (sets quit to CTRL \\)

stty eof   \^d   (sets end of file to CTRL d)

For the HP terminals and bit-mapped displays:

stty erase \^h   (sets erase character to backspace)

For the DEC VT100 and VT220 terminals:

stty erase   \^? (sets erase character to delete (DEL))
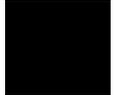
To resize a debugger window in an X-Window larger than 24 lines by 80 columns

The following procedure describes how to resize a debugger window in an X-Windows larger than 24 lines by 80 columns.

1 Check to see that the *LINES* and *COLUMNS* shell environment variables are correctly set for the window that you are working in so that the debugger knows what the screen limits are. By default, hpterm will set the screen limits correctly when a window is created. However, if you change its size, you should execute the command:

```
eval '/usr/bin/X11/resize'
```

from your shell. See the man page for resize(1) for details about how this works.

To resize a debugger window in a window larger than 24 lines by 80 columns

The following procedure describes how to resize a debugger window in a windows larger than 24 lines by 80 columns.

1 Check to see that the stty settings for *rows* and *columns* are correct for the window that you are working in so that the debugger knows what the screen limits are. Do this by entering the command:

```
stty size
```

The first number given is rows, the second number is columns. By default, the screen limits are set correctly when you change the size of the window. However, if the values for rows and columns are not correct for your window, you can set them by entering the *stty* command from your shell. For example, the command

```
stty rows 40 columns 100
```

will define a window to be 40 rows by 100 columns.

2 Start the debugger and load an executable file.

You need an executable in order to move windows in the source-level screen.

3 Move the status windows (command line) first. Start with the high-level status window (window number 5). For example, the command:

Window **N**ew 5 **H**igh_Level 46,0,49,79

will place the command line at the bottom of a window with 50 rows.

Then move the assembly-level status window (window number 15) to the bottom of the window. For example:

Window **N**ew 15 **A**ssembly 46,0,49,79
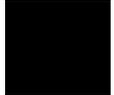
You can edit the previous command to save some typing. Remember the following information when moving the status windows:

1 The coordinates start from (0,0) in the upper left.

2 The maximum width of the command line is 80.

3 The top_row and bottom_row coordinates must be exactly three apart.

4 Move and resize the remainder of the windows by selecting each window and using the *Window Resize* command. Don't forget to resize the alternate view of each window. Use the *Window Toggle_View* command to select the alternate view.

5 You may want to set the *View* options of the Stdio, View, and Breakpoint windows to *On* (the windows swap by default). See the command description for the *debugger Option View* command for details.

6 Save your changes by executing the *File Startup* command. For example, the command:

File **S**tartup YourNameHere

will create a startup file named *YourNameHere.rc*. You can then use your custom window configuration by using the **-s YourNameHere** command.

The debugger will also automatically look in *./db68040.rc* for a startup file if you don't want to use the -s command line option.

Glossary

absolute file An executable module generated by compiling, assembling, and linking a program. Absolute files must have an extension of .x.

action key User-definable buttons in the graphical interface which allow quick access to often-used commands.

application default file A file containing default X resource specifications for an X Window System application.

background monitor An emulation monitor program that does not execute as part of the user program. See "emulation monitor".

BBA The Hewlett-Packard Branch Validator. It is a software tool you can use to analyze your testing, create more complete test suites, and measure your level of testing.

breakpoint A location in the program at which execution should stop.

cascade menu A secondary menu that appears when you select an item from a pull-down menu.

click To press and immediately release a mouse button. The term comes from the fact that pressing and releasing the buttons of most mice makes a clicking sound.

command file An ASCII file containing debugger commands.

command line An area at the bottom of the debugger window where commands may be entered using softkeys or pushbuttons. All **standard interface** commands are entered using the command line.

command token The smallest part into which a command may be broken—usually one word. Command tokens appear as pushbuttons on the command line.

concurrent usage model Describes an interface in which the user can perform most comands at the same time that code is being executed under emulation.

configuration file See "emulator configuration file".

cooked keyboard I/O mode The I/O mode in which keyboard input is processed. This lets you type and then edit the line to correct errors.

cut buffer A synonym for "entry buffer".

dialog box Sometimes called a secondary window, the dialog box is called by the user from the application's main window. A dialog box contains controls or settings, and sometimes prompts for text entry.

display area The part of the debugger window which shows windows containing information such as high-level code and breakpoints.

double-click To press the mouse button twice, quickly.

