GNX Debugging Tools

DBUG

Version 4

Series 32000®

GNX — Version 4.4
Symbolic Debugger (DBUG)
Reference Manual

Customer Order Number 424511103-004

June 1992

REVISION RECORD

VERSION	RELEASE DATE	SUMMARY OF CHANGES
4.0	May 1990	First Release. Support for new CPUs and monitors. Provides partial symbolics mode for very large executable files. Addition of support for fast communications via Ethernet.
4.1	Sep 1990	Support for new HP Emulators.
4.2	Feb 1991	Miscellaneous feature updates.
4.3	August 1991	Miscellaneous feature updates.
4.4	June 1992	Miscellaneous feature updates. MS-DOS support added.

A complete list of National Semiconductor's international offices may be found on the inside back cover of this manual.

The information contained in this manual is for reference only and is subject to change without notice.

PREFACE

This is a reference manual for DBUG, National Semiconductor Corporation's symbolic debugger. DBUG can be used for symbolic debugging of high-level language programs generated by the GNX™ optimizing compilers as well as for assembly language programs generated by the GNX assembler.

DBUG supports two different user interfaces, one for graphic (X-windows) and one for alphanumeric (ASCII) terminal environments. Mouse, function keys, and menus facilitate command entry.

This reference manual provides guidelines for using the debugger.

CONTENTS

FIGURES

TABLES

INDEX

Chapter 1

INTRODUCTION AND OVERVIEW

1.1 Introduction

This manual describes the function, operation and use of the GNX Symbolic Debugger, DBUG. DBUG is an interactive debugger that can debug programs developed with National Semiconductor's GNX Software Development Package.

1.2 Features

DBUG may be used for symbolic debugging of programs compiled with the GNX assembler and the C, Pascal and FORTRAN77 compilers.

DBUG provides the following features:

- Support for two modes of operation: *native* and *remote*. In *native mode*, both DBUG and the program being debugged run on the host system. Native mode is available only on Series 32000-based computers (SYS32 development systems) running under the UNIX Operation System. In remote mode, DBUG runs on the host system while the program being debugged runs on the Series 32000 based target system (usually one of the NS development boards).

- Support for the HP64700 family of In-System Emulators (ISE) for the Series 32000 CPUs. DBUG also supports the SPLICE Development Tool for the NS32CG16 CPU.

- Fast communication between DBUG and development boards via Ethernet.

- Fast communication, via Ethernet, between DBUG and the HP64700 family of in-system emulators.

- Extensive breakpoint, trace and print capabilities. Breakpoints and traces may be conditional. General expressions may be specified in high-level language syntax.

- Command files and Log files. DBUG can create and execute a file of debugger commands. A log of these commands and the responses generated during a debug session may be saved in a file.

- Symbolic disassembly.

- Command aliasing, to enable command invocation using abbreviations.

In addition, the UNIX and VMS versions of DBUG provide:

- Advanced user interface for both (graphic) terminal (X-windows) environments and alphanumeric terminal environments.

- Function keys - any DBUG command can be attached to a function key.

- Command history mechanism, as part of the user interface.

1.3 Documentation Conventions

The following documentation conventions are used throughout the manual.

1.3.1 General Conventions

Character names within brackets < > indicate non-printing characters. For example, <ctrl/b> indicates that the user should press the keys marked *ctrl* and *b* simultaneously.

Constant width letters are used in examples.

1.3.2 Conventions in Syntax Description

The following conventions are used in syntax descriptions.

Bold letters indicate reserved words. Type them as shown.

Italics indicate inputs supplied by you. The italicized word stands for the actual operand, which you enter.

Spaces or blanks must be entered as shown. Multiple blanks or horizontal tabs may be used in place of a single blank.

{ } Large braces enclose two or more items of which one, and only one, must be used. The items are separated from each other by the vertical bar sign (|).

[ ] Large brackets enclose optional item(s).

| Vertical bar sign separates items of which one, and only one, is used.

... Three consecutive periods indicate optional repetition of the preceding item(s). If a group of items is repetitive, the group is enclosed in large parentheses ( ).

( ) Large parentheses enclose items that must be grouped together for optional repetition. If three consecutive commas or periods follow an item, only that item may be repeated.

1.4 Manual Organization

The remainder of this manual is organized as follows:

Chapter 2: how to invoke DBUG.

Chapter 3: the user interfaces for both bitmapped (graphic) and alphanumeric terminals. The chapter describes the DBUG windows, menus, and function keys. This chapter is only for users of the UNIX and VMS versions of DBUG.

Chapter 4: general concepts for using DBUG commands.

Chapter 5: the DBUG command set.

Chapter 6: the ISE and SPLICE interface and specific commands.

Appendix A: a DBUG tutorial for users with bitmapped terminals. The tutorial introduces the DBUG interface and main commands. New users are STRONGLY ENCOURAGED to work the tutorial. This appendix is only for users of the UNIX and VMS versions of DBUG.

Appendix B: a DBUG tutorial for users with alphanumeric terminals. The tutorial introduces the DBUG interface and main commands. New users are STRONGLY ENCOURAGED to work the tutorial. This appendix is only for users of the UNIX and VMS versions of DBUG.

Appendix C: DBUG commands, listed in functional groupings.

Appendix D: DBUG glossary.

1.5 Reference Documents

- GNX Software Tools Document Set
- GENIX™ V.3 Reference Manuals
- HP64772 Emulator Terminal Interface: NS32532 and NS32GX32 Emulator Users Guide
- SPLICE III Hardware Reference Manual

Chapter 2

INVOKING DBUG

2.1 Invoking DBUG

Invoke DBUG with the following command:

SYNTAX

dbug [*options*] [*objfile* [*coredump*]]

DESCRIPTION

Objfile is the name of the executable file being debugged. This file must have been produced by the GNX tools. The *objfile's* default name is *a.out* in native mode and *a32.out* in remote mode.

DBUG can examine a core file that was created by a program that terminated abnormally. The *coredump* parameter specifies the name of the file containing the program's core dump. This feature is only available in *native mode*.

Upon invocation, DBUG will execute commands in the files *.dbuginit* (in UNIX) or *dbug.ini* (in VMS and MS-DOS), if these files exist in either the current directory or the user's home directory, unless you specify the **-c** or the **-noc** options (see invocation options, below).
The invocation line options are described in the table below.
After initialization, DBUG prompts and waits for commands.

UNIX/MS-DOS	VMS	DESCRIPTION
-I *dir*	**/i=***dir*	Add *dir* to list of directories to search for a source file. (Normally DBUG looks for source files in current directory and in directory that contains *objfile*.) The directory search path may also be set with the **use** command.
-c *file*	**/c=***file*	Execute DBUG commands in *file* before accepting commands from the user. Without this option, DBUG attempts to execute commands in *.dbuginit* file.
-noc	**/NOC**	Do not execute any command file initially (including *.dbuginit*).
-l *ttyname*	**/l=***linkname*	Selects a communications link between host and target in *remote mode*. If specified, DBUG establishes this connection on invocation.
-cpu *cpuname*	**/cpu=***cpuname*	Specifies the CPU name on the target board. Possible names: *gx32, cg16, cg160, fx16, gx320, 532, 332, 032, 016.*
-mmu *mmuname*	**/mmu=***mmuname*	Specifies Memory Management Unit (MMU) name on target board. Possible names: *082, 382, onchip* or *nommu.*
-fpu *fpuname*	**/fpu=***fpuname*	Specifies Floating Point Unit (FPU) name on target board. Possible names: *081, 381, 181, 580* or *nofpu.*
-mon *name*	**/mon=***name*	Specifies monitor name on target board. Possible names: *16, 32, 332, 332b, 532, cg16, gx32, ise532, cg160, fx16, gx320, gx32e, fx16fax, cg160lx, isefx16, isecg16, isecg160, isegx32, isegx320.*
-LC	**/LC**	If specified, DBUG converts each command to lowercase before analyzing it. (This is useful when debugging Pascal programs, where variable names are converted to lowercase.)
-wx *number* **-wy** *number*	not available	Specifies position of upper lefthand corner of DBUG frame in pixels. Default values for Hercules display: `-wy0` and `-wx0`. For Viking display: `-wx143` and `-wy107`. This option is valid only for graphic display environment.
-wh *number* **-ww** *number*	not available	Specifies height and width of DBUG frame. Minimum width and height are 200 pixels. Default is `-wh640`, `-ww853` for Viking display. In Hercules display, the complete screen is used as the DBUG window. This option is valid only for graphic display environment.
-n *nodename*	**/n=***nodename*	Specifies node name of target board on the local area network.
-p	**/p**	Specifies that DBUG should load partial symbolic information. Useful for large executable files.
-b *number*	**/baud=***number*	**Baud** *number* sets the communication baud rate for *stand-aside mode*. Possible values are: 110, 150, 300, 600, 1200, 2400, 4800, 9600, 19200, 38400. The default baud rate is 9600. This parameter is ignored if the node option is selected.
-stx *number*	**/stx=***number*	**Stx** *number* changes the "stx" character used when communicating with the development board and protocol B is in use.

Chapter 3

THE DBUG INTERFACE

3.1 Introduction

This chapter describes the DBUG interface for graphic and alphanumeric terminals.

Both interfaces work on the same principles. With few exceptions, commands for manipulating windows and keys are identical for both interfaces. Thus, users familiar with one interface will have little difficulty learning the other.

The two interfaces are described in the following sections:

- Section 3.2 describes the graphic terminal interface.
- Section 3.3 describes the alphanumeric terminal interface.

3.2 The Graphic Terminal Interface

This section describes the DBUG user interface that runs on a graphic terminal with mouse capabilities. This interface is referred to as the graphic interface.

The DBUG graphic interface consists of the following elements:

- **Windows**. Multiple windows may be displayed within the DBUG frame, creating a structured display of the DBUG session information. Window display is controlled with the window manipulation commands.

- **Command Menus**. Command menus list DBUG commands. These commands are executed when selected with the mouse. You can use the menu definition commands to extend and customize menus to fit your specific needs.

- **Function Keys**. DBUG commands may also be invoked by function keys. All function keys are user definable using key definition commands.

For detailed descriptions of windows, menus, or function keys, see the following sections:

"The DBUG Windows" (section 3.2.2)

"Command Menus" (section 3.2.3)

"Function Keys" (section 3.2.4)

3.2.1 Invoking Commands and Specifying Parameters

You can invoke a DBUG command in one of three ways: by keyboard, by function key, or by using a command menu. Many DBUG commands require parameters. The two most common ways of specifying a parameter are described below:

1. Typing it after the command (example: `print var1`),
2. Marking program text as it appears in the CODE window, and selecting certain command menu options or function keys. This method requires use of the mouse.

3.2.2 Using The Mouse

The mouse cursor is represented on the screen by a large "X". A number of important functions are performed through use of the mouse. These include:

1. Executing commands through the command menu.
2. Marking text or selecting a window as a command parameter.

3.2.3 The dbug Windows

The DBUG windows and their functions are listed below:

CODE: displays the program being debugged; displays user-defined breakpoints and marks the next line to be executed.

DIALOG: provides a line for entering commands through the keyboard and a window that echoes all commands and DBUG responses.

HELP: lists DBUG command syntax.

TRACE: displays trace information produced by the HP64772 in-system emulator.

The DBUG windows are displayed within the DBUG frame.

The DBUG frame is the same, except for some minor differences, in all environments.

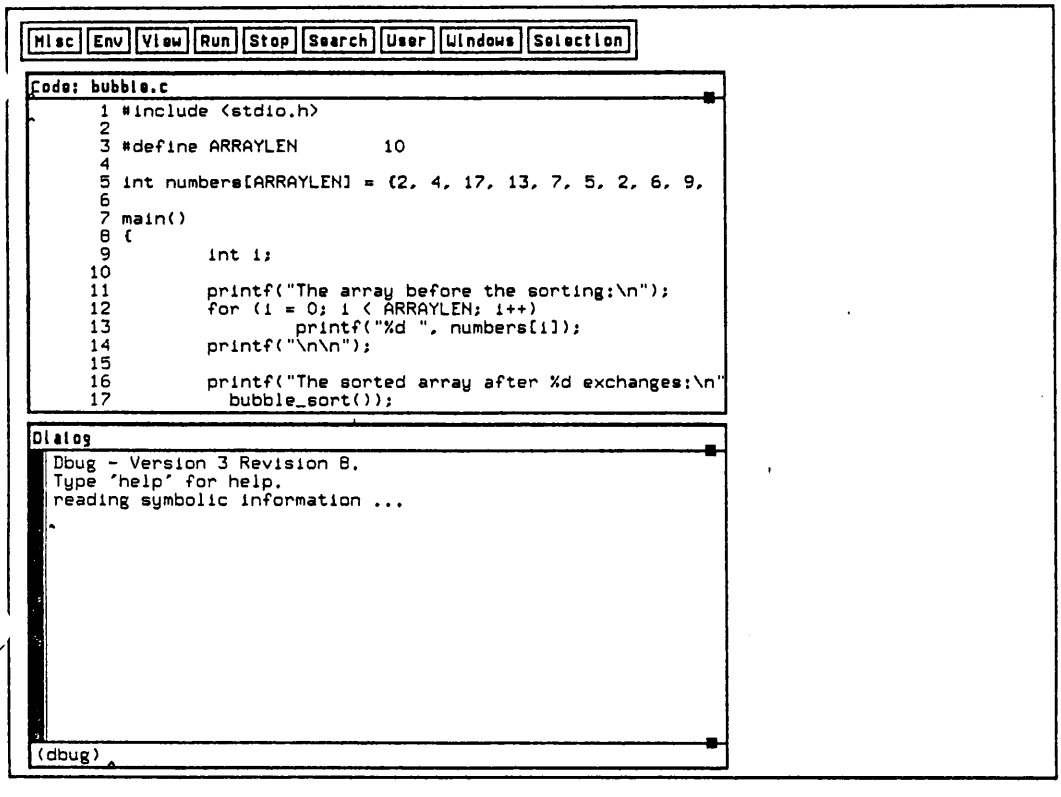


Figure 3-1. The Opening Dbug Frame for the Graphic Interface

3.2.3.1 The DBUG Frame

A DBUG session begins when the DBUG frame appears on the screen (Figure 3-1). All work takes place within the DBUG frame. The opening DBUG frame contains three rectangles and a large empty area on the right side. The uppermost rectangle stretches horizontally across the frame, and houses nine smaller windows with labels such as "Misc" and "Env". These are command menus, and they will be discussed later in this chapter (see "Command Menus," section 3.2.3).

The two remaining rectangles within the frame are windows for displaying information about the execution state of the debugged program. These two windows appear by default on the opening DBUG frame. They are called the CODE window and the DIALOG window.

You control the size and location of the DBUG frame and its windows. The windows may overlap or be located anywhere.

The only time during the debugging session when the mouse cursor should be located outside the DBUG frame is to enter input to the program being debugged. This is because the standard I/O of the debugged program is carried out in the *shell* window from which DBUG was invoked.

3.2.3.2 The Current Window

When working in a multiwindow environment, you must keep the mouse cursor within the DBUG frame. You may type in only one window at a time. The window where the mouse cursor is currently located is called the *current window*. The *current window* is the window that "listens" to the keyboard.

For example, you may enter commands along the command line only when the mouse cursor is within the DIALOG window.

3.2.3.3 The Selected Window

Some DBUG commands accept a window as a parameter. You can select a window that will serve as a default window for those commands. This window is called the *selected window*. To select a window, move the cursor into that window and press the **select** key (<pf3> by default), or execute the **wnext** command (see chapter 5, section 5.1.48). To unselect a window, put the mouse cursor in the DBUG frame and press the select key.

3.2.3.4 The CODE Window

The CODE window displays the section of your program code that is currently executing and provides a visual representation of the debugging session. DBUG marks lines that have breakpoints with an asterisk (*), and marks the next line to be executed with an arrow (=>).

The header along the upper border displays the window name "Code", and the file name of the program being debugged.

The program code is displayed in one of two modes:

1. **Source.** Source mode presents the program source code as originally written. Line numbers are displayed on the left side of the window.
2. **Assembly.** Assembly mode presents the disassembly of the target program. Each instruction's address is displayed on the left side of the window.

DBUG determines which mode to use. It selects source mode when the current module has symbolic information (compiled with **-g** option). Assembly mode is selected in all other cases.

To scroll through the CODE window, do one of the following:

- Use the cursor. CODE must be the current window. A small cursor (caret) marks the position in the file. Pressing the up arrow while the caret is adjacent to the upper border scrolls the window one line up. Pressing the down arrow while the caret is adjacent to the bottom border scrolls the window one line down. (This scrolling method is supported by all DBUG windows.)

- Scroll with the **wscroll** command. This command scrolls a number of lines up or down the CODE window.

- Scroll with the **wgo** or **search** commands. These commands scroll to a specific line number in the CODE window.

The CODE window is the only window in which text selection may take place. A full description of text selection appears later in this chapter.

3.2.3.5 The DIALOG Window

The DIALOG window is divided into four areas: the header band, the output area (the largest area in the window), the command line (along the window's bottom border), and the scroll bar (the thick black area along the lefthand border of the output area).

Use the command line (identified by the word "(dbug)" in the bottom left corner) to issue commands. There are two ways to enter a command along the command line. The first is by typing the command on the keyboard. The mouse cursor must be located on the command line. When DBUG starts running, the mouse cursor is automatically placed on this line. You can also bring the cursor there by pressing the **commline** key (<kp0> by default), or by moving the mouse.

The second way to enter a command through the command line is to use the **history buffer**. The history buffer records all user commands that were issued through the command line during the debugging session. You can access these commands by aligning the cursor on the command line and scrolling through the command line buffer with the up-down arrows until the appropriate command appears in the line. Press the return key, and the command is executed. Commands that have been invoked by function keys or menus are not kept in the history buffer, and thus cannot be accessed by scrolling.

You can move back and forth along the command line by pressing the left and right arrows (<-,->). The backspace key or your default erase character will delete characters.

The output area displays the session history, including user commands and DBUG prompts and echoes. Lines longer than the window width are wrapped around to the next line. Echoes of commands issued through a function key are prefixed by "-k-". Echoes of commands issued through a menu are prefixed by "-m-".

The DIALOG window, even if deleted, continues to log the debugging session in its output area. The window can be redisplayed with the **commline** command.

The scroll bar is used to scroll through the output area. To scroll, put the mouse cursor in the scroll area (the cursor will become a vertical line with arrows on either side) and press the appropriate mouse button. Within the scroll bar, mouse buttons function as follows:

left - scroll towards the end of the file. Pressing the left button causes the text next to the cursor (inside the scroll bar) to move to the top of the window.

middle - (for 3-button mouse) scroll to the absolute position in the text by positioning the cursor within the scroll bar.

right - scroll towards the beginning of the file. Pressing the right button causes the top line of the window to move to the mouse's position.

The scroll bar is also found in the DIALOG, HELP and TRACE windows.

Scrolling may also be performed with the up and down arrows on the keyboard.

3.2.3.6 The HELP Window

The HELP window provides information on the use of DBUG, window, and function key commands. The HELP window is manipulated like any other window. But unlike the other windows, it can be opened by invoking the **help** command. The HELP window may be closed with the **reset** function key. The default **reset** key is <ESC>.

3.2.3.7 The TRACE Window

The TRACE window is used only in an ISE configuration. It displays the results of the hardware trace output after the **traceh list** command has been issued.

The default location of the TRACE window is along the righthand border of the DBUG screen.

Issuing the **traceh list** command when the TRACE window is not present sends results to the DIALOG window.

3.2.3.8 The Window Manipulation Commands

The DBUG window manipulation commands are listed below. These commands allow the user to control the display of information on the screen and customize DBUG's default appearance.

For a complete description of each command and its syntax, see Chapter 5.

wdisplay - display a window.

wdelete - delete a window.

wgo - go to a line in the file.

wmove - move or resize a window.

wpop - pop a window.

wpush - push a window.

wreset - reset windows.

wscroll - scroll through a window.

wnext - select the next window.

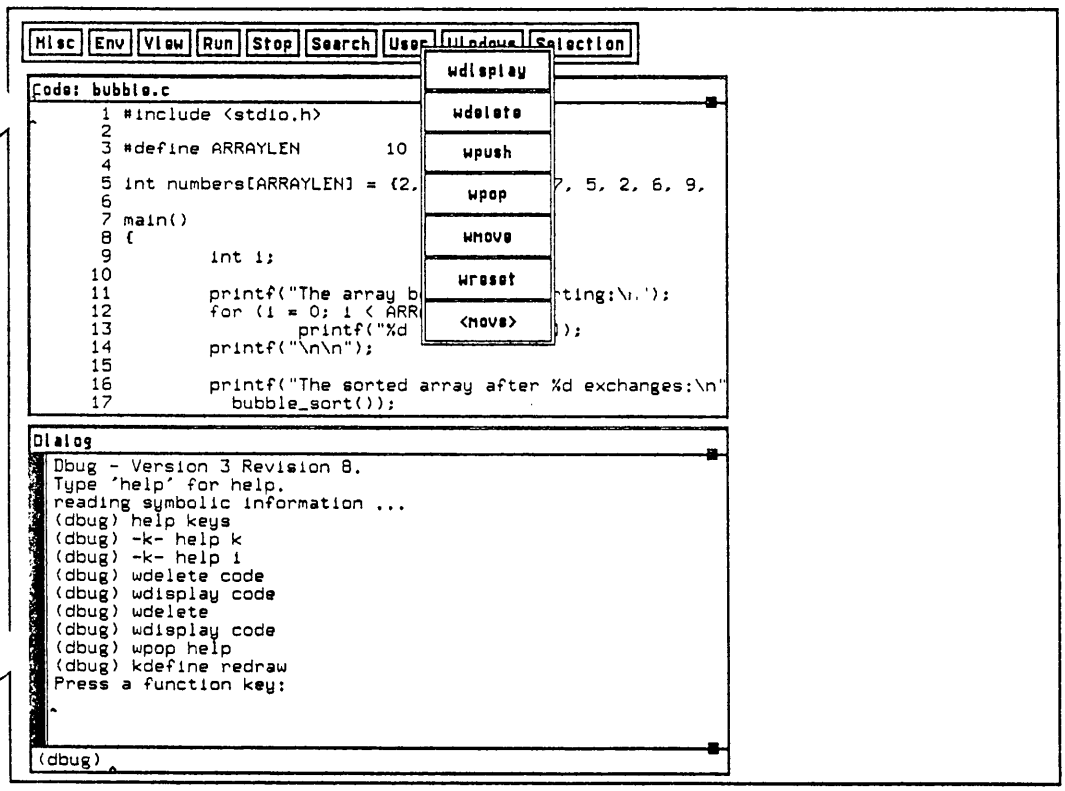


Figure 3-2. Temporary Menu

3.2.4 Command Menus

DBUG commands may be entered through a menu-driven interface. Command menu labels are displayed along the upper border of the DBUG frame. To display the command menus themselves, select the desired label with the mouse cursor.

Command menus may be extended by the user. This feature enables you to tailor the command menu interface to your needs, or to those of a specific application.

3.2.4.1 Command Menu Labels

The long, narrow rectangle along the upper border of the DBUG frame contains nine smaller rectangles with labels such as "Misc", "Env" and "View". These are command group labels. Command group labels cannot be moved, and their shape is automatically adjusted to the shape of the DBUG frame. Menus that relate to each command group label are displayed by aligning the mouse cursor within the appropriate rectangle and clicking a mouse button.

3.2.4.2 Temporary vs. Permanent (Hanging) Menus

Command menus may be **temporary** (displayed for the duration of a command or until another menu is selected), or **permanent** (for as long as you want). Temporary command menus are displayed from the base of the command menu label (Figure 3-2). Permanent menus are also known as "hanging menus". You create them by selecting the <**move**> option from a temporary menu, moving the mouse cursor to the point where you want to hang the top of the menu, and pressing any mouse button. You can remove a temporary menu without selecting an option by pressing the **reset** key (<ESC> by default). Hanging menus display a variety of frequently used commands on screen, all the time. You can remove a hanging menu by selecting the <**exit**> option.

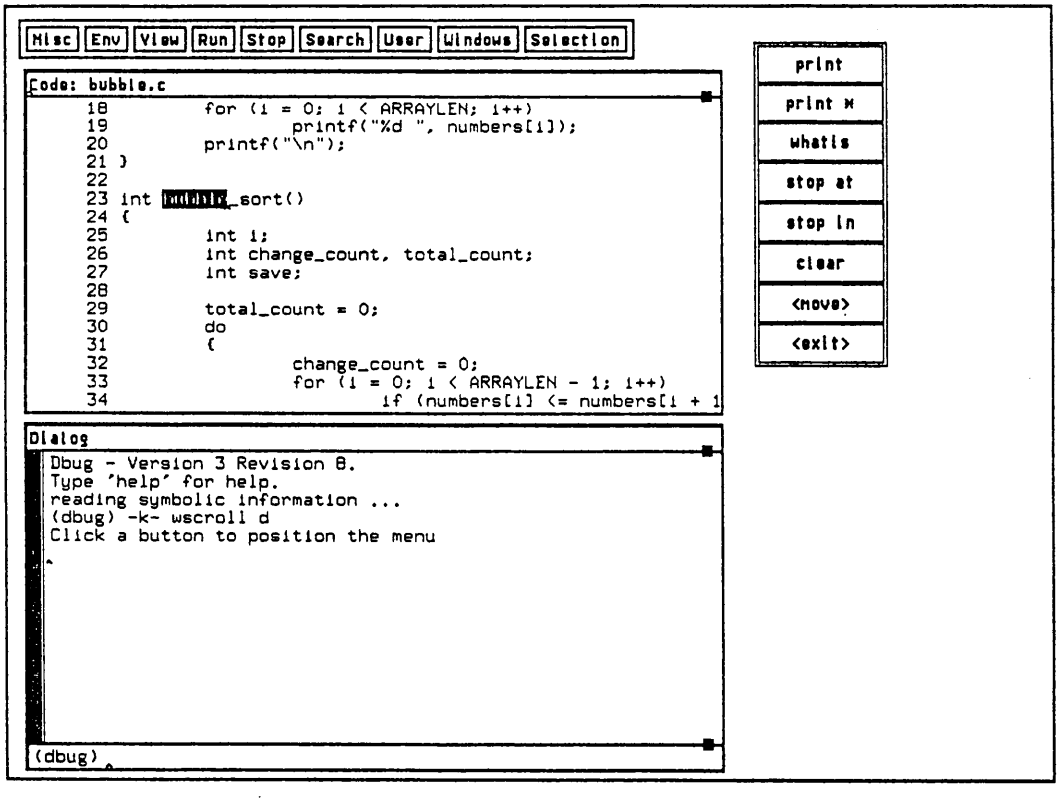


Figure 3-3. Marked Text and a Hanging Menu

3.2.4.3 Invoking Menu Commands

The three types of command menu options are:

1. Options with no parameter specified, or with a parameter known by default. These options are invoked by clicking the left mouse button on the desired entry in the corresponding menu.
2. Options with parameters specified through prompt lines that pop onto the screen when the menu option is selected. (Menu entries with "pop-up" prompts are highlighted around the edges.) These options are issued after you enter the parameter along the pop-up line and press <CR>.
3. Options with parameters specified before the option is selected. Such parameters are specified by marking text with the mouse. Command menu options which require selected (marked) parameters are found within the SELECTION menu. They may also be found in the USER menu, if you so define them (see the **addmenu** command in Chapter 5). These options are issued by selecting text **before** the option is selected. Only text in the CODE window can be selected.

A command created by one of these methods is immediately executed. An echo for the command is sent to the DIALOG window. The echo is prefixed with '-m-' to distinguish it from commands typed on the command line.

3.2.4.4 Marking Text With The Mouse

When executing a command through the command menu, you may specify parameters with the mouse. To mark text, use one of these methods:

1. *Dragging the cursor.* Align the cursor over a character that belongs to text (within the CODE window) from the desired parameter. Depress the left-hand mouse button and move the mouse over the text to be selected. Release the mouse button (Figure 3-3).

2. *Delimiting the area to be highlighted.* Align the cursor next to the text to be selected. Click the left button. This spot is marked by a small caret. Move the mouse cursor to the end of the text to be selected. Click the right button. The area between the clicks of the left and right buttons is highlighted.

3. *Selecting a word.* Align the mouse cursor on the word to be selected. Rapidly click the left button twice. This selects a string of contiguous non-blank characters.

4. *Selecting a line.* Align the mouse cursor on the line to be selected. Rapidly click the left button three times. All text selected in this way is highlighted. Pressing the left button once *unselects* the text.

3.2.4.5 Interpretations of Marked Text

DBUG can interpret marked text (also known as selected text) in a variety of ways. The particular interpretation depends on the command issued. The three possible interpretations are:

1. *Text.* Interpret the selection literally. The parameter is recognized exactly as that text which is highlighted, and nothing more. Specify **<text>** when defining a command with a text parameter.

2. *Expression.* This selection is also interpreted literally. However, DBUG expands the selected material to include the longest string of connected alphanumeric characters or underscores. Thus a long expression can be selected simply by marking one character in the expression. When defining a command with an expression parameter, specify **<expr>**.

3. *Line.* Interpret the selection as the line number containing the selected text. When defining a command with a line parameter, specify **<line>**.

Note: Only the first line of the marked text is considered if more than one line is selected.

3.2.4.6 Menu Definition Commands

Users can define command menu entries. All user-defined entries appear in the menu beneath the USER menu label. Commands for adding and deleting options to and from this menu appear below. These commands, which are described in full in Chapter 5, are:

addmenu - add a menu entry.

delmenu - delete a menu entry.

3.2.4.7 The Menus

This section lists the menu labels and the entries within each one. Command parameters that follow the command (example: `print <expr>`) are entered within pop-up forms that appear after the command is selected. Command parameters that precede the command (example: `<expr> print`) are entered by marking text before invoking the command. In both cases, DBUG interprets the command by adding the parameter to the end of the selected command. The commands are described in Chapter 5.

Misc

The command menu behind the MISC label supports several functions not associated with each other or any of the other labels. Most important of these is the <command> function. Selecting the <command> function generates a command line from which commands may be entered. This command line may be used as a substitute for the command line within the DIALOG window.

The MISC menu:

- help
- <command> <comm>
- source <file name>
- quit

Where:
<comm> is a DBUG command.

Env

The ENV menu contains commands for controlling the program's symbolic environment

The ENV menu:

- func <function name>
- file <file name>
- up
- down
- env

View

The VIEW menu contains commands for checking the value of variables and the status of the program.

The VIEW menu:

- where
- print <expression>
- status

Run

The RUN menu contains commands for executing the program under DBUG.

The RUN menu:

- step
- next
- cont
- clear
- return
- rerun
- run

Stop

The STOP menu contains commands for setting breakpoints.

The STOP menu:

- stop in <function name>
- stop at <line number>

Search

The SEARCH menu gives you information about program symbols (variables, functions, files, ...).

The SEARCH menu:

- whatis <symbol>
- which <symbol>
- whereis <symbol>

Where:
<symbol> is a variable or function name.

User

The USER menu is a user-defined menu. All entries are determined by you. Commands for adding and deleting options to and from this menu are presented later in this manual (see Chapter 5, "addmenu," "delmenu").

Windows

The WINDOWS menu contains commands for manipulating the DBUG window interface. Note that **wdelete**, **wpush**, **wpop** and **wmove** operate on the currently selected window.

The WINDOWS menu:

- wdisplay <window name>
- wdelete
- wpush
- wpop
- wmove
- wreset

Selection

The SELECTION menu contains commands for printing and identifying variables, and setting and clearing breakpoints. Parameters for SELECTION menu commands are identified by marking text.

The SELECTION menu:

- **<expr>** print
- **<expr>** print *
- **<expr>** whatis
- **<line>** stop at
- **<expr>** stop in
- **<line>** clear

Where:
<expr> Expand the selected material to include the longest string of connected alphanumeric characters or underscores. This selection is interpreted literally. Only a fraction of the required text needs to be selected, and the system selects the rest automatically. Note that the first or last letter of the expression must be an alphanumeric character or an underscore.

<line> Interpret the selection as the line number containing the selected text.

print * Print the reference of a pointer.

3.2.5 Function Keys

DBUG commands may be invoked by function keys. All function keys are user definable. The command attached to the pressed key is immediately executed, and (if it is not one of the "special function key" commands) is echoed in the DIALOG window. The echo is prefixed by "-k-". Commands generated by function keys are not recorded by the history mechanism.
Like menus, function keys may take marked text as a parameter.

A predefined setup exists for configurations with a numeric keypad. These definitions may be overridden using the **kdefine** command. The **help keys** command displays the current function key definitions.

A function key is a sequence of one or two key strokes.

Syntax:
[*color*] *keypad-key*
control-key
color alpha-key

Where:	*color*	A key to use to prefix other function keys. The two predefined color keys are the *gold* (<pf1>) and the *blue* (<pf4>) keys. Pressing a color key before the function key changes the function key's meaning. This is useful for increasing the number of available function keys. The *blue* and *gold* keys default definitions can be changed.
	keypad-key	One of the keypad keys. These are designated with the following symbolic form: <kp0>, <kp1>, <kp3>, <kp5>, <kp7>, <kp9>, <kpf->, <kpf+>, <kpf.>, <pf1>..<pf4>
	control-key	A keyboard key is clicked while the **ctrl** key is depressed. The symbolic representation for a control key is: <ctrl/a>...<ctrl/z>
	alpha-key	One of the alphanumeric keypad keys. The symbolic representation for a keypad key is: <a>..<z>, <0>..<9>

3.2.5.1 Function Key Definition Commands

The user may customize function keys to fit the needs of a particular application with the following commands:

kdefine - attach a function key to a command.

kreset - reset function keys to their default value (see Appendix C).

These commands are fully described in Chapter 5.

3.2.5.2 Special Function Key Commands

Commands that can be entered from the function keys alone are listed below. These commands may by specified with the **kdefine** command. Note that special function key commands are not echoed in the DIALOG window.

blue	Execute the "blue" prefix.
commline	Move the cursor to the command line. This command causes the DIALOG window to be redisplayed if necessary.
expand	Expand the current window to full DBUG frame size. Re-executing the *expand* command returns the window to its previous size.
gold	Execute the "gold" prefix.
redraw	Redraw the DBUG windows.
repeat	Repeat the last command issued from the command line.
reset	Remove a temporary menu, pop-up form (if existing), and the HELP window.
select	Define the current window as the selected window. This command also uncovers the selected window if it is hidden.

The default key definitions for these commands are specified in Appendix C.

3.3 The Alphanumeric Terminal Interface

This section describes the DBUG user interface that runs on an alphanumeric terminal. It describes all interface features supported by DBUG in this mode. The current implementation supports the following terminal types: `vt100`, `opus-pc` and `sun`. Select the terminal type by setting the environment variable: TERM.

This description of the alphanumeric interface is divided into three main sections.

1. Overview of the Alphanumeric Terminal Interface
2. The DBUG Windows
3. Function Keys

3.3.1 Overview of the Alphanumeric Terminal Interface

The DBUG alphanumeric user interface has the following main features:

- **Windows.** The debugging session may be monitored and controlled through a multi-window environment. Window display is user controlled.

- **Flexible Command Invocation.** Invoke a DBUG command in one of two ways: type it on the keyboard, or press predefined function keys. Any command that can be invoked through the command line can also be issued by pressing a function key. All function keys are user definable (see Chapter 5, "kdefine").

- **Flexible Parameter Specification.** Many DBUG commands require parameters. A parameter is entered on the keyboard as part of a command (example: `stop at 10`). If the parameter is a line number, place the cursor on the desired line and press the function key that executes the appropriate command.

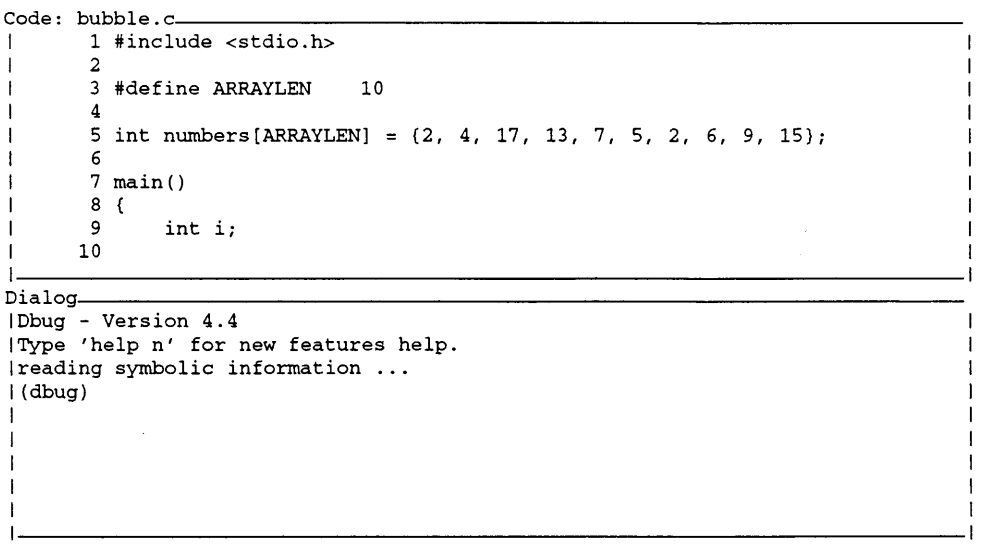


Figure 3-4. Opening Dbug Frame for an Alphanumeric Interface

3.3.2 The dbug Windows

The DBUG windows and their general functions are listed below:

CODE: displays the program being debugged, including user-defined breakpoints and a pointer to the next line to be executed.

DIALOG: provides a prompt for entering commands, and an area for echoing all commands and displaying the DBUG responses.

PROGRAM: provides an area for the user-program input and output exchanges.

HELP: lists DBUG command syntax and other help information.

TRACE: displays the trace information produced by the in-system emulator.

3.3.2.1 The Selected Window

The window where the cursor is placed is called the *selected window*. The selected window defines where work may take place. Window commands will take the selected window as a default parameter. Thus, instead of specifying "wdelete code", you can execute a **wdelete** function key while the cursor is in the CODE window. Change the selected window by executing the **wnext** command, or by pressing a function key defined as **wnext** (<ctrl/n> or <kpf.> by default).

3.3.2.2 The DBUG Frame

Each DBUG session begins with the appearance of the DBUG frame on the screen (Figure 3-4). All work takes place within the DBUG frame. The opening (default) DBUG frame contains two windows: CODE and DIALOG.
Note that none of the DBUG windows may exceed the screen boundaries. The coordinates of the DBUG frame are:

1. (0, 0) for the upper lefthand corner of the `vt100`, `opus-pc` and `sun`.
2. (22, 79) for the lower righthand corner of the `vt100`, and (23, 79) for the `opus-pc`. For the `sun`, the lower righthand corner of the DBUG frame will be the lower righthand corner of the parent window.

The rest of this section describes the windows supported by DBUG when it is running on an alphanumeric terminal. These are: CODE, DIALOG, PROGRAM, HELP and TRACE.

3.3.2.3 The CODE Window

The CODE window displays the word "CODE" and the name of the current source file along its upper lefthand corner.

The CODE window displays the section of your program code that is currently executing and provides a visual representation of the debugging session. DBUG marks lines that have breakpoints with an asterisk (*), and marks the next line to be executed with an arrow (=>).

The program code is displayed in one of two modes:

1. **Source.** Source mode presents the program source code as originally written. Line numbers are displayed on the left side of the window.
2. **Assembly.** Assembly mode presents the disassembly of a relevant segment of your program. Each instruction's address is displayed on the left side of the window.

DBUG determines which mode to use. It selects source mode when the current module has symbolic information (compiled with **-g** option). Assembly mode is selected in all other cases.

Lines which exceed the window width are truncated.

To scroll through the CODE window, choose one of the following:

- Use the cursor. CODE must be the current window. Pressing the **up** arrow while the cursor is next to the upper border scrolls the window one line up. Pressing the **down** arrow while the cursor is next to the bottom border scrolls the window one line down.
- Use the **wscroll** command. This command scrolls a number of lines up or down the CODE window.
- Use the **wgo** or **search** commands. These commands scroll to a specific line number in the CODE window.

The CODE window is the only window in which line selection may take place.

3.3.2.4 The DIALOG Window

The DIALOG window displays the word "Dialog" along the upper lefthand border. It has two logical functions: 1) to provide a line for entering commands, and 2) to display debugger responses.

The line where the cursor is located is called the command line. The command line is usually identified by the prompt **(dbug)**.

There are two ways to enter a command. The first is to type the command on the keyboard, with the cursor located on the command line. DBUG puts the cursor on this line when entering the DIALOG window. You can also move the cursor there by pressing the function key defined as **commline** (<kp0> by default), or by pressing a function key defined as **wnext** (next window) until the cursor comes around to the DIALOG window (default <ctrl><n>).

The second way to enter a command is with the aid of the **history buffer**. All commands entered through the command line are saved in a history buffer. You can retrieve these commands by pressing up-down arrows (scrolling through the history buffer) until the appropriate command appears in the line. Press the return, and this command is executed. Commands that have been invoked using function keys however are not saved in the history buffer.

