CompactRISC

Debugger Reference Manual

Part Number: 424521426-004

February 1997

REVISION RECORD

VERSION	RELEASE DATE	SUMMARY OF CHANGES
0.6	August 1995	First beta release.
0.7	January 1996	Minor changes and corrections.
1.0	August 1996	CR16A Product Version. CR32A Beta Version.
1.1	February 1997	Minor changes and corrections.

PREFACE

Welcome to the CompactRISC Debugger. The debugger can be used for symbolic debugging of high-level language programs generated by the CompactRISC C Compiler, as well as for assembly language programs generated by the CompactRISC Assembler.

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

CONTENTS

FIGURES

Chapter 1
OVERVIEW

1.1 INTRODUCTION

The CompactRISC Debugger is a GUI debugger for the CompactRISC family. It is available for IBM PC-compatible computer, running Mictosoft Windows™, Windows 95 or Windows NT. It supports symbolic debugging of code written in C, or in assembly language. The Release Letter, supplied with your software, contains instructions for the installation and configuration of the debugger and the remaining CompactRISC tools.

To help you evaluate a CompactRISC microprocessor, or to develop software for it, National Semiconductor provides a target board kit, for your particular member of the CompactRISC family, consisting of a Debugger Extension Box (DEX) and an Application Development Board (ADB).

Intended audience

This manual assumes a knowledge of the C language, and is addressed to embedded-systems engineers involved in:

- evaluating a CompactRISC processor
- developing software for a CompactRISC processor

This manual is applicable to processors and Evaluation/Development Boards for the entire CompactRISC family. For information unique to a specific processor and board you are using, see the appropriate documents.

1.2 DEBUGGER FEATURES

The CompactRISC Debugger offers the following features and benefits:

- Supports C or assembly programs
- Initialization files for automatic initialization and restart
- On-line Help
- User-configurable buttons
- Executes commands from a preconstructed command (input) file
- Records debugging session, for later analysis and/or re-execution. Facilitates automation of regression test suites
- Function call stack display in a window
- Traverse data structures (lists, etc.) through pointers using a mouse
- **ALIAS** and **SET** commands, for customizing the command interface
- Recall, editing, and re-execution of previous commands
- Multiple commands on a single line
- Mixed-mode debugging
- Save and restore debug environment
- Call user subroutines/functions from command level
- Access to host file system using Virtual I/O (see Section 5.1 of the *CompactRISC C Compiler Manual*)

1.3 DEVELOPMENT ENVIRONMENT

The symbolic debugger, with its instruction-level simulator, comprises a complete, software-based, evaluation or development environment. It supports all the capabilities necessary for effective, efficient development of your target program. However, if your requirements (speed, etc.) are such that you require hardware to aid in your development, you can purchase a target board kit, consisting of a pair of DEX and ADB Boards, from National Semiconductor Corporation, and connect them to your system for hardware-assisted debugging. In terms of using the debugger, there is no noticeable difference to you in using either the Simulator or a Development Board kit. You use exactly the same tools and interface.

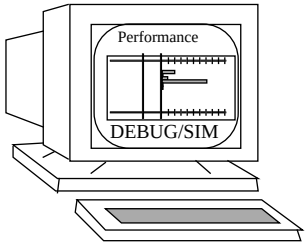

Figure 1-1. Debugging Using a Functional/Performance Simulator

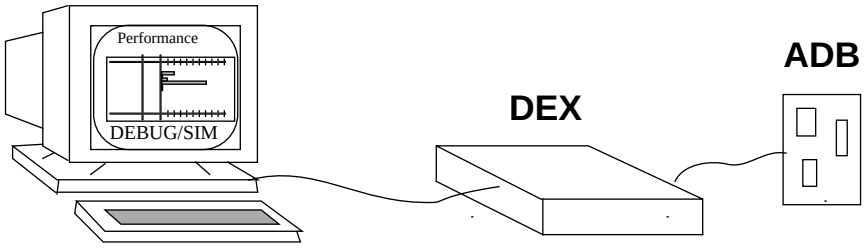

Figure 1-2. Debugging the Program on a Target ADB Board

1.3.1 Instruction-Level Simulator (ILS) Environment

The major functions of the Simulator are:

- Provide a tool for evaluating a particular target chip (CPU) before the first silicon is available.
- Provide a software tool (less expensive than hardware) for software application development.

Features

The Simulator provides the following functional capabilities:

- Instruction-level simulation of the operation of your program.
- Instruction traces of the operation of all or part of your program.
- Performance evaluation of all, or parts, of your program, including accurate execution-time estimates and program profiling. To make this more accurate, memory wait states may be simulated.

The following two features, while operable, are not fully supported.

- The ability, in primitive form, to simulate the effect of peripherals.
- The ability to simulate the effects of up to thirty-two unique non-maskable and maskable interrupts and traps on program flow.