E/A The Emulator/Analyzer window.

emul700dmn The UNIX background process which coordinates the actions and message traffic of the major emulation interfaces.

emulation memory Memory provided by the emulator to be used in place of target system memory.

emulation monitor A program that is executed by the emulation processor that allows the emulation controller to access target system resources. For example, when you display target system memory locations, the monitor program executes the microprocessor instructions that read the target memory locations and send their contents to the emulation controller. See also "foreground monitor" and "background monitor".

emulator An instrument that performs just like the microprocessor it replaces, but at the same time, it gives you information about the operation of the processor. An emulator gives you control over target system execution and allows you to view or modify the contents of processor registers, target system memory, and I/O resources.

emulator configuration file A file that contains configuration settings and memory map definitions for the emulator.

entry area A section of the **command line** area where commands are built. When you use menus or softkeys, the actual command which the debugger will execute appears in the entry area.

entry buffer The part of the graphical interface which contains "input" for commands. The symbol for the entry buffer is "()".

foreground monitor An emulation monitor program that executes as part of the user program. See "emulation monitor".

graphical interface The debugger interface program that uses graphics-oriented software such as windows, menus, and icons to make interaction easy.

host shell A UNIX command interpreter.

iconify The act of turning a window into an icon.

journal file A file that contains commands entered during a debug session and any output generated by the debugger. Journal files contain everything that is written to the debugger's journal window.

log file A command file that is created by the debugger when you record commands.

macro A C-like function consisting of debugger commands and C statements and expressions. Macros are most often used to patch C source code, create conditional breakpoints, return values to expressions, or execute a set of commands.

menu bar The row of words at the top of the graphical interface window. Clicking on the menu bar will display a menu of debugger commands.

monitor See "emulation monitor".

patch A small, temporary change to executable code.

PITS cycle Programming In The Small cycle. The repeating process of editing, compiling, and executing code to eliminate bugs.

pointer The symbol on your computer's screen which shows where the mouse is pointing. The pointer may be a hand, an arrow, or another shape.

pop-up menu A menu that pops up when you press and hold the right mouse button. Pop-up menus are available whenever the mouse pointer changes to a "hand-cursor".

predefined macro See also "macro".

pull-down menu A menu that appears to "pull down" from the menu bar at the top of the interface window.

pushbutton A graphic control that simulates a real-life pushbutton. Use the pointer and mouse to push the button and immediately start an action.

raw keyboard I/O mode The I/O mode in which each keystroke produces a character that is sent to the target program that is reading from the keyboard.

recall buffer A text entry field which remembers its previous value.

resource See "X resource".

scheme file A file that contains X resource specifications for a particular group of resources, for example, for a particular type of display, computing environments, or language.

scroll bar A scroll bar is used to move a window so that you can see information beyond the window's edge.

sequential usage model Describes a user interface in which user code execution must be stopped before the interface can perform most commands.

shell See "host shell".

simulated I/O The debugger feature that lets user programs read input from, and write output to, the same keyboard and display (respectively) that are used

to control the debugger. Simulated I/O also lets user programs use the UNIX file system and run UNIX commands.

simulated program interrupt User program interrupts that are simulated by the debugger. Simulated interrupts can be one-time interrupts or periodic interrupts.

simulator A software tool that simulates a microprocessor system for the purpose of debugging user programs.