The DIALOG window displays the session history, including commands and DBUG prompts and echoes. Commands entered by function keys are echoed with the "-k-" prefix.

If the DIALOG window is deleted or covered by other windows, executing the **commline** command automatically redisplays the DIALOG window.

3.3.2.5 The PROGRAM Window

If the PROGRAM window is displayed, all I/O to and from the program being debugged takes place in this window. Thus, when the PROGRAM window is displayed:

1. Your program's output is automatically displayed in this window.
2. All inputs to the program should be entered through this window. You can input data only when the PROGRAM window is the selected window.

When DBUG is invoked, the PROGRAM window is not displayed. Output is displayed in the DIALOG window. Program input can be entered from the command line by preceding the text with an escape character "@".

To display the PROGRAM window execute the following DBUG commands:

To display the program window use:

```
wdisplay program
```

at this point, parts of the CODE and DIALOG windows are covered by the PROGRAM window. To shrink the CODE and DIALOG windows use the following commands:

```
wmove code vrd 120
wmove dialog vrd 120
```

(NOTE: Don't confuse the "@" escape character, which is used for passing input to the debugged program, with the "|" escape, which is used for passing commands to the remote target monitor.)

3.3.2.6 The HELP Window

The HELP window provides information about DBUG, window, and function key commands. The HELP window is manipulated like any other window. But unlike the other windows, it can be opened by invoking the **help** command. The HELP window may be closed with the **reset** function key. The default **reset** key is <ESC>. This command and its parameters are described in Chapter 5.

3.3.2.7 The TRACE Window

The TRACE window is used only in an HP64772 ISE configuration. It displays the results of the hardware trace output after the **traceh list** command has been issued.

The default location of the TRACE window is along the righthand border of the DBUG screen.

Issuing the **traceh list** command when the TRACE window is not present sends results to the DIALOG window.

3.3.2.8 The Window Manipulation Commands

This section lists the commands for manipulating the DBUG windows. These commands allow the user to control the display of information on the screen and customize DBUG's default appearance.

The DBUG window manipulation commands are listed below. A complete description of each command and its syntax is found in Chapter 5.

wdisplay - display a window.

wdelete - delete a window.

wgo - go to a line in the file.

wmove - move or resize a window.

wpop - pop a window.

wpush - push a window.

wreset - reset windows.

wscroll - scroll through a window.

wnext - go to the next window.

3.3.3 Function Keys

DBUG commands may be invoked by specific function keys. All function keys are user definable. The attached command (unless it is a "special function key" command, section 3.3.3.2) is echoed in the DIALOG window. The echo is prefixed by "-k-". Commands generated by function keys are not recorded by the history mechanism. Note that the full power of the function key mechanism is best exploited by a keyboard with a keypad.

A predefined setup exists for configurations with a numeric keypad. These definitions may be overridden using the **kdefine** command. The **help keys** command displays the current function key definitions.

A function key is a sequence of one or two key strokes.

Syntax:

[*color*] *keypad-key*
control-key
color alpha-key

Where:

color — A key to use to prefix other function keys. The two predefined color keys are the *gold* (<pf1>) and the *blue* (<pf4>) keys. Pressing a color key before the function key changes the function key's meaning. This is useful for increasing the number of available function keys.

keypad-key — One of the keypad keys. These are designated with the following symbolic form:

<kp0>..<kp9>, <kpf->, <kpf,>, <kpf.>, <pf1>..<pf4>

Actual key assignments for a given host is described in the Appendix C.

control-key — A keyboard key is clicked while the **ctrl** key is depressed. The symbolic representation for a control key is: <ctrl/a>...<ctrl/z>

Note that the following keys may **not** be defined as function keys: <ctrl/m>, <ctrl/q>, <ctrl/s>

alpha-key — One of the alphanumeric keypad keys. The symbolic representation for an keypad key is: <a>..<z>, <0>..<9>

3.3.3.1 Function Key Commands

You may customize function keys to fit the needs of your particular application with the following commands:

kdefine - attach a function key to a command.

kreset - reset function keys to their default value.

Function key commands may require a line number as a parameter (example: `stop at <line no>`). You can bind a function key to a command that requires a line parameter (see Chapter 5, "**kdefine**"). To execute function key commands with line parameters, go to the CODE window, position the cursor on the desired line, and press the function key. The line number is appended to the end of the command.

3.3.3.2 Special Function Key Commands

Commands that can be entered only from the function keys are listed below. These commands may be specified with the **kdefine** command. Note that special function key commands are not echoed in the DIALOG window.

blue	Execute the "blue" prefix.
commline	Move the cursor to the command line. This command will also cause the DIALOG window to be displayed if it is not already on screen (default definition is keypad 0).
expand	Expand the current window to full screen size. Executing the *expand* command again returns the window to its previous size.
gold	Execute the "gold" prefix.
redraw	Redraw the screen.
repeat	Repeat the last command issued from the command line.
reset	Remove the HELP window if it is displayed.

These commands' key definitions are specified in Appendix C.

Chapter 4

USING DBUG

4.1 Introduction

This Chapter describes the basic concepts and conventions required to use DBUG commands effectively.

DBUG may be used for symbolic debugging of programs compiled with the GNX assembler and the C, Pascal, and FORTRAN compilers. The symbolic information is produced for DBUG when the compiler is called with the **-g** flag (for UNIX and MS-DOS) or the **/debug** flag (for VMS).
DBUG may also be used to debug programs compiled without the either the **-g** or **/debug** flags. In this case, only global variables and procedures are recognized by their names, and no line number or data type information is available.

4.2 DBUG Operating Modes

DBUG supports two operating modes: *native mode* and *remote mode*.

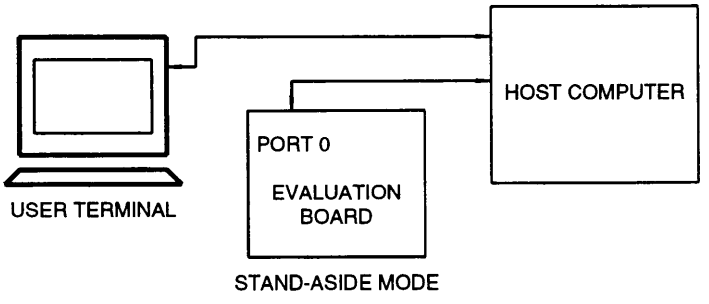
4.2.1 Native Mode

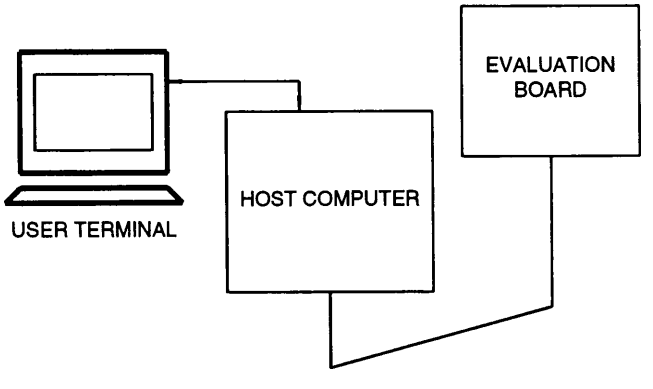
DBUG operates in *native mode* by default. In *native mode*, both DBUG and the user's program are executed on the host computer. DBUG may be used as a native debugger on the following National Semiconductor development systems using the UNIX Operating System:

- SYS32/30
- SYS32/50

4.2.2 Remote Mode

In *remote mode*, DBUG runs on the host computer while the user program is downloaded and executed on the Series 32000-based target system. The target system can be a development board, ISE or SPLICE. Program loading, execution and debugging are performed by entering DBUG commands from the host terminal. DBUG cooperates with the target board monitor (the control program for the target board) by carrying out the downloading and debugging process. DBUG translates debugger commands into monitor commands.





FT-01-0-U

Figure 4-1. Operating Modes

The host and target systems can be connected via a serial line or via LAN (ethernet). When an ethernet connection is used, the target system is considered as another host on the local area network. Section 4-4 describes in detail DBUG and development board monitor communications over ethernet.

Remote mode operation is available with National Semiconductor's Evaluation Boards, the SPLICE Development Tool, and the Series HP64000 emulators for the NS32000 family CPUs.

4.2.3 DBUG and GNX Tools

DBUG interacts with other GNX tools such as the assembler, linker and optimizing compilers.

DBUG also reads the *.gnxrc* (*gnx.ini* on VMS and MS-DOS) file, which is usually created by the GNX Target Setup Utility (GTS). The *.gnxrc* file specifies default parameters such as the CPU, monitor in use, communication link name, MMU, and FPU (see the GNX Commands and Operation manual, for details).

4.2.4 Initializing DBUG

On *Series 32000*-based systems DBUG operates in *native mode* by default. In *native mode*, DBUG reads the symbolic debugging information of the executable file, and prompts for commands.

DBUG operates in *remote mode* when the **-l** option is specified in the invocation line, or when the user issues the **connect** command from within DBUG. In *remote mode* DBUG establishes communications with the monitor program on the target board and prepares the environment for loading the user program onto the target board. Once the program is loaded onto the target board (using the **load** command), the debugging session may proceed.

4.2.5 Debugging Session

A typical debugging session includes setting breakpoints, running the program until breakpoints are reached, stepping, and examining program variables and registers.

The following example illustrates DBUG operating in *remote mode*. Assume the executable file `a32.out` contains the program to be executed on the target hardware. To invoke DBUG, enter:

```
% dbug a32.out
```

DBUG displays the following message:

```
dbug Version 4.0
reading symbolic information...
```

Next, connect to the target hardware using the **connect** command. Assume the communications link with the remote target hardware is `/dev/tty5`.

(dbug) `connect link tty5 with mon gx32 fpu 381`

Note that these parameters can be prespecified using GTS. You may also use the **config** command to specify configuration before issuing the **connect** command.

(dbug) `config mon gx32 fpu 381 sp 0x100000`
(dbug) `connect link ttys`

Load the executable file

(dbug) `load with sp 0x100000`

```
loading...
```

DBUG prompts when loading is complete. The debugging session can now begin.

4.3 Basic Terms

4.3.1 Current Environment

A name in a program may have different meanings, depending on its context and scope. To eliminate any possible ambiguity in the interpretation of a name, DBUG uses the concept of *current environment*. The *current environment* is defined by the combination of the *current file* and the *current procedure*. When the execution of the program stops for any reason and control returns to DBUG, the procedure at which execution stopped is called the *current procedure*. The source file that contains this procedure is called the *current file*. The *current environment* may be changed and displayed using the **func, file** or **env** commands.

The commands that restart program execution (i.e,. **cont**, **step**, or **next**, etc.), continue from the location where the execution halted (i.e. the environment at the halting point).

4.3.2 Constants

In addition to constants defined in the program (such as enumeration constants), DBUG recognizesthe following types of constants: numbers, characters, strings and boolean constants. Constants may be used as part of general expressions in a DBUG command.

Numbers typed as part of a command may be decimal, hexadecimal (with **0x** prefix), or octal (with **0** prefix).

Examples of legal numbers are:

Decimal: `2, -123, 32.5, 1.2e3`

Hexadecimal: `0x1f, -0x432, 0x7ffffda4`

Octal: `0123, 0277`

Floating point constants used as part of DBUG commands are interpreted as double precision numbers.

Character constants are enclosed in single quotes. An ASCII character or the octal numeric equivalent may be used if it is preceded by the "\" character.

Examples of legal character constants: `'a', '\121', '\03', '\0121'`

The character `'\121'` denotes the ASCII character with the octal value of 121 (corresponds to `Q`).

Strings are enclosed by double quotes (" "). For example, the following statement enclosed in quotes is considered a string: `"this is a string"`. If the " (*double quote*) character is part of a string, enter it into the string by preceding it with the backslash (\) character (i.e., "this is a \"string", results in the following: `this is a "string`).

The boolean constants recognized by DBUG are `true` and `false`.

4.3.3 Symbols and Names

All program identifiers, such as variables, procedures, type definitions, registers, file and module names are recognized by DBUG. These identifiers are called *symbols*. A symbol may be qualified by the module or procedure to which it belongs (i.e. variable `i` of procedure `main` can be referred to as `main.i`).

Qualified symbols are called **names**. The general form of a name is:

module.function1.function2...functioni.symbol

Global symbols that do not belong to a particular module are qualified by a period (.). For example, the qualified name of global symbol `i` is `.i`. When a symbol with no qualifier is used in DBUG commands, DBUG first searches the static scope to locate the symbol. If the search fails, DBUG searches the dynamic scope (i.e. call stack). If the search fails again, DBUG reports that the symbol is undefined or inactive.

4.3.4 File Names

Internally, DBUG defines a module for each program source file. The module names are used to qualify symbols defined in the specified module (symbols may be procedures, functions, types, static variables, etc.). For example, file `foo.c` defines the module `foo`. The function `f` in file `foo.c` is accessed by `foo.f`.

4.3.5 Registers

All Series 32000 CPU, FPU and MMU register names are recognized by DBUG when the register name is preceded by a dollar sign ($). Table 4-1 lists the legal register names, register descriptions, and the operating modes in which the registers are recognized.

4.3.6 Line Numbers

The line numbers are used to access source information in the source files. The syntax of the line number specification is:

[*"filename"*:]*number*

For example, to stop execution when the program arrives at line 50 of file `eval.c`, type: `stop at "eval.c":50`. If `eval.c` happens to be the current file, this can also be accomplished by typing `stop at 50`.

Table 4-1. Register Names
Sheet 1 of 2

NAME	DESCRIPTION
$r0-$r7	general purpose registers
$fp	frame pointer
$sp	stack pointer
$pc	program counter
$f0-$f7	floating point registers
$fsr	FPU status register
$l0-$l7	long floating registers
$psrmod	PSR and MOD registers
$psr	program status register
$mod	module table register
$sb	static base register
$is	interrupt stack pointer
$us	user stack pointer
$in	interrupt base register
$cf	configuration register
$dcr	debug condition register
$dsr	debug status register
$car	compare address register
$bpc	breakpoint program counter
$msr	MMU status register
$ptb0-$ptb1	page table base 0 and 1
$eia	error invalidate address
$bpr0-$bpr1	breakpoint registers 0, 1
$bcnt	breakpoint count register
$bar	breakpoint address register
$bmr	breakpoint mask register
$bdr	breakpoint data register
$bear	bus error address register
$tear	translation exception address register
$ivar0-$ivar1	invalid virtual address

Table 4-1. Register Names
Sheet 2 of 2

NAME	DESCRIPTION
$bpcr	BPU cntrl register
$bprmr	BPU right mask register
$bplmr	BPU left mask register
$bpmr	BPU mask register
$bfsr	BPU function select register
$bpcntr	BPU counter register
$adc0-$adc1	DMA channel 0-1 source address counter register
$adr0-$adr1	DMA channel 0-1 source address register
$bltc0-$bltc1	DMA channel 0-1 transfer complete register
$bltr0-$bltr1	DMA channel 0-1 block transfer register
$adcb0-$adcb1	GX320 device B source address counter register
$adrb0-$adrb1	GX320 device B source address register
$mode0-$mode1	DMA channel 0-1 mode control register
$ctl0-$ctl1	DMA channel 0-1 control register
$stat	DMA status register
$imsk	DMA status register
$dstat	DMA status register
$ivct	ICU interrupt vector register
$isrv	ICU interrupt in-service register
$tc0-$tc2	Timer 0-2 registers
$tcra0-$tcra2	Timer 0-2 autoload register A
$tcrb0-$tcrb2	Timer 0-2 autoload register B
$tcntl0-$tcntl2	Timer 0-2 control registers
$y0-$y1	FX16 FAM multiple input register
$a0-$a1	FX16 FAM accumulator
$dptr0	FX16 complex multiply registers data pointers
$dptr1	FX16 complex multiply registers data pointers
$cptr	FX16 coefficient memory vector pointer
$ctl	FX16 FAM control register
$st	FX16 FAM status register
$c0-$c95	FX16 FAM coefficient RAM array

4.3.7 Address

Addresses may be symbolic or absolute. Absolute addresses are numbers, which may be entered as decimal, octal or hexadecimal numbers. Symbolic addresses are specified by an ampersand (&), which must precede the symbol. The plus (+) and minus (-) operators can be used to create an address.

For example, the notation

```
&main
```

denotes the address of the symbol `main`.

```
&main + 5
```

denotes the address of the symbol `main` plus `5`.

4.3.8 Address Range

The format for address range is:

address, *address* or *address..address*

The first format, where the address range is separated by a comma (,), is used with most DBUG commands.

The second format, where the address range is separated by two periods (..), is only used with the HP64000 In-System Emulator related mapping commands.

For example, the command

(DBUG) `&main+0x17,&main+0x4c/i`

disassembles the range of addresses `main+0x17` through `main+0x4c`.

The following DBUG command

(dbug) `0x1000,0x1020/X`

prints the contents of the memory range `0x1000` through `0x1020` in hexadecimal format.

4.3.9 C Block Variables

The C programming language allows different variables defined in the same procedure to have the same name when they are located in different block levels. The variable names, which are defined in the inner blocks, can also be qualified.

DBUG creates a dummy procedure for each block called *$bn*, where `n` is a serial number assigned by the debugger. To find the qualified name of the internal variable, use the **whereis** command (see Chapter 5, DBUG Commands).

4.3.10 Expressions

General types of expressions can be used as part of DBUG command syntax. The common subset of expressions that are legal for Pascal and C programming languages are supported by DBUG.

Indirection may be specified using either C or Pascal notation, i.e., using the asterisk (*) prefix or the caret (^) suffix.

Array references are specified by using brackets ([ ]). Array references for multidimensional arrays are specified by `arr[i][j]` for programs written in C and Pascal, and by `arr[i,j]` for programs written in FORTRAN. The record/structure field reference is specified by either the period (.) or arrow (->).

For instance, the following notations are equivalent: `rec.fld`, `rec->fld`, `rec^.fld`

All of the standard arithmetic operators of Pascal and C are recognized by DBUG. This does not include standard language specific built-in functions (such as *sizeof* in C, *chr* in Pascal, ** in FORTRAN etc.). If a boolean expression includes `and` or `or` operators, the arguments of these operators must be enclosed in parentheses. For example:

```
stop if (i=j) or (k=l)
```

is legal, where as

```
stop if i=j or k=l
```

is not.

The following is the list of the recognized operators:

Table 4-2. Recognized Operators

OPERATOR	EXPLANATION
+	add
-	subtract
*	multiply
/	divide
div	integer divide
mod	modulus (integers only)
&	address of
*, ^	contents of pointer
<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to
== or =	equal to
!= or < >	not equal to
. (period)	structure field reference
and	logical and
or	logical or

The period operator may be used with pointers to records, as well as with records.

4.3.11 Types

DBUG recognizes all the data types defined by your program. These types may be used in expressions (via a typecasting mechanism), in addition to the predefined basic types, such as integers, reals and boolean, which are inherent to DBUG. The predefined basic type names are; *$integer, $char, $real, $boolean. $char* and *$boolean* are 1 byte long, *$integer* is 4 bytes long, *$real* is 8 bytes long. DBUG assigns these types to constants and expressions in DBUG commands.

DBUG performs the required conversions between different languages' types. It also performs the necessary conversions between types appearing in the same expression within the same language.

DBUG performs all the necessary type checking when evaluating expressions. It also supports a type interpretation mechanism which permits expressions to be treated as if they were of a particular type.

For example, if the debugged program, written in C, contains declarations:

```
typedef struct {
        int _cnt;
        char *_ptr;
        char *_base;
        char _flag;
        char _file;
} FILE;

char p[8];
int ptr;
```

enter:

(dbug) `print FILE(p)`

to print the contents of `p`, as if it were a structure of type `FILE`. Similarly, the command

(dbug) `print *&FILE(ptr)`

interprets `ptr` as a pointer to the type `FILE`. The command prints the variable `ptr` as if it were of structure type `FILE`.

(dbug) `print *&$integer(*&$integer($sp)) X`

Performs double indirection on `$sp` so it prints the value of 0(0(sp)).

If a C structure definition has a tag, then the tag name preceded by *$$* is used to identify the tag. For example, if the C program contains the following type definition:

```
struct  person {
        char name[];
        int age;
};
```

Enter the command

(dbug) `whatis $$person`

to print the definition of the structure tag `person`. This notation is also used for type interpretation purposes as described above.

DBUG recognizes and checks the types of your program variables in the following cases:

When printing variables, DBUG automatically selects the output format according to their type's *implicit radix*. You can over-ride the *implicit radix* by specifying an

explicit radix as part of the **print** command.

When assigning values to program variables, DBUG checks the validity of the assignment by comparing the variable types involved. This check may be eliminated by setting the *$unsafeassign* built-in variable (see Chapter 5, the **set** command).

DBUG verifies that the combination of variables in an expression is legal.

DBUG verifies that subscripts do not cross array boundaries. If they do, DBUG issues an error message.

4.3.12 Special Characters

Certain characters have a special meaning to the DBUG command interpreter.

(crosshatch) - The crosshatch character, as the first non-blank character in the line, denotes a comment line.

**** (backslash) - The backslash character is the continuation character within a command file. If a command is too long to fit onto one line, use the backslash (\) character as the last character of a line to indicate that the next line continues the command.

| (vertical bar) - The vertical bar character is used with a monitor escape. When | is detected as the first non-blank character in the line, the remainder of the line is passed directly to the monitor.

; (semi-colon) - The semi-colon character is used to enable the multiple command line.

. (period) - The period character is used in the low level print command to indicate the next address.

4.3.13 Debugger Files

DBUG uses the following files:

Objfile - contains the COFF program that is being debugged. This file is used as an input to DBUG.

Corefile - is created when the application program terminated abnormally while running under UNIX. In the *native mode*, DBUG uses this file to view the contents of the core dump. Core dump analysis is not available in *remote mode*.

Source files - the debugger uses *source files* to access source information.

Command files - contain debugger commands that are read and executed by the debugger. The debugger reads and executes the commands using the **source** command. One source file may activate another source file. DBUG sets the nesting level limit to ten (10). The host system may restrict this to a smaller number.

.dbuginit - (*dbug.ini* on VMS and MS-DOS) is the initialization file that is executed by the debugger as a part of the initialization process. If DBUG is invoked with either **-c** or **-noc**, the initialization file is not executed. DBUG will first search for the initialization file in the current directory. If found, the *.dbuginit* file is executed. If it is not found, DBUG will then search the user's home directory for the initialization file.

.gnxrc - (*gnx.ini* on VMS and MS-DOS) DBUG seeks and reads the target *setup file* in the current directory, the users's home directory or in the GNX directory (in that order) to set up the default configuration information.
The setup file is created by the GTS - the GNX Target Setup Utility. The following definitions of the setup file are meaningful for DBUG: CPU name, MMU name, FPU name, communications link, monitor name.

Log file - a *log file* containing a log of the debugging session can be created. The log format may be **full** or **short**. In the **short** format, only the user commands are logged. In the **full** format, DBUG's responses are recorded as comments in the *log file* (see Chapter 5, the **log** command).

4.3.14 Breakpoints and Traces

DBUG provides users with extensive breakpoint and tracing capabilities. Breakpoints are used to stop debugged program execution at specified places, upon occurrence of specified conditions, or to examine/modify variables, registers and memory locations. Your program can be traced without stopping its execution. You can trace variable values, calls to procedures, execution of particular procedures, etc.

You can specify that program execution should stop if the value of a specific variable or memory location changes. This is true for global and local variables. However, DBUG does not support tracing or stopping on value changes of variables that are allocated in registers.

When doing conditional tracing or stopping to a particular procedure, DBUG sets all the necessary breakpoints and removes them internally. For example, if you use the DBUG command: `trace in proc`, where `proc` is a procedure in your program, DBUG will set an internal breakpoint at the entry to the `proc`, turn on the tracing flag (which causes DBUG to report tracing information), and set another breakpoint at the exit from `proc`. DBUG turns off the tracing flag when it reaches the breakpoint at `proc`'s exit. Both internal breakpoints are removed when your program leaves `proc`.

In *native mode*, there is no limitation on the number of breakpoints you can set. In *remote mode*, the number of breakpoints is target dependent. Refer to your target board monitor manual for this information.

NOTE: DBUG assumes that all procedures in your program use `enter` and `exit` machine instructions for proper frame allocations and removal on the calling stack. Still, some procedures may use `enter` and `exit` instructions (e.g. some library routines). If you set breakpoints or traces in these procedures, DBUG might fail to provide accurate call stack information or fail to set internal breakpoints in correct places.

DBUG supports two forms of program tracing: software and hardware. Software tracing single-steps the debugged program. Hardware tracing executes the debugged program in real time (tracing without disturbing program execution). Chapter 6 provides a detailed description of DBUG's hardware tracing capabilities.

4.4 Ethernet Support

DBUG supports the ethernet connection between your development board and host system using the Internet Protocol (IP). This allows you to download and debug your application using a Local Area Network (LAN). A LAN provides for both convenience in linking hosts to targets and greater communication speed.

The packet exchange between DBUG and your development board is done using the User Datagram Protocol (UDP). The message portion of each datagram consists of the string of bytes that the debugger and monitor exchange over a serial link. However, the UDP is not a reliable protocol (i.e. packets can get lost during communications). In order to overcome this problem, DBUG implements a mechanism for lost packet detection and recovery.

The DBUG **connect** command supports the ethernet connection. The **-n** invocation flag is used to specify the target board name. The ethernet connection between DBUG and the target board can be lost if the target board is reset. The **connect** command with the **node** option is used to re-establish the connection (see Section 5.1.8 for details on the **connect** command).

4.4.1 Description of Ethernet Operation

1. DBUG establishes communication with a monitor. Once this communication is established, the monitor does not accept any other communication requests from DBUG or other hosts.
2. The debugging session involves communication between DBUG and your monitor.
3. After the debugging session has finished (issuing the DBUG **quit** command), the connection with the monitor is terminated by DBUG.

4.4.2 Example

```
%dbug a32.out
dbug - Version 4.0
Type 'help' for help
(dbug) connect node db01 with epu cg16 mon cg16 fpu nofpu
connection with db01 [xx.yy.zz.uu] established
setup in remote mode
(dbug)
```

4.5 Partial Symbolics Mode

In the default mode of operation, DBUG loads all the symbolic information of the executable file during the startup stage. This provides immediate availability of symbolic information, which in turns enables fast execution of complex symbolic information processing.

For very large executable files, the loading of symbolic information can result in a long startup time and large memory requirements. DBUG therefore provides a partial symbolic mode.

The partial symbolics mode is invoked using the **-p** (**/P** in VMS) invocation line parameter. In this mode the symbolic information is loaded on demand; explicitly by using the **file** command, or by stopping in or stepping to a new module. Once DBUG stops in a module whose symbolic information is not loaded, dbug loads the information automatically. The symbolic information of previously activated modules remains in the symbol table.

Chapter 5

DBUG COMMAND SET

5.1 Introduction

This chapter contains the command set for DBUG. Commands are presented in alphabetical order. For a functional presentation of the commands and the associated command syntax, see Appendix C.

5.1.1 ADDMENU - add a menu entry

SYNTAX

addmenu [**<text>** | **<expr>** | **<line>**] *command*

DESCRIPTION

Addmenu is used to define new command menu entries when using the graphic terminal interface.

All user-defined entries appear in the USER menu.

The *text*, *expr* and *line* options define how parameters should be interpreted when selected as text.

<text> Interpret the selection literally.

<expr> Expand the selected material to include the longest string of connected alphanumeric characters or underscores. Like **<text>**, this selection is interpreted literally. But you need select only a fraction of the required text, and the system selects the rest automatically. Note that the first or last letter of the expression must be an alphanumeric character or an underscore.

<line> Interpret the selection as the line number containing the selected text.

Command is a parameterless DBUG command or alias. The alias is only decoded at execution time, so it is possible to define the alias even after the **addmenu** command is issued.

EXAMPLES

(dbug) `alias si "stopi at"`
(dbug) `addmenu <line> si`

The `si` alias is added to the user menu. This entry allows you to interactively create breakpoints in assembly mode.

(dbug) `addmenu where`

Add the `where` command to the user menu.

SEE ALSO

delmenu

LIMITATION

Not supported in MS-DOS.

5.1.2 ALIAS and UNALIAS - define command aliases

SYNTAX

alias *name* [*name* | *"string"*]

alias *name(parameters)* *"string"*

alias

unalias *name*

DESCRIPTION

When commands are processed, DBUG determines if the first entered word in a command line is an alias. If it is an alias, DBUG treats the input as though the corresponding string (with values substituted for any parameters) were entered. The parameters must be separated by a comma (,).

If **alias** is called with no parameters, all the defined aliases are printed. When only one parameter is specified, **alias** prints its alias definition.

The debugger has a predefined set of aliases. Use the **alias** command with no parameters to print the predefined aliases. You may redefine or delete these aliases.

The **unalias** command removes the alias definition for *name*.

The **unalias** command should not be used in the same line (after a semicolon) as **begin** or **load** commands.

EXAMPLES

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

Defines an alias called `b` that sets a breakpoint (stop) at a line `x`. For example, `b(56)` is equivalent to the `stop at 56` command.

(dbug) `alias st stop`

This command defines `st` to be an alias for the `stop` command.

The following command lists the alias definitions

```
(dbug) alias
st stop
b(x) stop at x
```

5.1.3 ASSIGN - assign a value to a variable

SYNTAX

assign {*variable* | *address*} = *expression* [*size*]

DESCRIPTION

The **assign** command sets a specified *variable* or *address* to the value of an *expression*.

size determines the assignment operation size: **b** - byte, **x** (or **d** or **o**) - short word, **X** (or **D** or **O**) - long word.

DBUG performs type-checking when performing assignments. If *variable* and *expression* types are not compatible, DBUG issues an error message. The type-checking procedure may be bypassed by setting the DBUG variable *$unsafeassign*.

EXAMPLES

(dbug) `assign total = 123.4`

Assigns the floating variable `total` the value `123.4`.

(dbug) `assign 0x1000 = 0x200 d`

Replaces a short-word at address `0x1000` with the value `0x200`

(dbug) `assign array[20][d] = 2 * array[20][22] + $r3`

Assigns the value of the expression `array[20][22]` multiplied by `2` and incremented by the contents of CPU register `r3` to `array[20][d]`.

(dbug) `assign $r0 = 0x23efbbb`

Assigns the hexadecimal value `23efbbb` to the CPU register `r0`.

SEE ALSO

set

5.1.4 BEGIN - begin debugging an objfile

SYNTAX

begin *objfile* [*corefile*]

DESCRIPTION

The **begin** command starts debugging the *objfile*. This command is useful if you want to switch from the file currently being debugged to another file without leaving the DBUG session. If the optional *corefile* is specified and exists, then it may be used for crashed program analysis.

All the breakpoints, traces and any other programmed events are deleted. All the alias and internal variable definitions are preserved.

EXAMPLES

Suppose you are debugging an object file called `a.out`.

(dbug) `begin test`

DBUG reads symbolic information of `test` and returns the prompt "(dbug)". The file `test` becomes the file being debugged.

SEE ALSO

run, rerun, load

5.1.5 CALL - excecute a procedure

SYNTAX

call *procedure()*

DESCRIPTION

Exeute the procedure. Parameters cannot be passed to the called procedure. The **call** command applies to the *remote* mode only. *$callproc* must be set.

EXAMPLE

```
(dbug) set $callproc
(dbug) call foo()
```

LIMITATION

This command is not supported when using an ISE.

5.1.6 CATCH and IGNORE catch/ignore signals

SYNTAX

catch [*signal number* | *signal name*]

ignore [*signal number* | *signal name*]

DESCRIPTION

These commands are supported only in the *native mode*. The **catch/ignore** commands advise DBUG to react to, or **ignore**, a signal trap before the signal is sent to your program.

When your program reacts to a signal and the signal is caught (by **catch**), DBUG gains control before the signal arrives at the program. You are notified that a specific signal was caught. Signal names are case-insensitive and the prefix SIG is optional. By default, all signals are trapped except SIGCHILD, SIGALRM, SIGHUP and SIGKILL. When no parameters are specified, the signals currently affected by the **catch** or **ignore** commands are listed.

EXAMPLE

(dbug) `catch SIGINT`

The debugger halts the program being debugged when it encounters the SIGINT signal.

SEE ALSO

cont, signal (2) Unix manual

5.1.7 CLEAR - clear breakpoints

SYNTAX

clear [*number*]

DESCRIPTION

The **clear** command clears all breakpoints at the selected source line *number* or the selected address. If no parameter is supplied, the current position (line or address) is used.

Clear is context sensitive. If the current file contains source line information, the *number* is interpreted as a source line, otherwise, the *number* is interpreted as a memory address.

Clear is particularly useful when working in the graphic environment. With this command you can remove breakpoints using the mouse.

EXAMPLE

(dbug) `clear 19`

If the current file has source line information (i.e compiled with **-g** option), all breakpoints on line 19 are cleared. Otherwise, all the breakpoints at address 19 are cleared.

SEE ALSO

delete, status, stop

5.1.8 CONFIG - configure for remote target system

SYNTAX

config [**verbose** {**on** | **off**}] [**baud** *number*] [**cpu** *name*] [**mmu** *name*] [**fpu** *name*] [**mon** *name*] [**sp** *number*] [**load** {**hex** | **binary**}] [**stx** *number*]

DESCRIPTION

The **config** command is used to modify or toggle all items relevant for working in *remote* operation mode.

The **config** parameters are closely related to the DBUG invocation line flags. When no parameters are specified, the current target system configuration is printed. The **config** command may be used both before and after the connect command has been issued.

load selects the communication protocol for loading programs to the target board. DBUG has two communication protocols for executable file loading. These are the binary protocol, in which binary data is sent to the monitor, or ASCII protocol, in which each byte is downloaded as two bytes of ASCII data. The binary protocol is appropriate when the communication line between the host and the target is sensitive to control characters that might otherwise be sent over the serial line (the default communication protocol is binary).

The **sp** option specifies the default stack pointer value when downloading program to the target system.

The **verbose** option is used to either enable or disable the verbose communication mode. In this mode, DBUG displays the messages exchanged by the debugger and the monitor or the Emulator. The command:

(dbug) `config verbose on`

may be used to enter the verbose communication mode, even if the **connect** command was previously issued.

Baud *number* sets the communication baud rate for *stand-aside mode*. Possible values are: 110, 150, 300, 600, 1200, 2400, 4800, 9600, 19200, 38400. The default is 9600. This parameter is ignored if the node option is selected.

Stx *number* changes the "stx" character used when communicating with development board and protocol B is in use. *number* is the ASCII value of the desired character. The monitor should be set to use the same character, by means of the mode control command. (See GNX Development Boards Monitor Manual.)

Cpu specifies the target board CPU name. Valid names are: 016, 032, 332, 532, gx32, cg16, gx320, fx16, cg160, fx164, am160 (fx16 and cg16 refer to the same monitor).

Mmu specifies the target board MMU name. Valid names are: 082, 382, onchip and nommu.

Fpu specifies the target board floating-point unit name. Valid names are: 081, 181, 381, 580, nofpu.

Mon specifies the version (variant level) of the monitor on the target board. Possible names are: 16, 32, 332, 332b, 532, cg16, gx32, gx32e, gx320, cg160, fx16 (fx16 and cg16 refer to the same monitor), fx16fax, fx164, cg160lx, am160 and cmon. cmon is a special monitor name that is accepted with any CPU, and allows you to use cmon specific commands.

The default values for the **cpu**, **mmu** and **fpu** options are described by your GNX target setup file *.gnxrc*, the previous **connect** command, or the invocation line.

EXAMPLES

(dbug) `config cpu cg16 mon cg16 fpu 381`

Specifies that the target board has an NS32CG16 CPU, a MONCG16, and an NS32381 FPU.

(dbug) `config verbose on`

Specifies that all message exchanges between DBUG and the monitor on the target system are displayed (*stand-aside remote mode*). The load protocol will be binary, which is the default.

(dbug) `config`

Prints the current target system configuration.

SEE ALSO

begin, load, connect

5.1.9 CONNECT - connect to a remote target system

SYNTAX

connect [**link** *linkname* | **node** *nodename*] [**with** {[**baud** *number*] [**cpu** *name*] [**mmu** *name*] [**fpu** *name*] [**mon** *name*] [**nofast**] [**list**] [**stx** *number*] }]

DESCRIPTION

When DBUG is initialized, no communciation link is defined. Therefore, you must select a link before issuing any debugger command that communicates with the target board.

The **connect** command is used to switch to the *remote* operation mode. **connect** selects the communications channel, through which DBUG and the target board communicate, and sets configuration parameters. The **connect** parameters are closely related to the DBUG invocation line flags. Several **connect** commands may be issued during one DBUG session. When **connect** is issued for the second time, it terminates the current connection and establishes a new connection. A similar version of this command is used to connect with an In-System Emulator (see Chapter 6 for description).

At least one parameter must be specified when the **with** clause is used.

link *linkname* identifies the serial communication line between the host and the target board. The default *linkname* is the last name given by the previous **connect** command, or as selected in the invocation line (**-l** parameter), or as specified in the *.gnxrc* file. The **link** parameter cannot be used if the **node** parameter is already given.

The **node** parameter selects the fast communication channel (LAN) between DBUG and the target development board. *nodename* is the name of the development board, as recognized by the host system. The default *nodename* is either the last name given by the previous **connect** command, the name selected in the invocation line, or the name specified in the .gnxrc file.

Nofast selects the communication protocol for loading programs to the target board. DBUG has two communication protocols for executable file loading. These are the binary protocol, in which binary data is sent to the monitor, or ASCII protocol, in which each byte is downloaded as two bytes of ASCII data.
The binary protocol is called the **fast** protocol (and is the default). **Nofast** option selects the ASCII protocol. This protocol is appropriate when the communication line between the host and the target is sensitive to control characters that might

otherwise be sent over the serial line.

The **list** option is used to enable the verbose communication mode. In this mode, DBUG displays the messages exchanged by the debugger and the monitor or the Emulator.

The **config** command is used to toggle between the fast/nofast and list parameters.

Baud *number* sets the communication baud rate for *stand-aside mode*. Possible values are: 110, 150, 300, 600, 1200, 2400, 4800, 9600, 19200, 38400. The default baud rate is 9600. This parameter is ignored if the node option is selected.

Stx *number* changes the "stx" character used when communicating with development board and protocol B is in use. *number* is the ASCII value of the desired character. The monitor should be set to use the same character, by means of the mode control command. (See GNX Development Boards Monitor Manual.)

Cpu specifies the target board CPU name. Valid names are: 016, 032, 332, 532, gx32, cg16, gx320, fx16, cg160, am160 and fx164.

Mmu specifies the target board MMU name. Valid names are: 082, 382, onchip, and **nommu**.

Fpu specifies the target board floating-point unit name. Valid names are: 081, 181, 381, 580, **nofpu**.

Mon specifies the version (variant level) of the monitor on the target board. Possible names are: 16, 32, 332, 332b, 532, cg16, gx32, gx32e, gx320, cg160, fx16, fx16fax, fx160, cg160lx, am160, fx164 and cmon. cmon is a special monitor name that allows you to use cmon specific commands.

The default values for the **cpu**, **mmu** and **fpu** options are described by your GNX target setup file *.gnxrc*, the previous **connect** command, or the invocation line.

Once the **connect** command has been issued for the first time, and the connection established, the **connect** command can be issued any number of times to toggle the **list** and **nofast** parameters. These must be specified each time you want them enabled. Issuing **connect** without these parameters will cause them to be disabled. They are different from other parameters, for which the previously specified value is in effect unless otherwise specified.

COMMENT: Although DBUG allows changing links during operation, this exercise is not recommended.

EXAMPLES

(dbug) `connect node db01 with cpu cg16 mon cg16 fpu 381`

Selects the board `db01` as the target and specifies that the communications with the target must be carried out via LAN.

(dbug) `connect link tty7`

Selects the channel `tty7`. The load protocol will be **fast** since the **nofast** parameter is not specified.

(dbug) `connect`

Reconnects the target system using the parameters selected by the previous `connect` and `config` commands.

(dbug) `alias do_connect (linkname) 'connect link linkname with baud 38400 cpu gx32 fpu 381'`

Defining an alias/macro makes the connect command easier to use. Issuing the following command selects `tty4` as your communication channel. This macro may be inserted in your *.dbuginit* file so that it will be recognized each time DBUG is entered.

(dbug) `do_connect (tty4)`

SEE ALSO

begin, load, config

5.1.10 CONT - continue program execution

SYNTAX

cont [*signal number* | *signal name*]

DESCRIPTION

Cont instructs DBUG to resume your program execution. The arguments are applicable only in the *native mode*. If an argument is supplied, the debugged program resumes execution as if it had received the specified signal. The signal name can be specified by its full name (SIGINT or sigint, etc) or signal number. The prefix, SIG, is optional.

EXAMPLES

(dbug) `cont`

continues program execution from where it was halted.

(dbug) `cont 2`

(dbug) `cont INT`

These two commands are equivalent. They send signal 2 to the program being debugged before execution is resumed.