Limitations

The Simulator has the following limitations:

- ILS does not simulate the hardware aspects of instruction execution, such as caching or pipelining. However, it does include factors in its performance estimation to allow for caching, etc.
- ILS simulates all CompactRISC instructions, except **`WAIT`**.
- Performance information available from the ILS is a very accurate estimate, but is not exact.
- The ILS does not simulate the operation of any memory management peripherals. Therefore, any addresses used within a program are physical, not logical or virtual.
- The maximum real memory that can be simulated is limited, and is dependent on the host.

1.3.2 Board Environment

The development board kit is comprised of two boards, the Debugger Extension Box (DEX) and the Application Development Board, which communicate with one another and together provide a hardware development environment in which you can develop your CompactRISC applications. The DEX is a standard board; the ADB is product-specific, supporting a particular member of the CompactRISC family.

The DEX contains a resident Monitor which provides a number of capabilities (Section 1.3), most of which are supported by CompactRISC Debugger. You may take advantage of some of the hardware capabilities not directly supported by CompactRISC Debugger by using it to send commands directly to the monitor and to display communications to and from the monitor. The monitor commands are described in Appendix D.

Board/host connection

You can connect the board to your host system via one of the UART (RS-232) connections, or via ETHERNET.

See the relevant board Reference Manual for general installation and operating instructions.

Both the DEX and the ADB have a monitor to aid debugging which provides the following features:

- Control of program execution and debugging.
  The debugger can download your program to on-board memory, after which the monitor can execute commands for starting, stopping, single-stepping, and setting breakpoints within your program.
- Data exchange.
  The DEX works with the debugger to display and change data located in the on-board memory, and CPU general-purpose and special-purpose registers.
- Run-time environment.
  The ADB-monitor contains routines that can be used by your program to access the host computer file system via the debugger (Chapter 2, Virtual I/O).
- Reset and initialization.
  Upon reset, the Monitor initializes the board. It resets the board I/O devices and the CPU, and performs a small set of diagnostic tests to ensure that the board is operational.

See Appendix D for information about the Monitor interface and command set. See also the appropriate development board Reference Manual, and the *CompactRISC Toolset - Debugger Extender (DEX) Reference Manual.*

1.4 MANUAL ORGANIZATION

Chapter 2 *Debugger Features*, explains how the commands are used to perform various debugging operations. It describes the CompactRISC Debugger environment, and shows how to invoke it.

Chapter 3 *Debugger User Interface*, describes the debugger graphical user interface, including the purpose and content of the windows and the CompactRISC Debugger menus.

Chapter 4 *The Debugger Commands*, is a reference section which describes the syntax of the commands, the equivalent GUI facility and provides some examples.

Chapter 5 *Instruction-level Simulator Commands*, describes the simulator commands. The format is similar to Chapter 4.

Appendix A *Quick Reference Guide*, summarizes the CompactRISC Debugger commands and arguments.

Appendix B *Trouble-shooting Hints*, suggests solutions to some common problems, including responses to error messages, and RS-232 or ETHERNET interface problems.

Appendix C *Peripheral Simulation System - Example*, contains listings for the files giving an example of the usage of the Peripheral Simulation System (PSS) and virtual I/O in a C-language program.

Appendix D *Monitor Interface*, describes the interface and command set of the DEX Monitor. These commands may be executed through the CompactRISC Debugger.

Appendix E *Debugger Limitations*, lists the known limitations.

Appendix F *Performance Simulation Configuration File*, shows how to configure the wait-state parameters of the simulated memory through a simulator configuration file..

Appendix G *Performance Simulation Trace Output*, describes detailed program execution information the simulator sends to a file.

1.5 REFERENCE DOCUMENTS

The following National Semiconductor publications provide related study and reference material:

1. CompactRISC Toolset - Introduction
2. CompactRISC Toolset - C Compiler Reference Manual
3. CompactRISC Toolset - Assembler Reference Manual

4. CompactRISC Toolset - Object Tools Reference Manual

5. Reference Manuals for each of the microprocessors in the CompactRISC family.

6. User's Manuals for the various Evaluation/Development Boards provided by National Semiconductor Corporation.

Chapter 2
DEBUGGER FEATURES

2.1 INTRODUCTION

The debugger is used to debug programs, either by executing them on a National Semiconductor development board, or with the built-in Instruction-Level Simulator. The debugger provides the same interface for both the development board and the Simulation environment.

2.2 DEBUGGING WITH THE DEBUGGER

2.2.1 Installing the Debugger

The debugger must be installed in your host as described in the release letter. It requires a number of external files (help file, resource files for window, etc.) which are supplied with the release package. These files are installed in the installation directory, which must be included in your system's **PATH** variable.