SPA The HP Software Performance Analyzer.

standard interface The traditional debugger interface designed for use with several types of terminals, terminal emulators, and bitmapped displays. When using the standard interface, commands are entered from the keyboard.

startup file A file that contains information regarding debugger options and screen configurations.

state file A file that contains the CPU state (including register values) and a memory image. This file is saved within a debugger session and can be loaded at a later time to return to a particular state of execution.

status line A line which displays debugger information such as the CPU type, the current module name, and the current debugger operation.

sticky slider A scrollbar slider which is designed for local navigation in a large file. Moving the slider moves the contents of the active window just a few pages at a time.

storage qualifier A bus cycle state description that causes only particular states to be stored in the analyzer trace.

trace A collection of states captured on the emulation bus (in terms of the emulation bus analyzer) or on the analyzer trace signals (in terms of the external analyzer) and stored in trace memory.

trace event A bus state consisting of a combination of address, data, and status values.

trigger The captured analyzer state about which other captured states are stored. The trigger state specifies when the trace measurement is taken.

window A window inside the debugger's display area. See also "X window".

working directory The current directory from which the debugger loads and saves files.

X resource A piece of data that controls an element of appearance or behavior in an X application.

X server A program that controls all access to input devices (typically a mouse and a keyboard) and all output devices (typically a display screen). It is an interface between application programs you run on your system and the system input and output devices.

X window A window on your computer's display. The debugger's graphical interface runs inside an X window. See also "window".

Index

() entry buffer, **557**
/dev/simio/display reserved symbol, **180**
/dev/simio/keyboard reserved symbol, **180**
68040 FPU simulation, **88**
@@ as access status pseudoregister, **159**
@cycles cycle count pseudoregister, **103, 159**
@exc exception handling keyword, **159**
@pi previous instruction pseudoregister, **159**
@SP stack pointer, **159**
@wait_state wait state pseudoregister, **104**

A absolute file, **555**
absolute files, **94–95**
access status pseudo-register @as, **461**
access status pseudoregister @as, **159**
action keys, **7, 555**
  custom, **259**
  operation, **66**
  with command files, **259**
  with entry buffer, **65–66**
activating windows, **14**
active window
  changing, **147**
  description of, **147**
  displaying the alternate view of, **148**
  viewing information in, **149–150**
add configured terminals to HP-UX, **540**
add symbol, **136**
address operator, **29, 45**
address registers @A0-A7, **461**
addresses, **441–442**
  assembly level code, **441**
  code, **441**
  data, **441**
  displaying variable, **29, 45**
  ranges, **441**

Index

Certification and Warranty

Certification

Hewlett-Packard Company certifies that this product met its published specifications at the time of shipment from the factory. Hewlett-Packard further certifies that its calibration measurements are traceable to the United States National Bureau of Standards, to the extent allowed by the Bureau's calibration facility, and to the calibration facilities of other International Standards Organization members.

Warranty

This Hewlett-Packard system product is warranted against defects in materials and workmanship for a period of 90 days from date of installation. During the warranty period, HP will, at its option, either repair or replace products which prove to be defective.

Warranty service of this product will be performed at Buyer's facility at no charge within HP service travel areas. Outside HP service travel areas, warranty service will be performed at Buyer's facility only upon HP's prior agreement and Buyer shall pay HP's round trip travel expenses. In all other cases, products must be returned to a service facility designated by HP.

For products returned to HP for warranty service, Buyer shall prepay shipping charges to HP and HP shall pay shipping charges to return the product to Buyer. However, Buyer shall pay all shipping charges, duties, and taxes for products returned to HP from another country. HP warrants that its software and firmware designated by HP for use with an instrument will execute its programming instructions when properly installed on that instrument. HP does not warrant that the operation of the instrument, or software, or firmware will be uninterrupted or error free.

Limitation of Warranty

The foregoing warranty shall not apply to defects resulting from improper or inadequate maintenance by Buyer, Buyer-supplied software or interfacing, unauthorized modification or misuse, operation outside of the environment specifications for the product, or improper site preparation or maintenance.

No other warranty is expressed or implied. HP specifically disclaims the implied warranties of merchantability and fitness for a particular purpose.

Exclusive Remedies

The remedies provided herein are buyer's sole and exclusive remedies. HP shall not be liable for any direct, indirect, special, incidental, or consequential damages, whether based on contract, tort, or any other legal theory.

Product maintenance agreements and other customer assistance agreements are available for Hewlett-Packard products.

For any assistance, contact your nearest Hewlett-Packard Sales and Service Office.