SEE ALSO

catch, ignore, rerun, run

5.1.11 CONT UNTIL - continue execution until value in range

SYNTAX

cont until (*name* [**and** *mask*] [**not**] **in** *range*) [*/size*]

DESCRIPTION

Cont until instructs DBUG to resume execution of your program, and to stop execution if the value of the specified *name*, (variable or register), is in a specified range. **Cont until** is only applicable in *remote mode*, under the GNX monitor.

range is *expression..expression*. The expressions are evaluated before the debugger starts executing the command. Execution will stop when the value of the *variable*, or register, is within the evaluated range, when **in range** is specified, or when the value is out of this range when **not in range** is specified.

mask can be specified if not all of the bits of the variable or register are to be used. The keyword **and** can also be written as **&&**.

When evaluating the variable and the range, the debugger automatically determines whether a byte, word or double-word should be tested, according to the size of the specified variable. This can be changed using *size*. *size* is a single letter parameter which may take the following are values:

b - byte
x (or **d** or **o**) - word
X (or **D** or **O**) - double word.

EXAMPLES

(dbug) `cont until ( i in 1..3)`

The debugger will continue execution until `i` is greater than, or equal to, 1 and less than, or equal to, 3.

(dbug) `cont until ( x and 0x000000ff in a+3..b+4 )`

Execution will continue until the value of the right byte of `x` is between `a+3` and `b+4`. Both `a+3` and `b+4` are evaluated once, before the command is

execution.

(dbug) `cont until ( $r0  not in -17..i+3) /b`

Execution will stop if the value of the most significant byte of `$r0` is outside the range `(-17, i+3)`.

5.1.12 DELETE - delete breakpoint and trace events

SYNTAX

delete *number* [*number ...*]

DESCRIPTION

Delete removes the specified events (e.g. breakpoint, trace) from the active events list. *Number* is the number assigned by the **stop**, **stopi**, **trace**, or **tracei** commands, as displayed by the **status** command.

Delete can handle several arguments in one call. The arguments must be separated by spaces or tabs.

EXAMPLES

(dbug) `delete 3`

This command deletes event number 3.

SEE ALSO

clear, status, stop, stopi, trace, tracei

5.1.13 DELMENU - removes a menu entry

SYNTAX

delmenu *entry*

DESCRIPTION

This command removes the specified entry from the USER menu.

EXAMPLE

(dbug) `delmenu where`

Deletes the `where` entry from the user menu. The last added entry is deleted if more than one entry with the same name is present.

SEE ALSO

addmenu

LIMITATION

Not supported in MS-DOS.

5.1.14 DOWN - move down in call stack

SYNTAX

down [*n*]

DESCRIPTION

The **down** command moves the *current procedure* down the stack *n* levels.

If *n* is not specified, the default is 1 level. This command is intended for viewing purposes. It does not change the debugged program's execution state.

EXAMPLES

(dbug) `where`

```
fact.fact(n = 5), line 17 in "fact.c"
fact.fact(n = 6), line 19 in "fact.c"
fact.fact(n = 7), line 19 in "fact.c"
main(argc = 2, argv = 0x1ffffd3c), line 11 in "fact.c"
```

(dbug) `up 2`

```
fact.fact(n = 7), line 19 in "fact.c"
```

(dbug) `down`

```
fact.fact(n = 6), line 19 in "fact.c"
```

The `where` command displays the program call stack.
The `up` command moves the current procedure up 2 levels, within the stack.
The `down` command moves the current procedure down 1 level within the stack.

SEE ALSO

env, file, func, up, where

5.1.15 DUMP - dump procedure variables

SYNTAX

dump [*procedure*]

DESCRIPTION

The **dump** command prints the names and values of variables in the selected *procedure*, or the *current procedure*, if no procedure is specified. If the selected *procedure* is a period (.), then all active variables of the program are dumped.

The *procedure* must be an active procedure in the calling stack. If there are several occurrences of *procedure* in the stack, use the **up** and **down** commands to locate the desired occurrence and then use the **dump** command.

EXAMPLE

The following example illustrates the dump command:

(dbug) `where`

```
proc(a = 1, b = 2, c = 4), line 65 in "testfile.c"
main(argc = 1, argv = 0x1ffffd58), line 58 in "test.c"
```

(dbug) `dump`

```
proc(a = 1, b = 2, c = 4), line 65 in "test.c"
i = 10
j = 4306052
```

SEE ALSO

env, func, print, where, file

5.1.16 ENV - restore the environment

SYNTAX

env

DESCRIPTION

DBUG preserves the environment (*current file* and *current procedure*) when the program execution stops. If you make changes in the environment using the **up**, **down**, **file**, or **func** commands, you may use the **env** command to restore the environment to what it was.

EXAMPLES

(dbug) `env`

```
current procedure is bubble_sort
current file is bubble.c
```

SEE ALSO

down, file, func, up, where

5.1.17 FILE - change current file

SYNTAX

file [*filename*]

DESCRIPTION

The **file** command changes the current source file name to *filename*. The current source file is used as a default file name for referring to line numbers in DBUG commands and listing source lines.

The **file** command changes the contents of the code window to the selected file. When DBUG operates in partial symbolic mode, the **file** command also loads the symbolic information for the module that was created from the source of *filename*.

If no *filename* is specified, the **file** command prints information about all the modules in the executable file.

EXAMPLES

(dbug) `file bubble.c`

(dbug) `file`

```
    Name   Address   Language   Symbols
1   main   0xe010        C         yes
2   eval   0xe100        C         yes
3   sbrk   0xe210      ASM          no
```

SEE ALSO

down, env, func, up

5.1.18 FUNC - change current procedure

SYNTAX

func [*procedure*]

DESCRIPTION

The **func** command changes the current procedure. If no *procedure* is specified, the current procedure name is printed.

Changing the current procedure implicitly changes the current source file to the file that contains the procedure; it also changes the current scope used for name resolution. In the partial symbolic mode, **func** loads the symbolic information of its module.

The **func** command changes only the default path for symbol name interpretations and default source locations. It does not affect the program state.

The **func** command changes the contents of the code window, which displays the code associated with the selected function.

EXAMPLE

(dbug) `func get_a_charl`

SEE ALSO

down, env, file, up, where

5.1.19 HELP - explain dbug commands

SYNTAX

help

help e

help c

help k[**eys**]

help i[**nterface**]

help *command*

help n

DESCRIPTION

This command displays a short explanation of the DBUG commands. The help information is displayed in the HELP window. (It pops up and may be removed by the *reset* key).

The **keys** parameter is used to display the definitions of function keys. The **interface** parameter is used to display window manipulation commands. The keys and interface parameters may be abbreviated to **k** and **i**, respectively.

If a command name is specified, information on the command is printed.

help n specifies the new features supported in DBUG Revision 4.4.

help e describes the command-line edit function.

help c provides a list of the most common DBUG commands.

LIMITATION

help k[**eys**], **help e** and **help i**[**nterface**] are not supported in MS-DOS.

5.1.20 KDEFINE - bind function key to command

SYNTAX

kdefine [**<text>** | **<expr>** | **<line>**] {*command* | *"string"*} [*pfkey*]

DESCRIPTION

The **kdefine** command binds a function key to a DBUG command, alias, or string.

The DBUG command is then executed each time the key is pressed. Previous definition's are overwritten by the **kdefine** command. Keys defined in this way cannot be preceded by a *color* (gold, blue) key specification. To see the updated key binding, use the **help keys** command.

Pfkey is the function key to which the command is bound. *Pfkey* may be specified directly (see Chapter 3 - "Function Keys"). This is useful in command files. If *pfkey* is missing, DBUG will prompt you to press a function key. This command may be combined with text selection.

<text> Interpret the selection literally. Available only in graphic mode.

<expr> Expand the selected material to include the longest string of connected alphanumeric characters or underscores. Like **<text>**, this selection is interpreted literally. The difference is that you need only to select a fraction of the required text, and the system automatically expands the rest to include contiguous text of alphanumerics and underscores. This option is available only in graphic mode.

<line> In the graphic mode, interpret the selection as the line number containing the selected text. In the alphanumeric mode, interpret the selection as the line number along which the cursor is aligned.

EXAMPLES

(dbug) `kdefine help`

Entering this command on the command line generates a prompt which asks you to press a function key. This key will subsequently perform the `help` function. Pressing <ctrl/h> binds the <ctrl/h> key to the `help` function.

(dbug) `kdefine <line> clear <blue> <kp8>`

Having entered this command, you need only press the <blue> key and then <kp8> in order to perform a breakpoint clear of the selected line.

(dbug) kdefine "print var1" <ctrl> <p>

This command (typed exactly as shown) defines the function key <ctrl/p> to perform print var1.

(dbug) kdefine <line> "stop at"

Entering this command causes a prompt; pressing a function key defines a function key that sets a breakpoint in the selected source line. Combining *clear* and *set* breakpoints may be useful in both graphic and alphanumeric mode. Source line in an alphanumeric terminal environment can be selected by moving the cursor to the CODE window and placing it on the desired line.

SEE ALSO

kreset

LIMITATION

Not supported in MS-DOS.

5.1.21 KRESET - reset function keys to default

SYNTAX

kreset

DESCRIPTION

The **kreset** command resets the keys to their initial default definitions.

SEE ALSO

kdefine

LIMITATION

Not supported in MS-DOS.

5.1.22 LIST - print source code lines

SYNTAX

list [*procedure*]

list *fromline*[,*toline*] [**i**]

DESCRIPTION

The **list** command is used to view a range of source lines in the currently active source file (*current file)*, or a given procedure.

If the *toline* is specified, its value must be greater than or equal to *fromline.*

If only *fromline* is specified, a window of 8 lines, starting at *fromline* is displayed. If **list** is called with no parameters, the listing continues from the last displayed line, plus one.

If a *procedure* is specified, a window is formed around the procedure declaration.

If the **i** parameter is specified, the source lines are listed with embedded disassembly for the source lines.

The output of the **list** command is displayed in the DIALOG window.

EXAMPLE

DBUG `list 2`

Lists lines 2 through 9 in the current file.

(dbug) `list 1,10`

Lists lines 1 to 10 in the current file.

(dbug) `list`

Lists lines 11 to 18 (the last **list** command stopped at line 10).

(dbug) `list 25,27 i`

Lists lines 25 to 27 with each line followed by the corresponding assembly code.

SEE ALSO

wgo, print

5.1.23 LOAD - load program to a target system

SYNTAX

load [*objfile*] [**with** {[**nocode**] [**nodata**] [**protect**] [**zerofill**] [**sp** *address*]}]

DESCRIPTION

The **load** command downloads *objfile* and sets up the environment for executing the *objfile*. In *remote mode*, *objfile* is loaded to the target board memory. This command cannot be used in *native mode*.

When working with the emulator you can specify which portions of the *objfile* to load to the target board memory, and which portions to load to the emulator memory. The **map** command maps memory prior to using the **load** command. All memory is assumed to be on the target board by default.

The following registers must be set to initialize the environment for executing the *objfile*:

$pc - points to the program's starting address.

$sb - is set to the value of the static base entry in the main module table.

$mod - points to the entry of the main module in the global module table.

$sp - holds the address specified by the **sp** parameter or the default value: 0x3ffffc, if **sp** was not specified.

$fp - is set to 0 (end of call chain indicator).

$intbase - is set to the address of the monitor exception table.

$psr - is set to 0x300 in preparation for execution in user mode.

In *remote mode*, DBUG sets these registers when the **load** command is issued. When working with the emulator, DBUG sets only the *psr*, *pc*, *sp* and *fp* registers. The *intbase*, *mod* and *sb* registers are set to the correct values when DBUG configures the emulator's foreground monitor immediately after establishing connection with the emulator. You may change the value of the intbase register, if necessary, before you issue the **run** command to start your program. The dispatch table will be updated so that it will point to the TRC trap and BPT trap handlers in the emulator's foreground monitor before execution begins.

The **load** command also fills the bss (uninitialized data) area with zeros. The **run** command is required to start program execution, after loading.

Objfile is the executable file to be loaded. The default is the last specified name from the previous **load** command or from the invocation line. Default assumption if name is not specified is `a.out`.

The **nocode** parameter specifies that the *objfile*'s code segment is not loaded.

The **nodata** parameter specifies that the data section not be loaded. The *pc*, *sp*, *fp* and *psr* registers are modified in this case, as described above.

The **protect** parameter is not valid when working with the emulator. When **protect** is specified, standard protection is established. The default is no protection. Standard protection provides for read-only access to the code segment, and read/write access to the data and stack. The **protect** option is only effective when an MMU is present on the target board. The *ds* bit in the *$mmcr* register determines whether the *$ptb0* or *$ptb1* is used (default is *$ptb0*). For table addresses, see *Development Board Monitor Reference Manual* - Installation and Protocol. The **protect** parameter is ignored if the **nocode** and **nodata** parameters are specified.

When **zerofill** is specified, the general registers *$r0* through *$r7* are set to zero. Registers *$f0* through *$f7* are set to zero if an FPU is present and has been specified.

The **sp** option sets the end address of the stack to *address*. **Load** is normally only used with this parameter. The default value 0x3ffffc is assumed if *sp* is not specified. The **sp** value may also be specified by the **config** command.

A **load** command may be issued any number of times during a debugging session.

A **load** command deletes all breakpoints and events unless the **nocode** option is specified.

EXAMPLES

(dbug) `load sample with sp 0x1fff0`

Loads the executable file `sample`, and sets *$sp* to `0x1fff0`.

(dbug) `load a32.out with protect`

Loads the executable file `a32.out` and initializes the *$pc, $mod, $sb, $psr*, and *$sp* registers. The `protect` option enables standard protection. This option is not valid when working with the emulator.

(dbug) `load with nocode`

Operates on the executable file specified by the previous **load** command. The `nocode` option suppresses the reloading of a code segment of the file.

SEE ALSO

connect, protect, begin, map, run

5.1.24 LOG - log a program to the log file

SYNTAX

log [*logfile*]

log [*logfile*] **with append**

log [*logfile*] **with save**

log [*logfile*] **with full**

log [*logfile*] **with full append**

DESCRIPTION

The **log** command starts logging the debugging session on the *logfile*. The default file for *logfile* is *dbug.log*.

DBUG records all the commands and responses of the debugging session in a temporary file. You can save the complete recording of a session when quitting DBUG (the **quit** command). If you save the session via the **log** command, the recording starts from the point where the **log** command is invoked.

Save specifies that logging to this log file must stop, and *logfile* is closed. With this option you record a portion of the debugging session. You can return to that portion by replaying the *logfile* using the **source** command. *logfile* is immediately closed after this option is specified. If *logfile* is specified, it must be the current log file name.

Full specifies that both the DBUG commands and the debugger responses to the commands are recorded. Responses are recorded as comments and do not prevent replaying the file. To record only the commands, omit the **Full** option.

Append specifies that if a *logfile* with the same name exists, the new text is appended to the end of the existing *logfile*. Otherwise, *logfile* is overwritten if it exists.

Only one active *logfile* at a time is permitted.

EXAMPLES

(dbug) `log hist1 with full`

Logs the full session in the file `hist1`.

(dbug) `log with save`

This command saves the current session in the currently open log file, and then closes the file.

(dbug) `log`

Starts logging the debugging session on the file `dbug.log`.

SEE ALSO

quit, source

5.1.25 NEXT and NEXTI - execute one line/instruction

SYNTAX

next

nexti

DESCRIPTION

The **next** command executes the current source-line and breaks execution when the next source line is reached. If the current line contains a call to a procedure or function, the call is executed as part of the source line, without breaking the execution.

The **nexti** command executes one machine-level instruction instead of a source-line level instruction. All branch-subroutine machine instructions (*bsr, jsr, cxp, cxpd*) are performed and the execution breaks on the instruction following the branch subroutine instruction in the source.

The **next** and **nexti** commands are sensitive to active breakpoints, and when a breakpoint is reached, program execution halts.

SEE ALSO

step, stepi

5.1.26 PCPU PMMU ... - print all registers

SYNTAX

pcpu

pfpu

pmmu

pbpu

pdma

picu

ptimer

pcomplex [*int, int*]

DESCRIPTION

The **pcpu** command prints the contents of CPU registers *r0* through *r7, fp, sp, pc, psr* and *mod.*

The **pmmu** command prints the memory management unit (MMU) registers.

The dumped registers depend on the type of MMU (32082, 32382, or onchip). This command is only available in *remote mode.*

The **pfpu** command prints the floating-point unit (FPU) registers.

The dumped registers depend on the FPU type (32081, 32381, 32181 or 32580). This command is available only in *remote mode.*

The **pbpu** command is available only in *remote mode.* It prints the contents of all on-chip BPU registers for the NS32CG160. These registers are the *bpcr, bprmr, bplmr, bpmr, bfsr* and the *bpcntr* registers. This option is valid only when working with the NS32CG160 CPU.

The **picu** command is available only in *remote mode.* It prints the contents of the on-chip interrupt control unit (ICU) registers of the NS32CG160 and the NS32GX320 CPUs. These registers are the *ivct* and the *isrv* registers. This option is valid only when working with the NS32CG160 or the NS32GX320

CPUs.

The **pdma** command is available only in *remote mode*. It prints the contents of the on-chip DMA registers of the NS32CG160 and the NS32GX320 CPUs. For the NS32CG160 CPU, the printed registers are *adc0, adc1, adr0, adr1, bltc0, bltc1, bltr0, bltr1, mode0, mode1, ctl0, ctl1, stat, iereg, dstat.* For the NS32GX320 CPU, the registers will include in addition the registers *adcb0, adcb1, adrb0, adrb1.* This option is valid only when working with the NS32CG160 or the NS32GX320 CPUs.

The **ptimer** command is available only in *remote mode*. It prints the contents of the on-chip timer registers of the NS32CG160 and the NS32GX320 CPUs. The printed registers are *tc0, tc1, tc2, tcra0, tcra1, tcra2, tcrb0, tcrb1, tcrb2, tcntl0, tcntl1, tcntl2*. This option is valid only when working with the NS32CG160 or the NS32GX320 CPUs.

The **pcomplex** command is available only in *remote mode*. It prints the contents of the on-chip complex-multiplier registers of the NS32FX16 CPU. The printed registers are *y0, y1, a0, a1, dptr0, dptr1, cptr, ctl, st*. If a range is specified (e.g. pcomplex 0,95), the registers *$c0*, through *$c95* are also displayed. Any partial range within 0,95 may be specified.

SEE ALSO

print

5.1.27 PRINT - print variables and expressions

SYNTAX

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

DESCRIPTION

The **print** command computes the given expressions, and prints their values.

You may use general types of expressions as part of a command. The common subset of expressions that are legal for Pascal, Modula-2 and C programming languages are supported (see Section 4.3.10, "Expressions").

When printing variables, the debugger automatically selects the output format according to the variable type. This can be changed by the *radix* parameter, which specifies the output format. The *radix* is a single letter parameter. Following are legal values for the *radix.*

d - print a short word as an unsigned decimal
D - print a long word in decimal
o - print a short word in octal
O - print a long word in octal
x - print a short word in hexadecimal
X - print a long word in hexadecimal
b - print a byte in hexadecimal
c - print a byte as a character
s - print a string of characters terminated by a null byte
f - print a single-precision real number
g - print a double-precision real number
B - print binary (assembly level only)

ASSEMBLY-LEVEL PRINTING

In addition to the high-level **print** command, DBUG supports assembly-level printing. The formats for assembly-level print instructions are:

address , *address*/ [*radix*] [> *file*]

address / [*count*] [*radix*] [> *file*]

DESCRIPTION

Assembly-level print commands print memory contents starting at the first *address* and continuing up to the second *address* or until *count* items are printed. The output may be redirected to a file.

If the address is **.** (dot), the address following the address most recently printed is used. The *radix* specifies how memory is to be printed (as in **print** command). If the *radix* is omitted, the *radix* of the previously specified mode is used. The initial radix is **X**.

Assembly-level printing disassembles memory locations, using the special radix **i**. The **list** command may also be used for disassembling source lines.

Symbolic addresses for machine-level printing are specified by placing the prefix & (ampersand) before the symbol.

The **B** radix instructs the debugger to print in binary mode. When this radix is specified, you must print only to a file.

EXAMPLES

Print the contents of the array named `numbers`.

```
(dbug) print numbers
(2, 4, 17, 13, 7, 5, 2, 6, 9, 15)


(dbug) print numbers x
(0002, 0004, 0011, 000d, 0007, 0005, 0002, 0006, 0009, 000f)
```

Print a member of two-dimensional array of integer:

```
(dbug) print two_d_array[2][4]
12
```

Print expression value:

```
(dbug) print "average=" (total^.number + total^.nxt^.number)/2.0
average= 17.7
```

Print boolean expression value:

```
(dbug) print (i>=0) or (i<0)
true
```

Print structure variable:

```
(dbug) whatis strvar
struct {
    char        *name;
    int         age;
    float       salary;
    char        rank;
} strvar;
(dbug) print strvar
(name = "john", age = 35, salary = 1234.56, rank = 'b')

(dbug) print $fp x
1ffffd2c
```

EXAMPLES (assembly-level):

To disassemble 6 instructions of `main` procedure of the `bubble.c` program, use

```
(dbug) &main/6 i
main    :       enter   [], 0x4
main+3  :       movd    $0x400f60, tos
main+9  :       bsr     printf
main+e  :       adjspb  $-0x4
main+11 :       movqd   0x0, -0x4(fp)
main+14 :       cmpd    $0xa, -0x4(fp)
```

To disassemble 4 more instructions use:

```
(dbug) ./4 i
main+1b :       ble     main+47
main+20 :       movd    -0x4(fp), r1
main+23 :       movd    numbers[r1:d], tos
main+2a :       movd    $0x400f7f, tos
```

To print the contents of the memory location pointed to by the frame-pointer register (FP) use:

```
(dbug) &$fp/X
1ffffd2c:  00000000
```

To print the contents of register relative address (10 double words starting from 4(r0)):

```
(dbug) &$r0+4/10 X
1ffffd4c:  1ffffdb4 1ffffdd2 1ffffde1 1ffffdfc
1ffffd5c:  1ffffe18 1ffffe21 1ffffe2f 1ffffe44
1ffffd6c:  1fffff83 1fffffad
```

SEE ALSO

list

5.1.28 PROTECT - set memory protection

SYNTAX

protect *address range* [{**read** | **write**} **for** {**u** | **s**}] [{**set** | **clear**} {[**v**] [**r**] [**m**]}] [**start** *address*] [**on primary**] [**using** {**ptb0** | **ptb1**}]

DESCRIPTION

Protect creates protection/translation by writing to the protection and the translation fields in the primary and secondary page tables.

Protect is allowed only in the *remote mode*, and if the target board includes an MMU. The **protect** command defines the following parameters:

Address range is the address range to be protected. It is rounded outward to include page boundaries.

{**read** | **write**} **for** {**u** | **s**} specifies the protection level for user (u) or for supervisor (s).

{**set** | **clear**} {[**v**] [**r**] [**m**]} specifies which bit should be cleared or set. The bits are: **v** for valid, **r** for referenced, and **m** for modified. If **set** or **clear** are specified, then **v**, **r** or **m** must also be specified.

start *address* specifies the address for page translation.

on primary specifies operation on the primary page table. Otherwise, it writes to the secondary page table(s).

using {**ptb0** | **ptb1**} specifies which page table register is used. The default is PTB1.

The **protect** command uses the current contents of either the PTB0 or PTB1 register, according to the register specified in the **using** option to access the page tables. The command does not change the contents of the register.

To create user-defined protection, you must first use the **on primary** option, i.e., you must define the protection level, the bits that are valid, referenced, or modified, and specify the translation address of the primary page table before defining the secondary page tables.

If **protect** *address* is specified, the *address*'s protection is printed.

EXAMPLES

(dbug) `protect &num_of_loops clear v`

Clears the valid bit in the secondary page table for the page containing the address of the variable `num_of_loops`. All other fields remain unchanged.

(dbug) `protect 1024,2047 set r m on primary`

Sets the referenced and modified bits in the primary page table for the pages containing the address range 1024 through 2047.

(dbug) `protect 0,2000 set v start 10000`

Sets the valid bit and defines the translation address for the pages containing the address range `0` through `2000`. The translation table address begins at address `10000`. The address range `0,2000` is rounded outward (0 to 4k) to include page boundaries.

SEE ALSO

load

5.1.29 QUIT - terminate debugging session

SYNTAX

quit [**with save** [*filename*]]

DESCRIPTION

The **quit** command terminates the debugging session. You can save the complete session in a log file by specifying the **save** parameter.

If no file is specified, `dbug.save` (`dbug.sav` on MS-DOS) is assumed.

EXAMPLE

Quit the debugging session

(dbug) `quit`

Quit debugging session and save the history in a file named `history`

(dbug) `quit with save history`

Quit DBUG session and save the history in a file called `dbug.save`.

(dbug) `quit with save`

SEE ALSO

log

5.1.30 RETURN - return from current procedure

SYNTAX

return [*procedure*]

DESCRIPTION

The **return** command executes the program until the return from the *current procedure*. If *procedure* is specified, the program continues executing until the *procedure's* return is executed. The *procedure* must be on a calling stack (i.e. must be active), otherwise DBUG issues an error message.

EXAMPLE

(dbug) `where`

```
proc2(i=17), line 47 in file test.c
proc1(j=5, c='x'), line 85 in file test.c
main(argc=1, arg2=0x1ffffd5c), 23 in file test.c
```

(dbug) `return`

(dbug) `where`

```
proc1(j=5, c='x'), line 85 in file test.c
main(argc=1, arg2=0x1ffffd5c), 23 in file test.c
```

SEE ALSO

down, env, up, where

5.1.31 RUN and RERUN - run the loaded program

SYNTAX

run [*args*] [< *ifile*] [> *ofile*]

rerun [*args*] [< *ifile*] [> *ofile*]

DESCRIPTION

The **run** command starts the execution of the loaded program, passing the *args* as command line arguments. The parameter *ifile* can be specified for program input redirection (Unix shell style). *Ofile* is used for output redirection.

The **rerun** command with no parameters runs the same program again using the arguments from the last **run** or **rerun** command. The **rerun** command with parameters is equivalent to the last **run** or **rerun** command, appending the arguments of **rerun** to the previous arguments.

In *remote mode*, **rerun** is equivalent to the following sequence:

```
load with nocode zerofill
run
```

Args are applicable only in the *native mode*.

COMMENTS

In the *remote mode*, the **run** command should be issued only once for a loaded program. Subsequent **run** commands continue execution from the current PC. Use **rerun** to run the program again.

EXAMPLE

(dbug) `run /tmp/xxx < ifile`

runs the program being debugged with the single argument `/tmp/xxx`. The standard input file for the debugged program is `ifile`.

SEE ALSO

begin, cont, load

5.1.32 SEARCH - search for patterns in the source file

SYNTAX

/ regular expression

? *regular expression*

DESCRIPTION

The specified *regular expression* can be searched for in the forward direction using /, or the reverse (backward) direction using **?**.

DBUG searches from the current line to the end/start of the current file. If the search reaches the file start/end without finding the search pattern, the search continues cyclically (i.e., if DBUG reaches the end of file during forward search without finding *regular expression*, the search continues from the first line to the current line).

If no pattern (*regular expression*) is specified, the last pattern expressed is assumed, and the search continues for the next occurrence of the last pattern.

If **search** locates the requested *regular expression*, the CODE window is updated. The line where the *expression* is found is located at the top of the CODE window. Subsequent **search** commands resume searching from the top line of the CODE window.

EXAMPLES

(dbug) `?array1`

Searches the current source file for the pattern `array1`, from the current line (backwards) to the beginning of the file.

(dbug) `/main`

Searches the current source file from the present line to the end of the file for string `main`.

(dbug) /

Searches for the occurrence of the string `main` in the forward direction.

SEE ALSO

REGEXP (3) *UNIX Reference Manual*, **wgo, wscroll**

5.1.33 SET and UNSET - set dbug variables

SYNTAX

set

set *variable* [= *string*]

unset *variable*

DESCRIPTION

Set defines DBUG variables. If no parameter is given, **set** prints all DBUG set variables. With *variable* as the parameter, the debugger variable *variable* is defined. These variables are used as flags. If a variable name is specified as part of the DBUG command, it is substituted by *string*. DBUG has a predefined set of internal variables: *$hexchars, $hexints, $hexstrings, $unsafeassign, $filedisasm, $callproc, $checkstack* and *$newdisasm*.
Unset deletes the definition of a specified debugger variable.

$hexchars - If set, DBUG prints characters in hexadecimal format

$hexints - If set, DBUG prints integers in hexadecimal format

$hexstrings - Normally, if you ask to print a pointer to character, DBUG prints the character string pointed to by the pointer. If *$hexstrings* is set, DBUG prints the hexadecimal value of the pointer.

$unsafeassign - If set, DBUG suppresses the type compatibility check when assigning values to variables. DBUG also refuses to perform the assignment if the operand sizes of the source and destination are unequal.

$filedisasm - If set, DBUG uses the data from the object file for disassembly, instead of memory.

$callproc - If set, a procedure can be excecuted from debugger level using the call command.

$checkstack - If set, checks the stack's consistency during execution. The debugger verifies that:
All saved pc values point to text area.
All saved fp values point to stack area.
Each saved fp value is smaller than its predecessor.
Last saved fp value is 0.
A warning is issued when one of these conditions is not met.

$newdisasm - If set, extended disassembly format is used. The command address is printed as an absolute hexadecimal value. Negative numbers are printed with a '-' sign.

SEE ALSO

assign

5.1.34 SOURCE - execute command file

SYNTAX

source *filename*

DESCRIPTION

The **source** command executes the debugger commands in *filename*.

The command files may be nested. The nesting level is limited by the host operating system. DBUG does not echo the commands in the command file. The **source** command is considered as one DBUG command, i.e. only the command output, and not the commands in the *filename*, are echoed.

SEE ALSO

log

5.1.35 STATUS - list active breakpoints and traces

SYNTAX

status

DESCRIPTION

The **status** command displays the currently active **trace, tracei, stop**, and **stopi** commands, and their event number. The numbering is not consecutive. Other commands, such as **delete**, may reference the event by their assigned event numbers.

EXAMPLE

Suppose that, during the debug session of the `bubble.c` file, you issued the following stop and trace commands:

```
stop in main
stop at 13
trace i in main
stop in main if (i > 10)
```

The status command will print:

(dbug) `status`

```
[1] stop in main
[2] stop at "bubble.c":13
[3] trace bubble.main.i in main
[4] stop if bubble.main.i > 10 in main
```

SEE ALSO

clear, delete, trace, tracei, step, stepi

5.1.36 STEP and STEPI - step over one line/instruction

Step - step over current source line

Stepi - step over current machine instruction

SYNTAX

step

stepi

DESCRIPTION

Step executes one source line. If the "stepped" source line contains a call to a procedure, and the called procedure has source line information (compiled with the **-g** option on UNIX, or the **/debug** option on VMS), DBUG stops in the called procedure.

Stepi steps over one machine instruction. If the instruction being stepped is the branch-subroutine instruction, DBUG steps into the called subroutine.

The **step** command is sensitive to active breakpoints. When a breakpoint is detected, the user is notified.

SEE ALSO

next, nexti

5.1.37 STOP - set breakpoints (source level)

SYNTAX

stop if *condition*

stop at *source-line-number* [**if** *condition*]

stop in *procedure* [**if** *condition*]

stop *variable* [**if** *condition*]

DESCRIPTION

The **stop** command allows you to control target program execution by setting breakpoints. Breakpoints may be set for a *condition*, at a selected *source-line-number*, on entry to a *procedure (function)*, or on a change of a *variable*.

In stop if, the execution continues until the specified condition becomes true.

Stop at sets a breakpoint at the specified source line, in the specified source file. The execution stops before the specified line.

If the **stop in** *procedure* is specified, program execution halts before the first line of *procedure* is executed. If *condition* is specified, execution stops any time the condition is true and *procedure* is a current procedure. Note that global symbols of an assembler module are regarded by DBUG as procedures; thus **stop in** can be applied to them.

If **stop** *variable* is specified, the debugged program breaks any time the *variable*'s value is changed. If the **in** *procedure* is not specified, then the **stop** *variable* command remains in effect as long as the procedure/scope containing the variable is active.

If *condition* is specified, it is checked prior to any action (even sampling the initial value of *variable*).

The *condition* is any valid debugger expression. If *condition* is specified, the execution breaks only if the *condition* evaluation yields true.

In *remote mode*, the number of the defined breakpoints is limited to 16. No such limitation is imposed in *native mode*.

Each **stop** command is assigned an event number. The *event number* is an identifier for the **status** and **delete** commands.

Each **stop** command is assigned an event number. The *event number* is an identifier for the **status** and **delete** commands.

NOTE

Specifying *condition* in the **stop** command causes the program to execute single step. This reduces program execution speed significantly.

EXAMPLES

(dbug) `stop at 15`

Stop at line `15` in the current file.

(dbug) `stop at "bubble.c":17`

Set a breakpoint at line `17` of file `bubble.c`.

(dbug) `stop in bubble_sort`

Stop every time the procedure `bubble_sort` is entered.

(dbug) `stop i`

Stop every time the variable `i` changes its value.

(dbug) `stop in hash if (val > 17) and (val < 32)`

Stop in procedure `hash` if integer variable `val`'s value is in the range: 18-31. Note that the condition is checked for each instruction as long as `hash` is active.

FAST MONITOR-LEVEL CONDITIONAL STOP

In addition to the high-level **stop** command, DBUG supports a monitor-level **stop** command. The formats for the monitor-level **stop** command are:

stop at *line* / *count*

stop at *line* (**if** *name* [**and** *mask*] [**not**] **in** *range*) [/ [*size*] *count*]

DESCRIPTION

Unlike the other conditional breakpoints, the monitor-level conditional **stop** command is executed entirely by the GNX "CMON" monitor code (version 4.4 and up). The program will be stopped, at the specified line when the specified condition is fulfilled. *count* indicates the number of times the breakpoint condition must be true before the program is stopped. *name* can be an integer variable or a register.

range is: *integer-expression..integer-expression*

mask can be used to specify which bits of the variable or register are to be evaluated.

The range check is performed by the debugger according to the type and size of the specified variable. This can be changed using *size*. *size* is a single letter parameter which may take the following values:

b - byte
x (or **d** or **o**) - word
X (or **D** or **O**) - double word

By default the debugger will stop whenever the variable, or the register, is within the range. If *count* is specified, the program execution will stop, when the debugger reaches the specified line, after the value has been within the specified range *count* times. The keyword **and** can also be written as **&&**.

EXAMPLES

(dbug) `stop at 15/5`

Execution will stop after line 15 has been reached five times.

(dbug) `stop at 10 (if i in 1..3)`

The execution will stop at line 10 only if $1 \le i \le 3$.

(dbug) `stop at 123 (if $r0 not in 0..0)`

Execution will stop at line 123 only if the value of the register `r0` is non zero.

(dbug) `stop at 150 (if b in a+1..c*25)/5`

Execution will stop when, at line 150, the value of `b` is greater than, or equal to, `a+1` and less than, or equal to, `c*25`, for the fifth time and for each successive time. Both `a+1` and `c*25` are evaluated when the breakpoint is set and are not affected by changes made during execution.

(dbug) `stop at 65 ( if x in 0x35..0x45) /b`

In this case only the least significant byte of `x` will be compared to the specified range.

SEE ALSO

clear, delete, status, stopi, where, trace, tracei

GNX Version 4.4 Development Board Monitor Reference Manual.

5.1.38 STOPI - set breakpoints (machine level)

SYNTAX

stopi [**write**] *address*

stopi *address* [**if** *condition*]

stopi at *address* [**if** *condition*]

DESCRIPTION

The **stopi** command sets a breakpoint upon different accesses to *address*. This is the machine-level equivalent of the **stop** command.

If **write** is specified, program execution halts when the specified address is written. This option activates CPU debugging features (*remote mode* only). If **stopi** *address* is specified, execution is stopped when the value of the specified address is changed. If **stopi at** *address* is specified, execution is stopped before executing the specified address.

If *condition* is specified, it is checked prior to any action.

Upon reaching a breakpoint, the definition of the **stopi** command that caused the breakpoint is printed.

The number of the defined breakpoints in the *remote mode* is limited to 16 (including the **stop** command).

Each **stopi** command has an event number as an identifier for the **status** and **delete** commands.

COMMENTS

The **write** option uses Series 32000 CPU debug features. Therefore, only one breakpoint for data may be set with the NS32532 or NS32GX32 CPUs and two breakpoints for the NS32082 or NS32382 MMUs.

The **write** breakpoint option is not available for registers or variables that are allocated in registers.

EXAMPLES

(dbug) `stopi at 0x124`

Sets breakpoint at address `0x124`.

(dbug) `stopi write &i`

Stops on **write** to variable `i`.

FAST MONITOR-LEVEL CONDITIONAL STOPI

In addition to the high-level **stopi** command, DBUG supports a monitor-level **stopi** command. The formats for the monitor-level **stopi** command are:

stopi at *address / count*

stopi at *address* (**if** *name* [**and** *mask*] [**not**] **in** *range*) [/ [*size*] *count*]

DESCRIPTION

Unlike the other conditional breakpoints, the monitor-level conditional **stopi** command is executed entirely by the GNX "CMON" monitor code (version 4.4 and up). The program will be stopped, at the specified address when the specified condition is fulfilled.

In all other respects, the **stopi** command is similar to the **stop** command. See the **stop** command for a detailed description and examples.

SEE ALSO

clear, delete, status, stopi, tracei

GNX Version 4.4 Development Board Monitor Reference Manual.

5.1.39 TRACE and TRACEI - trace variables and execution

SYNTAX

trace [**if** *condition*]

trace {*line* | *procedure*} [**if** *condition*]

trace in *procedure* [**if** *condition*]

trace *variable* [**in** *procedure*] [**if** *condition*]

trace *expression* **at** *line* [**if** *condition*]

tracei in *procedure* [**if** *condition*]

tracei at *address* [**if** *condition*]

tracei *address* [**if** *condition*]

tracei [**if** *condition*]

DESCRIPTION

The **trace** command prints tracing information while the program is executed. Each trace command is identified by an event number for use by **status** and **delete** commands. **Trace** or **tracei** without parameters reports program progress for each line or machine instruction in the current scope.

The first argument describes what is to be traced. If it is a *line*, then the line is printed immediately before being executed. Source line numbers in a file other than the current one must be preceded by the name of the file in quotes and a colon, e.g. `"foo.c":17`.

If the argument is a procedure name, then whenever it is called, information is printed stating which routine called it and from what source line it was called.

If the argument is a variable name, then the variable's value is printed whenever it changes. Execution is substantially slower during this form of tracing.

The **trace in** *procedure* restricts tracing information to be printed only while executing inside the given procedure or function.

Trace at causes DBUG to print the instruction at *address* immediately before it is executed.

Condition is a boolean expression and is evaluated prior to executing the tracing information; if it is false then the information is not printed.

The **tracei** command is similar to **trace** except it operates on machine instruction level instead of source line level.

NOTE: Specifying *condition* for **trace** command implies single-step execution of DBUG. This results in a significant performance degradation.

SEE ALSO

delete, status, stop, stopi

5.1.40 UP - move up in call stack

SYNTAX

up [*n*]

DESCRIPTION

The **up** command moves the *current procedure* up the stack *n* levels. If *n* is not specified, the default is 1. This command is intended for viewing purposes. It does not change the execution state of the debugged program.

EXAMPLES

(dbug) `where`

```
fact.fact(n = 5), line 17 in "fact.c"
fact.fact(n = 6), line 19 in "fact.c"
fact.fact(n = 7), line 19 in "fact.c"
main(argc = 2, argv = 0x1ffffd3c), line 11 in "fact.c"
```

(dbug) `up 2`

```
fact.fact(n = 7), line 19 in "fact.c"
```

(dbug) `down`

```
fact.fact(n = 6), line 19 in "fact.c"
```

The `where` command displays the program call stack.
The `up 2` command moves the current procedure up 2 levels, within the stack.
The `down` command moves the current procedure down 1 level within the stack.

SEE ALSO

down, env, file, func, return, where

5.1.41 USE - set source search path

SYNTAX

use *dir* [*dir* ...]

DESCRIPTION

DBUG assumes that source files and the object file reside in the same directories. To override this assumption, the **use** command can select a group of search directories in which the source file is located. This command is related to the **-I** invocation line option, which can set an initial search path for source files.

The directory names must be separated by blanks.

EXAMPLE

(dbug) `use /usr/src/cmd /users/softw/john/src`

Defines `/usr/src/cmd` and `/users/softw/john/src` as the current search path for source information.

5.1.42 WDELETE - delete a window

SYNTAX

wdelete [*wname*]

DESCRIPTION

The **wdelete** command removes the window *wname*. The default window is the currently selected window.

DBUG keeps track of deleted windows and remembers where they were before deletion. Specifying **wdisplay** without specifying a location displays the window at the location it occupied before deletion.

EXAMPLES

(dbug) `wdelete code`

This command deletes the CODE window.

(dbug) `wdelete`

This command deletes the currently selected window.