2.2.2 Debugger Initialization and Configuration

The debugger uses the following initialization and configuration files:

Global environment file (`crdb.env`) This optional file contains configuration commands for color, size and position of the windows, and communication parameters. The debugger first looks for this file in the directory indicated by the environment variable, **CRDBENV**. If **CRDBENV** is not set, the debugger looks in the current directory. See Section 2.2.13.

Local initialization file (`crdb.ini`) This optional file contains local setup commands, used to customize the debugging environment for a particular project e.g., alias definitions, standard input command file, executable file to be downloaded, source path directories and standard logging file. It must reside in the current working directory, and may contain any legal debugger command.

2.2.3 Invoking the Debugger

You can invoke the debugger by double-clicking on the debugger icon from the CompactRISC Program Group. If you plan to run the debugger from the File Manager's Run menu, use the following invocation syntax

crdb [*executable_file*] [**-e=***command_file*] [**-l=***log_file*]

Executable file

executable_file is the file name (including the path, if necessary) of the COFF file to be debugged. This file must have been generated using the CompactRISC C Compiler with the debug option. Refer to the *CompactRISC Toolset - C Compiler Reference Manual* for more information about compiling for debugging.

The CompactRISC Linker generates executable files in COFF format. A COFF file consists of two major parts: code and data that make up your program, and the symbol table, which contains debugging information about your program. The manual, *Introduction to the CompactRISC Tools* describes the procedure for compiling and linking your program.

Command file

When you invoke the debugger, it immediately executes the debugger commands in **command_file**. You can use your text editor to create a file of the debugger commands, or use a log file previously created by the debugger (Section 4-16). You can also execute commands from a file, during the debugging session, using the **INPUT** command (Section 4.15).

Log file

log_file is the file which holds the information logged (recorded) by the debugger during the debugging session. Specify the **-l** option on the command line to set up a log file to record the debugging session.

Startup and initialization

When invoked, the debugger executes commands from the two initialization files, **crdb.env** and **crdb.ini**, in that order.

If you invoke the debugger without any arguments, it completes its initialization, and awaits further commands from you.

If you specify the executable file, the debugger reads and downloads the file to the target.

If you specify the **-e** option, the debugger executes the commands from the **command_file**.

Debugging session

To enter the commands, select the command line edit area in the upper part of the Debugger Window by clicking it with the left-button of the mouse, and type in the command (see Chapter 4 for details of the debugger commands). After entering the command, press **ENTER** to process the command.

Exiting the debugger

Terminate a debugging session with the **QUIT** command, or open ***FILE*** and select ***QUIT***.

2.2.4 Accessing On-line Help

Select the **help** button, on the main menu, to obtain help information. Select a topic from the list by double clicking on the left button of the mouse.

Alternatively, enter a question mark (?) as the only argument to a command to get help with its syntax.

2.2.5 Selecting the Target

You must select the target environment in which you want to debug the program. This can be either simulator, or development board.

Simulator

Specify the **TARGET** command (section 4.38), or open ***CONFIG*** and select ***SIMULATOR***, to select the simulator.

Development board

Before you can use the development board, you must connect it to your host through either a serial or an ETHERNET connection.

Set up the communication parameters, which tell the debugger how to communicate with the board. The **COMM** command specifies the communication method, and the **TARGET** command specifies the target. You can put these commands in either of the initialization files, or execute them manually.

For example:

```
comm -l 2
comm -b 19200
target edb
```

Alternatively, you can open ***CONFIG*** and select ***EDB***.

For detailed information, and installation instructions, for your DEX/ADB, see the user's manual supplied with your board.

2.2.6 Working with Executables and Source Files

Specifying an executable file

The **DEBUG** command (Section 4.9) specifies the name of the executable COFF file to be used by the debugger, and optionally downloads code and symbols. **DEBUG** updates the Source Window (Section 3.2) display. After a file has been downloaded, the debugger issues a **RESET** command (Section 4.24).

C or Assembly mode display

SRCMODE (Section 4.31) specifies the display mode for your source files. You can specify source code only, or mixed source and assembly code.

Source file operations

The current source file, or the currently requested text file, is always displayed in the Source Window. To look at a particular area in your source file, if you know its location, use the **LIST** command (Section 4.16). If you do not know where to look, but you know something about the context, such as some structure, or variable, referenced there, you can use **FINDSRC** (Section 4.12) to find each of the possible locations. **SYNC** (Section 4.37) returns the Source Window to the execution point in the source file.

Specifying directories

If your source files are in several directories, use **SRCPATH** (Section 4.31) to tell the debugger where to find them.

CD (Section 4.6) sets the current working directory for the debugger; **INFO** (Section 4.14) displays the setting of **CRDBENV**, the current directory paths used by the debugger when searching for source files, and the names of the files currently in use.