SEE ALSO

wdisplay

LIMITATION

Not supported in MS-DOS.

5.1.43 WDISPLAY - display window

SYNTAX

wdisplay *wname* [**at** [*wloc*]]

DESCRIPTION

The **wdisplay** command creates the window *wname*, **at** the location *wloc*.

If the window is being displayed for the first time, and **at** *wloc* is not specified, then the default placement is used. However, if the window was previously displayed and then removed by **wdelete**, and **at** *wloc* is not specified, then the window is displayed at the position it occupied before deletion.

wname is the name of the window to be displayed.

wloc is the location where the window should be placed. There are two ways to specify window locations: 1) With a mouse. If *wloc* is not specified, you should move the mouse cursor to the points where you want the two diagonal corners of the window. These points determine placement and size of the window. To identify the points, move the mouse cursor to the desired location and press a mouse button. This method is available only in graphic mode.

2) Without a mouse. Give the explicit coordinates for locating the two diagonal corners of the window.
The syntax is:

(*row1*,*col1*),(*row2*,*col2*)

Where (*row1*,*col1*) defines the coordinates for one corner of the window, and (*row2*,*col2*) defines the coordinates for the opposite corner of the window. Both *row1* and *row2* must be integers and indicate a legal row number. Both *col1* and *col2* must be integers and indicate a legal column number. Graphic environments (with X-windows) interpret the row and column values as pixels, while alphanumeric environments interpret the row and column values as screen character positions.

EXAMPLES

(dbug) `wdisplay code at (100,200),(400,500)`

Creates and places the code window at the specified coordinates. These parameter values are only valid with a graphic environment.

(dbug) `wdisplay code at`

Displays the CODE window. DBUG prompts for two points to determine where and how large the window is to be. This is legal only in systems with a graphic interface.

(dbug) `wdisplay code`

Displays the CODE window. If this is the first time the window is displayed, it appears at its default location, otherwise it is placed in the last position it occupied before it was deleted.

SEE ALSO

wmove, wdelete

LIMITATION

Not supported in MS-DOS.

5.1.44 WGO - go to a line in file

SYNTAX

wgo *number*

DESCRIPTION

The **wgo** command places the source line *number* of the current file at the middle of the CODE window. This command is available only in source mode.

SEE ALSO

func, file, search, wscroll

LIMITATION

Not supported in MS-DOS.

5.1.45 WHATIS - describe a symbol

SYNTAX

whatis {*symbol* | *address*}

DESCRIPTION

The **whatis** *symbol* command prints the declaration of the specified symbol.

The symbol can be a variable, procedure, module, type definition, or type definition tag. Array members and record structure variables are not accepted as parameters.

The **whatis** *address* command prints the absolute and symbolic addresses of the specified *address*. If *address* is in the code area, the **whatis** command prints the address' symbolic pc.

EXAMPLES

(dbug) `whatis my_array`
`address = 0x51472 = 0x16(bss)  int my_array[10]`

Displays the absolute address 0x51472, offset 0x16 from the beginning of the bss area, and the type of the symbol `my_array`.

(dbug) `whatis 0x9000`
`address = 0x9000 = init.driver_init line 432`

Displays the absolute address 0x9000 and the symbolic pc `init.driver_init line 432` of the address 0x9000. This means that line 432 in module `init` is within procedure `driver_init` and mapped to the absolute address `0x9000`.

SEE ALSO

which, whereis

5.1.46 WHERE - print active call stack

SYNTAX

where

DESCRIPTION

The **where** command displays the currently active call stack. The following information is provided for each called stack:

procedure name

the procedure argument values

the line or address where the caller routine will resume after return.

EXAMPLE

(dbug) `where`

```
fact.fact(n = 5), line 17 in "fact.c"
fact.fact(n = 6), line 19 in "fact.c"
fact.fact(n = 7), line 19 in "fact.c"
main(argc = 2, argv = 0x1ffffd3c), line 11 in "fact.c"
start ( ) at 0x012a
```

SEE ALSO

up, down, env

5.1.47 WHEREIS - find all occurrences of a symbol

SYNTAX

whereis *symbol*

DESCRIPTION

The **whereis** command displays the qualified names of all the occurrences of *symbol*. Enumeration constants or union/structure fields are not displayed.

EXAMPLES

(dbug) `whereis i`

```
main.i bubble_sort.i
```

SEE ALSO

whatis, which

5.1.48 WHICH - print symbol qualifier

SYNTAX

which *symbol*

DESCRIPTION

The **which** command displays a qualified name for the currently active *symbol*.

EXAMPLE

```
(dbug) print i
12

(dbug) which i
bubble.bubble_sort.i
```

SEE ALSO

whatis, which

5.1.49 WMOVE - move or resize window

SYNTAX

wmove [*wname*] *wcorner wshift*

DESCRIPTION

The **wmove** command enables you to move a window to a different location or to resize it.

If *wcorner* is not specified, or equals **v**, the window is moved. Otherwise the command stretches (resizes) the window according to the specified *wcorner* and *wshift*.

wname is the name of the window to move.

The selected window is moved when *wname* is not specified.

wcorner specifies one or more corners on the selected window. The specification changes the size of the selected window by stretching it from the identified corner(s). This specification must be used in conjunction with the *wshift* specification. Specifying all four corners will move the entire window.

The format for *wcorner* specification is:

[v[u | d][l | r]]

Where:
vu - indicates the upper two corners.
vd - indicates the lower two corners.
vl - indicates the left two corners.
vr - indicates the right two corners.
vul - indicates the upper left corner.
vur - indicates the upper right corner.
vdl - indicates the down left corner.
vdr - indicates the down right corner.
v (or empty) - indicates all corners.

wshift - specifies shift movement.

There are two ways to enter the shift specification:

1) With a mouse. Click a mouse button at two points on the screen: the first click specifies a point in a window, while the second click specifies the position of the same point at the new location of the window. The distance between the two points is automatically translated to the corresponding shift parameter. This method is available only in graphic mode.

2) Without a mouse. The shift specification is entered alphanumerically in the following format:

[{**u** | **d**}*number*] [{**l** | **r**}*number*]

Where:
u - indicates up shift.
d - indicates down shift.
l - indicates left shift.
r - indicates right shift.
number is a positive number, specified in pixels for graphic systems, and characters for alphanumeric systems.

NOTE: This command's numeric parameters are interpreted as pixels in the graphic environment, and as lines and columns in the alphanumeric terminal environment.

EXAMPLES

(dbug) `wmove dialog u5 r10`

Moves the Dialog window up 5 lines, and 10 columns to the right.

(dbug) `wmove vl l10`

Moves the left side of the selected window 10 columns to the left.

(dbug) `wmove vdr d10 l4`

Moves the selected window from its lower right-hand corner.

The window is moved down 10 lines and 4 columns to the left.

(dbug) `wmove code vu`

Moves the upper border of the code window. The shift is entered with the mouse. Only the vertical shift component is considered. This command is only legal in graphic mode.

(dbug) `wmove`

Moves the selected window according to the shift specified by the mouse.

This command is legal only in graphic mode.

SEE ALSO

wdisplay, wreset

LIMITATION

Not supported in MS-DOS.

5.1.50 WNEXT - select a window

SYNTAX

wnext [*wname*]

DESCRIPTION

The **wnext** command changes the selected window to *wname*.

If *wname* is not specified, the default window is the window that comes after (in a predefined cyclic order) the currently selected window.

The command's effect varies with the mode in which DBUG is operating. In graphic terminal environments the selected window is popped-up and its border is highlighted. In alphanumeric terminal environments, the selected window is popped-up and the cursor moves to it. (This is the only way to move the cursor between windows in alphanumeric mode.)

wname is a window name.

This command's default key definition is <ctrl/n>.

EXAMPLES

(dbug) `wnext trace`

Selects the trace window.

(dbug) `wnext`

Selects the next window as defined by a predefined cyclic order.

(dbug) `<ctrl/n>`

The next window becomes the selected window. In the alphanumeric terminal environment, the cursor is moved to the next window.

LIMITATION

Not supported in MS-DOS.

5.1.51 WPOP - pop window

SYNTAX

wpop [*wname*]

DESCRIPTION

The **wpop** command uncovers a specified window that is covered (either in part, or completely). If no window is specified, the selected window is popped-up.

wname is the name of the window to be popped-up.

EXAMPLES

(dbug) `wpop help`

If the HELP window is covered by another window, the `wpop help` command brings the HELP window to the top (i.e., no part of it is obstructed from view).

(dbug) `wpop`

Uncover the selected window.

SEE ALSO

wpush

LIMITATION

Not supported in MS-DOS.

5.1.52 WPUSH - push window

SYNTAX

wpush [*wname*]

DESCRIPTION

The **wpush** command covers the specified window beneath other displayed windows. The default *wname* is the selected window.

Wname is the name of the window to be pushed.

EXAMPLES

(dbug) `wpush help`

This command pushes the HELP window below any windows it currently sits on.

(dbug) `wpush`

Pushes the selected window.

SEE ALSO

wpop

LIMITATION

Not supported in MS-DOS.

5.1.53 WRESET - reset windows

SYNTAX

wreset

DESCRIPTION

The **wreset** command resets the display, by returning the currently displayed windows to their default sizes and locations.

SEE ALSO

wmove

LIMITATION

Not supported in MS-DOS.

5.1.54 WSCROLL - scroll window

SYNTAX

wscroll [*wname*] *wshift*

DESCRIPTION

The **wscroll** command exercises a vertical scroll on the contents of the specified window. The scroll size is determined by the vertical component of *wshift*.

wname is the name of the window to scroll.
Currently, **wscroll** works only with the CODE window, so CODE is the default window.

wshift specifies how much the window should be shifted. The shift specification is typed in the following format:

{**u** | **d**} [*number*]

Where:
u - indicates up shift.
d - indicates down shift.
number indicates number of lines to scroll. If omitted, default is one window size.

EXAMPLES

(dbug) `wscroll code d10`

Scrolls the contents of the CODE window down 10 lines.

(dbug) `wscroll u`

Scrolls the contents of the CODE window up one page.

SEE ALSO

search, wgo

LIMITATION

Not supported in MS-DOS.

Chapter 6

INTERFACE WITH EMULATORS

6.1 Introduction

DBUG provides an interface for remote debugging with in-system emulators. These emulators are: the HP64772 emulator for the NS32532 and NS32GX32 CPUs; the HP64778 emulator for the NS32GX320 CPU; the HP64779 emulator for the NS32CG16, NS32FX16, NS32FX164 and NS32CG160 CPUs; and the SPLICE Development Tool for the NS32CG16 CPU.
The interface includes commands supported by DBUG's *native* and *remote* modes. Additional features support the capabilities of the Series HP64000 In-System Emulators such as real time trace, counters and memory mapping. The SPLICE capability of memory mapping is also supported.

The rest of this section provides general information on DBUG supported emulators. Sections 2 and 3 are an overview of the invocation and command set for the HP64772/HP64778 and HP64779, respectively. Section 4 describes the invocation and command set for SPLICE.

Further information on the Series HP64000 In-System Emulators can be found in the *User's Reference Guide* and the *Emulators Terminal Interface* for each emulator. See the *SPLICE Hardware Manual* for more information on the SPLICE Development Tool.

6.1.1 Downloading A Program

The first step before executing and debugging a program is to download it to the emulation or target memory. This is done with the **load** command. Note that if you are using the emulation memory, the memory map should be initialized before downloading (see the **map** command in this chapter).

When using the SPLICE emulator, programs should be linked so that down-loading to RAM located on the SPLICE board will not cause overwriting of the monitor wakeup area. The on-board SPLICE monitor (`spmon`) uses 2 Kbytes of RAM as its scratch pad, starting from a location pointed to by the monitor static base register. The monitor static base register is initially set to 0. You can change the setting of the monitor static base register by using the **configh mon sb** command. The new address must be within the first 64 Kbytes of address space to minimize the impact of the SPLICE board on the target system.

6.1.2 Tracing

The trace function is an important feature provided by the Series HP64000 emulators. Trace allows you to monitor the status of the CPU pins during each cycle, without interrupting the execution of the application program. This includes viewing a selected subset of pins during specified cycles.

You can use the trace mechanism to store the values monitored in each pin-group during each cycle. A line composed of the values of the pin-groups is entered into a trace buffer after each cycle. The trace buffer stores a maximum of 1024 lines.

Begin the trace by issuing the **traceh start** command. As the trace runs, pin data from each cycle of program execution is read into the trace buffer until it is full. The user can control which cycles to store (and thereby maximize storage efficiency) with the **traceh define** command. The **traceh define** command defines which cycles write to the trace buffer.

You can display entries that have been stored in the trace buffer with the **traceh list** command. The **traceh format** command controls the *radix* in which the value of each pin-group in the trace buffer is displayed. Only labels with a recognized *radix* are displayed when the **traceh list** command is issued. It is also possible to specify symbolic disassembly format. This feature is another way to screen out unnecessary information. The **traceh stop** command halts the currently active trace. The **traceh reset** command empties the contents of the buffers and resets all definitions and display formats to their default values. The **traceh status** command prints current definitions and selected formats.

Traceable CPU pins (channels) are grouped by logical function into pin-groups. Each group has a matching label known to DBUG. The labels and their pins for each emulator are presented in the following tables:

Table 6-1. HP64772/8 Emulator Pin-Group Assignments

(for the NS32532, NS32GX32 and NS32GX320 CPUs)

dbug name	PIN-GROUP	SIGNAL	FUNCTION
abus	0..31	ADDR0..ADDR31	processor address bus
stat	32..35	ST0..4	processor status
be	36..39	BE0..BE3	processor byte enables
bw	40..41	BW0..BW1	processor bus width
ddin	42	DDIN	processor data direction
us	43	U/S	processor user / supervisor
dbus	48..79	DATA0..DATA 31	processor data bus

Table 6-2. HP64779 Emulator Pin-Group Assignments

(for the NS32CG16, NS32CG160 and NS32FX16 CPUs)

dbug name	PIN-GROUP	SIGNAL	FUNCTION
abus	0..23	ADDR0..ADDR31	processor address bus
stat	24..27	ST0..4	processor status
us	28	BE0..BE3	processor byte enable
bpu	29	BPU	processor BITBLT processing cycle
spc	30	SPC	processor SLAVE process/control cycle
ias	31	IAS	processor internal address strobe
hbe	32	HBE	processor high byte enable
ddin	33	DDIN	processor data direction
dak	34..35	DAK1..DAK0	DMA channel 1/0 cycle
pfs	36..39	PFS3..PFS0	processor PFS count
dbus	48..63	DATA0..DATA 31	processor data bus

6.1.3 Counter

The Series HP64000 emulators contain a counter that is directly connected to the trace function. This counter can be used as a clock for measuring performance or keeping track of execution sequences. It can also count user-defined events, such as references to memory locations, and the occurrence of particular conditions. The counter is activated by issuing the **traceh start** command. The **counter define** command determines what the counter is used for. The **traceh format** command enables the user to control whether counter information is displayed in absolute (accumulative) or relative (discrete) forms when the output is generated by the **traceh list** command. The **counter status** command displays current definitions.

6.1.4 Memory mapping

DBUG provides the ability to map or unmap either target memory or emulation memory. Memory blocks can be referred to as either part of the target or emulation memory. The **map** command can specify which of the target memory address ranges should be referred to as emulation memory. This feature is especially useful when debugging code that is in a ROM, because otherwise you cannot set software breakpoints.

Memory blocks which have been defined as part of the emulation memory can be redefined as part of the target memory with the **unmap** command.

6.2 The HP64772 and HP64778 Emulators

6.2.1 Invocation

DBUG is connected to an in-system emulator either by RS-232 or via Ethernet.
There are two ways to invoke DBUG's interface with the HP64772 and HP64778 In-System Emulators. One is through the invocation line, and the other is through the **connect** or **config** commands. Both approaches require that you specify the **mon** parameter as **ise532**.

A sample invocation line is as follows:

```
dbug -mon ise532 -cpu gx32 -fpu 381 -l tty4 a.out
```

This invocation line invokes DBUG, automatically connects to the serial line `tty4` (the **-l** parameter), and expects to find the HP64772 Series In-System Emulator (the **-mon** parameter), with an NS32GX32 CPU and an NS32381 FPU. The executable file is `a.out`.

If the required parameters do not appear in the invocation line, the **connect** command may be used as follows:

(dbug) `connect link tty4 with mon isegx320 cpu gx320 fpu 381`

This command yields the same results as the previous example.

6.2.2 Virtual PC (VPC)

The emulator can trace values of virtual addresses of each executed instruction. These values are stored in **abus**. In the trace list (displayed with the **traceh list** command), all lines matching a virtual PC have the label **VPC** appended to them. The **traceh format** command allows you to display additional information for these lines, such AAAAAas disassembly of the instruction matching the line's PC value, the source line number and source filename matching that instruction, etc. You may specify that only addresses of executed instructions be stored, by issuing the appropriate **traceh define** command. The emulator uses the **stat** pin-group to mark the **VPC** cycles by assigning it the value 0.

For the HP64778 emulator, the value of 1 on the **stat** pin-group denotes a lost **VPC** cycle during a burst sequence.

6.2.3 Condition option

Several of the commands relevant in this context allow for an optional specification of a condition. The condition in these cases is restricted to the following:

pin_group {== | !=} {*range* | *value*}

or

(*pin_group* {== | !=} {*range* | *value*}) {**and** | **or**} (*pin_group* {== | !=} {*range* | *value*})

Where:

1) *pin-group* is one of the labels described in Table 6-1.

2) *value* is any expression that can be expressed as a numeric value.

3) *range* is an expression of the type: *value .. value.*

Note that only one range definition may be used at any one time. Respecifying an existing range will cause the existing range to be overwritten.

6.2.4 Traceh Mode

This mode determines which target cycles are to be sampled by the emulator. The syntax of the *traceh_mode* specification is:

{**bus** | **vpc** | **all**}

Where **bus** indicates that only bus cycles should be sampled, **vpc** indicates that only executed instruction addresses should be sampled, and **all** means that all target cycles will be sampled.

The value of the *traceh_mode* is set with the **traceh define** command.

Several commands may be influenced by the *traceh_mode* value. Among them are: **traceh define**, **traceh stop**, **traceh start**, **counter defined** and **stoph**.

6.2.5 HP64772/8 BREAKH - stop execution

SYNTAX

breakh

DESCRIPTION

The **breakh** command issues a break to the monitor, causing it to stop executing the user program and to report its position. If the emulator is in the reset state when the **breakh** command is issued, it will be released from reset.

EXAMPLE

(dbug) `run`

```
<control c>
<BREAK>
```

(dbug) `breakh`

```
emulator run aborted
stopped in main at line 1 in file "loop.c"
1 main()  { int i,j; for (i=1;i<100000000;i++) { j=10; } }
```

6.2.6 HP64772/8 CONFIGH - set configuration parameters

SYNTAX

configh [[**mon bg** | **mon fg** *address*]
[**wait number**] [**lock** {**enable** | **disable**}]
[**burst** {**enable** | **disable**}]
[**realtime** {**enable** | **disable**}]
[**clock** {**inter** | **exter**}]
[**cfg** *number*]
[**excp** [**all** | **none** | *exception*]]
[**dma** {**u** | **s**}]
[**ic** {**enable** | **disable**}]
[**dc** {**enable** | **disable**}]
[**mod** *address*]
[**sp0** *address*]
[**nmi** {**enable** | **disable**}]
[**hold** {**enable** | **disable**}]
[**dbg** {**enable** | **disable**}]
[**target** {**all** | **ignore** | [**dbg** | **int** | **nmi** | **hold**]}]

DESCRIPTION

When no parameters are specified, the current configuration is given.

mon bg | **mon fg** *address* specifies the type of monitor. **mon bg** specifies a background monitor, **mon fg** specifies a foreground monitor starting at *address*. The address can be specified in hex or physical and is aligned by DBUG to a 4k byte boundary for the 64772 emulator, and to a 32K byte boundary for the 64778 emulator.

wait *number* selectes the number of wait states during emulation memory access for the HP64772 emulator. *number* can be either 0, no wait states, or 1, wait state. (For information on wait state configuration for the HP64778 emulator, refer to the **map** command.)

lock enable | **disable** locks the emulation memory access to target system control for the HP64772 emulator. (For information on locking memory for the HP64778 emulator, refer to the **map** command.)

burst enable | **disable** controls the burst mode in emulation memory.

realtime enable | **disable** controls the real time restrictions. When **realtime** is disabled the emulator allows you to issue commands that interfere with the application, and thus affect its real time. Such commands, e.g. **pcpu** (print cpu

registers) are not allowed when **realtime** is enabled.

clock inter | exter determines internal or external emulator clock source selection. The **inter** option selects the internal 50 MHz clock. The **exter** option selects the clock provided by the target system.

cfg *number* sets the cfg register when the NS32532, NS32GX32, or NS32GX320 CPUs are reset by the emulator. *number* must specify a 32 bit value. If no *number* is specified the cfg will not be initialized.

target all | ignore | dbg | int | nmi | hold configures the target system signal. The **all** option enables all four target signals. The **ignore** option disables all four target signals. Signals can be specified as **dbg, int, nmi** or **hold**. Signals not specified will be disabled. (Valid only for the HP64772 emulator.)

nmi enable | disable controls the target system signal NMI configuration. (Valid only for the HP64778 emulator.)

hold enable | disable controls the target system signal HOLD configuration. (Valid only for the HP64778 emulator.)

dbg enable | disable controls the target system signal DBG configuration. (Valid only for the HP64778 emulator.)

mod *address* sets the mod register when the CPU is reset by the emulator. *address* must be a 16 bit value. The default value is fff0 hex. (Valid only for the HP64778 emulator.)

sp0 *address* sets the sp0 register when the NS32GX320 CPU is reset by the emulator. *address* must be a 32 bit value. (Valid only for the HP64778 emulator.)

excp controls the CPU exception setup. The dispatch table is modified so that the occurance of specified exceptions will result in an entry into the foreground monitor. The setup is based on the current exception mode (direct or indirect) and the contents of the intbase and mod registers. This option is only valid when the foreground monitor is used. *exception* is one of the following names: NVI, NMI, RESERVED, SLAVE, ILL, SVC, DVZ, FLG, BPT, TRC, UND, RBE, NBE, OVF, DBG; or the numbers 0 through 14. **all** specifies that all exceptions will result in an entry into the foreground monitor. **none** specifies that no exceptions will be handled by the foreground monitor. Note however that the BPT and TRC exceptions are always used by the foreground monitor for breakpoints and single stepping.

dma allows you to define the User/Supervisor signal to the emulation memory during DMA transfers for the NS32GX320 CPU. This signal is not defined during DMA transfers, but may be defined by the target system for target system memory. For the DMA channel, specifying **u** defines a User signal, specifying **s** defines a Supervisor signal. (Valid only for the HP64778 emulator.)

dc causes the processor input CIIN directly to be driven directly by the target system data transfers (data cache enabled). This is the default value. (Valid only for the HP64778 emulator.)

ic specifies that the processor input CIIN will be true (instruction cache enabled). (Valid only for the HP64778 emulator.)

EXAMPLES

```
(dbug)  configh
        monitor  foreground, address 0x30000
        clock    internal
        realtime disabled
        cfg      0x1f0
        target   dbg,int,nmi,hold
        burst    enabled
        lock     disabled
        wait     1

(dbug)  configh  mon bg burst disable lock enable target int
        monitor  background
        clock    internal
        realtime disabled
        cfg      0x1f0
        target   int
        burst    disabled
        lock     enabled
        wait     1
```

SEE ALSO

connect

6.2.7 HP64772/8 CONNECT - connect to a system emulator

SYNTAX

connect [**link** *linkname* | **node** *nodename*] [**with** [**nofast**] [**list**] [**baud** *number*] [**cpu** *name*] [**mmu** *name*] [**fpu** *name*] [**mon ise532** | **isegx320**]]

DESCRIPTION

The **connect** command is used to switch to the *remote* operation mode. **Connect** selects the communications channel through which DBUG and the target board communicate, and sets configuration parameters. The **connect** parameters are closely related to the DBUG invocation line flags. Several **connect** commands may be issued during one DBUG session to alter the configuration or communication parameters.

Once the **connect** command has been issued for the first time, and the connection established, the **connect** command can be issued any number of times. When the **connect** command is issued for the second time, it terminates the current connection and establishes a new one. The **config** command is used to toggle the fast/nofast and list parameters. When **connect** is issued for the second time it terminates the current connection and establishes a new connection.

At least one parameter must be specified when the **with** clause is used.

Link *linkname* identifies the serial communication line between the host and the target board. The default *linkname* is the last name given by the previous **connect** command, or as selected in the invocation line (**-l** parameter), or as specified in the GNX target specification file.

When working with the emulator, DBUG supports two types of link interfaces:

1. RS232 link. In this case, under UNIX, the *linkname* should have the form `tty`*xx*, where *xx* is the link identifier (i.e. the device would be: `/dev/tty`*xx*). Under MS-DOS, the linkname should have the form `com`*xx*.
2. The HP64037 high-speed RS422 interface card (referred to as HP COMCARD). In this case the *linkname* should have the form `com`*xx*. `Com`*xx* is the name of the IBM PC port in which the HP COMCARD is installed (i.e. `connect link com6`). The HP COMCARD can be used only with SYS32/30 or SYS32/50 hosts.

The **node** parameter selects the fast communication channel (LAN) between DBUG and the emulator. *nodename* is the name of the emulator, as recognized by the host system. The default *nodename* is either the last name given by the previous **connect** command, the name selected in the invocation line, or the

name specified in the GNX target specification file.

The **node** parameter is only valid for isegx320.

In the **fast** protocol (the default value), DBUG uses the *HP X* Binary file format and transfer protocol when downloading an executable file to a target system. The **fast** protocol should be used with the HP COMCARD, to achieve high baud rates. Specifying **nofast** causes DBUG to use the *Intel-hex* file format downloading the executable file to a target system. The **nofast** option can be used when the serial communication line is unreliable.

The **list** option enables the verbose communication mode. In this mode, DBUG displays the messages exchanged by the debugger and the monitor or the emulator.

The command:

(dbug) `connect with list`

may be used to enter the verbose communication mode, even if the **connect** command was previously issued. An additional **connect** command with no parameters disables the verbose mode.

Baud *number* sets the communication baud rate for *stand-aside mode*. Possible values for tty communication are: 300, 600, 1200, 2400, 4800, 9600, 19200, and 38400. Possible values for HP COMCARD communication are: 300, 1200, 2400, 4800, 9600, 19200, 38400, 57600, 115200 and 230400. The default baud rate is 9600.

Cpu specifies the target board CPU name. Valid names are gx32, 532 and gx320.

Mmu specifies the target board MMU name. Valid names are: 382, **onchip**, and **nommu**.

Fpu specifies the target board floating-point unit name. Valid names are: 381, 580, **nofpu**.

The default values for the **cpu**, **mmu** and **fpu** options are described by your GNX target specification file, the previous **connect** command, or the invocation line.

If the emulator is plugged into a target that has ROM at address 0, you should first initialize the target and only then issue the **connect** command. For

example, if the emulator is plugged into NS32GXEB evaluation board, you should press the RESET button on the evaluation board before issuing the **connect** command.

When DBUG is initialized, no communication link is defined. Therefore, the user must select a link before issuing any debugger command that communicates with the target board.

NOTE: Although DBUG allows you to change links during the debugging session, this exercise is not recommended.

EXAMPLES

(dbug) `connect`

Reconnects the target system using the parameters selected by the previous `connect` command. The **list** and **nofast** parameters are disabled because they are not specified. The **fast** protocol will be used for downloading the executable file, and the communication between DBUG and the monitor will not be displayed.

(dbug) `connect link tty7 with baud 38400 mon ise532 cpu gx32`

Selects the channel `tty7`, and communicates with the emulator over a 38400 baud rate serial line. In this case the ISE should be reset to work at 38400 baud. This is done by setting the dip-switches on the back of the emulator to the correct value and turning the emulator on.

(dbug) `connect link com4 with mon isegx320 baud 230400 cpu gx320`

Selects your HP COMCARD installed as port COM4 in your IBM PC as your communication channel. The connection is established with the emulator using 230400 baud rate and the RS422 protocol provided by the emulator and the HP COMCARD. The load protocol will be **fast** since the **nofast** parameter is not specified.

(dbug) `alias do_connect (linkname) 'connect link linkname with baud 38400 cpu gx32 fpu 381 mon ise532'`

Defining an alias/macro makes the connect command easier to use. Issuing the following command selects `tty4` as your communication channel. This macro may be inserted in your *.dbuginit* (*dbug.ini* in VMS/MS-DOS) file so that it will be recognized each time DBUG is entered.

(dbug) `do_connect (tty4)`

(dbug) `connect node ise01 with mon isegx320 cpu gx320.`

Selects, and connects to, the ISE whose Ethernet name is `ise01` for use with the NS32GX320.

SEE ALSO

begin, load

6.2.8 HP64772/8 COUNTER DEFINE - counts time or events

SYNTAX

counter define {**none** | **time** | *condition*}

DESCRIPTION

The **counter define** command qualifies the emulator's counter to count **time** or *events*.

When counting **time**, the counter acts as a clock. When counting *events*, the counter counts the number of times the specified *condition* is true. Counting begins from zero. The counter can only be incremented (cannot be decremented).

The counter is invoked by issuing the **traceh start** command and can be viewed with the **traceh list** command, in either absolute or relative formats. The complete procedure for viewing the counter is discussed in the description of the **traceh format** command.

The default setting for the counter is **none**. The counter is not active during this setting.

If the counter is defined to count either time or events, the clock rate used by the emulation analyzer must be reduced accordingly. If time is counted, this clock rate is set by DBUG to "slow" (less than or equal to 16 MHz). If events are counted (**counter define** *condition*), DBUG sets the clock rate to "fast" (between 16 and 20 MHz). When the counter is defined as **none**, the clock speed for the emulation analyzer is set by DBUG to "very fast" (speed between 20 MHz and 25 MHz).

You must use the **traceh define** command to ignore either bus cycles or vpc cycles if you define the counter to count either time or events. Otherwise, lines may be missing at random from the trace buffer, as the emulation analyzer's clock speed will not be fast enough to ensure that all cycles are buffered.

For more information, refer to the *HP64772/8 Emulator Terminal Interface: NS32GX32 Emulator User's Guide.*

EXAMPLES

(dbug) `counter define time`

Defines the counter to count time.

(dbug) `counter define (abus == 0xffffffff) or (stat == 0x0)`

This command causes the counter to count all the events in which either the **abus** (address bus) equals `0xffffffff`, or the `stat` (status) equals 0x0.

SEE ALSO

counter status, traceh start, traceh format, traceh define

6.2.9 HP64772/8 COUNTER STATUS - print counter qualification

SYNTAX

counter status

DESCRIPTION

This command prints the counter's current qualification. This may be **none**, **time** or a *condition*.

EXAMPLES

(dbug) `counter status`

```
The counter definition is:
   counter define (abus == 0xffffffff) or (stat == 0x0)
```

This example indicates that the counter was defined to be incremented each time the condition: `(abus == 0xffffffff) or (stat == 0x0)` is true.

SEE ALSO

counter define, traceh format, traceh define, traceh start

6.2.10 HP64772/8 LOADMON - load a foreground monitor

SYNTAX

loadmon *filename*

DESCRIPTION

This commands allows you to load your private version of the foreground monitor into the emulator. The **configh** command can then be used to specify the foreground monitor. For more details on modifying and compiling a foreground monitor, refer to the *HP64772/8 Emulator Terminal Interface: NS32GX32 Emulator User's Guide*. *filename* is the name of the executable foreground monitor file.

The symbolic information of a debugged program is lost when a new foreground monitor is loaded. The **load** command can be used to reread this information. Using the **loadmon** command at the beginning of your session can avoid the need to reread the symbolic information.

EXAMPLES

(dbug) `loadmon my_mon`

Specifies that `my_mon` is loaded into the emulator as a monitor program.
(dbug) `configh mon fg 0x30000`

Specifies that `my_mon` is used as the foreground monitor, mapped to address 30000 hex.

(dbug) `load my_prog`

The debugged program `my_prog` is now loaded using the new foreground monitor.

SEE ALSO

configh, load

6.2.11 HP64772/8 MAP - map emulation memory

SYNTAX

map [*range* [**rom** | **ram** | **trom** | **tram**]
[**lock** | **wait** *number*] [**with copy**]]

DESCRIPTION

This command maps the specified address range to the emulator's memory. *range* is an address range that is boundary aligned outward to 4-Kbyte boundaries for the HP64772 emulator, and to 32-Kbyte boundaries for the HP64778 emulator.

Memory blocks can be referred to as either part of the target or emulation memory. Memory blocks that are mapped to emulation memory may be characterized as either RAM or ROM. All memory is referred to as target RAM immediately after the emulator is powered-up. The **map** command specifies which of the target memory address ranges should be referred to as emulation memory, and whether they should be characterized as RAM or ROM.

Memory blocks that have been referred to as part of the emulation memory can be redefined as part of the target memory with the **unmap** command.

If **rom** is specified, the *range* is treated as part of a ROM (no write to target ROM is allowed) and mapped to emulator memory. If **ram** is specified, the *range* is treated as part of a RAM. If **tram** is specified, the *range* is treated as target RAM. If **trom** is specified, the *range* is treated as target ROM.

If none of the options **ram**, **rom**, **tram** or **trom** are specified, the default is **ram**.

If **with copy** is specified, the contents of the address range in the target memory are copied to the same location in the emulation memory. Otherwise, the emulator memory block might not have the same contents as the corresponding target memory block.

To set software breakpoints on part of the program that lies in target ROM, first map the relevant address range to emulation memory, characterize it as **ram**, and specify **with copy**.

DBUG assigns a *map term* to each memory block mapped to emulation memory. Each term consists of the term number, the mapped address range, and its RAM/ROM status. Map terms are displayed each time terms are added or deleted, or when the **map** command is issued without parameters. The term numbers will be rearranged in ascending order each time a new term is added or

deleted. The HP64772/8 emulator supports seven map terms and 512 Kbyte emulation memory.

If *range* is not specified, the mapping of all mapped address ranges is displayed.

For the HP64778 emulator (NS32GX320 CPU), only addresses in the range 0..0xfffefff can be mapped. Memory addresses in the IO range of 0xffff0000..0xffffffff will result in internal processor cycles and are not affected by the memory map. Therefore, memory commands may be used to access the IO registers.

Additional attributes for the HP64778 emulator can be specified for the emulation memory terms. **lock** determines whether the access to the selected address range is according to target system timing or emulation memory timing (default). The **wait** *number* may be used to specify the number of wait states the defined memory should emulate. The number of wait states may be 0, 1, 2 or 3. Note that the emulation memory containing the foreground monitor (if used) is confgured seperately with the **configh** command.

EXAMPLES

```
(dbug) map
[1] 0x00000000..0x00000fff emulator ram
[2] 0x00001000..0x00002fff emulator rom
[3] 0x00030000..0x00030fff emulator ram
```

Prints the current map. The ranges are in hexadecimal format. Term [3] is the memory block used for the foreground monitor. It is set by DBUG when the connection with the emulator is established.

```
(dbug) map 8830..8840 rom
[1] 0x00000000..0x00000fff emulator ram
[2] 0x00001000..0x00002fff emulator rom
[3] 0x00008000..0x00008fff emulator rom
[4] 0x00030000..0x00030fff emulator ram
```

Maps the block containing the specified address range to emulator memory. The address range is considered part of the target ROM. The contents of the address range on the target board are not copied to the emulator's memory. Note that the address range is rounded to 4-Kbyte page boundaries.

```
(dbug) map 0x9000..0x9fff trom
[1] 0x00000000..0x00000fff emulator ram
[2] 0x00001000..0x00002fff emulator rom
[3] 0x00008000..0x00008fff emulator rom
[4] 0x00009000..0x00009fff target rom
[5] 0x00030000..0x00030fff emulator ram
```

Maps the block containing the specified address to target ROM.

SEE ALSO

unmap

6.2.12 HP64772/8 RESETH - reset CPU

SYNTAX

reseth

DESCRIPTION

The **reseth** command resets the emulation CPU. This causes certain registers to be reset to zero, and others to be reset to an undefined value. Executing this command also places the emulator in a reset state. You must return system control to the monitor before attempting to change or print registers. Do this by issuing the **breakh** command.

6.2.13 HP64772/8 STOPH - set a hardware breakpoint

SYNTAX

stoph if *stop_condition*

DESCRIPTION

This command uses the emulator's hardware to force your program to halt execution after the *stop_condition* becomes true. *Stop_condition* is:

pin_group {== | !=} {*value* | *range*}

Only one range definition may be used at one time for all of the emulator commands. In addition, up to 4 hardware breakpoints may be defined with this command. Each defined hardware breakpoint is assigned an event number. Use the **status** command to see a list of the currently defined events.

The hardware trace begins automatically when a hardware breakpoint is defined. DBUG activates the hardware trace in order to keep the hardware breakpoints active. You may use the **traceh stop** command to stop the hardware trace and disable all the currently defined hardware breakpoints. The hardware breakpoints will become active again when you issue the **traceh start** command or define another hardware breakpoint.

If execution stops because of a hardware breakpoint, the trace buffer contains up to 512 entries, stored just before execution was halted. DBUG will restart the hardware trace when you resume execution, in order to keep the hardware breakpoints active.

The **traceh start at** and **traceh stop at** commands may not be used when a hardware breakpoint is defined. To delete a hardware breakpoint, you must delete the event associated with it. Use **status** to see a list of the currently defined events, and then use **delete** to delete the appropriate event(s).

Note that hardware breakpoints are sensitive to *traceh_mode* as defined with the **traceh define** command.

EXAMPLES

(dbug) `stoph if abus ==&array..&array + array_size`

Causes execution to halt when the array `array` is referenced. `array` and `array_size` are assumed to be variables defined in your program.

(dbug) `stoph if us==1`

Causes execution to halt when execution mode is switched to supervisor mode.

(dbug) `status`

The status command displays the previously defined hardware breakpoints with the matching event numbers. These appear here:

```
[1] stoph if abus==array..array+array_size
[2] stoph if us==1
```

(dbug) `traceh status`

```
Hardware trace is running
The hardware trace definition is:
      traceh define all
The hardware trace format is:
      traceh format abus:x dbus:x
```

(dbug) `delete 1`

Deletes the first hardware breakpoint

(dbug) `status`

```
[2] stoph if us==1
```

This shows that the first hardware breakpoint has been deleted.

SEE ALSO

stop, traceh start, traceh stop, traceh status, status, delete

6.2.14 HP64772/8 TRACEH DEFINE - define hardware trace

SYNTAX

traceh define [*condition*] [*traceh_mode*]

DESCRIPTION

This command performs two functions:

1. Changes the value of *traceh_mode*. This mode determines which types of target cycles will be sampled by the emulator.
2. Defines which cycles of the specified type will be stored in the trace buffer.

Traceh_mode specifies which types of target cycles will be sampled by the emulator. Possible values for this parameter are:

bus - only bus cycles will be sampled.
vpc - only execution cycles will be sampled.
all - all cycles will be sampled.

The *traceh_mode* definition affects all commands that are sensitive to it. These include: **stoph**, **counter define**, **traceh stop** and **traceh start**.

Note that only the **bus** and **vpc** modes are supported when the emulator's counter is active (the **counter define time** or **counter define condition** commands have been issued). Otherwise, in **all** mode, random cycles may be missing from the trace buffer.

If *traceh_mode* is not specified, its value remains unchanged. The initial value of *traceh_mode* is **all**.

Condition specifies which cycles should be stored in the trace buffer. Cycles which match the condition and of *traceh_mode* type are stored. The default is to match all cycles.

The **traceh define** command cannot be issued if a trace is already in progress. The trace must first be stopped with the **traceh stop** command.

The **traceh list** and **traceh format** command should be used to view the accumulated trace buffer entries.

EXAMPLES

(dbug) `traceh define abus==0x90ff`

Defines that only trace entries with the address values 90ff (hexadecimal) are stored (and therefore, displayed later).

In the following example, only the changes to the variables `i` and `j` defined in `main` will be of interest. The executable program is assumed to be loaded to memory.

(dbug) `list 1,20`

```
1   main()
2   {
3       int i=0;
4       int j=10;
5
6       i = 1;
7       j = 9;
8       i = 2;
9       j = 8;
10      i = 3;
11      j = 7;
12      i = 4;
13      j = 6;
14      i = 5;
15      j = 5;
16      i = 6;
17      j = 4;
18      i = 7;
19      j = 3;
20      i = 8;
21      j = 2;
22      i = 9;
23      j = 1;
24      i = 10;
25      j = 0;
26      i = 11;
27      j = -1;
28      f();
29  }
```

A breakpoint is specified, so that execution will be stopped in the function `main`. (`i` and `j` are local variables of the function `main`).

(dbug) `stop in main`

```
[1] stop in main
```

(dbug) `run`

```
[1] stopped in main at line 3 in file "t_trace.c"
      3 int i=0;
```

Next, the addresses of `i` and `j` are printed.

(dbug) `print &j`

```
0x24ff0
```

(dbug) `print &i`

```
0x24ff4
```

As the addresses are adjacent, the following definition will capture all the accesses to these memory locations in user mode.

(dbug) `traceh define (abus==&j..&i+3) and (us==1)`

Since these variables are allocated on the stack, these addresses are only of interest while we are in function `main`. The following hardware trace definition will solve this problem:

(dbug) `traceh stop at &f`

Now, continue execution

(dbug) `cont`

```
execution completed
```

It is now easy to follow the values assigned to the variables `i`, `j`, and to compare them to the source listing previously displayed. The value `24ff4` in the `abus` column is the address of the variable `i` and the value `24ff0` is the address of `j`. the values assigned to these variables are the values displayed in the `dbus` column. First `i` is assigned `0`, then `j` is assigned `0xa` etc.

(dbug) `traceh list -24,-15`

```
line  abus      dbus
----  ----      ----
 -24  00024ff4  00000000
 -23  00024ff0  0000000a
 -22  00024ff4  00000001
 -21  00024ff0  00000009
 -20  00024ff4  00000002
 -19  00024ff0  00000008
 -18  00024ff4  00000003
 -17  00024ff0  00000007
 -16  00024ff4  00000004
 -15  00024ff0  00000006
```

The format can be changed to allow a different view of the same information, for example:

(dbug) `traceh format mnemonic`

(dbug) `traceh list -24,0`

```
line  mnemonic
----  --------
 -24  00000000  usr data       write
 -23  0000000a  usr data       write
 -22  00000001  usr data       write
 -21  00000009  usr data       write
 -20  00000002  usr data       write
 -19  00000008  usr data       write
 -18  00000003  usr data       write
 -17  00000007  usr data       write
 -16  00000004  usr data       write
 -15  00000006  usr data       write
 -14  00000005  usr data       write
 -13  00000005  usr data       write
 -12  00000006  usr data       write
 -11  00000004  usr data       write
 -10  00000007  usr data       write
  -9  00000003  usr data       write
  -8  00000008  usr data       write
  -7  00000002  usr data       write
  -6  00000009  usr data       write
```

```
-5  00000001  usr data       write
-4  0000000a  usr data       write
-3  00000000  usr data       write
-2  0000000b  usr data       write
-1  100effff  usr data       write
 0  VPC ---------------------------
    ---   f    :    enter   [], 0x8
```

Note that the trace was stopped when the function `f` was about to be entered.

SEE ALSO

traceh format, traceh start, traceh status, traceh stop

6.2.15 HP64772/8 TRACEH FORMAT - define trace display format

SYNTAX

traceh format [*pin_group*:*radix* ...] [**absolute** | **relative**] [**lines**] [**disasm**] [**mnemonic**]

DESCRIPTION

The *pin_group* parameter selects any of the following labels which correspond to different channels (pin-groups) as found in Table 6-1: **abus**, **dbus**, **stat**, **ddin**, **be**, **bw**, **us**. The counter may also be incorporated into the trace listing. This can be done by specifying either **absolute** or **relative** format. The absolute format displays the cumulative value of the counter (for all trace lines). The relative format displays the counter's accumulated value per trace line.

The trace listing columns include pin values that correspond to the selected labels in the specified format.

If **disasm** is specified, the disassembly of instructions matching a VPC cycle is appended to the displayed line (the disassembly of the instructions reside in the address specified by the **abus** value in that cycle). If **lines** is specified, the source line number and the source file name are appended to each line matching a VPC cycle. If **mnemonic** is specified, all the other specified columns are ignored and a verbal description of their contents is given instead. This description includes the description of the execution cycles. When activated, DBUG defines the initial trace format as mnemonic.

EXAMPLES

(dbug) `traceh format abus:x dbus:d`

Specifies that the address bus (**abus**) is displayed in hexadecimal format, and the data bus (**dbus**) is displayed in decimal format. No other information is displayed.

SEE ALSO

traceh define, traceh start, traceh status, traceh stop

6.2.16 HP64772/8 TRACEH LIST - display trace buffer

SYNTAX

traceh list [*number* | *number*, *number*]

DESCRIPTION

This command displays the trace on the screen. If *numbers* are specified, only those entries within the specified range are displayed. If one number is specified, the current line and the few lines following it are displayed. If no number is specified, the next few lines are displayed. All lines which correspond to a virtual PC cycle have the letters **VPC** appended to them.

The trace is displayed in a format defined by the **traceh format** command. The **traceh format** command makes it possible to redisplay the trace buffer according to different formats without having to resume execution.

EXAMPLES

(dbug) `traceh list 3,9`

```
line  abus      dbus      count
----  ----      ----      -----
  3   00000001  --------  1.240 uS  VPC
  4   00000000  --------  1.480 uS  VPC
  5   00000000  127feaa2  1.880 uS
  6   00000004  ff785634  2.600 uS
  7   00000001  --------  3.120 uS  VPC
  8   00000000  --------  3.360 uS  VPC
  9   00000000  127feaa2  3.760 uS
```

Lines 3 through 9 of the trace buffer are displayed, according to the specified format (in this case the default format). The address bus and data bus are displayed in hexadecimal notation, the counter is displayed in absolute format, and VPC cycles are noted.

SEE ALSO

traceh format, traceh start, traceh status, traceh stop

6.2.17 HP64772/8 TRACEH RESET - reset trace definitions

SYNTAX

traceh reset

DESCRIPTION

This command resets the trace and counter definitions to their initial settings. Any active trace is halted. The trace definition is reset to its default value, which specifies that all cycles should be stored. The *traceh_mode* is set to **all**. The trace format is also reset to its default value, which specifies **abus** and **dbus** in hexadecimal notation, and counter definition is reset to **none**.

EXAMPLES

The following command sequence demonstrates the effect of the **traceh reset** command. First the trace status is displayed by the **traceh status** command, which shows the current value of the **traceh define** and **traceh format** commands. Next the **traceh reset** command gives new (default) values to the hardware trace definition and format. These values are displayed by re-issuing the **traceh status** command. Finally, the fact that the buffer is now empty is demonstrated by issuing the **traceh list** command.

```
(dbug) traceh status

Hardware trace is stopped
The hardware trace definition is:
    traceh define stat==0 all
hardware trace format is:
    traceh format abus:x ddin:d
(dbug) traceh reset

(dbug) traceh status

Hardware trace is stopped
The hardware trace definition is:
    traceh define all
The hardware trace format is:
    traceh format mnemonic
```

(dbug) `traceh list`

```
line  abus      dbus      count
----  ----      ----      -----

  ** Trigger not in memory **
```

SEE ALSO

traceh define, traceh format, traceh start, traceh status, traceh stop

6.2.18 HP64772/8 TRACEH START - start hardware trace

SYNTAX

traceh start [**at** *address*]

DESCRIPTION

This command starts the emulator's trace function. Issuing this command first causes the trace buffer to be reinitialized (all current entries are removed). Further, all cycles that match the trace definition are stored in the emulator's trace buffer once execution resumes.

If an address is specified, trace will start when execution reaches that address. All previous addresses specified with a **traceh start** or **traceh stop** command are ignored.

The trace buffer contents are displayed using the **traceh list** command according to the format currently specified by the **traceh format** command. The buffer size is 1024 entries. Tracing is suspended when the buffer is full. Clear the buffer using the **traceh reset** command, or by re-issuing the **traceh start** command. A trace in progress should be stopped with the **traceh stop** command, before using the **traceh start** command.

Note that the trace trigger point is affected by the *traceh_mode*.

EXAMPLES

(dbug) `traceh start`

(dbug) `traceh list`

```
line  abus      dbus      count
----  ----      ----      -----

  ** Trigger not in memory **
```

The command resets the trace buffer contents. Only new entries are buffered, once execution resumes.

The following example will start a hardware trace when the first instruction of a function f is executed. To demonstrate this fact more clearly, only the execution (VPC) cycles are stored (using the appropriate **traceh define** command).

(dbug) `traceh define stat==0`

(dbug) `traceh start at &f`

The executable program is assumed to be loaded to memory.

(dbug) `run`

```
execution completed
```

(dbug) `traceh format abus:x lines disasm`

(dbug) `traceh list 0`

```
line  abus
----  ----
0     0000e06b VPC "t_trace.c":32  f      : enter   [], 0x8
1     0000e06e VPC "t_trace.c":35  f+0x3  : movqd   0x1, -0x4(fp)
2     0000e071 VPC "t_trace.c":36  f+0x6  : addr    @0x9, -0x8(fp)
3     0000e075 VPC "t_trace.c":37  f+0xa  : movqd   0x2, -0x4(fp)
4     0000e078 VPC "t_trace.c":38  f+0xd  : addr    @0x8, -0x8(fp)
5     0000e07c VPC "t_trace.c":39  f+0x11 : movqd   0x3, -0x4(fp)
6     0000e07f VPC "t_trace.c":40  f+0x14 : movqd   0x7, -0x8(fp)
7     0000e082 VPC "t_trace.c":41  f+0x17 : movqd   0x4, -0x4(fp)
8     0000e085 VPC "t_trace.c":42  f+0x1a : movqd   0x6, -0x8(fp)
9     0000e088 VPC "t_trace.c":43  f+0x1d : movqd   0x5, -0x4(fp)
10    0000e08b VPC "t_trace.c":44  f+0x20 : movqd   0x5, -0x8(fp)
```

The list shows the addresses, source line numbers and instructions executed. The trace starts at the entrance to the function `f`.

SEE ALSO

traceh define, traceh format, traceh start, traceh status, traceh stop

6.2.19 HP64772/8 TRACEH STATUS - display current status of emulator trace

SYNTAX

traceh status

DESCRIPTION

This command displays the current definition and status of the emulator's trace. The first item displayed is the trace status. Status may be: **running, stopped** or **complete**. The next items displayed are the current value of the trace format and the trace define.

EXAMPLES

(dbug) `traceh status`

```
Hardware trace is stopped
The hardware trace definition is:
    traceh define stat==0 vpc
 The hardware trace format is:
    traceh format abus:x
```

The trace is stopped because a **traceh stop** command was issued or the trace buffer is full. A previous **traceh define** command specified that only entries with 0 in the status field should be buffered. Additionally, only **VPC** cycles are traced (as set by a previous **traceh define** command). Other possibilities are either **bus** or **all**. When the buffer is displayed (using the **traceh list** command), the **abus** is displayed for each entry.

SEE ALSO

traceh define, traceh format, traceh start, traceh status, traceh stop

6.2.20 HP64772/8 TRACEH STOP - stop hardware trace

SYNTAX

traceh stop [**at** *address*]

DESCRIPTION

This command stops the emulator's trace if it is currently in progress. Executions following the **traceh stop** command do not affect the trace buffer, unless a **traceh start** command is issued. If an address is specified, the trace is initialized and started. The trace will stop during execution, when the executed address matches the specified address. The line numbers will start from -512. Line number zero will be the last number entered. All previously invoked traces stop before this trace begins. Only one start or stop address can be specified at one time. All previously issued start and stop addresses are ignored. Trace buffer contents are displayed (using the **traceh list** command) according to the format currently specified by the **traceh format** command.

Note that the **traceh stop** command is affected by the *traceh_mode*.

EXAMPLES

```
(dbug) traceh stop

(dbug) traceh status

Hardware trace is stopped
The hardware trace definition is:
    traceh define all
The hardware trace format is:
    traceh format abus:x dbus:x absolute
(dbug) traceh stop
traceh is not running
```

The first command stops the trace mechanism, so that no more entries are buffered. This is demonstrated by the output of the **traceh status** command. Finally, issuing the **traceh stop** command confirms that the trace has been halted.

The following example will demonstrate how a hardware trace is stopped when a function `g` is called. Currently the hardware trace is stopped.

```
(dbug) list 52,66

     52        j = 1;
     53        i = 10;
     54        j = 0;
     55        i = 11;
     56        j = -1;
     57        g();
     58    }
     59
     60    g()
     61    {
     62    int k,m;
     63
     64        k = 1;
     65        m = 9;
     66        k = 2;

(dbug) traceh stop at &g

(dbug)  print &g
  0xe0bf

(dbug) traceh status

Hardware trace is running
The hardware trace definition is:
    traceh define all
The hardware trace format is:
    traceh format abus:x, dbus:x, absolute
The hardware trace stop address is: 0xe0bf
```

The status shows that the trace is now running, and that the stop address is 0xe0bf, matching the entrance point to the function `g`. Note that when an address is specified for the **traceh stop** command, it implies that the hardware trace should be started (as can be seen in the output of the **traceh status** command). Entries, however, will not be entered to the trace buffer after the function `g` is entered.

```
(dbug) run
  execution completed

(dbug) traceh format relative lines disasm
```

(dbug) `traceh list -10,0`

```
line  count
----  -----
-10   0.200 uS
-9    0.160 uS
-8    0.040 uS  VPC  "t_trace.c":55  f+0x44 : addr @0xb, -0x4(fp)
-7    0.200 uS
-6    0.160 uS
-5    0.240 uS  VPC  "t_trace.c":56  f+0x48 : movqd -0x1, -0x8(fp)
-4    0.120 uS
-3    0.120 uS  VPC  "t_trace.c":57  f+0x4b : bsr g
-2    0.080 uS
-1    0.240 uS
 0    0.280 uS  VPC  "t_trace.c":61  g       : enter [], 0x8
```

The example shows that lines 55, 56 and 57 were executed just before function `g` was entered at line 61. `g` is entered.

SEE ALSO

traceh define, traceh format, traceh start, traceh status, traceh stop

6.2.21 HP64772/8 UNMAP - delete an emulator map term

SYNTAX

unmap [*number* | **all**]

DESCRIPTION

The **unmap** command deletes a specified emulator *map term*.

Number is the number of the *map term* as displayed by the **map** command.

The **unmap** command can redefine memory blocks (presently defined as part of emulation memory) as target memory.

EXAMPLES

(dbug) `map`

```
[1] 0x00000000..0x00000fff emulator ram
[2] 0x00001000..0x00002fff emulator rom
[3] 0x00008000..0x00008fff emulator rom
[4] 0x00030000..0x00030fff emulator ram
```

(dbug) `unmap 2`

```
[1] 0x00000000..0x00000fff emulator ram
[2] 0x00008000..0x00008fff emulator rom
[3] 0x00030000..0x00030fff emulator ram
```

The second map term is deleted. Note that the term numbers for the remaining terms are changed accordingly.

SEE ALSO

map

6.3 The HP64779 Emulator

6.3.1 Invocation

There are two ways to invoke DBUG's interface with the HP64779 In-System Emulator. One is through the invocation line, and the other is through the **connect** command. Both approaches require that you specify the **mon** parameter as **isecg16** **isefx16**, **isecg160** or **isefx164**

A sample invocation line is as follows:

```
dbug -mon isecg16 -fpu 381 -l tty4 a.out
```

This invocation line invokes DBUG, automatically connects to the serial line `tty4` (the **-l** parameter), and expects to find the HP64779 Series In-System Emulator (the **-mon** parameter), with an NS32381 FPU. The executable file is `a.out`.

If the required parameters do not appear in the invocation line, the **connect** command may be used as follows:

(dbug) `connect link tty4 with fpu 381 mon isecg16`

This command yields the same results as the previous example.

6.3.2 Condition option

Several of the commands relevant in this context allow for an optional specification of a condition. The condition in these cases is restricted to the following:

pin_group {**==** | **!=**} {*range* | *value*}

or

(*pin_group* {**==** | **!=**} {*range* | *value*}) {**and** | **or**} (*pin_group* {**==** | **!=**} {*range* | *value*})

Where:

1) *pin-group* is one of the labels described in Table 6-1.

2) *value* is any expression that can be expressed as a numeric value.

3) *range* is an expression of the type: *value .. value.*

Note that only one range definition may be used at any one time. Respecifying an existing range will cause the existing range to be overwritten.

6.3.3 HP64779 BREAKH - stop execution

SYNTAX

breakh

DESCRIPTION

The **breakh** command issues a break to the monitor, causing it to stop executing the user program and to report its position. If the emulator is in the reset state when the **breakh** command is issued, it will be released from reset.

EXAMPLE

(dbug) `run`

```
<control c>
<BREAK>
```

(dbug) `breakh`

```
emulator run aborted
stopped in main at line 1 in file "loop.c"
1 main()  { int i,j; for (i=1;i<100000000;i++) { j=10; } }
```

6.3.4 HP64779 CONFIGH - set configuration parameters

SYNTAX

configh [[**mon bg** | **mon fg** *address* [**lock**]]
[**clock** {**inter** [*number*] | **exter**}]
[**nmi** {**enable** | **disable**}]
[**hold** {**enable** | **disable**}]
[**realtime** {**enable** | **disable**}]
[**cfg** *number*]
[**mod** *address*]
[**sp0** *address*]
[**dma** {[**u** | **s**] [**u** | **s**]}]
[**excp** {**all** | **none** | *exception* }]
[**target** {**piped** | **buffered**}]

DESCRIPTION

When no parameters are specified, the current configuration is given.

mon bg | **mon fg** *address* specifies the type of monitor. **mon bg** specifies a background monitor. **mon fg** specifies a foreground monitor starting at *address*. The address is aligned by the emulator to a 4k byte boundary. The **lock** option specifies that references to the location of the fg monitor in emulation memory will be based on target board memory signals rather than emulation memory signals.

clock inter | **exter** determines internal or external emulator clock source selection. The **inter** option selects the internal clock. The internal clock can be configured for different frequencies by specifying an appropriate number. The emulator supports a 30, 40 and 50 MHz clock. By default, the NS32CG16 emulator uses an internal 30 MHz clock, the NS32FX16 and NS32CG160 emulators uses an internal 50 MHz clock. The **exter** option selects the clock provided by the target system.

nmi enable | **disable** controls the target system signal NMI configuration.

hold enable | **disable** controls the target system signal HOLD configuration.

realtime enable | **disable** controls the restriction to real time runs. When disabled, the emulator allows the user to issue commands that effect its application timing, while the application is running, e.g. print cpu registers.

cfg *number* sets the cfg register when the CPU is reset by the emulator. *number* must be an 8 bit value. If no *number* is specified the cfg will not be initialized.

mod *address* sets the mod register when the CPU is reset by the emulator. *address* must be a 16 bit value. The default value is fff0 hex.

sp0 *address* sets the sp0 register when the CPU is reset by the emulator. *address* must be a 32 bit value. If no *address* is specified the sp0 register will not be initialized.

target piped | buffered selects the target interface mode.

excp controls the CPU exception setup. The dispatch table is modified so that the occurance of specified exceptions will result in an entry into the foreground monitor. The setup is based on the current exception mode (direct or indirect) and the contents of the intbase and mod registers. This option is only valid when the foreground monitor is used. *exception* is either one of the following names: NVI, NMI, RESERVED, SLAVE, ILL, SVC, DVZ, FLG, BPT, TRC, UND; or a number 0 through 14. **all** specifies that all exceptions will result in an entry into the foreground monitor. **none** specifies that no exceptions will be handled by the foreground monitor. Note however that the BPT and TRC exceptions are always used by the foreground monitor for breakpoints and single stepping.

dma allows you to define the User/Supervisor signal to the emulation memory during DMA transfers for the NS32CG160 CPU. This signal is not defined during DMA transfers, but may be defined by the target system for target system memory. For each DMA channel, specifying **u** defines a User signal, specifying **s** defines a Supervisor signal. Note that the User/Supervisor signal seen by the target is determined by the processor, and is not affected by the configuration.

EXAMPLES

```
(dbug) configh
        monitor
foreground, address 0x30000
        clock           internal
        realtime        disabled
        excp            none
        target          piped
        cfg             0x2
        sp0             0x30c80
        mod             0xfff0
        nmi             enabled
        hold            enabled
(dbug) configh mon bg excp svc
        monitor         background
        clock           internal
        realtime        disabled
        excp            5
        target          piped
        cfg             0x2
        sp0             0x30c80
        mod             0xfff0
        nmi             enabled
        hold            enabled
```

SEE ALSO

connect

6.3.5 HP64779 CONNECT - connect to a system emulator

SYNTAX

connect [**link** *linkname* | **node** *nodename*][**with** [**nofast**][**list**][**baud** *number*] [**cpu** *name*][**mmu** *name*][**fpu** *name*][**mon monname**]]

DESCRIPTION

The **connect** command is used to switch to the *remote* operation mode. **Connect** selects the communications channel through which DBUG and the target board communicate, and sets configuration parameters. The **connect** parameters are closely related to the DBUG invocation line flags. Several **connect** commands may be issued during one DBUG session to alter the configuration or communication parameters.

Once the **connect** command has been issued for the first time, and the connection established, the **connect** command can be issued any number of times to toggle the **list** and **nofast** parameters. These must be specified each time you want them enabled. Issuing **connect** without the parameters will disable them. This situation is different from that of other parameters, for which the previously specified value is in effect unless otherwise specified. When **connect** is issued for the second time it terminates the current connection and establishes a new connection.

monname must be one of the following: isecg16, isecg160, isefx16, isfx164.

At least one parameter must be specified when the **with** clause is used.

Link *linkname* identifies the serial communication line between the host and the target board. The default *linkname* is the last name given by the previous **connect** command, or as selected in the invocation line (**-l** parameter), or as specified in the GNX target specification file.

When working with the emulator, DBUG supports two types of link interfaces:

1. RS232 link. In this case the *linkname* should have the form `tty`*xx*, where *xx* is the link identifier (i.e. the device would be: `/dev/tty`*xx*).
2. The HP64037 high-speed RS422 interface card (referred to as HP COMCARD). In this case the *linkname* should have the form `com`*xx*. `Com`*xx* is the name of the IBM PC port in which the HP COMCARD is installed (i.e. `connect link com6`). The HP COMCARD can be used only with SYS32/30 or SYS32/50 hosts.

The **node** parameter selects the fast communication channel (LAN) between DBUG and the emulator. *nodename* is the name of the emulator, as recognized by the host system. The default *nodename* is either the last name given by the previous **connect** command, the name selected in the invocation line, or the name specified in the target specification file.

Dbug uses the intel-hex executable file format to download the executable file when working with the emulator.

In the **fast** protocol (the default value), DBUG uses the *HP X* Binary file format and transfer protocol when downloading an executable file to a target system. The **fast** protocol should be used with the HP COMCARD, to achieve high baud rates. Specifying **nofast** causes DBUG to use the *Intel-hex* file format downloading the executable file to a target system. The **nofast** option can be used when the serial communication line is unreliable.

The **list** option enables the verbose communication mode. In this mode, DBUG displays the messages exchanged by the debugger and the monitor or the emulator.

The command:

(dbug) `connect with list`

may be used to enter the verbose communication mode, even if the **connect** command was previously issued. An additional **connect** command with no parameters disables the verbose mode.

Baud *number* sets the communication baud rate for *stand-aside mode*. Possible values for tty communication are: 300, 600, 1200, 2400, 4800, 9600, 19200, and 38400. Possible values for HP COMCARD communication are: 300, 1200, 2400, 4800, 9600, 19200, 38400, 57600, 115200 and 230400. The default baud rate is 9600.

Cpu specifies the target board CPU name. Valid names are cg16, cg160, fx16 and fx164.

Mmu specifies the target board MMU name. If this parameter is specified it must be as **nommu**.

Fpu specifies the target board floating-point unit name. Valid names are: 081, 381, **nofpu**.

The default values for the **cpu**, **mmu** and **fpu** options are described by your GNX target setup file *.gnxrc*, the previous **connect** command, or the invocation line.

If the emulator is plugged into a target that has ROM at address 0, you should first initialize the target and only then issue the **connect** command. For example, if the emulator is plugged into NSCG16ED evaluation board, you should press the RESET button on the evaluation board before issuing the **connect** command.

When DBUG is initialized, no communication link is defined. Therefore, the user must select a link before issuing any debugger command that communicates with the target board.

NOTE: Although DBUG allows you to change links during the debugging session, this exercise is not recommended.

EXAMPLES

(dbug) `connect`

Reconnects the target system using the parameters selected by the previous `connect` command. The **list** and **nofast** parameters are disabled because they are not specified. The **fast** protocol will be used for downloading the executable file, and the communication between DBUG and the monitor will not be displayed.

(dbug) `connect link tty7 with baud 38400 mon isecg160`

Selects the channel `tty7`, and communicates with the emulator over a 38400 baud rate serial line. In this case the ISE should be reset to work at 38400 baud. This is done by setting the dip-switches on the back of the emulator to the correct value and turning the emulator on.

(dbug) `connect link com4 with mon isecg160 baud 230400`

Selects your HP COMCARD installed as port COM4 in your IBM PC as your communication channel. The connection is established with the emulator using 230400 baud rate and the RS422 protocol provided by the emulator and the HP COMCARD. The load protocol will be **fast** since the **nofast** parameter is not specified.

(dbug) `alias do_connect (linkname) 'connect link linkname with baud 38400 cpu 532 fpu 381 mon isecg160'`

Defining an alias/macro makes the connect command easier to use. Issuing the following command selects `tty4` as your communication channel. This macro may be inserted in your *.dbuginit* (*dbgini* in VMS/MS-DOS) file so that it will be recognized each time DBUG is entered.

(dbug) `do_connect (tty4)`

(dbug) `connect node ise01 with mon isefx164 cpu fx164`

Selects, and connects to, the ISE whose Ethernet name is `ise01` for use with the NS32FX164.

SEE ALSO

begin, load

6.3.6 HP64779 COUNTER DEFINE - counts time or events

SYNTAX

counter define {**none** | **time** | *condition*}

DESCRIPTION

The **counter define** command qualifies the emulator's counter to count **time** or *events*.

When counting **time**, the counter acts as a clock. When counting *events*, the counter counts the number of times the specified *condition* is true. Counting begins from zero. The counter can only be incremented (cannot be decremented).

The counter is invoked by issuing the **traceh start** command and can be viewed with the **traceh list** command, in either absolute or relative formats. The complete procedure for viewing the counter is discussed in the description of the **traceh format** command.

The default setting for the counter is **none**. The counter is not active during this setting.

For more information, refer to the *HP64779 Emulator Terminal Interface: Emulator User's Guide*.

EXAMPLES

(dbug) `counter define time`

Defines the counter to count time.

(dbug) `counter define (abus == &f) and (stat == 9)`

This command causes the counter to count all the events in which the address of function `f` is on the address bus and the status equals 9 (corresponding to a non-sequential fetch).

SEE ALSO

counter status, traceh start, traceh format, traceh define

6.3.7 HP64779 COUNTER STATUS - print counter qualification

SYNTAX

counter status

DESCRIPTION

This command prints the counter's current qualification. This may be **none**, **time** or a *condition*.

EXAMPLES

(dbug) `counter status`

```
The counter definition is:
counter define (abus == 57451) and (stat == 0x9)
```

This example indicates that the counter was defined to be incremented each time the condition: `(abus == 57451) or (stat == 9)` is true. The number 57451 is the address of the function `f()` used in the example for the **counter define** command.

SEE ALSO

counter define, traceh format, traceh define, traceh start

6.3.8 HP64779 LOADMON - load a foreground monitor

SYNTAX

loadmon *filename*

DESCRIPTION

This commands allows you to load your private version of the foreground monitor into the emulator.

The **configh** command can then be used to specify the foreground monitor. For more details on modifying and compiling a foreground monitor, refer to the *HP64779 Emulator Terminal Interface: Emulator User's Guide*. *filename* is the name of the executable foreground monitor file.

The symbolic information of a debugged program is lost when a new foreground monitor is loaded. The **load** command can be used to reread this information. Using the **loadmon** command at the beginning of your session can avoid the need to reread the symbolic information.

EXAMPLES

(dbug) `loadmon my_mon`

Specifies that `my_mon` is loaded into the emulator as a monitor program.

(dbug) `configh mon fg 0x30000`

Specifies that `my_mon` is used as the foreground monitor, mapped to address 30000 hex.

(dbug) `load my_prog`

The debugged program `my_prog` is now loaded using the new foreground monitor.

SEE ALSO

configh, load

6.3.9 HP64779 MAP - map emulation memory

SYNTAX

map [*range* [**rom** | **ram** | **trom** | **tram**] [**lock** | **wait** *number*] [**with copy**]]

DESCRIPTION

This command maps the specified address range to the emulator's memory. *range* is an address within the range 0..0xfffefff. Memory accesses in the IO range of 0xffff0000..0xffffffff will result in internal processor cycles and are not affected by the memory map. Therefore memory commands may be used to access the IO registers. Addresses in the range 0x1000000..0xfffefff will be truncated to a 24-bit address before being used.

Memory blocks can be referred to as either part of the target or emulation memory. Memory blocks that are mapped to emulation memory may be characterized as either RAM or ROM. All memory is referred to as target RAM immediately after the emulator is powered-up. The **map** command specifies which of the target memory address ranges should be referred to as emulation memory, and whether they should be characterized as RAM or ROM.

Memory blocks that have been referred to as part of the emulation memory can be redefined as part of the target memory with the **unmap** command.

If **rom** is specified, the *range* is treated as part of a ROM (no write to target ROM is allowed) and mapped to emulator memory. If **ram** is specified, the **range** is treated as part of a RAM. If **tram** is specified, the *range* is treated as target RAM. If **trom** is specified, the *range* is treated as target ROM.

If none of the options **ram**, **rom**,**tram** or **trom** are specified, the default is **ram**.

If **with copy** is specified, the contents of the address range in the target memory are copied to the same location in the emulation memory. Otherwise, the emulator memory block might not have the same contents as the corresponding target memory block.

To set software breakpoints on part of the program that lies in target ROM, first map the relevant address range to emulation memory, characterize it as **ram**, and specify **with copy**.

DBUG assigns a *map term* to each memory block mapped to emulation memory. Each term consists of the term number, the mapped address range, and its RAM/ROM status. Map terms are displayed each time terms are added or

deleted, or when the **map** command is issued without parameters. The term numbers will be rearranged in ascending order each time a new term is added or deleted. The HP64779 Emulator supports 15 map terms and 512 Kbyte emulation memory.

If *range* is not specified, the mapping of all mapped address ranges is displayed.

Additional attributes can be specified for the emulation memory terms. **lock** determines whether the access to the selected address range is according to target system timing or emulation memory timing (default). The **wait** *number* may be used to specify the number of wait states the defined memory should emulate. The number of wait states may be 0, 1, 2 or 3. Note that the emulation memory containing the foreground monitor (if used) is confgured seperately with the **configh** command.

Since the **map** command resets the emulation CPU, it is recommended that this command not be used once the debugging of your program has started.

EXAMPLES

(dbug) `map`

```
[1] 0x00000000..0x00000fff emulator ram lock
[2] 0x00001000..0x00002fff emulator rom nolock
[3] 0x00030000..0x00030fff emulator ram lock
```

Prints the current map. The ranges are in hexadecimal format. Term [3] is the memory block used for the foreground monitor. It is set by DBUG when the connection with the emulator is established.

(dbug) `map 8830..8840 rom wait 2`

```
[1] 0x00000000..0x00000fff emulator ram lock
[2] 0x00001000..0x00002fff emulator rom nolock
[3] 0x00008000..0x00008fff emulator rom wait 2
[4] 0x00030000..0x00030fff emulator ram lock
```

Maps the block containing the specified address range to emulator memory. The address range is considered part of the target ROM. This memory range will now emulate two wait states. The contents of the address range on the target board are not copied to the emulator's memory. Note that the address range is rounded to 4-Kbyte boundaries.

```
(dbug) map 0x9000..0x9fff trom
[1] 0x00000000..0x00000fff emulator ram
[2] 0x00001000..0x00002fff emulator rom
[3] 0x00008000..0x00008fff emulator rom
[4] 0x00009000..0x00009fff target rom
[5] 0x00030000..0x0003fff emulator ram
```

Maps the block containing the specified address to target ROM.

SEE ALSO

unmap, configh

6.3.10 HP64779 RESETH - reset CPU

SYNTAX

reseth

DESCRIPTION

The **reseth** command resets the emulation CPU. This causes certain registers to be reset to zero, and others to be reset to an undefined value. Executing this command also places the emulator in a reset state. You must return system control to the monitor before attempting to change or print registers. Do this by issuing the **breakh** command.

6.3.11 HP64779 STOPH - set a hardware breakpoint

SYNTAX

stoph if *stop_condition*

DESCRIPTION

This command uses the emulator's hardware to force your program to halt execution after the *stop_condition* becomes true. *Stop_condition* is:

pin_group {== | !=} {*value* | *range*}

Only one range definition may be used at one time for all of the emulator commands. In addition, up to 4 hardware breakpoints may be defined with this command. Each defined hardware breakpoint is assigned an event number. Use the **status** command to see a list of the currently defined events.

The hardware trace begins automatically when a hardware breakpoint is defined. DBUG activates the hardware trace in order to keep the hardware breakpoints active. You may use the **traceh stop** command to stop the hardware trace and disable all the currently defined hardware breakpoints. The hardware breakpoints will become active again when you issue the **traceh start** command or define another hardware breakpoint.

If execution stops because of a hardware breakpoint, the trace buffer contains up to 512 entries, stored just before execution was halted. DBUG will restart the hardware trace when you resume execution, in order to keep the hardware breakpoints active.

The **traceh start at** and **traceh stop at** commands may not be used when a hardware breakpoint is defined. To delete a hardware breakpoint, you must delete the event associated with it. Use **status** to see a list of the currently defined events, and then use **delete** to delete the appropriate event(s).

Note that hardware breakpoints are sensitive to *traceh_mode* as defined with the **traceh define** command.

EXAMPLES

(dbug) `stoph if abus == &array..&array + array_size`

Causes execution to halt when the array `array` is referenced. `array` and `array_size` are assumed to be variables defined in your program.

(dbug) `stoph if us==1`

Causes execution to halt when execution mode is switched to supervisor mode.

(dbug) `status`

The status command displays the previously defined hardware breakpoints with the matching event numbers. These appear here:

```
[1] stoph if abus==array..array+array_size
[2] stoph if us==1
```

(dbug) `traceh status`

```
Hardware trace is running
The hardware trace definition is:
      traceh define all
The hardware trace format is:
      traceh format abus:x dbus:x
```

(dbug) `delete 1`

Deletes the first hardware breakpoint

(dbug) `status`

```
[2] stoph if us==1
```

This shows that the first hardware breakpoint has been deleted.

SEE ALSO

stop, traceh start, traceh stop, traceh status, status, delete

6.3.12 HP64779 TRACEH DEFINE - define hardware trace

SYNTAX

traceh define [*condition*] [**all**]

DESCRIPTION

This command defines which cycles of the specified type will be stored in the trace buffer.

Condition specifies which cycles should be stored in the trace buffer. Cycles which match the condition are stored. The default is to match all cycles.

The **traceh define** command cannot be issued if a trace is already in progress. The trace must first be stopped with the **traceh stop** command.

The **traceh list** and **traceh format** command should be used to view the accumulated trace buffer entries. However, if non-sequential fetches are ignored (e.g., by the command `traceh define stat!=9`), the **traceh list** command will not display disassembly of instructions when the traceh format is specified as mnemonic, disasm or lines.

EXAMPLES

(dbug) `traceh define abus==0x90ff`

Defines that only trace entries with the address values 90ff (hexadecimal) are stored (and therefore, displayed later).

In the following example, only the changes to the variables `i` and `j` defined in `main` will be of interest. The executable program is assumed to be loaded to memory.

(dbug) `list 1,20`

```
1   main()
2   {
3       int i=0;
4       int j=10;
5
6       i = 1;
7       j = 9;
8       i = 2;
9       j = 8;
```

```
10        i = 3;
11        j = 7;
12        i = 4;
13        j = 6;
14        i = 5;
15        j = 5;
16        i = 6;
17        j = 4;
18        i = 7;
19        j = 3;
20        i = 8;
21        j = 2;
22        i = 9;
23        j = 1;
24        i = 10;
25        j = 0;
26        i = 11;
27        j = -1;
28        f();
29    }
```

A breakpoint is specified, so that execution will be stopped in the function `main`. (`i` and `j` are local variables of the function `main`).

(dbug) `stop in main`

```
[1] stop in main
```

(dbug) `run`

```
[1] stopped in main at line 3 in file "t_trace.c"
      3 int i=0;
```

Next, the addresses of `i` and `j` are printed.

(dbug) `print &j`

```
0x24ff0
```

(dbug) `print &i`

```
0x24ff4
```

As the addresses are adjacent, the following definition will capture all the accesses to these memory locations in user mode.

(dbug) `traceh define (abus==&j..&i+3) and (us==1)`

Since these variables are allocated on the stack, these addresses are only of interest while we are in function `main`. The following hardware trace definition will solve this problem:

(dbug) `traceh stop at &f`

Now, continue execution

(dbug) `cont`

```
execution completed
```

It is now easy to follow the values assigned to the variables `i`, `j`, and to compare them to the source listing previously displayed. The value `24ff4` in the `abus` column is the address of the variable `i` and the value `24ff0` is the address of `j`. the values assigned to these variables are the values displayed in the `dbus` column.

(dbug) `traceh format abus:x dbus:x`

(dbug) `traceh list -24,-15`

```
line  abus      dbus
----  ----      ----
 -24  024ff4    0006
 -23  024ff6    0000
 -22  024ff0    0004
 -21  024ff2    0000
 -20  024ff4    0007
 -19  024ff6    0000
 -18  024ff0    0003
 -17  024ff2    0000
 -16  024ff4    0008
 -15  024ff6    0000
```

The format can be changed to allow a different view of the same information, for example:

(dbug) `traceh format mnemonic`

(dbug) `traceh list -24,0`

```
line  mnemonic
----  --------
 -24  0006  usr data     write
 -23  0000  usr data     write
 -22  0004  usr data     write
 -21  0000  usr data     write
 -20  0007  usr data     write
 -19  0000  usr data     write
 -18  0003  usr data     write
 -17  0000  usr data     write
 -16  0008  usr data     write
 -15  0000  usr data     write
 -14  0002  usr data     write
 -13  0000  usr data     write
 -12  0009  usr data     write
 -11  0000  usr data     write
 -10  0001  usr data     write
  -9  0000  usr data     write
  -8  000a  usr data     write
  -7  0000  usr data     write
  -6  0000  usr data     write
  -5  0000  usr data     write
  -4  000b  usr data     write
  -3  0000  usr data     write
  -2  ffff  usr data     write
  -1  ffff  usr data     write
   0     f     :         enter    [], 0x8
```

Note that the trace was stopped when the function `f` was about to be entered.

SEE ALSO

traceh format, traceh start, traceh status, traceh stop

6.3.13 HP64779 TRACEH FORMAT - define trace display format

SYNTAX

traceh format [*pin_group:radix ...*] [**absolute** | **relative**] [**mnemonic**]

DESCRIPTION

The *pin_group* parameter selects any of the following labels which correspond to different channels (pin-groups) as found in Table 6-1: **abus**, **dbus**, **stat**, **ddin**, **us**, **bpu**, **spc**, **ias**, **hbe**, **dak**, **pfs**. The counter may also be incorporated into the trace listing. This can be done by specifying either **absolute** or **relative** format. The absolute format displays the cumulative value of the counter (for all trace lines). The relative format displays the counter's accumulated value per trace line.

The trace listing columns include pin values that correspond to the selected labels in the specified format.

If **mnemonic** is specified, all the other specified columns are ignored and a verbal description of their contents is given instead. This description includes the description of the execution cycles. When activated, DBUG defines the initial trace format as mnemonic.

EXAMPLES

(dbug) `traceh format abus:x dbus:d`

Specifies that the address bus (**abus**) is displayed in hexadecimal format, and the data bus (**dbus**) is displayed in decimal format. No other information is displayed.

(dbug) `traceh format disassm lines relative`

Specifies that for lines corresponding to execution cycles, the disassembly of the executed instruction is displayed. The corresponding source file and source line number are also displayed; and for each line in the trace buffer, the counter is displayed in its relative format.

SEE ALSO

traceh define, traceh start, traceh status, traceh stop

6.3.14 HP64779 TRACEH LIST - display trace buffer

SYNTAX

traceh list [*number* | *number*, *number*]

DESCRIPTION

This command displays the trace on the screen.

If *numbers* are specified, only those entries within the specified range are displayed. If one number is specified, the current line and the few lines following it are displayed. If no number is specified, the next few lines are displayed.

The trace is displayed in a format defined by the **traceh format** command. The **traceh format** command makes it possible to redisplay the trace buffer according to different formats without having to resume execution.

EXAMPLES

```
(dbug) traceh list 0,22

line  mnemonic
  ----  --------
        ---- start of list ----
     0  0400  sup fetch:seq
     1  108f  sup fetch:seq
     2  e000  sup data     read
     3  0000  sup data     read
     4  a252  sup fetch:seq
     5  0000  sup data     read
     6  0300  sup data     read
     7  4100  usr data     read
     8  0000  usr data     read
     9    start    :   jsr      main
    10  00c0  usr fetch:seq
    11  10e0  usr fetch:seq
    12*   start+0x6  :  movd    r0, tos
    13*   start+0x8  :  jsr     exit
    14  00c0  usr fetch:seq
    15    main    :     enter   [], 0x8
    16    main+0x3  :   movqd   0x0, -0x4(fp)
    17  e006  usr data     write
    18  0000  usr data     write
    19  7cc0  usr fetch:seq
```

```
20   0000   usr data      write
21   0000   usr data      write
22     main+0x6   :   addr     @0xa, -0x8(fp)
```

This command displays the first 22 lines of the trace buffer.

Note that lines corresponding to fetches aligned to instructions are disassembled (lines 9, 11, 12, 15, 16, 22). DBUG also concluded that lines 12 and 13 correspond to a sequential fetch that was not executed, and therefore these lines are marked with a '*'.

It is possible to change the display format of the buffer. The following definition will instruct DBUG to display only disassembly and source information for the relevant lines in the buffer.

(dbug) `traceh format lines disasm relative`

(dbug) `traceh list 0,17`

```
line  count
----  -----
      ---- start of list ----
   8   0.160 uS
   9   0.520 uS       start     :      jsr     main
  10   0.160 uS
  11   0.160 uS
 12*   0.160 uS       start+0x6  :     movd     r0, tos
 13*   0.160 uS       start+0x8  :     jsr      exit
  14   0.160 uS
  15   0.520 uS     "t_trace.c":2  main     :  enter   [], 0x8
  16   0.160 uS     "t_trace.c":3  main+0x3  :  movqd   0x0, -0x4(fp)
  17   0.160 uS
  18   0.160 uS
  19   0.520 uS
  20   0.160 uS
  21   0.160 uS
  22   0.160 uS     "t_trace.c":4  main+0x6  :  addr @0xa, -0x8(fp)
```

SEE ALSO

traceh format, traceh start, traceh status, traceh stop

6.3.15 HP64779 TRACEH RESET - reset trace definitions

SYNTAX

traceh reset

DESCRIPTION

This command resets the trace and counter definitions to their initial settings. Any active trace is halted. The trace definition is reset to its default value, which specifies that all cycles should be stored.

The trace format is also reset to its default value (mnemonic) and counter definition is reset to **none**.

EXAMPLES

The following command sequence demonstrates the effect of the **traceh reset** command. First the trace status is displayed by the **traceh status** command, which shows the current value of the **traceh define** and **traceh format** commands. Next the **traceh reset** command gives new (default) values to the hardware trace definition and format. These values are displayed by re-issuing the **traceh status** command. Finally, the fact that the buffer is now empty is demonstrated by issuing the **traceh list** command.

```
(dbug) traceh status

Hardware trace is stopped
The hardware trace definition is:
    traceh define stat==8
hardware trace format is:
    traceh format abus:x ddin:d

(dbug) traceh reset

(dbug) traceh status

Hardware trace is stopped
The hardware trace definition is:
    traceh define all
The hardware trace format is:
    traceh format mnemonic
```

```
(dbug) traceh list

line  abus      dbus      count
----  ----      ----      -----

  ** Trigger not in memory **
```

SEE ALSO

traceh define, traceh format, traceh start, traceh status, traceh stop

6.3.16 HP64779 TRACEH START - start hardware trace

SYNTAX

traceh start [**at** *address*]

DESCRIPTION

This command starts the emulator's trace function.

Issuing this command first causes the trace buffer to be reinitialized (all current entries are removed). Further, all cycles that match the trace definition are stored in the emulator's trace buffer once execution resumes.

If an address is specified, trace will start when that address is first fetched by the CPU. All previous addresses specified with a **traceh start** or **traceh stop** command are ignored.

The trace buffer contents are displayed using the **traceh list** command according to the format currently specified by the **traceh format** command. The buffer size is 1024 entries. Tracing is suspended when the buffer is full. Clear the buffer using the **traceh reset** command, or by re-issuing the **traceh start** command. A trace in progress should be stopped with the **traceh stop** command, before using the **traceh start** command.

Note that the trace trigger point is affected by the *traceh_mode*.

EXAMPLES

(dbug) `traceh start`

(dbug) `traceh list`

```
line  abus      dbus      count
----  ----      ----      -----

  ** Trigger not in memory **
```

The command resets the trace buffer contents. Only new entries are buffered once execution resumes.

The following example will start a hardware trace when the first instruction of a function `f` is executed. First, select only those cycles corresponding to a sequential or non-sequential fetch to be stored.

(dbug) `traceh define (stat==8) or (stat==9)`

Next, define the counter to count the sequential fetches.

(dbug) `counter define stat==8`

(dbug) `traceh start at f`

(dbug) `rerun`

```
loading...
loaded 0 bytes of code, 3556 bytes of data
total of 3556 bytes_loaded

execution completed
```

(dbug) traceh list 0,25

```
line  count
----  -----
      ---- start of list ----
   0        ---
   1          1
   2          1
   3          0       start    :      jsr     main
   4          1
   5          1
   6*         1       start+0x6  :    movd    r0, tos
   7*         1       start+0x8  :    jsr     exit
   8          1
   9          0     "t_trace.c":2  main    :               enter[], 0x8
  10          1     "t_trace.c":3  main+0x3  :  movqd   0x0, -0x4(fp)
  11          1
  12          1     "t_trace.c":4  main+0x6  :  addr    @0xa, -0x8(fp)
  13          1
  14          1     "t_trace.c":6  main+0xa  :  movqd   0x1, -0x4(fp)
  15          1     "t_trace.c":7  main+0xd  :  addr    @0x9, -0x8(fp)
  16          1
  17          1     "t_trace.c":8  main+0x11 :  movqd   0x2, -0x4(fp)
  18          1
  19          1     "t_trace.c":9  main+0x14 :  addr    @0x8, -0x8(fp)
  20          1
  21          1     "t_trace.c":10  main+0x18 : movqd   0x3, -0x4(fp)
  22          1     "t_trace.c":11  main+0x1b : movqd   0x7, -0x8(fp)
  23          1
  24          1     "t_trace.c":12  main+0x1e : movqd   0x4, -0x4(fp)
  25          1     "t_trace.c":13  main+0x21 : movqd   0x6, -0x8(fp)
```

(dbug) traceh list 46,55

```
  46          1     "t_trace.c":26  main+0x4b : addr    @0xb, -0x4(fp)
  47          1
  48          1     "t_trace.c":27  main+0x4f : movqd   -0x1, -0x8(fp)
  49          1
  50          1     "t_trace.c":28  main+0x52 : bsr     f
  51          1
  52*         1     "t_trace.c":29  main+0x57 : exit    []
  53*         1       main+0x59 :   ret       0x0
  54*         1     "t_trace.c":32  f     :     enter   [], 0x8
  55          0     "t_trace.c":32  f     :     enter   [], 0x8
```

The list shows the addresses, source line numbers and instructions executed.

SEE ALSO

traceh define, traceh format, traceh start, traceh status, traceh stop

6.3.17 HP64779 TRACEH STATUS - display current status of emulator trace

SYNTAX

traceh status

DESCRIPTION

This command displays the current definition and status of the emulator's trace.

The first item displayed is the trace status. Status may be: **running**, **stopped** or **complete**. The next items displayed are the current value of the trace format and the trace define.

EXAMPLES

(dbug) `traceh status`

```
Hardware trace is stopped
The hardware trace definition is:
    traceh define stat==10
 The hardware trace format is:
    traceh format abus:x
```

The trace is stopped because a **traceh stop** command was issued or the trace buffer is full.

A previous **traceh define** command specified that only entries with 0 in the status field should be buffered. When the buffer is displayed (using the **traceh list** command), the **abus** is displayed for each entry.

SEE ALSO

traceh define, traceh format, traceh start, traceh status, traceh stop

6.3.18 HP64779 TRACEH STOP - stop hardware trace

SYNTAX

traceh stop [**at** *address*]

DESCRIPTION

This command stops the emulator's trace if it is currently in progress.

Executions following the **traceh stop** command do not affect the trace buffer, unless a **traceh start** command is issued. If an address is specified, the trace is initialized and started. The trace will stop during execution, when the address fetched by the CPU matches the specified address. The line numbers will be negative. Line number zero will be the last number entered.

All previously invoked traces stop before this trace begins. Only one start or stop address can be specified at one time. All previously issued start and stop addresses are ignored. Trace buffer contents are displayed (using the **traceh list** command) according to the format currently specified by the **traceh format** command.

Note that the **traceh stop** command is affected by the *traceh_mode*.

EXAMPLES

(dbug) `traceh stop`

(dbug) `traceh status`

```
Hardware trace is stopped
The hardware trace definition is:
    traceh define all
The hardware trace format is:
    traceh format abus:x dbus:x absolute
```

(dbug) `traceh stop`

```
traceh is not running
```

The first command stops the trace mechanism, so that no more entries are buffered. This is demonstrated by the output of the **traceh status** command. Finally, issuing the **traceh stop** command confirms that the trace has been halted.

The following example will demonstrate how a hardware trace is stopped when a function g is called. Currently the hardware trace is stopped.

```
(dbug) list 52,66

     52        j = 1;
     53        i = 10;
     54        j = 0;
     55        i = 11;
     56        j = -1;
     57        g();
     58    }
     59
     60    g()
     61    {
     62    int k,m;
     63
     64        k = 1;
     65        m = 9;
     66        k = 2;

(dbug) traceh stop at &g

(dbug)  print &g

  0xe0bf

(dbug) traceh status

Hardware trace is running
The hardware trace definition is:
    traceh define all
The hardware trace format is:
    traceh format abus:x, dbus:x, absolute
The hardware trace stop address is: 0xe0bf
```

The status shows that the trace is now running, and that the stop address is 0xe0bf, matching the entrance point to the function g. Note that when an address is specified for the **traceh stop** command, it implies that the hardware trace should be started (as can be seen in the output of the **traceh status** command). Entries, however, will not be entered to the trace buffer after the function g is entered.

```
(dbug) run

  execution completed

(dbug) traceh stop at &g

(dbug) rerun
  loading...
  loaded 0 bytes of code, 3556 bytes of data
  total of 3556 bytes_loaded

(dbug) traceh format mnemonic

(dbug) traceh list -50,1

  line  mnemonic
  ----  --------
   -50  df7c  usr fetch:seq
   -49  78c7  usr fetch:seq
   -48  c002  usr fetch:seq
   -47  0000  usr fetch:seq
   -46  9209  usr fetch:seq
   -45  1200  usr fetch:seq
   -44  8200  usr fetch:seq
   -43    f    :        enter   [], 0x8
   -42  0800  usr fetch:seq
   -41    f+0x3  :       movqd   0x1, -0x4(fp)
   -40    f+0x6  :       addr    @0x9, -0x8(fp)
   -39  09ae  usr fetch:seq
   -38    f+0xa  :       movqd   0x2, -0x4(fp)
   -37  7cc1  usr fetch:seq
   -36    f+0xd  :       addr    @0x8, -0x8(fp)
   -35  7808  usr fetch:seq
   -34    f+0x11 :       movqd   0x3, -0x4(fp)
   -33    f+0x14 :       movqd   0x7, -0x8(fp)
   -32  78c3  usr fetch:seq
   -31    f+0x17 :       movqd   0x4, -0x4(fp)
   -30    f+0x1a :       movqd   0x6, -0x8(fp)
   -29  78c3  usr fetch:seq
   -28    f+0x1d :       movqd   0x5, -0x4(fp)
   -27    f+0x20 :       movqd   0x5, -0x8(fp)
   -26  78c2  usr fetch:seq
   -25    f+0x23 :       movqd   0x6, -0x4(fp)
   -24    f+0x26 :       movqd   0x4, -0x8(fp)
```

```
-23  78c2  usr fetch:seq
-22    f+0x29 :       movqd   0x7, -0x4(fp)
-21    f+0x2c :       movqd   0x3, -0x8(fp)
-20  78c1  usr fetch:seq
-19    f+0x2f :       addr    @0x8, -0x4(fp)
-18  7c08  usr fetch:seq
-17    f+0x33 :       movqd   0x2, -0x8(fp)
-16    f+0x36 :       addr    @0x9, -0x4(fp)
-15  09ae  usr fetch:seq
-14    f+0x3a :       movqd   0x1, -0x8(fp)
-13  78c0  usr fetch:seq
-12    f+0x3d :       addr    @0xa, -0x4(fp)
-11  7c0a  usr fetch:seq
-10    f+0x41 :       movqd   0x0, -0x8(fp)
 -9    f+0x44 :       addr    @0xb, -0x4(fp)
 -8  0bae  usr fetch:seq
 -7    f+0x48 :       movqd   -0x1, -0x8(fp)
 -6  78c7  usr fetch:seq
 -5    f+0x4b :       bsr     g
 -4  0000  usr fetch:seq
 -3*   f+0x50 :       exit    []
 -2*   f+0x52 :       ret     0x0
 -1*   g    :         enter   [], 0x8
  0    g    :         enter   [], 0x8
  1  0800  usr fetch:seq
     ---- end of list ----
```

Note that lines -3, -2, -1 were fetched but not executed, and are therefore marked with a "*". function `g` was entered at line 61. `g` is entered.

SEE ALSO

traceh define, traceh format, traceh start, traceh status, traceh stop

6.3.19 HP64779 UNMAP - delete an emulator map term

SYNTAX

unmap [*number* | **all**]

DESCRIPTION

The **unmap** command deletes a specified emulator *map term*.

Number is the number of the *map term* as displayed by the **map** command.

The **unmap** command can redefine memory blocks (presently defined as part of emulation memory) as target memory.

EXAMPLES

(dbug) `map`

```
[1] 0x00000000..0x00000fff emulator ram wait 2
[2] 0x00001000..0x00002fff emulator rom
[3] 0x00008000..0x00008fff emulator rom wait 0
[4] 0x00030000..0x00030fff emulator ram lock
```

(dbug) `unmap 2`

```
[1] 0x00000000..0x00000fff emulator ram nolock
[2] 0x00008000..0x00008fff emulator rom wait 0
[3] 0x00030000..0x00030fff emulator ram lock
```

The second map term is deleted. Note that the term numbers for the remaining terms are changed accordingly.

SEE ALSO

map

6.4 The SPLICE Emulator

The SPLICE Development Tool provides a communication link between a Series 32000 target and a development system host. This connection allows you to download and map software onto target memory and then debug this software using DBUG. Currently DBUG supports the SPLICE board with the NS32CG16 Development Board.

6.4.1 Invocation

There are two ways to invoke DBUG interface with the SPLICE board: through the invocation line or by the **connect** command. Both approaches require that you specify the **mon** parameter as **spmon** and the **cpu** parameter as **cg16**.

A sample invocation line is:

```
dbug -mon spmon -cpu cg16 -fpu 381 -l tty4 a.out
```

This invocation line invokes DBUG, automatically connects it to the serial line `tty4` (the **-l** parameter), and specifies that the target system is NS32CG16 SPLICE with the NS32381 FPU. The executable file is `a.out`.

The **connect** command may be used as follows (see Section 5.1.8 for more details):

(dbug) `connect link tty4 with fpu 381 mon spmon cpu cg16`

This command yields the same results as the invocation example above.

6.4.2 SPLICE CONFIGH MON SB - setting monitor static base

SYNTAX

configh mon sb *address*

DESCRIPTION

This command allows you to set the monitor-time static base register of SPLICE to *address*. The register is set by default to 0. SPLICE uses 2 Kbytes for the module table scratch pad and interrupt stack, relative to the static base register. This command should be used before you down-load your program. *address* must be located in a 4-Kbyte boundary within the first 64-Kbytes of the address space.

The current value of the Monitor Static Base is displayed as part of the output of the **map** command.

EXAMPLE

(dbug) `configh mon sb 0x1000`

This example sets the Monitor Static Base register to 0x1000.

SEE ALSO

map, unmap

6.4.3 SPLICE MAP - map SPLICE memory

SYNTAX

map [*address* [**b** *n_blocks*] **p** *n_partition*]

DESCRIPTION

This command is used to map blocks from the target address space to the SPLICE memory.

address specifies the starting address of the target address space to be mapped to RAM. *address* must be at a block boundary and cannot overlap addresses already mapped.

The **b** *n_blocks* option specifies the number of blocks to be used in mapping. The default is 1 block. A block is the smallest mappable unit. Currently it is 64 Kbytes for SPLICE.

The **p** *n_partition* option specifies the partition number of the mappable SPLICE memory to which the target address space will be mapped. *N_partition* must be a number between 1 and the number of blocks in a partition.

If no parameter is given, **map** prints the SPLICE memory-mapping information (i.e., the sizes of starting partitions and blocks, how the SPLICE memory mapping is used). The address of the monitor static base is also displayed.

EXAMPLES

(dbug) map

```
SPLICE monitor static base address: 0x0
SPLICE Mapping Memory:
         Partition size = 0x10000   Block size = 0x10000
Partition            Mapped to target address space
    0                    0x0..0xffff
    1                    not used
    2                    0x300000..0x30ffff
    3                    not used
```

This example prints out the SPLICE memory-mapping information. Partition 0 is mapped to target address range 0x0 to 0xffff. Partition 2 is mapped to target address range 0x300000 to 0x30ffff. Partitions 1 and 3 are not mapped.

(dbug) `map 0x30000 b 1 p 1`

This example maps 1 block starting at 0x30000 into partition 1.

(dbug) `map 0x10000 p 3`

This example maps an address range starting from 0x10000,and of a size equal to the SPLICE memory partition, into partition 3.

SEE ALSO

unmap, configh

6.4.4 SPLICE UNMAP - unmapping SPLICE memory

SYNTAX

unmap *n*

DESCRIPTION

This command will undo the mapping for SPLICE memory partition *n*.

The target address space that is currently mapped into memory partition *n* will be mapped back to the target. Unmapping a partition necessary for the operation of SPLICE (*e.g.*, a partition containing the address space pointed to by the monitor static base register when the corresponding address space on the target board is not writable) is not allowed, and attempts to do so will result in an error message.

EXAMPLE

(dbug) `unmap 2`

This example unmaps partition 2.

SEE ALSO

map, configh

Appendix A

DBUG TUTORIAL FOR GRAPHIC TERMINALS

A.1 Introduction

This sample session shows how to use DBUG in a graphic environment to debug a simple sort program.

The purpose of the sample session is to give all levels of users, a clear understanding of the basics of debugging with DBUG in an X-windows and a mouse environment.

Readers with an alphanumeric terminal environment should work through the example in Appendix B.

A.2 Organization and Use of the Sample Session

A.2.1 Organization of the Sample Session

The sample session is divided into a number of logical modules. These are:

1. The Hardware Environment
2. Beginning the Program Run
3. Introduction to DBUG Windows
4. Explanation of the Sample Program
5. The DBUG Session

While each module builds on the previous, a you may skip those topics with which you are already familiar, without disrupting the complete session. Each module is further divided into subsections which describe elements of the major topic area.

A.2.2 Working with the Sample Session

The sample session is designed to be worked through on a "live" system. You are presented with the commands you must execute along with an explanation of what has and what should happen. Reading through the chapter and executing the commands as they are described will give you a strong foundation on which to build your DBUG expertise.

A.2.3 Command Presentation

Commands and explanations are presented through a tabular structure. Session logic progresses from left to right, top to bottom. Text in the left column are commands. There are three types of commands: those that are entered by typing on the keyboard and pressing enter, those that are entered by pressing a function key and those that are entered by selecting a menu option with the mouse cursor. The sample table below describes what you should do when you see some typical commands.

COMMAND	EXPLANATION
wreset	Type "wreset" and press return.
<kp0>	Press the keypad key labeled with the number 0.
<ESC>	Press the key labeled "ESC".
<pf3>	Press the key labeled "f3".
middle button	Press the middle button on the mouse.
<pf1> <pf2>	Press the function key labeled "f1", then press the function key labeled "f2".
<ctrl><l>	Press the key marked Ctrl (control) and the key marked "L" simultaneously.

A.3 The Hardware Environment

This example runs on a bitmapped (graphic) terminal environment with a mouse, on SYS32/20 or SYS32/30 development systems.

NOTE: before running this session, please make sure that the application keypad on your terminal setup is active.

A.4 Beginning the Program Run

The following commands are used to begin the program run.

COMMAND	EXPLANATION

	Copy the sample program and its source to your directory. Both files reside in the same directory where DBUG was installed. The command for this is:
	`cp` *gnxdir*`/usr/bin/bubble* .`
	
	The sample program is called `bubble`, and is written in C. The program was compiled earlier with the command:
	`cc -g bubble.c -o bubble`
	using the GNX C compiler.
	
	For subsequent compilations, please make sure that you are using the GNX C compiler.

dbug bubble	Begin the DBUG sample session.

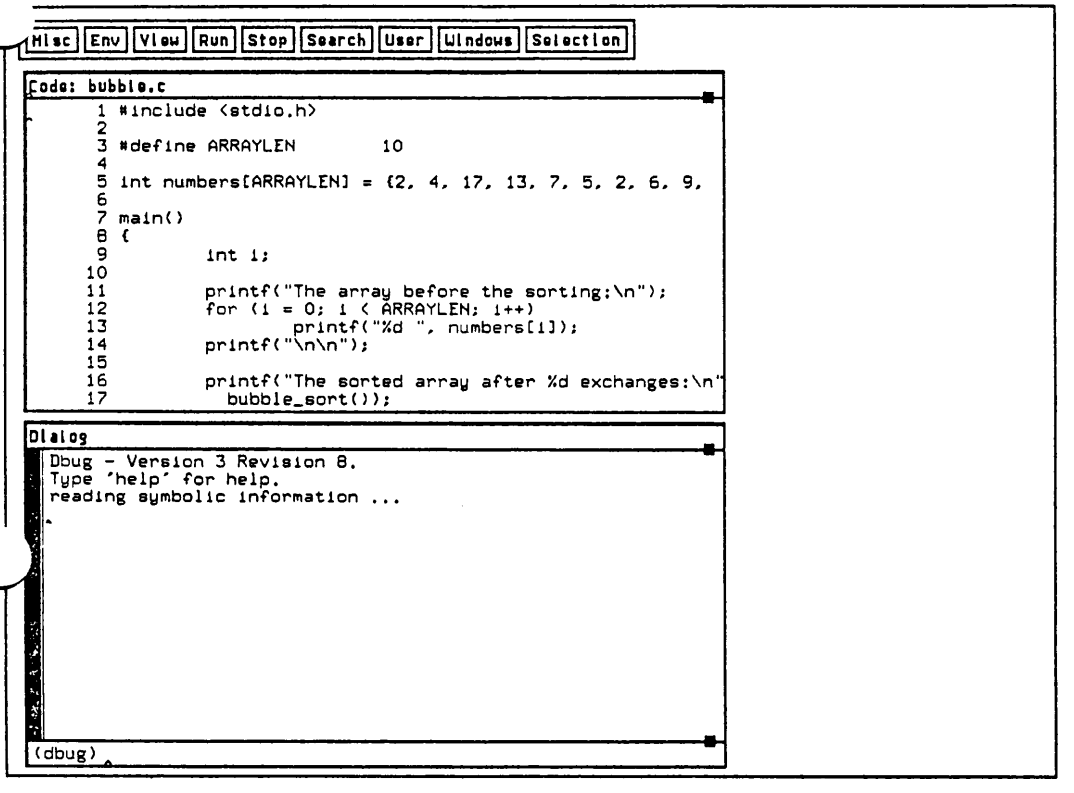


Figure A-1. Opening Frame of the Graphic Interface

A.5 Introduction to the Graphic Interface Windows

A.5.1 Opening Screen Layout

COMMAND	EXPLANATION
	The DBUG session begins with the appearance of the DBUG frame on the screen (Figure A-1). The opening DBUG frame contains three major rectangles, and a large empty area on the right side. The first rectangle stretches along the upper border of the frame, and houses nine smaller windows with labels like "Misc" and "Env". These are known as command menus, and will be discussed a little later in this section. The second and third major rectangles are windows for displaying information about the run. These windows are divided into a number of areas. Both windows contain a narrow band along their upper border. This band displays the name of the window. Notice that the upper window is called the CODE window and the lower window is called the DIALOG window. The CODE window contains the currently executing code of the program under debug. Notice, too, that this program is `bubble.c`. The CODE window displays the code of the current program under debug. As expected, this code belongs to the program: `bubble.c`. Unlike the CODE window, the DIALOG window is divided into four separate areas: the header band, the output area (the largest area in the window), the command line (along the window's bottom border), and the scrolling area (the thick black bar along the lefthand border of the output area). The output area displays the session history, including user commands and DBUG prompts and echoes. The command line (denoted by the word "(dbug)" in the left corner) is used to type commands into the debugger.

A.5.2 Scrolling through the Scroll Bar

COMMAND	EXPLANATION
	Finally, the scroll bar is used to scroll through the output area. Scrolling is accomplished by placing the mouse cursor in the scroll area (the cursor will become a vertical line with arrows at either end) and clicking the appropriate mouse button. The left button is used to scroll forward, the right button to scroll backward, and the middle button to scroll to an absolute position in the text. Let's scroll to absolute positions.
	Place the mouse cursor on the **lowermost point of the scroll bar.**
middle button	To scroll to the bottom of the DIALOG window.
	Place the mouse cursor on the **uppermost point of the scroll bar**.
middle button	To scroll to the top of the DIALOG output area.

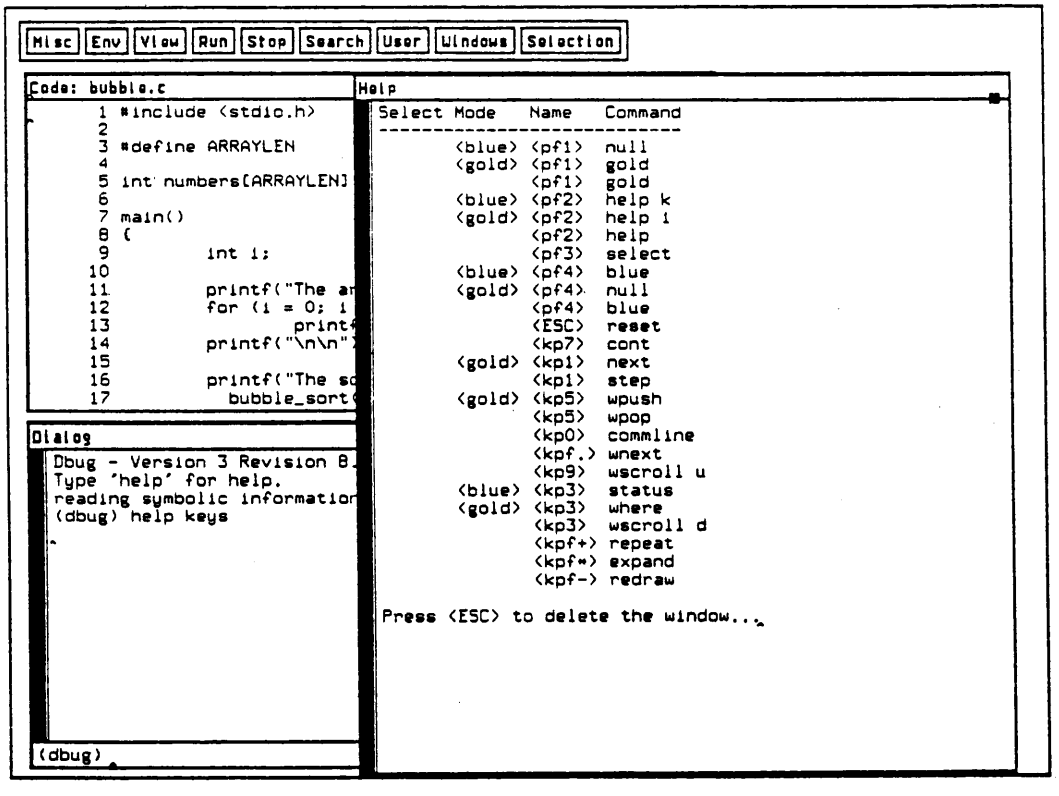


Figure A-2. HELP Window

A.5.3 Moving Between Windows and Entering Commands

COMMAND	EXPLANATION
	When working in a multiwindow environment, you must be sure that the mouse cursor is within the DBUG frame. From within the DBUG frame, you may only write or do work in one window at a time. The window where you currently working is known as the current window. The *current window* is defined when the mouse cursor (identified by the large X) is placed within its area.
move the mouse	Control the placement of the mouse cursor by moving the mouse.
<kp0>	Press the keypad key marked with a zero (0). This key has been defined to execute a command which brings the mouse cursor into the command area. Let's enter a command. A valuable command at this point would be one which describes the various previously defined function keys.
help keys	If you've done everything right (like align the mouse cursor in the command line of the DIALOG window), typed without errors and pressed return, then a new window has appeared on the screen (Figure A-2). This window is called the HELP window, and lists the power of various predefined function keys. Let's look through the list and execute a few of these functions right now.
<ESC>	Reset. The help screen disappears.
<pf4> <pf2>	Display the help keys once again, this time by pressing function keys.

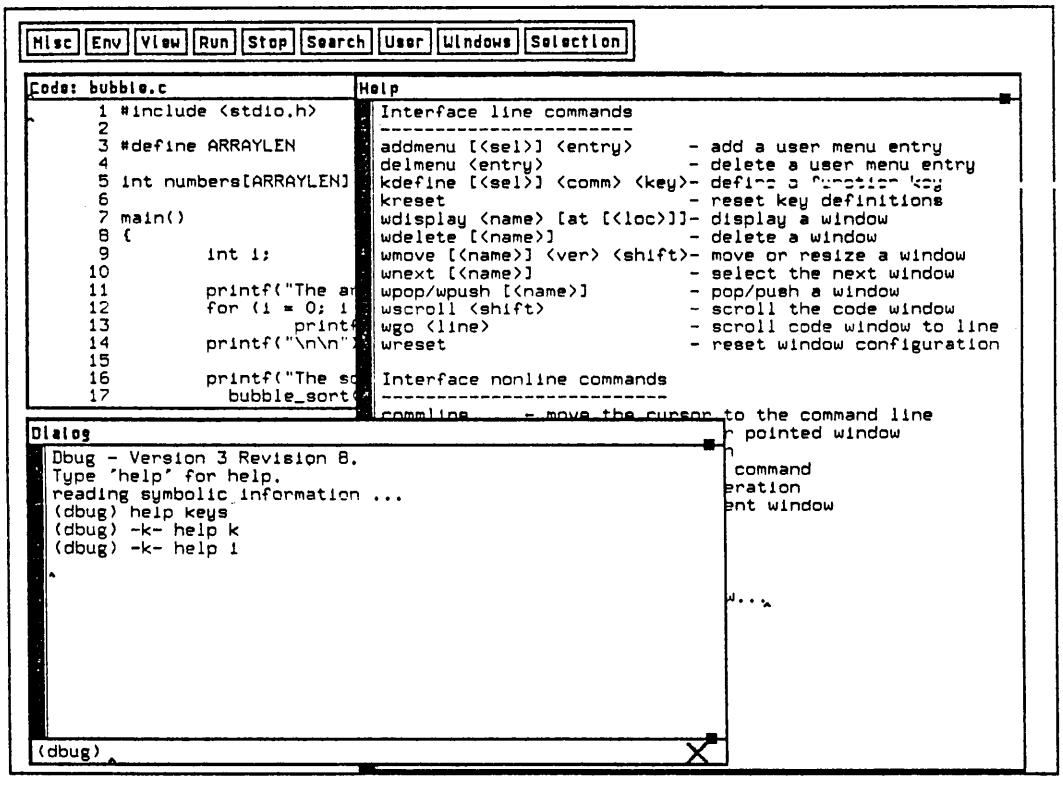


Figure A-3. On the Command Line

COMMAND	EXPLANATION
<pf1> <pf2>	List the commands for manipulating the DBUG windows. Note that the window is divided into two parts. The top half, called interface line commands, lists those commands that must be entered along a command line via the keyboard. The bottom half is called interface nonline commands, these are commands that are entered through predefined function keys.
<kp0>	Go to the command line. Notice that the DIALOG window overlaps part of the HELP window (Figure A-3).
wdelete code	Delete the CODE window
wdisplay code	Display the CODE window.

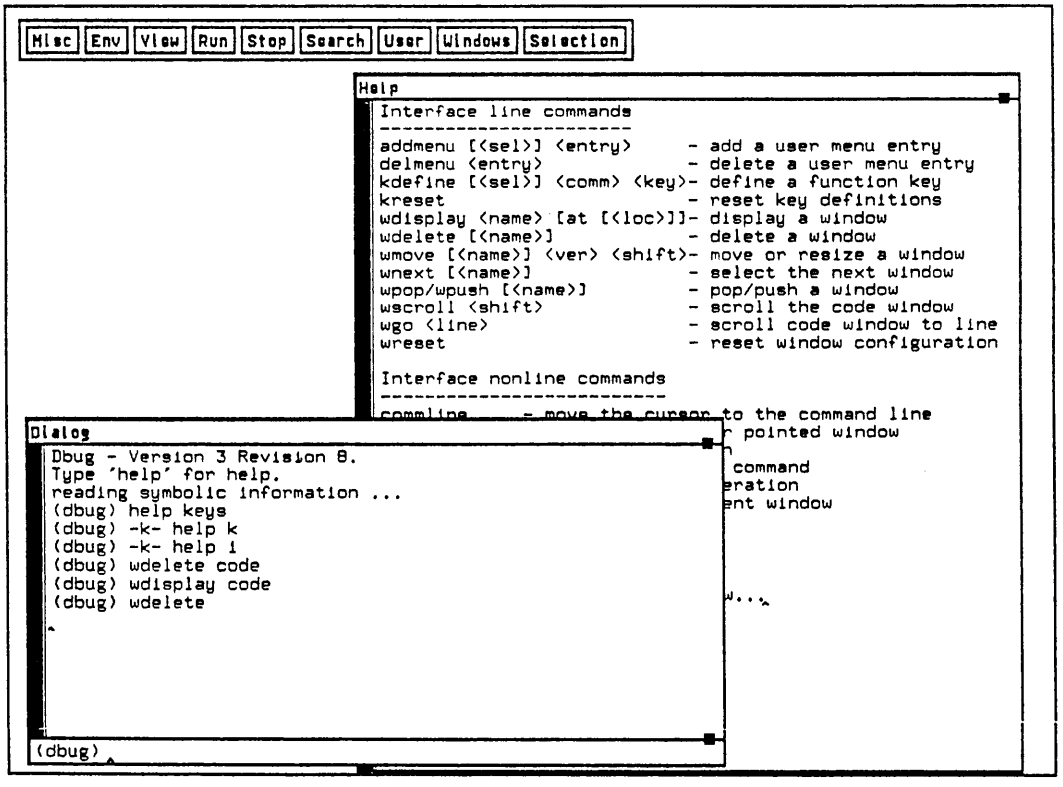


Figure A-4. wdelete code

COMMAND	EXPLANATION
	The *selected window* is that window on which a particular command will be executed by default. Thus, instead of specifying "move code", or "wdelete code", you may, when CODE is the selected window, simply invoke "wdelete" (for example) to see the CODE window disappear from the face of the screen. A window is selected by moving the mouse cursor into the area of the window and pressing <pf3>.
	Move the cursor into the code window now.
<pf3>	Let's execute the wdelete command once again.
<kp0>	Go to the command line.
wdelete	There is no need to specify CODE, because it is already the selected window (Figure A-4).
wdisplay code	Redisplay the CODE window.
wpop help	Lay the overlapping part of the help menu over the CODE and DIALOG windows. Read the HELP window.
	Now that you're familiar with all the options described in the HELP window (a small, but very handy subset of the complete range of commands available to you), you can define a function key that can carry out one of these commands. The ability to define function keys will save you countless seconds of typing time. So, let's define <ctrl><y> as our function key for executing the **redraw** command (a command for clearing any garbage from the screen).
<kp0>	Go to the command line. From here we'll define a function key.

COMMAND	EXPLANATION
kdefine redraw	Begin the procedure for defining a function key to have the command **redraw**. The message "Press a function key:" appears in the output area of the DIALOG window. Simultaneously, the "(dbug)" prompt disappears from the command line.
<ctrl><y>	Press <ctrl><y>. The "(dbug)" prompt reappears on the command line. <ctrl><y> now has the function: redraw.
<ctrl><y>	Redraw the screen. Note that the ability to define keys is one of DBUG's strengths, for it allows you to define functions that are important for your particular application, and in a manner that is easy to use.
<ESC>	Reset. The help screen disappears.

A.5.4 Command Menus

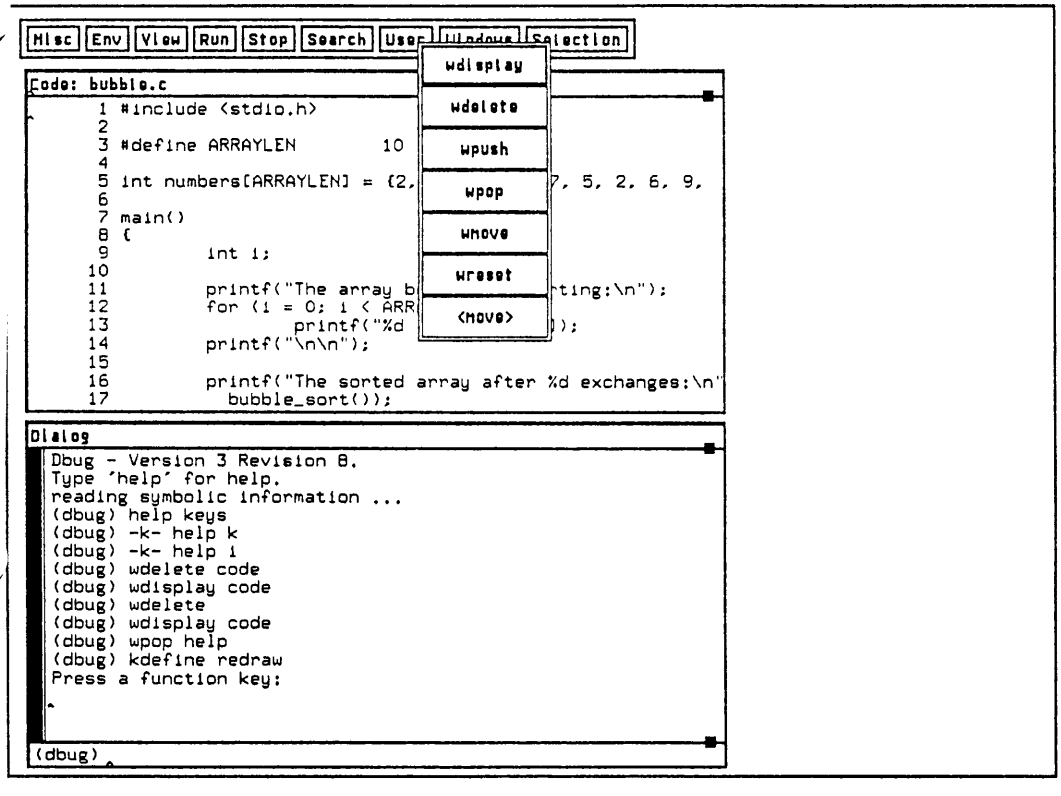


Figure A-5. Temporary Menu

Until now, we've discussed two ways to enter a command: by typing it in, and by pressing predefined keys. Yet there is a third way: by selecting a menu option.

Menus are placed in the narrow rectangle along the top border of the DBUG frame. Inside this rectangle are nine small windows. Each window is labeled with words like "Misc", "Env", and "View". Each of these small windows holds a menu list of commands which relate to the label. The window labeled "Windows", for instance, holds commands which are used to manipulate the windows in the DBUG frame.

Move the mouse cursor to the rectangle labeled "Windows". Notice that its border is highlighted in bold when the mouse cursor is within its realm. Let's look at the contents of this window.

left button

Assuming you've pressed the left button on the mouse while the cursor was positioned in the "Windows" window, you now see before you a list of menu options. This is called a "temporary menu", and should contain words like "wdisplay" "wdelete", "wreset" and "<move>" (Figure A-5).

Commands are executed from this temporary menu by the following procedures: 1) a window is selected 2) the mouse cursor is placed on the desired command, and 3) the mouse's left key is pressed. You can remove the temporary menu by pressing <ESC> (but DON'T do it now).

We'll execute this procedure with the "wmove" as an example. The **wmove** command is used to move a window within the DBUG frame. Let's move the DIALOG window.

Position the mouse cursor in the DIALOG window.

COMMAND	EXPLANATION
<pf3>	Select the DIALOG window. Note that it is now outlined. **Move the mouse cursor to the "wmove" menu option.**
left button	The "Windows" hanging menu disappears. A prompt appears in the DIALOG window. The prompt reads: "Click a button to enter the first point". This point, together with a second point to be entered defines how the window will shift. Place the mouse cursor on the **lower righthand corner of the DIALOG window.**
left button	The prompt reads: "Click a button to enter the second point". Move the cursor to the **lower righthand corner of the dbug frame.** This point, together with the previous point, defines the DIALOG window shift.
left button	The DIALOG window shifts to the right. Let's clean the screen by pressing our recently defined **redraw** key.

COMMAND	EXPLANATION
<ctrl><y>	Press the **redraw** key to refresh DBUG frame.
	It is possible to prevent the temporary menu from disappearing with the execution of a command by moving it to to another location before executing a command. Once it's been moved it won't disappear unless you ask it to. Such menu is called a "hanging" menu. Let's try it now.
	Move the mouse cursor to the rectangle labeled Windows.
left button	Displays the temporary menu. **Position the mouse cursor in the option titled: <move>.**
left button	The <move> option is highlighted. The DIALOG window prompts:
	"Click a button to position the menu"
	Move the cursor to a **location to the right of the CODE window** and towards the top of the DBUG frame.
middle button	The menu is hung at the point you identified with the click of the mouse key.
	This feature, allows you to display commands without obstructing any other windows. You can remove the hanging menu by selecting the <exit> option. Note that the <exit> option does not appear on the menu until it has been moved.
	Position the mouse cursor on the **<exit>** option.
left button	Press the left button. This causes the hanging menu to disappear.

COMMAND	EXPLANATION
	Another interesting menu is the "Misc" menu. **Position the cursor on the "Misc" window** (until it is outlined in bold).
left button	Three interesting commands in this menu are "help" - to display the HELP window, "quit" - to quit DBUG, and <command>. <command> generates a command line that can be used for entering commands through the keyboard. This line may be used instead of the command line in the DIALOG window. **Align the mouse cursor on the <command> option** from the "Misc" menu.
left button	Causes the command line to appear. Align the mouse cursor on the command line from the Misc menu.
quit	Leave DBUG We are now ready for debugging the sample program.

A.6 Explanation of the Sample Program

This section describes the sample program to be debugged during the sample session.

A.6.1 The Sample Program Logic

The program is designed to execute a bubble sort. The bubble sort algorithm reorders a set of given numbers in descending order by comparing two adjacent numbers, and exchanging them if the number to the right is greater than the number to the left. The sorter compares every pair in the set until it reaches the end of the list. The sorter continues to make passes on the set until all numbers appear in descending order.

The following table displays bubble sort algorithm applied to a set with four values ordered as follows: 1 4 2 3.

NUMBER SET	PROGRAM OPERATION
1 4 2 3	(0) Original state of set
4 1 2 3	(1) Pass 1, values 1 and 4 compared and exchanged.
4 2 1 3	(2) Pass 1, values 1 and 2 compared and exchanged.
4 2 3 1	(3) Pass 1, values 1 and 3 compared and exchanged.
4 2 3 1	(4) Pass 2, values 4 and 2 compared but not exchanged.
4 3 2 1	(5) Pass 2, values 3 and 2 compared and exchanged.
4 3 2 1	(6) Pass 2, values 2 and 1 compared but not exchanged.
4 3 2 1	(7, 8, 9) Pass 3, values 4 and 3, 3 and 2, and 2 and 1 compared, but not exchanged. Bubble sort complete.

A.7 The Sample Program

The C language program that implements this algorithm is found in a file called `bubble.c`. The contents of this file appears on the following page.

Program variables include `numbers`, `total_count`, `change_count` and `save`. `ARRAYLEN` equals the number of values in the list. `numbers` is an array of length `ARRAYLEN` which holds the numbers in the set. `total_count` counts the total number of times an exchange takes place between two adjacent array cells. `change_count` also counts exchanges, but is reset to zero each time the sorter begins a new pass on the set. `save` is a temporary variable which acts as a holding area for exchanges between two adjacent array cells.

The first program activity is to print the array before it is sorted. The program then calls a routine called `bubble_sort` which performs the bubble sort logic. The routine is exited when a pass is completed without any exchanges taking place

(change_count = 0) or when the total number of passes exceeds the theoretical maximum (total_count >= ARRAYLEN squared). The program then prints the result of the run.

```
#include <stdio.h>

#define ARRAYLEN        10

int numbers[ARRAYLEN] = {2, 4, 17, 13, 7, 5, 2, 6, 9, 15};

main()
{
    int i;

    printf("The array before the sorting:\n");
    for (i = 0; i < ARRAYLEN; i++)
            printf("%d ", numbers[i]);
    printf("\n\n");

    printf("The sorted array after %d exchanges:\n",
      bubble_sort());
    for (i = 0; i < ARRAYLEN; i++)
            printf("%d ", numbers[i]);
    printf("\n");
}

int bubble_sort()
{
    int i;
    int change_count, total_count;
    int save;

    total_count = 0;
    do
    {
            change_count = 0;
            for (i = 0; i < ARRAYLEN; i++)
                    if (numbers[i] <= numbers[i + 1])
                    {
                            save = numbers[i+1];
                                    numbers[i+1] = numbers[i];
                            numbers[i] = save;
                            change_count++;
                            total_count++;
                    }
    }
    while (change_count > 0 && total_count <= ARRAYLEN * ARRAYLEN);
    return(total_count);
}
```

A.8 The dbug Session

A.8.1 Run the Sample Program

COMMAND	EXPLANATION

bubble	Run `bubble` and look at the results. The program prints:
	
	`The array before sorting: 2 4 17 13 7 5 2 9 6 15`
	
	`The sorted array after 101 exchanges: 543516756 17 15 13 9 7 6 5 4 2`
	
	Notice anything strange about the output? For one thing, the number 543516756 (or something similar) was not part of our original input set, and for another, the program carried out 101 exchanges (more than the theoretical maximum) before exiting.
	
	Let's use DBUG to find the problem.

A.8.2 Running dbug and Devising a Debug Strategy

COMMAND	EXPLANATION
dbug bubble	Run DBUG
	Let's turn our attention to the DBUG frame. More specifically, let's examine `bubble.c` in the CODE window.
cursor in CODE	Place the mouse cursor in the display area of the CODE window.
up-down arrows	Scroll through `bubble.c` by pressing the up and down arrows. You may also scroll by using the pg-up, pg-down keys. Notice that the small "window cursor" (looks like an upside down "v") marks your place in the window.
	The first question we must address as we examine `bubble.c` is where to look for the bug. If we glance at the output we see that the first bug appears only after we've called the routine called `bubble_sort`. So our first task will be to run the program until we reach the `bubble_sort` routine in line 23, then examine different variables in order to determine where and how the problem occurs.
<kp0>	Go to command line.

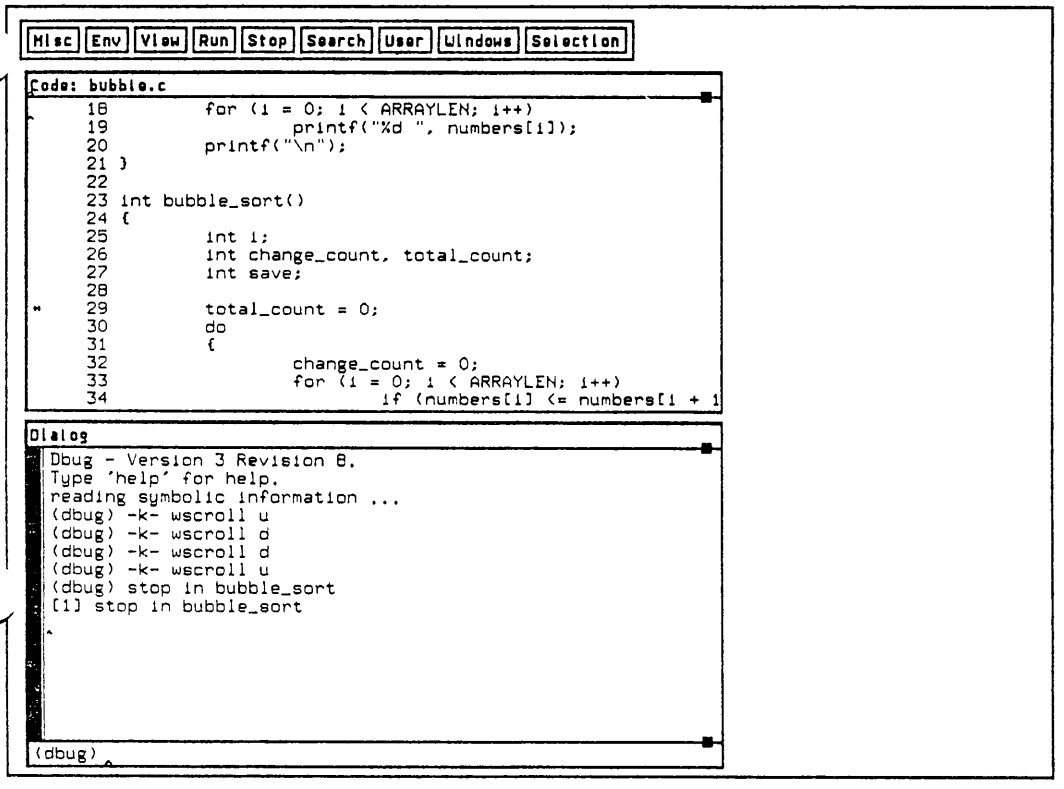


Figure A-6. stop in bubble_sort

A.8.3 First Debug Commands

COMMAND	EXPLANATION
stop in bubble_sort	Set a flag which tells the program to halt when it reaches the first executable command in `bubble_sort`. Notice that this command is `total_count = 0` in line 29, and that it is now marked by an asterisk (*) along the lefthand border of CODE window (Figure A-6).
<kp0>	Go to command line.
run	Execute `bubble.c` Notice that an arrow appears on the left side of the CODE window screen and points to the current program line under execution. This is line 29, the first executable line of `bubble_sort`, and is also, the point where we told the program to stop. Notice that this information is also displayed in the output area of the DIALOG window.
step	Continue to execute the program one executable instruction at a time. Notice that the arrow moves to the current line under execution.
step	Execute the next line. You may find it boring to type "step" over and over. One short cut available to you is to type "s", the alias for **step**.
s	Step one more line by executing the alias for **step**.
p numbers	We can also print the value of different variables. "p" is the alias for print. `numbers`, of course, is the array where our set is stored. Notice that the array is in its original state. No exchanges have taken place.
alias	This command prints a list of all available aliases.

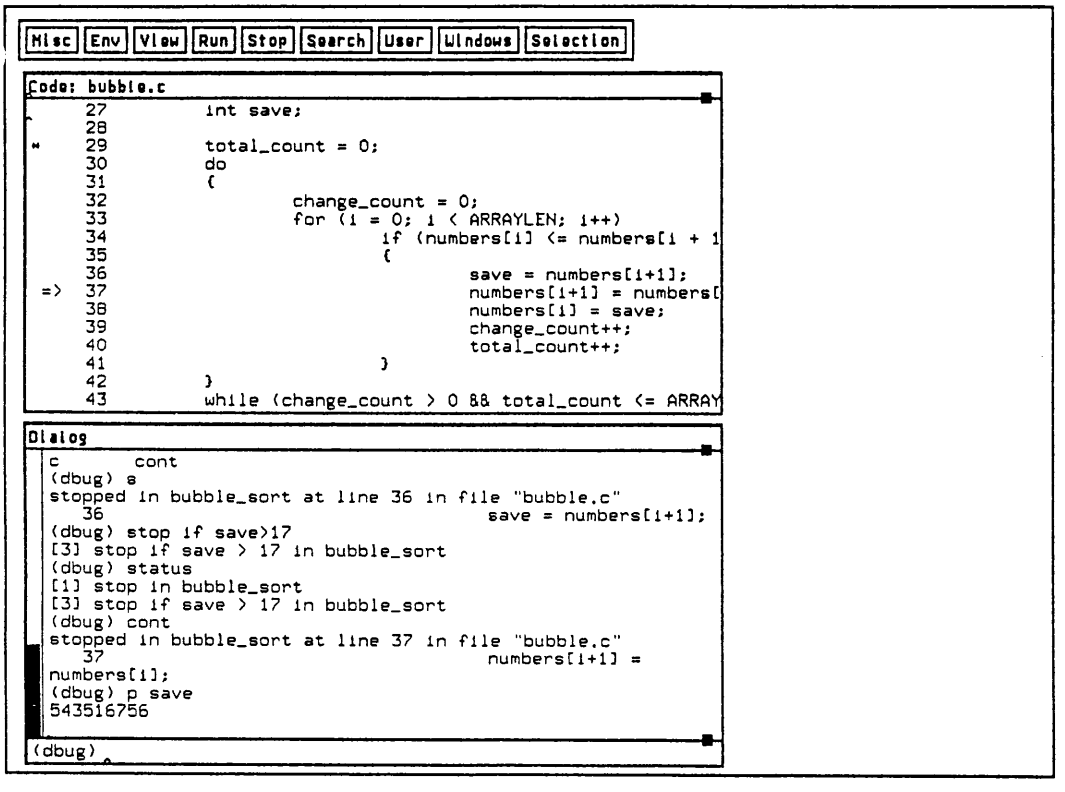


Figure A-7. p save

COMMAND	EXPLANATION
s	Step. This process of single stepping, while potentially fruitful, is also time consuming, and perhaps not really appropriate for our current needs. Let's try another approach. Notice that all exchanges go through the temporary variable called `save` (line 36). That is, when `numbers[i] <= numbers[i+1]` then `save = numbers[i+1]` and `numbers[i] = save`. Thus, if we can identify the first time `save` receives an erroneous value, we'll be well on our way to discovering the bug.
stop if save > 17	Set a breakpoint if `save` gets a value that's greater than 17, the largest known number in our set. Unlike our earlier breakpoint, this one is not identified by an asterisk (*) because it doesn't have a specific location in the program. On the other hand, its scope does not extend beyond the `bubble_sort` routine.
status	List all of the currently declared breakpoints. There are two. The number to the left has been assigned by DBUG and is called *event number*.
cont	Continue program execution until the "stop if" breakpoint is reached. Notice that the program stops after a few moments. This is an indication that the value of `save` exceeds 17.
p save	Print the value of `save`. It equals 543516756 (or something equally wrong) (Figure A-7). Let's examine the array index to determine when this occurred.

COMMAND	EXPLANATION
p i	The "print i" command yields the information that the array index equals 9 when `save` gets the erroneous value. This means that `save` gets the erroneous value in line 36, from the assignment: `save = numbers[i+1]` (or in others words: `save = numbers[10]`). How could array index 10 receive such a value? The answer is clear. According to C language conventions, array indices begin from 0 and continue to n-1 (where n equals the size of the array). This means an array of size 10 would have the indices: 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9. Our array is also of size 10 (`ARRAYLEN = 10`). This means that array `numbers` is composed of variables `numbers[0]`, `numbers[1]`,...,`numbers[9]`. Variable `numbers[10]` is not even part of the array. Our problem comes from the fact that we've referenced an invalid variable (`numbers[10]`). The solution is to prevent array index `i` from exceeding 8, then the maximum value attainable by `i+1` will be 9 (instead of 10, as is currently). Let's fix the program.
q	Quit DBUG.

A.8.4 Fixing the First Bug

COMMAND	EXPLANATION
	Edit `bubble.c`.
	Go to line 33 where the loop `bubble_sort` is defined. Replace the line:
	`for (i=0; i < ARRAYLEN; i++)`
	with:
	`for (i=0; i < ARRAYLEN - 1; i++)`
	This ensures that the loop continues from 0 to 8, instead of from 0 to 9. Save the change.
cc -g bubble.c -o bubble	Recompile the program. Be sure you are using the GNX C compiler.
bubble	Let's run the program again to see if it runs correctly. The numbers seem to be in the right order, but the output message states:
	The sorted array after 101 exchanges.
	This is theoretically impossible - even if the array were completely backwards, for the program to need more than 100 exchanges to sort it correctly. Something is wrong, and we should use DBUG again.
dbug bubble	Run the debugger.

A.8.5 Hanging Menus and Selecting Text

COMMAND	EXPLANATION
	Unlike last time, we want to make the way we enter our commands as efficient as possible. Let's begin by selecting certain command windows, and posting their hanging menus in locations where they're easy to access.
	Position the mouse cursor on the **"Selection" menu.**
left button	Display the temporary menu by pressing the left key on the mouse.
	Move the mouse cursor to the **<move>** option.
left button	Press the left button on the mouse. The <move> option should now be highlighted.
	Move the mouse cursor to a point to the **right of the CODE window,** not far from the upper border of the DBUG frame.
left button	Press the left button on the mouse. The "Selection" hanging menu should now be displayed to the right of the CODE window.
	Position the mouse cursor in the **CODE window.**
	Once again we want to begin by setting a breakpoint in the `bubble_sort` routine with the command **stop in bubble_sort**. We take this action because we want to determine why the variable `total_count` returned the value 101 after exiting `bubble_sort`. Unlike in the past, we will issue the command by selecting text from the program, and then execute the "stop in" command from the hanging menu.

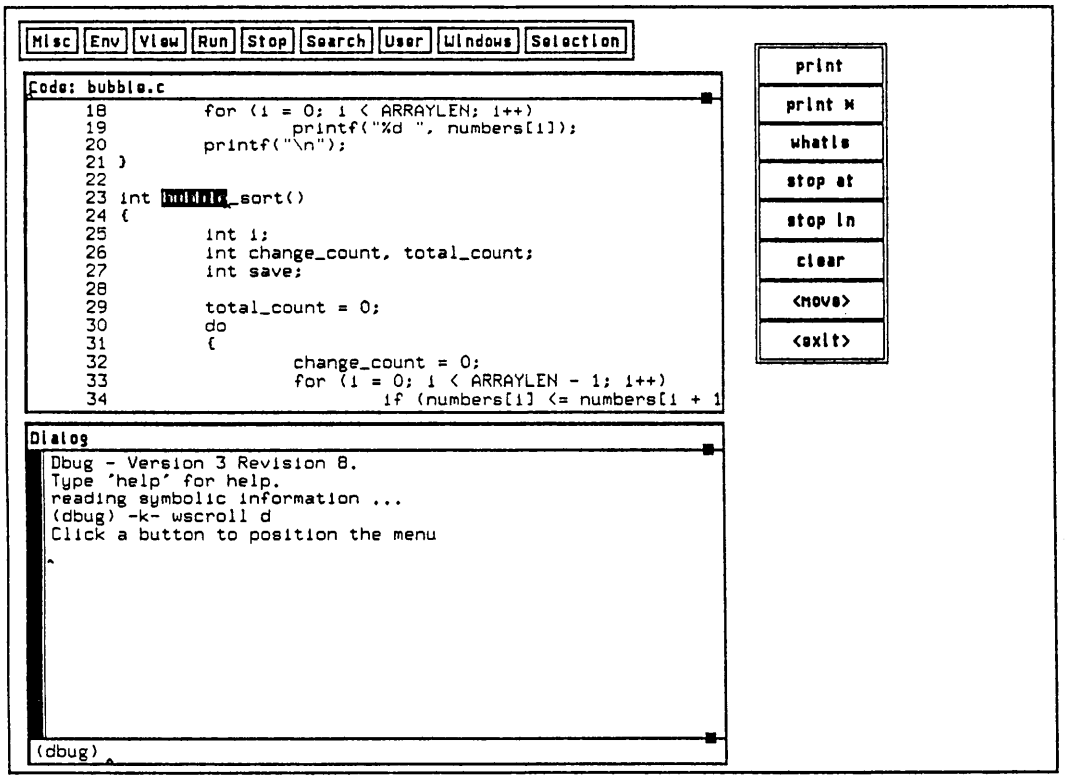


Figure A-8. Marking Text

COMMAND	EXPLANATION
kp3 (pg dn)	Scroll through the display of bubble.c until line 23: `int bubble_sort()` is in view. Position the mouse cursor **on the word: bubble_sort.**
Hold down left button	Press the left button on the mouse and **move the cursor to the right.** Notice that the line begins to be highlighted as you move. Ensure that at least one or two characters are highlighted. This process is used to select the name `bubble_sort` (Figure A-8).
Release left button	Notice that the highlighting remains even after you release the left button. Now move the cursor to the hanging menu **stop in** option.
left button	The **stop in bubble_sort** command is issued. An asterisk (*) appears next to line 29, the first executable code in the `bubble_sort` function. Note what we just did. We selected a couple of characters from a function name and then selected the command name we wanted executed with it. The complete command was issued without having to touch a single key on the keyboard. Now let's display the command menu labeled "Run". Move the mouse cursor to the **command window labeled Run.**

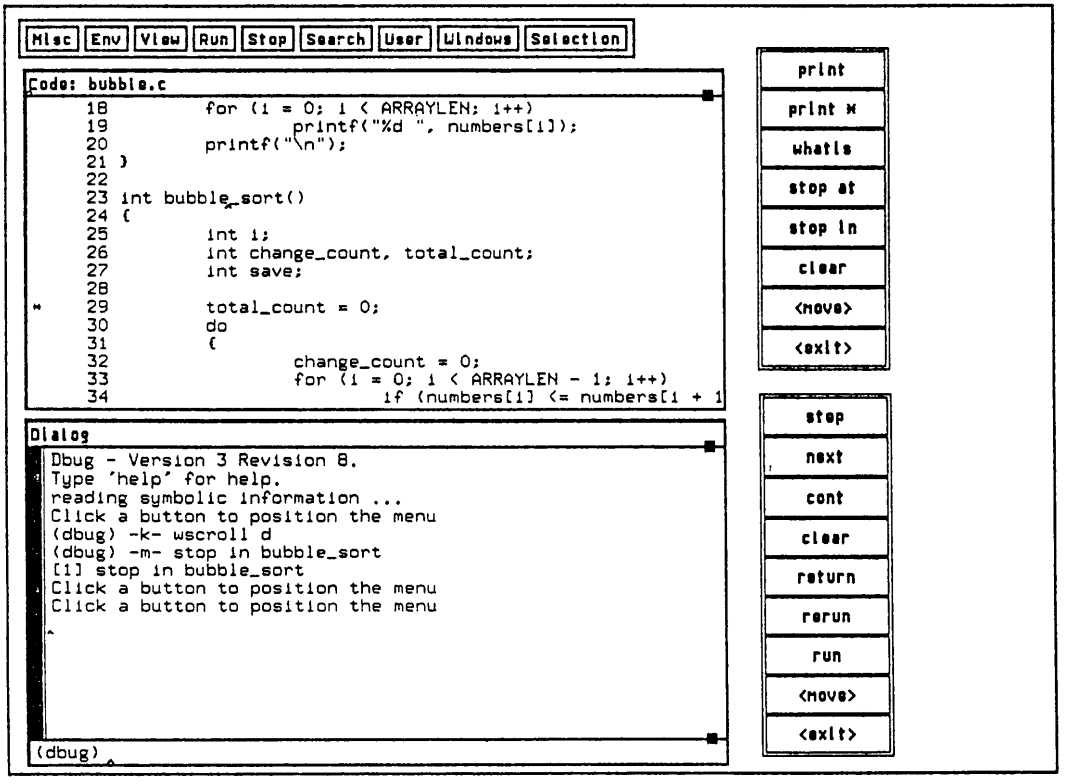


Figure A-9. Hanging Menus

COMMAND	EXPLANATION
left button	Display the hanging menu by pressing the left key on the mouse. Move the mouse cursor to the **<move> option.**
left button.	The <move> option should now be highlighted. Move the mouse cursor to a point just **below where the Selection menu is hanging.** Somewhere to the right of the DIALOG window (Figure A-9). Notice the prompt in the DIALOG output window which reads: "Click a button to position the menu.".
left button	The "Run" hanging menu should now be displayed to the right of the DIALOG window. Align the mouse cursor on the "**run" option** on the hanging menu.
left button	Run the program. Notice that execution stops to the side of the first executable command in the `bubble_sort` function. This is line 29 which states `total_count = 0`. Now move the mouse cursor into the **CODE window.** From here we may take a closer look at the program code, and decide what to do next.

COMMAND	EXPLANATION
up - down arrows	Gently scroll through the CODE window. Keep in mind, as we scroll, that the evidence of the bug comes from the fact that the value of `total_count` is too high, at least by one. Since checking each and every exchange would be too clumsy and time consuming, we would be advised to put in a breakpoint after each complete pass on the number set. Then we could check the array `numbers` after it had undergone a complete pass of the sorter. To do this, **position the mouse cursor over a character in line 33,** the line which reads: `for (i = 0; i < ARRARYLEN-1; i++).`
hold down left button	Press the left button on the mouse and **move the cursor to the right** for a couple of characters.

COMMAND	EXPLANATION
release the left button	Notice that the highlighting remains even after the mouse key is released. Position the mouse cursor over the "stop at" option displayed on the hanging menu.
left button	Issue the "**stop at**" line 33 command. Position the mouse cursor over the "cont" **option.**
left button	Issue the "cont" command. The program run continues until it reaches line 33, where we placed the breakpoint. Let's print the contents of the array `numbers` to determine if its values are in the correct order before beginning the first pass.
<kp0>	Go to command line.
p numbers	Print the contents of numbers. Look in the DIALOG window. The array prints out the following numbers in the following order: 2, 4, 17, 13, 7, 5, 2, 6, 9, 15 This is the order they should be in before sorting. Let's now continue following the same procedure we just executed: continue the run, stop at the breakpoint and print the value of `numbers`. We'll do this until a problem is detected.

COMMAND	EXPLANATION
	Continue the run by aligning the mouse cursor on the "cont" **option.**
left button	Issue the "cont" option. Wait for the program to reach the breakpoint.
<kp0>	Go to the command line.
p numbers	Print the value of the array `numbers`. Notice that after the first pass on the number set, the number which had previously been in the second position is now in the first position (4), and the number which had previously been in the first position, is now in the last position (2). This is evidence that at least until now, the sorting is proceeding correctly.

A.8.6 Issuing Commands with Predefined Keys

COMMAND	EXPLANATION
	At this point, let's define a function key that executes the **repeat** command. **repeat** re-executes the last command issued from the command line. Note that it does NOT re-execute commands issued from the hanging menu, keypad keys or function keys, even if they were issued more recently than the last command from the command line. The distinction of how a command is issued is also made in the output area of the DIALOG window where the program conversation is displayed, and all commands are echoed. Echoes of commands issued from the command line are preceded by the indicator "(dbug)" (example: **(dbug)** `p numbers`), while echoes of commands issued through a keypad key or function key are preceded by the indicator "-k-", while echoes of commands issued through the menu are preceded by the indicator "-m-" (example: `-m- cont`).
<kp0>	Go to command line
kdefine repeat	Define a key which executes the repeat function. Look in the DIALOG output area. DBUG now asks that you press a function key. This key will hold the function **repeat**.
<kp5>	Give keypad 5 the function **repeat** Align the mouse cursor on the **"cont" option.**
left button	Issue the "cont" option. Wait for the program to hit the breakpoint.

COMMAND	EXPLANATION
<kp0>	Go to the command line.
p numbers	Print numbers. We can't use our **repeat** key just yet because the last command issued from the command line was "kdefine repeat".
	Notice that the sorting mechanism seems to be proceeding normally, as higher numbers gravitate towards the front, and lower numbers move towards the back.
	Align the mouse cursor on the "**cont**" **option.**
left button	Issue the "cont" option. Wait for the program to hit the breakpoint.
<kp5>	Issue the "print numbers" command by pressing the key defined as **repeat**.
	Let's continue with this pattern (pressing **cont** and the **repeat** function key), until something suspicious comes up.
	Align the mouse cursor on the **"cont" menu option.**
left button	Execute the "cont" option by pressing the left button on the mouse.
<kp5>	Repeat the print numbers command.

COMMAND	EXPLANATION
	Until now everything seems to be ok. It could be, however, that our problem lies with `total_count` - that it is receiving an incorrect value. Let's print its value. However, instead of going to the command line and issuing the command there, let's select it graphically and issue the command from one of our hanging menus.
	Go to the where the program is displayed in the CODE window. **Position the cursor on total_count (line 40).**
Depress left button	**Move the cursor to the right** until a couple of characters in the word `total_count` are highlighted.
Release the left button	The highlighting remains.
	Now move the mouse cursor to the hanging menu **"print" option.**
left button	Execute the "print" option. The value of `total_count` is 25. This seems reasonable, so let's simply continue.
	Align the mouse cursor on the **cont menu option.**
left button	Execute the "cont" option by pressing the left button on the mouse.
<kp5>	Repeat the print numbers.
	Align the mouse cursor on the **cont menu option.**
left button	Execute the "cont" option by pressing the left button on the mouse.
<kp5>	Repeat the print numbers.

COMMAND	EXPLANATION
	This mode too, you may have noticed, can become a little tedious. It turns out that, DBUG provides you with still another way for entering commands. This option allows you scroll through the history of commands issued through the command area, and re-issue those commands that are of interest to you, by pressing return. Before going to the exact details of how to do this, let's first execute a couple of key commands through the command window.
<kp0>	Go to the command line.
cont	Continue the run.

A.8.7 Scrolling Through the Command Window

COMMAND	EXPLANATION
p numbers	print numbers. Now let's execute the same command series by scrolling through the command window (with the up-down arrows) and pressing return.
up arrow (2 times)	Scroll back through the history of the command line until the **cont** command appears.
press enter	Execute the **continue** command.
up arrow (2 times)	Scroll back until the **print numbers** command appears.
press enter	Execute the **print numbers** command. At this point everything looks all right. The number set has been reordered. As designed, the program should now end. Let's see what happens if we execute **cont** again.
up arrow (2 times)	Scroll back through the history of the command line of the command line until the **cont command** appears.
press enter	Execute the **continue** command.
up arrow (2 times)	Scroll back until the **print numbers** command appears.
press enter	Execute the **print numbers** command. Watch the result! The set remains in the same "correct" order, but the program fails to exit. Let's look at total_count, to determine if it's still making exchanges.

A.8.8 Narrowing Down the Problem

COMMAND	EXPLANATION
p total_count	`Total_count` equals 36. The program continues making exchanges even after the number set has been reordered. Let's try to find out why it continues executing `bubble_sort`. Let's put a breakpoint at the point where the exchange occurs. Go to the program display in the CODE window. **Align the mouse cursor on the word** `save` **in line 36.**
Depress left button	Press the left mouse key, move the cursor to the right for a character or two until you see the word begin to be highlighted.
Release the left button	`save` is highlighted.
stop at	Select "stop at" from the hanging menu. Next, clear the earlier breakpoint. **Align the cursor on some text from line 33.**
Depress left button	Press the left mouse key, **move the cursor to the right** for a character or two, until you see that some of the text is highlighted
Release the left button	The selected text remains highlighted. **Align the mouse cursor on the clear option** from the hanging menu.
left button	Execute the **clear** command. A message appears in the DIALOG window: `1 breakpoint deleted at bubble.c:33.` Align the mouse cursor on the "cont" **option** from the hanging menu.
left button	Execute the **continue** command. Now that the program has executed another pass, let's examine the numbers it is trying to exchange.

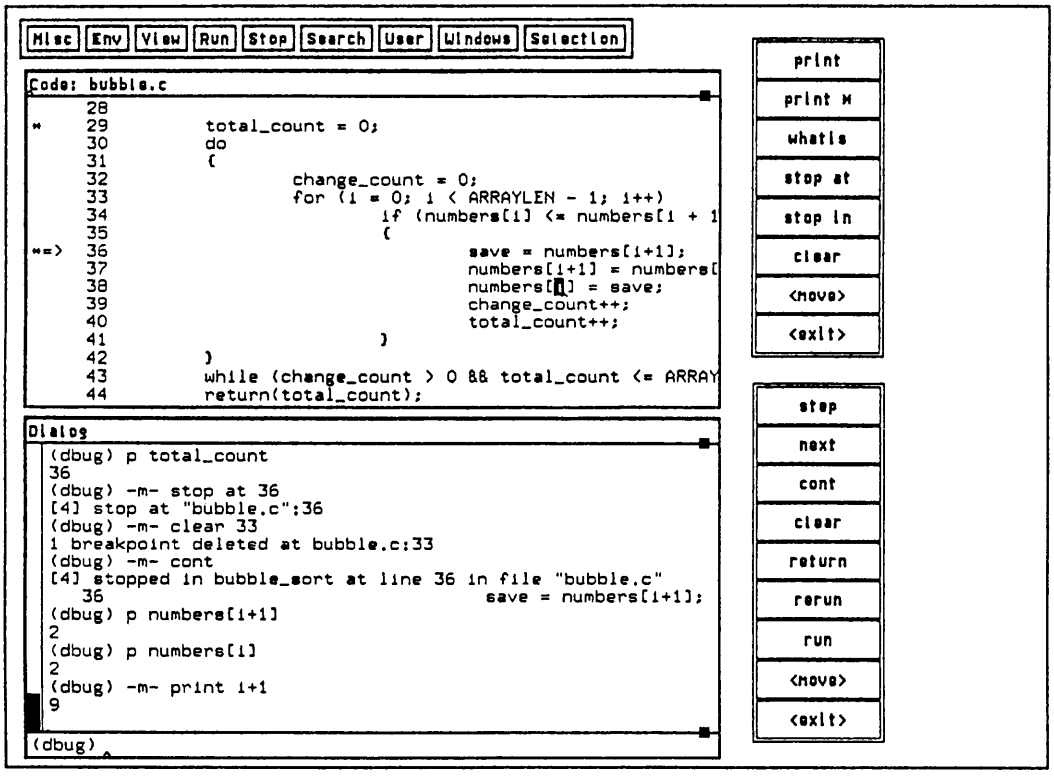


Figure A-10. Marking i

COMMAND	EXPLANATION
<kp0>	Go to command line.
p numbers[i+1]	The value of `numbers[i+1]` is 2.
p numbers[i]	The value of `numbers[i]` is 2. This means that the program is exchanging equal values. Let's examine value of `i` when it does this.
	Go to line 37 and **place the mouse cursor on i from the expression: i+1.**
Depress left button	Hold the left button down as you **highlight the complete expression: i+1.**
Release left button	`i+1` is highlighted.
	Align the mouse cursor on the **"print" menu option.**
left button	Select the "print" menu option. `i+1` equals 9.
	Go to the following line. Position the mouse cursor over the `i` in `numbers[i]` (Figure A-10).
Depress left button	Hold the button down until **just the i is highlighted.**

COMMAND	EXPLANATION
Release left button	The `i` is highlighted. Align the mouse cursor on the **print menu option.** The value of `i` is 8. It would seem that our program works fine when it compares two inequalities, but just doesn't stop when it compares two equal numbers. Look at the "if" statement which calls for the exchange in line 34: `if (numbers[i] <= numbers[i+1])`. Here is our problem. We don't want the program to execute the exchange if the values of two neighboring numbers are equal. We must fix the errant line to read: `if (numbers[i] < numbers[i+1])`
<kp0>	Go to command line.
quit	Quit DBUG
	Edit `bubble.c`. Make the change.
cc -g bubble.c -o bubble	Compile the program.
bubble	Run the program. The program generates the following message: The array before sorting: 2 4 17 13 7 5 2 9 6 15 The sorted array after 27 exchanges: 17 15 13 9 7 6 5 4 2 2 Everything looks good. This terminates the tutorial session of DBUG, but by no means did we cover the full range of DBUG's capabilities. Please refer to the previous chapters of this reference manual for the description of many additional commands and advanced options.

Appendix B

DBUG TUTORIAL FOR ALPHANUMERIC TERMINALS

B.1 Introduction

This sample session shows how to use **dbug** in an alphanumeric terminal environment to debug a simple sort program.

The purpose of the sample session is to give all levels of users, a clear understanding of the basics of debugging with **dbug** in an alphanumeric terminal environment.

B.1.1 Organization of the Sample Session

The sample session is divided into a number of logical modules. These are:

1. The Hardware Environment
2. Beginning the Program Run
3. Introduction to the Alphanumeric Interface
4. Explanation of the Sample Program
5. The **dbug** Session

While each module builds on the previous, you may choose to skip those topics with which you are already familiar, without disrupting the complete session. Each module is further divided into subsections which describe elements of the major topic area.

B.1.2 Working with the Sample Session

The sample session is designed to be worked through on a "live" system. You are presented with the commands you must execute along with an explanation of what has and what should happen. Reading through the chapter and executing the commands as they are described will give you a strong foundation on which to build your expertise.

B.1.3 Command Presentation

Commands and explanations are presented through a tabular structure. Session logic progresses from left to right, top to bottom. Text in the left column are commands. There are two types of commands: those that are entered by typing on the keyboard and pressing enter, and those that are entered by pressing a function key. The sample table below describes what you should do when you see some typical commands.

COMMAND	EXPLANATION
wreset	Type "wreset" and press return.
<kp0>	Press the keypad key labeled with the number 0.
<ESC>	Press the key labeled "ESC".
<pf3>	Press the function key labeled "f3".
<pf1> <pf2>	Press the function key labeled "f1", then press the function key labeled "f2".
<ctrl><l>	Simultaneously press the key marked "ctrl" (control) and the key marked "L".

B.2 The Hardware Environment

This example runs on an alphanumeric terminal environment, on SYS32/20 or SYS32/30 development systems.

NOTE: Make sure that the applications keypad on your terminal setup is active before running this session.

B.3 Beginning the Program Run

The following commands are used to begin the program run.

COMMAND	EXPLANATION
	Copy the sample program and its source to your directory. Both files reside in the same directory where **dbug** was installed. The sample program is called `bubble` (`bubble.exe` in VMS), and is written in C. The program was compiled using the GNX C compiler. For subsequent compilations, please make sure that you are using the GNX C compiler.
dbug bubble	Begin the **dbug** sample session.

B.4 Introduction to the Alphanumeric Interface

B.4.1 Opening Screen Layout

COMMAND	EXPLANATION
	The **dbug** session begins with the appearance of the **dbug** frame on the screen (Figure 3-4). The **dbug** frame contains two windows.
	The two windows appear one above the other. The upper window is called the CODE window and the lower window is called the DIALOG window. The CODE window displays the code of the current program under debug. This code belongs to the program: `bubble.c`.
	The DIALOG window supports two logical functions: 1) to provide a line for entering commands, and 2) to provide an area for echoing commands and displaying debugger messages.
	The line where the cursor is located is called the command line. The command line usually is identified by the prompt "(dbug)". **dbug** displays the prompt after a command completes execution.
	One way to enter a command along the command line is by typing the command through the keyboard.
	The output area of the DIALOG window displays the session history, including user commands and **dbug** prompts and echoes.
wdisplay program	Display the PROGRAM window. See Chapter 3 for a description of this window.
wmove code vrd l20	Shrink the CODE window.
wmove dialog vrd l20	Shrink the DIALOG window.

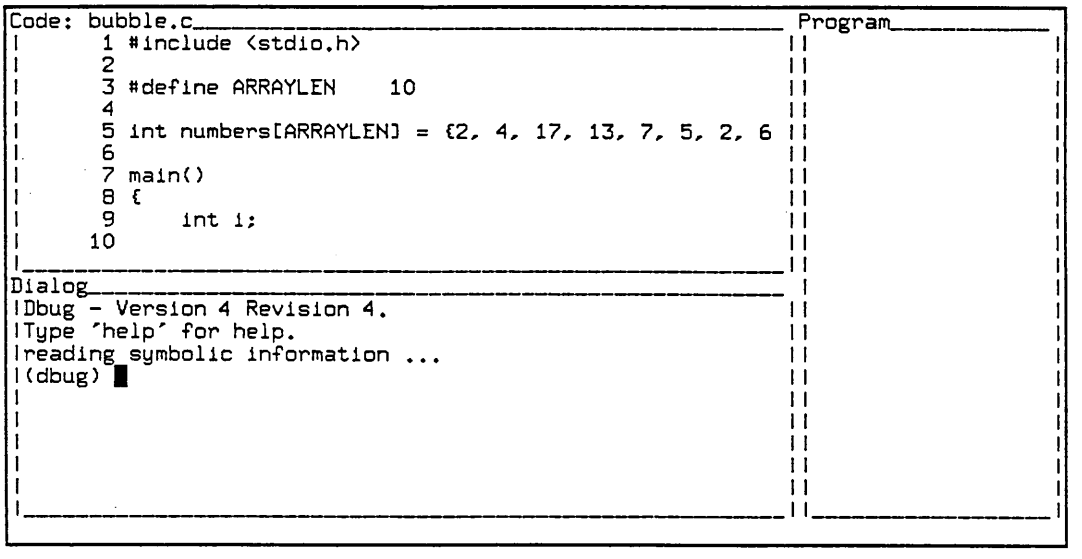


Figure B-1. DBUG Frame after the Code Window is Displayed

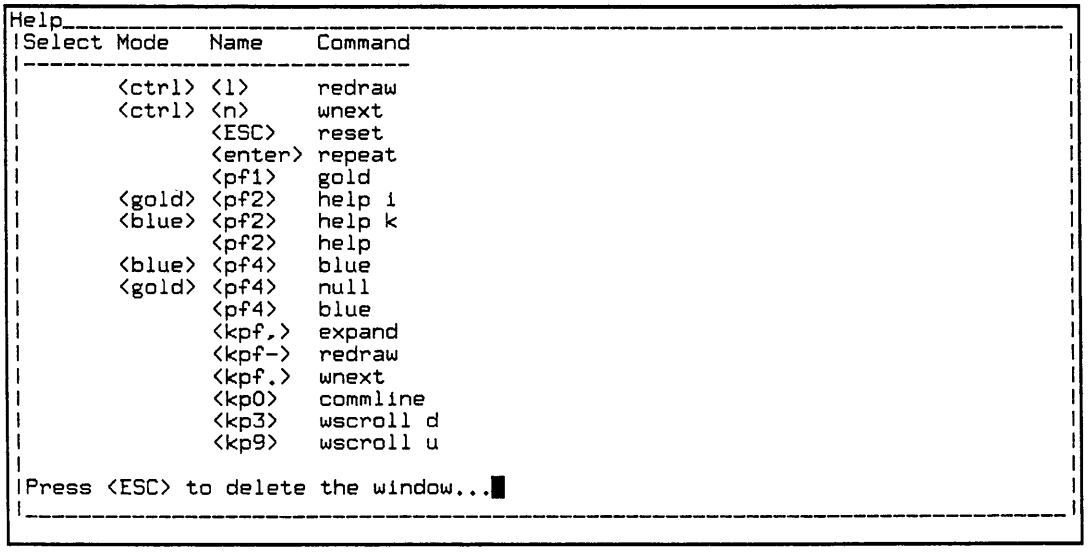


Figure B-2. The HELP Window

B.4.2 Moving Between Windows and Entering Commands

COMMAND	EXPLANATION
	From within the **dbug** frame, you may only write or do work in one window at a time. The window where the cursor is currently placed is known as the *selected window*. The selected window defines where work may take place. For example, you may only scroll through the DIALOG or CODE windows when the cursor is within their borders. In some cases, a command parameter will be a window. Window commands will take the selected window as a default parameter if nothing else is specified. It is possible to change the selected window by executing the **wnext** command, or by pressing a function key defined as **wnext** (<ctrl> <n> or <kpf.> by default).
<ctrl> <n>	Move the cursor from the DIALOG to the CODE window.
<ctrl> <n>	Move the cursor from the CODE to the PROGRAM window.
<kp0>	Press the keypad key marked with a zero (0). This key has been defined to execute a command which brings the cursor directly to the command line from wherever its current position may be. Let's enter a command. A valuable command at this point would be one which describes the power of various previously defined function keys.
help keys	If you've done everything right, like typed without spelling errors and pressed return, then a new window has taken over the screen (Figure B-2). This window is called the HELP window, and lists the power of various predefined function keys. Let's look through the list and execute a few of these functions right now.
<ESC>	Reset. The help screen disappears.
<pf4> <pf2>	Display the help keys once again, this time by pressing function keys.

COMMAND	EXPLANATION
<pf1> <pf2>	List the commands for manipulating the **dbug** windows. Note that the window is divided into two parts. The top half, called interface line commands, lists those commands that must be entered along a command line via the keyboard. The bottom half is called interface nonline commands, these are commands that are entered through predefined function keys.
<ESC>	Reset the screen.
wdelete code	Delete the CODE window.
wdisplay code	Display the CODE window.
	Now that you're familiar with all the options described in the HELP window (a small, but very handy subset of the complete range of commands available to you), we can define a function key that can carry out one of these commands. The ability to define function keys will save you countless seconds of typing time. So without any further ado, let's define <ctrl><d> as our function key for executing the "**wdelete**" command (a command for removing a window from the screen).
kdefine wdelete	Begin the procedure for defining a function key to have the value **wdelete** (window delete).
	The message "Press a function key:" appears in the output area of the DIALOG window. Simultaneously, the "(dbug)" prompt disappears from the command line.
<ctrl><d>	Press <ctrl><d>. The "(dbug)" prompt reappears on the command line. <ctrl><d> now has the function: **wdelete**.

COMMAND	EXPLANATION
	We mentioned before that the *selected window* is that window on which a particular command will be executed by default. Thus, instead of specifying "move code", or "wdelete code", you may, when code is the selected window, simply invoke **wdelete** (for example) to see the CODE window disappear from the face of the screen.
<ctrl> <n>	Move the cursor from window to window (press the <ctrl><n> command as often as necessary) until the cursor is in the CODE window. Let's execute the **wdelete** command once again.
<ctrl><d>	Execute the **wdelete** function key. There is no need to specify CODE, because it is already the selected window.
wdisplay code	Redisplay the CODE window.
	Now that you're familiar with the basics of the **dbug** interface, we can turn our attention to debugging the sample program.
	The first thing we must do is leave **dbug**, learn a few things about the sample program, run the program, and see if we, in fact, need to debug it.
quit	Leave **dbug**.

B.5 Explanation of the Sample Program

This section describes the sample program to be debugged during the sample session.

B.5.1 The Sample Program Logic

The program is designed to execute a bubble sort. The bubble sort algorithm reorders a set of given numbers in descending order by comparing two adjacent numbers, and exchanging them if the number to the right is greater than the number to the left. The sorter compares every pair in the set until it reaches the end of the list. The sorter continues to make passes on the set until all numbers appear in descending order.

The following table displays bubble sort algorithm applied to a set with four values ordered as follows: 1 4 2 3.

NUMBER SET	PROGRAM OPERATION
1 4 2 3	(0) Original state of set
4 1 2 3	(1) Pass 1, values 1 and 4 compared and exchanged.
4 2 1 3	(2) Pass 1, values 1 and 2 compared and exchanged.
4 2 3 1	(3) Pass 1, values 1 and 3 compared and exchanged.
4 2 3 1	(4) Pass 2, values 4 and 2 compared but not exchanged.
4 3 2 1	(5) Pass 2, values 3 and 2 compared and exchanged.
4 3 2 1	(6) Pass 2, values 2 and 1 compared but not exchanged.
4 3 2 1	(7, 8, 9) Pass 3, values 4 and 3, 3 and 2, and 2 and 1 compared, but not exchanged. Bubble sort complete.

B.6 The Sample Program

The C language program that implements this algorithm is found in a file called `bubble.c`. The contents of this file appears on the following page.

Program variables include `numbers`, `total_count`, `change_count` and `save`. `ARRAYLEN` equals the number of values in the list. `numbers` is an array of length `ARRAYLEN` which holds the numbers in the set. `total_count` counts the total number of times an exchange takes place between two adjacent array cells. `change_count` also counts exchanges, but is reset to zero each time the sorter begins a new pass on the set. `save` is a temporary variable which acts as a holding area for exchanges between two adjacent array cells.

The first program activity is to print the array before it is sorted. The program then calls a routine called `bubble_sort` which performs the bubble sort logic. The routine is exited when a pass is completed without any exchanges taking place (`change_count` = 0) or when the total number of passes exceeds the theoretical maximum (`total_count >= ARRAYLEN` squared). The program then prints the result of the run.

```
#include <stdio.h>

#define ARRAYLEN        10

int numbers[ARRAYLEN] = {2, 4, 17, 13, 7, 5, 2, 6, 9, 15};

main()
{
    int i;

    printf("The array before the sorting:\n");
    for (i = 0; i < ARRAYLEN; i++)
            printf("%d ", numbers[i]);
    printf("\n\n");

    printf("The sorted array after %d exchanges:\n",
      bubble_sort());
    for (i = 0; i < ARRAYLEN; i++)
            printf("%d ", numbers[i]);
    printf("\n");
}

int bubble_sort()
{
    int i;
    int change_count, total_count;
    int save;

    total_count = 0;
    do
    {
            change_count = 0;
            for (i = 0; i < ARRAYLEN; i++)
                    if (numbers[i] <= numbers[i + 1])
                    {
                            save = numbers[i+1];
                            numbers[i+1] = numbers[i];
                            numbers[i] = save;
                            change_count++;
                            total_count++;
                    }
    }
    while (change_count > 0 && total_count <= ARRAYLEN * ARRAYLEN)
    return(total_count);
}
```

B.7 The dbug Session

B.7.1 Run the Sample Program

COMMAND	EXPLANATION
bubble	Run `bubble` from the system shell and look at the results. The program prints: The array before sorting: 2 4 17 13 7 5 2 9 6 15 The sorted array after 101 exchanges: 543516756 17 15 13 9 7 6 5 4 2 Notice anything strange about the output? For one thing, the number 543516756 (or a similar unrealistic number) was not part of our original input set, and for another, the program carried out 101 exchanges (more than the theoretical maximum) before exiting. Let's use **dbug** to find the problem.

B.7.2 Running dbug and Devising a Debug Strategy

COMMAND	EXPLANATION
dbug bubble	Run **dbug** Let's turn our attention to the **dbug** frame. More specifically, let's examine `bubble.c` in the CODE window.
<ctrl><n>	Place the cursor in the display area of the CODE window.
up-down arrows	Scroll through `bubble.c` by pressing the up and down arrows. You may also scroll by using the pg-up, pg-down keys (default setup for pg-up is <kp9>, and for pg-down is <kp3>). The first question we must address as we examine `bubble.c` is where to look for the bug. If we glance at the output we see that the first bug appears only after we've called the routine called `bubble_sort`. So our first task will be to run the program until we reach the `bubble_sort` routine in line 23, then examine different variables in order to determine where and how the problem occurs.
<kp0>	Go to the command line.

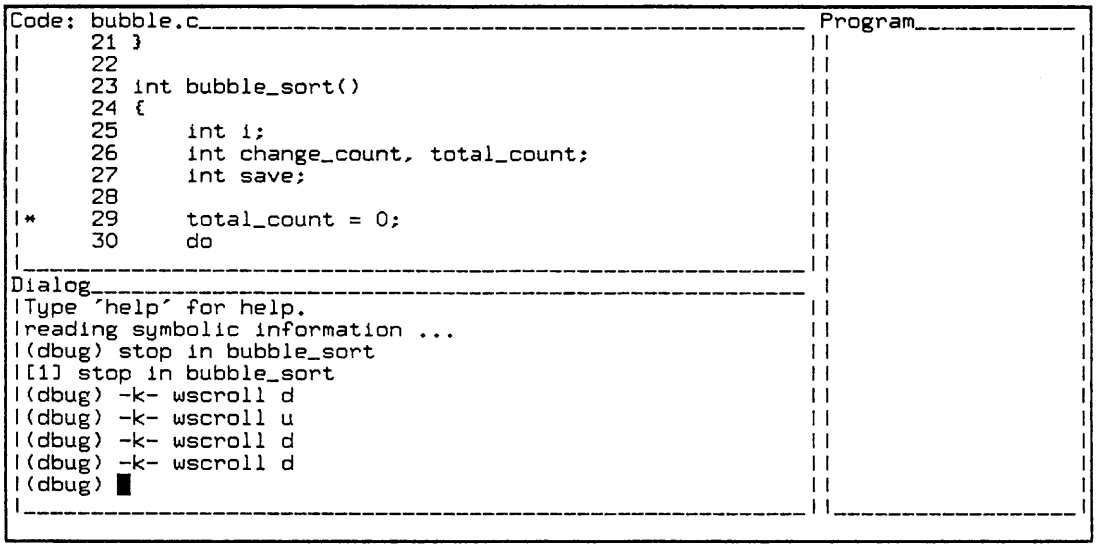


Figure B-3. Breakpoint Marked

B.7.3 First Debug Commands

COMMAND	EXPLANATION
stop in bubble_sort	Set a flag which tells the program to halt when it reaches the first executable command in `bubble_sort`. Notice that this command is `total_count = 0` in line 29, and that it is now marked by an asterisk (*) along the lefthand border of CODE window (Figure B-3).
run	Execute `bubble.c` Notice that an arrow appears on the left side of the CODE window screen and points to the current program line under execution. This is line 29, the first executable line of `bubble_sort`, and is also, not coincidently, the point where we told the program to stop. Notice that this information is also displayed in the DIALOG window. Also notice that the `bubble.c` program message is displayed in the PROGRAM window.
step	Continue to execute the program one executable instruction at a time. Notice that the arrow moves to the current line under execution.
step	Execute the next line. You may find it boring to type "step" over and over. One short cut available to you is to type "s", the alias for **step**.
s	Step one more line by executing the alias for **step**.
p numbers	We can also print the value of different variables. "p" is the alias for print. `numbers`, of course, is the array where our set is stored. Notice that the array is in its original state. No exchanges have taken place.
alias	This command prints a list of all available aliases.

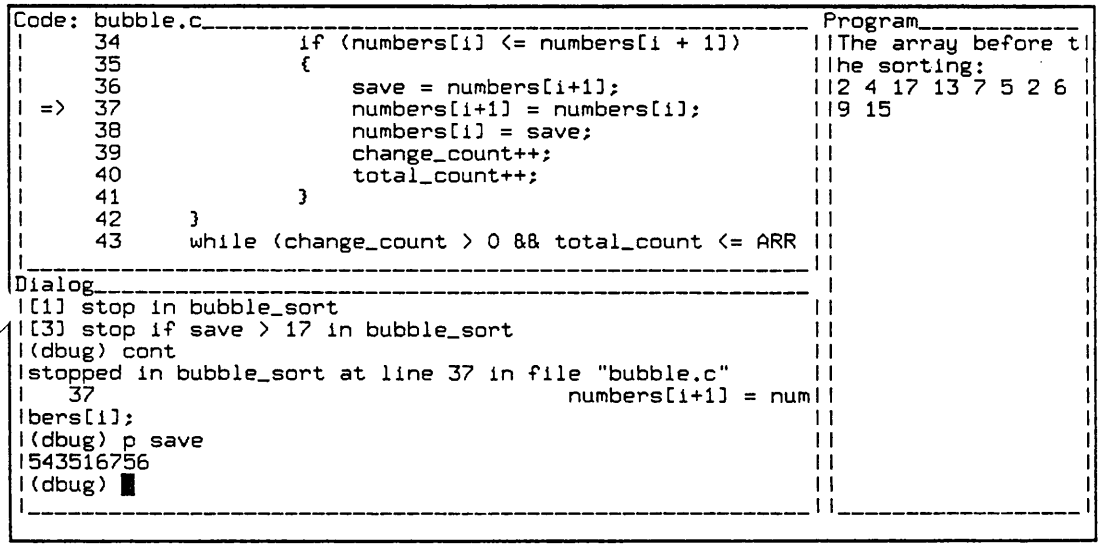


Figure B-4. p save

COMMAND	EXPLANATION
s	Step. This process of single stepping, while potentially fruitful, is also time consuming, and perhaps not really appropriate for our current needs. Let's try another approach. Notice that all exchanges go through the temporary variable called `save` (line 36). That is, when `numbers[i] <= numbers[i+1]` then `save = numbers[i+1]` and `numbers[i] = save`. Thus, if we can identify the first time `save` receives an erroneous value, we'll be well on our way to discovering the bug.
stop if save > 17	Set a breakpoint if `save` gets a value that's greater than 17, the largest known number in our set. Unlike our earlier breakpoint, this one is not identified by an asterisk (*) because it doesn't have a specific location in the program. On the other hand, its scope does not extend beyond the `bubble_sort` routine.
status	List all of the currently declared breakpoints. There are two. The number to the left has been assigned by **dbug** and is called the *event number*.
cont	Continue program execution until the "stop if" breakpoint is reached. Notice that the program stops after a few moments. This is an indication that the value of `save` exceeds 17.
p save	Print the value of `save`. It equals the sinister 543516756 (or something equally wrong) (Figure B-4). Let's examine the array index to determine when this occurred.

COMMAND	EXPLANATION
p i	The "print i" command yields the information that the array index equals 9 when `save` gets the erroneous value. This means that `save` gets the erroneous value in line 36, from the equation: `save = numbers[i+1]` (or in others words: "`save = numbers[10]`").
	How, could array index 10 receive such a value?
	The answer is clear. According to C language conventions, array indices begin from 0 and continue to n-1 (where n equals the size of the array). This means, an array of size 10 would have the indices: 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9. Our array is also of size 10 (`ARRAYLEN = 10`). This means that array `numbers` is composed of variables `numbers[0]`, `numbers[1]`,...`numbers[9]`. Variable `numbers[10]` is not a part of the array.
	Our problem comes from the fact that we've referenced an invalid variable (`numbers[10]`).
	The solution is to prevent array index `i` from exceeding 8, then the maximum value attainable by `i+1` will be 9 (instead of 10, as is currently).
	Let's fix the program.
q	Quit **dbug**.

B.7.4 Fixing the First Bug

COMMAND	EXPLANATION

	Edit `bubble.c`.
	
	Go to line 33 where the loop `bubble_sort` is defined. Replace the line:
	
	`for (i=0; i < ARRAYLEN; i++)`
	
	with:
	
	`for (i=0; i < ARRAYLEN - 1; i++)`
	
	This ensures that the loop continues from 0 to 8, instead of from 0 to 9. Save the change.

cc -g bubble.c -o bubble	Recompile the program. Be sure you are using the GNX C compiler.
bubble	Let's run the program again to see if it runs correctly. The numbers seem to be in the right order, but the output message states: The sorted array after 101 exchanges. Can it be that the program successfully carried out the sort after 101 exchanges? This can't be. It is theoretically impossible - even if the array were completely backwards, for the program to need more than 100 exchanges to sort the program correctly. Something is wrong, and we should use **dbug** again.
dbug bubble	Run the debugger.

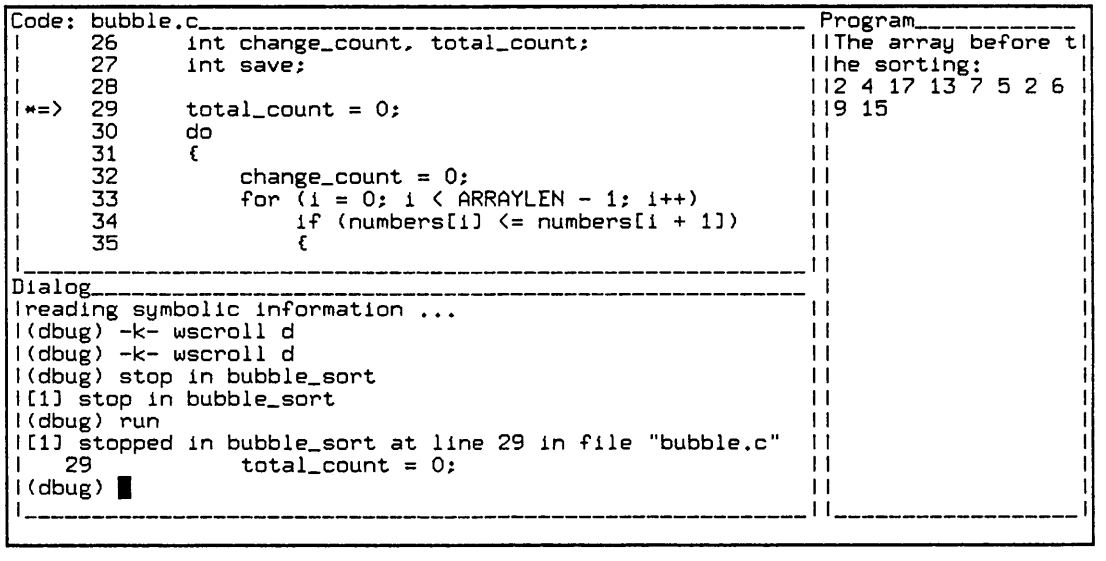


Figure B-5. stop in bubble_sort

B.7.5 Finding the Second Bug

COMMAND	EXPLANATION
	Once again we want to begin by setting a breakpoint in the `bubble_sort` routine with the command stop in bubble_sort. We take this action because we want to determine why the variable `total_count` returned the value 101 after exiting `bubble_sort`.
<ctrl><n>	Position the cursor in the CODE window.
<kp3> (pg dn)	Scroll through the display of `bubble.c` until line 23: `int bubble_sort()` is in view.
<kp0>	Go to the command line.
stop in bubble_sort	An asterisk (*) appears next to line 29, the first executable code in the `bubble_sort` function.
run	Run the program. Notice that execution stops to the side of the first executable command in the `bubble_sort` function. This is line 29 which states `total_count = 0` (Figure B-5).
<ctrl><n>	Now move the cursor into the **CODE window.** From here we may take a closer look at the program code, and decide what to do next.
up - down arrows	Gently scroll through the CODE window. Keep in mind, as we scroll, that the evidence of the bug comes from the fact that the value of `total_count` is too high, at least by one (and it would seem that performing even 100 exchanges is too much). Since checking each and every exchange would be too clumsy and time consuming, we would be advised to put in a breakpoint after each complete pass on the number set. Then we could check the array `numbers` after it had undergone a complete pass of the sorter.
<kp0>	Go to the command line.

COMMAND	EXPLANATION
stop at 33	Issue the **stop at** line 33 command.
cont	Issue the **cont** command. The program run continues until it reaches line 33, where we placed the breakpoint. Let's print the contents of the array `numbers` to determine if its values are in the correct order before beginning the first pass.
p numbers	Print the contents of `numbers`. Look in the DIALOG window. The array prints out the following numbers in the following order: 2, 4, 17, 13, 7, 5, 2, 6, 9, 15 This is the order they should be in before sorting. Let's consider a strategy. Continue the same procedure just executed: continue the run, stop at the breakpoint and print the value of `numbers`. We'll do this until the problem is detected.

COMMAND	EXPLANATION
	But first, let's make our job a little easier by defining a couple of function keys to carry out some of the more tedious commands.
kdefine cont	Begin the procedure for defining a function key to have the value **cont** (continue).
	The message "Press a function key:" appears in the output area of the DIALOG window. Simultaneously, the "(dbug)" prompt disappears from the command line.
<ctrl><o>	Press <ctrl> (control) o. The "(dbug)" prompt reappears on the command line. <ctrl><o> now has the function: **cont**.
	Now let's define a key to invoke the command "print numbers". This is a more complex command than before (we have to take the parameter `numbers` into account). We'll do the job in two stages. First, we'll create an alias for "print numbers" and next, we'll assign that alias to a function key.
alias pr "print numbers"	Give the alias "pr" the value "print numbers".
kdefine pr	Begin the procedure for defining a function key to have the value "pr" (print numbers).
	The message "Press a function key:" appears in the output area of the DIALOG window. Simultaneously, the "(dbug)" prompt disappears from the command line.
<ctrl><p>	Press <ctrl> (control) p. The "(dbug)" prompt reappears on the command line. <ctrl><p> now has the function: print numbers.

COMMAND	EXPLANATION
<ctrl><o>	Invoke the **cont** command. Wait for the program to reach the breakpoint. Note that the DIALOG window makes a distinction between how a command is issued when it echoes the program conversation. Echoes of commands issued from the command line are preceded by the indicator "(dbug)" (example: **(dbug)** `p numbers`), while echoes of commands issued through a keypad key or function key are preceded by the indicator "-k-".
<ctrl><p>	Print the value of the array `numbers`. Notice that after the first pass on the number set, the number which had previously been in the second position is now in the first position (4), and the number which had previously been in the first position, is now in the last position (2). This is evidence that at least until now, the sorting is proceeding correctly.
<ctrl><o>	Issue the **cont** command. Wait for the program to reach the breakpoint.
<ctrl><p>	Print `numbers`. Notice that the sorting mechanism seems to be proceeding normally, as higher numbers gravitate towards the front, and lower numbers move towards the back.
<ctrl><o>	Issue the **cont** command. Wait for the program to reach the breakpoint.
<ctrl><p>	Print `numbers`.
<ctrl><o>	Issue the **cont** command. Wait for the program to reach the breakpoint.

COMMAND	EXPLANATION
<ctrl><p>	Execute the **print numbers** command. Until now everything seems to be ok. It could be, however, that our problem lies with `total_count` - that it is receiving an incorrect value. Let's print its value.
p total_count	Execute the **print total_count** command. The value of `total_count` is 25. This seems reasonable, so let's simply continue.
<ctrl><o>	Execute the **cont** command.
<ctrl><p>	Repeat the **print numbers** command.
<ctrl><o>	Execute the **cont** command.
<ctrl><p>	Print numbers. This mode too, you may have noticed, can become a little tedious. **Dbug** provides you with still another way for entering commands. This option allows you scroll through the history of commands issued through the command area, and re-issue those commands that are of interest to you, by pressing return. Before going to the exact details of how to do this, let's first execute a couple of key commands through the command prompt. We do this because the command area buffer only saves those commands issued through the keyboard.
cont	Continue the run.

B.7.6 Scrolling Through the Command Window

COMMAND	EXPLANATION
p numbers	Print numbers. Now let's execute the same command series by scrolling through the command prompt buffer (with the up-down arrows) and pressing return.
up arrow (2x)	Scroll back through the history of the command line of the command line until the **cont** command appears.
press return	Execute the **cont** command.
up arrow (2x)	Scroll back until the **print numbers** command appears.
press return	Execute the **print numbers** command. At this point everything looks all right. The number set has been reordered. As designed, the program should now end. Let's see what happens if we execute **cont** again.
up arrow (2x)	Scroll back through the history of the command line of the command line until the **cont** command appears.
press return	Execute the **cont** command.
up arrow (2x)	Scroll back until the **print numbers** command appears.
press return	Execute the **print numbers** command. Watch the result! The set remains in the same "correct" order, but the program fails to exit. Let's look at `total_count`, to determine if it's still making exchanges.

B.7.7 Narrowing Down the Problem

COMMAND	EXPLANATION
p total_count	`Total_count` equals 36. The program continues making exchanges even after the number set has been reordered. Let's try to find out why it continues executing `bubble_sort`.
	First we put a breakpoint at the point where the exchange occurs.
<ctrl><n>	Go to the program display in the CODE window. Scroll to line 36. Let's set a breakpoint.
<kp0>	Go to the command line.
stop at 36	Now we can stop and examine the contents of `numbers[i+1]` and `numbers[i]` when they trade values.
	Next, let's clear the earlier breakpoint.
clear 33	Execute the **clear** command.
	A message appears in the DIALOG window: `1 breakpoint deleted at bubble.c:33.`
<ctrl><o>	Execute the **cont** command.
	Now that the program has executed another pass, let's examine the numbers it is trying to exchange.
p numbers[i+1]	The value of `numbers[i+1]` is `2`.

COMMAND	EXPLANATION
p numbers[i]	The value of `numbers[i]` is 2. This means that the program is exchanging equal values. Let's examine value of `i` when it does this.
p i+1	`i+1` equals `9`.
p i	The value of `i` is `8`.
	It would seem that our program works fine when it compares two inequalities, but just doesn't stop when it compares two equal numbers.
	Look at the "if" statement which calls for the exchange in line 34: `if (numbers[i] <= numbers[i+1])`. Here is our problem. We don't want the program to execute the exchange if the values of two neighboring numbers are equal. We must fix the errant line to read:
	`if (numbers[i] < numbers[i+1])`
quit	Quit **dbug**
	Edit bubble.c. Make the change.
cc -g bubble.c -o bubble	Compile the program.
bubble	Run the program.
	The program generates the following message:
	The array before sorting: 2 4 17 13 7 5 2 9 6 15
	The sorted array after 27 exchanges: 17 15 13 9 7 6 5 4 2 2
	Everything looks good. This terminates the tutorial session of **dbug**, but by no means did we cover the full range of **dbug**'s capabilities. Please refer to the previous chapters of this reference manual for the description of many additional commands and advanced options.

Appendix C

COMMAND LIST BY FUNCTIONAL GROUP

C.1 Introduction

This appendix lists all available commands by functional group.

C.2 Execution And Tracing Commands

COMMAND	EXPLANATION
begin *objfile* [*corefile*]	Begin debugging of a new file
run [*args*] [< *ifile*] [> *ofile*]	Run the loaded program
rerun [*args*] [< *ifile*] [> *ofile*]	Rerun the loaded program
stop if *condition* **stop at** *source-line-number* [**if** *condition*] **stop in** *procedure* [**if** *condition*] **stop** *variable* [**if** *condition*] **stop at** *line* (**if** *name* [**and** *mask*] [**not**] **in** *range*) [/ [*size*] *count*]	Set breakpoints (source level)
trace [**if** *condition*] **trace** {*line* | *procedure*} [**if** *condition*] **trace in** *procedure* [**if** *condition*] **trace** *variable* [**in** *procedure*] [**if** *condition*] **trace** *expression* **at** *line* [**if** *condition*]	Traces variables and execution
status [> *filename*]	List active breakpoints and traces
clear [*number*]	Clear breakpoints
delete *number* [*number* ...]	Delete breakpoint and trace events
cont [*signal number* | *signal name*]	Continue program execution
step	Step over current source line
next	Step over current source line (skip proc. calls
return [*procedure*]	Return from procedure to *procedure*
cont until (*name* [**and** *mask*] [**not**] **in** *range*) [/*size*]	Continue execution until value in range.

C.3 Remote Mode Commands

COMMAND	EXPLANATION
config [**link** *linkname* | **node** *nodename*] [**verbose** {*on* | *off*}] [**baud** *number*] [**cpu** *name*] [**fpu** *name*] [**mmu** *name*] [**mon** *name*] [**load** {*hex* | *binary*}]	
connect [**link** *tty*] [**with** | **node** *nodename*] [**baud** *number*] [**cpu** *name*] [**mmu** *name*] [**fpu** *name*] [**mon** *name*]]	Connect to a target
load [*objfile*] [**with** [**nocode**] [**nodata**] [**protect**] [**zerofill**] [**sp** *address*]]	Load executable file to target

C.4 Printing Variables And Expressions

COMMAND	EXPLANATION
assign {*variable* | *add*} = *exp* [*radix*]	Assign a value to a variable or address
up [*number*]	Move up in call stack
down [*number*]	Move down in call stack
dump [*procedure*] [> *filename*]	Dump procedure variables
pcpu	Print all cpu registers
pfpu	Print all fpu registers
pmmu	Print all mmu registers
pbpu	Print all on-chip BPU registers
picu	Print on-chip ICU registers
pdma	Print on-chip DMA registers
ptimer	Print on-chip timer registers
pcomplex	Print on-chip complex-multiplier registers
print *expression* [, *expression* ...] [*radix*]	Print on-chip registers
whatis *symbol*	Describe a symbol
where	Print active call stack
whereis *symbol*	Find all occurrences of a symbol
which *symbol*	Print symbol qualifier
address, *address*/[*radix*] [> *file*] *address*/[*count*] [*radix*] [> *file*]	Assembly level printing

C.5 Window Commands

COMMAND	EXPLANATION
wdelete [*wname*]	Delete a window
wdisplay *wname* [**at** [*wloc*]]	Display a window
wgo *number*	Go to a line in file
wmove [*wname*] *wcorner wshift*	Move or resize window
wnext [*wname*]	Select a window
wpop [*wname*]	Pop window
wpush [*wname*]	Push window
wreset	Reset windows
wscroll [*wname*] *wshift*	Scroll window

C.6 Menu Commands

COMMAND	EXPLANATION
addmenu [**<text>** | **<expr>** | **<line>**] *command*	Add a menu entry
delmenu *entry*	Removes a menu entry

C.7 Emulator Specific Commands

COMMAND	EXPLANATION
configh [[**mon bg** | **mon fg**,*address*] [**wait number**] [**lock** {**enable** | **disable**}] [**burst** {**enable** | **disable**}] [**realtime** {**enable** | **disable**}] [**clock** {**int** | **ext**}] [**cfg** *number*] [**target** {**all** | **ignore** | [**dbg** | **int** | **nmi** | **hold**}]]	
connect [**link** *tty* | **node** *nodename*] [**with** [**baud** *number*] [**cpu** *name*] [**mmu** *name*] [**fpu** *name*] [**monname**]]	**Connect to a target.**
counter define {**none** | **time** | *condition*}	**Counts time or events**
counter status	**Print counter qualification**
map [*range* [**rom** | **ram** | **tram** | **trom**] [**with copy**]]	**Map emulation memory**
reseth	**Reset the emulation CPU**
stoph if *stop_condition*	**Set a hardware breakpoint**
traceh define [*condition*] [**bus** | **vpc** | **all**]	**Define hardware trace**
traceh format [*pin_group*:*radix* ...] [**absolute** | **relative**] [**lines**] [**disasm**] [**mnemonic**]	**Define trace display format**
traceh list [*number* | *number*, *number*]	**Display trace buffer**

traceh reset	**Reset trace definitions**
traceh start [**at** *address*]	**Start hardware trace**
traceh status	**Display current status of emulator trace**
traceh stop [**at** *address*]	**Stop hardware trace**
unmap *number*	**Delete an emulator map term**

C.8 Accessing Source Files

COMMAND	EXPLANATION
/ *regular expression* [/]	Search forward for patterns in source file
? *regular expression* [**?**]	Search backward for patterns in source file
file [*filename*]	Select current file
func [*procedure*]	Select current procedure or function
list [*procedure*]	Print source code lines of a procedure
list *fromline* [, *toline*] [**i**]	Print source code lines with optional disassembly
use *dir* [*dir ...*]	Set source search path

C.9 Key Definition

COMMAND	EXPLANATION
kdefine [*selection*] *command* [*pfkey*]	Attach function key to command
kreset	Resets function key to default

C.10 Function Key Commands

COMMAND	EXPLANATION	GRAPHIC DEFAULT	OPUS-PC DEFAULT	VT100 DEFAULT	SUN DEFAULT
reset	Remove temporary displays	<ESC>	<ESC>	<ESC>	<ESC>
repeat	Repeat last typed command	<kpf+>	<pf7>	<enter>	<R15>
redraw	Redraw the screen	<kpf->	<pf8>	<kpf->	<ctrl><l>
gold	"gold" key	<pf1>	<pf1>	<pf1>	<R1>
blue	"blue" key	<pf4>	<pf4>	<pf4>	<R4>
expand	Expand the current window	<kpf*>	<pf5>	<kpf,>	<R5>
commline	Move to the command line	<kp0>	<pf9>	<kp0>	<R13>
select	Select a window	<pf3>	----	----	----

C.11 Assembly Level Commands

COMMAND	EXPLANATION
stepi	Execute next instruction
nexti	Execute next instruction (skip subr. calls)
stopi [{**read** | **write** | **access**}] *address*]	Set breakpoints
stopi *address* [**if** *condition*]	Set breakpoint on change of address contents
stopi at *address* [**if** *condition*]	Set breakpoint at address
stopi at *address* (**if** *name* [**and** *mask*] [**not**] **in** *range*) [/ [*size*] *count*]	
tracei in *procedure* [**if** *condition*]	Trace program execution
tracei at *address* [**if** *condition*]	
tracei *address* [**if** *condition*]	
tracei [**if** *condition*]	

C.12 Command Aliases And Variables

COMMAND	EXPLANATION
alias *name* [*name* | *"string"*]	
alias *name*(*parameters*) *"string"*	Define command aliases
alias	Print aliases
set [*variable*]	Set dbug variables
unalias *name*	Remove alias definition
unset *variable*	Unset debugger variable

C.13 Miscellaneous Commands

COMMAND	EXPLANATION
config [*config parameter*]	Set/Print the configuration parameters
env	Restore the environment
help	Explain debug commands
help k[**eys**]	Explain key bindings
help i[**nterface**]	Explain window manipulation commands
catch {*signal number* | *signal name*}	Catch signals
ignore {*signal number* | *signal name*}	Ignore signals
log [*logfile*]	Save debugging session log
log [*logfile*] **with append**	
log [*logfile*] **with save**	
log [*logfile*] **with full**	
log [*logfile*] **with full append**	

quit [**with save** [*filename*]]	Terminate debugging session
protect *address range* [{**read** | **write**} **for** {**u** | **s**}] [{**set** | **clear**} {[**v**] [**r**] [**m**]}] [**start** *address*] [**on primary**] [**using** {**ptb0** | **ptb1**}]	Set memory protection
source *filename*	Execute command file

Appendix D

GLOSSARY

.dbuginit (dbug.ini on VMS and MS-DOS) The default command file that is executed by the debugger as a part of the DBUG invocation.

.gnxrc (gnx.ini on VMS and MS-DOS) A GNX target specification file that is used by GNX tools to obtain the CPU, FPU, MMU, system bus-width, and OS target specifications.

Address In the DBUG command syntax, addresses may be symbolic or absolute. Absolute addresses are numbers, which may be entered as decimal, octal or hexadecimal numbers. Symbolic addresses are specified by an ampersand (&), which must precede the symbol. The plus (+) and minus (-) operators can be used to create an address.

Analyzer pin-group assignment Assignment of symbolic names by DBUG of those CPU pins traced by the HP Emulator.

Assembly mode Assembly mode presents the disassembly of the target program. Each instruction's address is displayed on the left side of the DBUG code window. This mode is activated when compiling without the -g option (/debug on VMS).

Background monitor A monitor operation mode for HP Emulator firmware. In this mode the entire address space of the CPU is available. System exception processing is disabled (including software breakpoints).

Breakpoint Breakpoint is the mechanism used to stop debugged program execution at specified places and upon occurrence of specified conditions. Breakpoints are defined by the STOP or STOPI commands. Reaching a breakpoint allows for the examination/modification of variables, registers and memory locations.

CODE window One of 5 DBUG windows, which displays the section of your program code that is currently executing and provides a visual representation of the debugging session.

COFF Acronym for the Common Object File Format. This is the standard object file format for the Unix System V operating system, and for the GNX software tools. A COFF file contains machine code and data and additional information for relocation and debugging purposes.

Command files Contain debugger commands that can be read and executed by DBUG.

Commline key Moves the cursor to the command line. This command causes the DIALOG window to be redisplayed if necessary.

Compilation flag The flag **-g** (**/debug**) is used to produce symbolic information for the debugging of programs compiled with the GNX assembler and the C, Pascal and FORTRAN compilers.

Corefile The file that is created when the application program is terminated abnormally while running under UNIX.

Current environment Defined in DBUG as the combination of the current file and the current procedure.

Current file The source file that contains the current procedure.

Current procedure The procedure at which execution stops and control is returned to DBUG.

Current window The window in which the mouse cursor is currently located.

DBUG frame In the graphic environment, the window in which DBUG executes. When an ASCII terminal is used, it is the full screen.

dbug.save file (`dbug.sav` on MS-DOS) The default file in which the history of a debug session can be saved.

DBUG windows DBUG supports five windows: code, dialog, help, trace and program. The code window displays the program being debugged. The dialog window is for command entry. The help window lists DBUG command syntax. The trace window displays trace information produced by emulators. The program window is available only when using an ASCII terminal.

DIALOG window One of 5 DBUG windows, which displays user commands and DBUG responses. All commands to DBUG are entered in the dialog window.

Disassembly The presentation of program memory as assembly commands.

Events Breakpoints and traces. Created by the stop and trace commands and displayed by the status command.

Executable file The file that contains the COFF program being debugged. This file is used as input to DBUG. Also referred to as an `objfile`.

Expand key Expands the current window to full DBUG frame size.

Expressions General types of expressions can be used as part of DBUG command syntax. The common subset of expressions that are legal for Pascal and C

programming languages are supported by DBUG.

Foreground monitor A monitor operation mode for HP Emulator firmware. In this mode the monitor occupies a portion of CPU address space and some entries in the interrupt table. This permits system exception processing.

HELP window One of 5 DBUG windows, which provides information on the use of DBUG, window, and function key commands.

HP Emulator The HP64772 In-System Emulator for the NS32532, NS32GX32 and the NS32GX320; the HP64779 In-System Emulator for the NS32FX16, NS32CG160, NS32CG16 and NS32FX164 CPUs.
These emulators can be plugged into a target board to provide features such as real-time trace, breakpoint, counters, and memory mapping.

Hanging menu A permanent menu which displays a variety of frequently used commands.

Log file Consists of a log of the debugging session.

Marked text Text in the code window is marked by using the following methods: dragging the cursor, delimiting the area to be highlighted, selecting a word, selecting a line.

Memory blocks Memory blocks can be referred to as either part of the target or emulation memory. Memory blocks that are mapped to emulation memory may be characterized as either RAM or ROM.

Native mode In native mode both DBUG and the program being debugged run on the host system. Native mode is available only on Series 32000-based computers (SYS32/20/30/50) running under the UNIX Operation System.

Node name The name of the target system as recognized by the host. Used with ethernet communications.

Option The UNIX term for a parameter, specified on the command line, that is used to control the utility.

Qualifier The VMS term for a parameter, specified on the command line, that is used to control the utility.

Radix parameter The radix parameter specifies the output format of the print command.

Redraw key Redraws the DBUG window.

Remote mode In remote mode DBUG runs on the host system while the program being debugged runs on the Series 32000 based target system.

Repeat key Repeats the last command issued from the command line.

Reset key Removes temporary menus and the HELP window.

Scroll bar The scroll bar is used to scroll through the output area (dialog or trace window).

Selected window A window that is selected as a default parameter for DBUG commands.

Selection menu The selection menu contains commands for printing and identifying variables, and setting and clearing breakpoints.

Select key Defines the current window as the selected window.

Serial link The RS232 line that connects the host running DBUG with the target board.

Shift specification Changes the size of the selected window by stretching it from the identified corner(s). Specifying all four corners will move the entire window.

Source file The file containing the program code. Can be either an HLL or assembly file. DBUG uses source files to access source information.

Source mode Source mode presents the program source code as originally written.

Symbol Symbols are defined as program identifiers. These include: variables, procedures, type definitions, registers, and file and module names.

TRACE window One of 5 DBUG windows, which is used with ISE configuration. It displays the results of the hardware trace output.

Target board (system) The Series 32000 based system that runs user applications.

Trace This capability allows you to trace your program without stopping its execution. You can trace variable values, calls to procedures, and execution of particular procedures.

Virtual PC A virtual address that is executed by the 532 and GX core CPUs and appears on specific CPU pins. The HP Emulator can display these pins.

INDEX

D

S

T

U

V

W