2.2.7 Working with Breakpoints

There are two types of breakpoints: hardware breakpoints, made available by either the development board or the CompactRISC processor, and software breakpoints, made available by the processor only. The hardware breakpoints (**BREAK**, Section 4.4) are less numerous and less flexible, but operate in near real-time. Software breakpoints (**SOFTBREAK**, Section 4.30) are more numerous and more flexible but do not operate in real-time.

Hard-break

Depending upon the implementation, a hard breakpoint may support breakpointing on data read/write/any access, and PC match. You can manage your hardware breakpoint list with the **BREAK** command.

PC match with occurrence count is implemented through software, and hence affects the real-time performance.

Soft-break

SOFTBREAK provides PC breakpointing based on occurrence count and/or based on a conditional expression. Use the **SOFTBREAK** command (Section 4.30) to manage the software breakpoint list. Double-click on a particular source line to set or unset a software breakpoint.

If a software breakpoint is set on a source file line displayed in the Source Window, the affected line is marked with an “S” at the left edge of the Source Window.

Temporary breakpoints

If you do not want to use a breakpoint for a while, you can disable it and later re-enable it. You can also set a temporary breakpoint, which is automatically removed after it has been executed once. If you set a new hardware breakpoint, and expect it to be executed immediately, you must ensure that your **BREAK** list is enabled.

You can use the **WATCH** command (Section 4.41) to set up a list of variables to display and **AUTOCOMMAND** (Section 4.3) to list the commands to be executed when the next breakpoint is recognized.

The number of breakpoints you may use depends on the target. Refer to your application board's Reference Manual for the availability of hardware breakpoints.

Attaching commands to break

AUTOCOMMAND specifies a set of commands to be executed whenever the program stops after execution due to breakpoints.

2.2.8 Executing the Program

Start execution

Use **GO** (Section 4.13), to start executing your program. When the program reaches a breakpoint, the debugger stops the execution, updates the Watch Variables Window, executes the **AUTOCOMMAND** list (Section 4.3), and updates the display in the Source Window to point to the current source line or instruction.

RESET and debugging of startup code

The **RESET** command (Section 4.24) issues a software reset to the ADB with the help of the monitor on the DEX board. By default, the debugger executes up to **main()**.

If you want to get control at the physical entry point of the program, to debug your startup code, use **DEBUGMODE** (Section 4.10) to stop at the entry point, or to execute up to **main()**, the logical 'C' start function. The Source Window display is updated accordingly.

Single stepping

The debugger provides single stepping at the source level (**STEP** Section 4.34) as well as at instruction level (**STEPINS** Section 4.35). It also provides stepping over at source level (**NEXT** Section 4.19) and stepping over at the instruction level (**NEXTINS** section 4.20).

Aborting execution

Open ***EXECUTE*** and select ***ABORT*** to abort any executing command. Control returns to the debugger prompt level. If you select abort while an input file is being executed, the execution of the input file is also aborted.

If you are executing your program on a development board, you may abort execution by pressing the ISE switch on the ADB board. You may have to restart the program later.

If the Simulator is executing your program when you abort, you may reset and restart the program later.

2.2.9 Looking at Memory and Variables

Whenever the program is stopped, the debugger provides various ways of looking at the memory or program variables.

Inspecting program stack

WHERE (Section 4.42) displays the current program context, and the current function stack in the Output Window. You can also open the Stack Window to see the current call chain; open ***SHOW*** and select ***STACK WINDOW***.

Looking at memory

LIST, with **-m** option (Section 4.16), and Memory window (Section 3.2) display memory in various widths and formats. You can use the Memory window or **MODIFY** to modify the memory location.

Viewing program variables

VIEW (Section 4.40) displays the program variables in a variety of C **printf**-like formats. The debugger also provides several different windows to view variables.

The Local Variables window (open ***SHOW*** and select ***LOCAL VARIABLES***) displays the arguments, and C-local variables.

The Watch window (open ***SHOW*** and select ***WATCH VARIABLES***) lets you define a set of variables or expressions to be automatically displayed whenever the program stops.

The Symbol Window (open ***QUERY MENU***) displays any variable in the C-Program context. This window also lets you expand or contract structures, and includes a facility to get symbol-type information via this window.

Viewing registers

VIEW also displays registers, using the Register window (open ***SHOW*** and select ***REGISTER***).

Modifying memory, registers and variables

MODIFY modifies memory location and variables. You can modify registers, variables, or memory by double clicking on the entry in the Register, Query, and Memory windows respectively.

2.2.10 Virtual I/O Support

Virtual I/O enables your program, running on an ADB, or under control of the Simulator, to access the host system, normally through the debugger. The low-level virtual I/O functions are included as part of the C library. Virtual I/O operations are listed in Chapter 7, and described in the *CompactRISC Toolset - C Compiler Reference Manual.*

STDIO (Section 4.33) directs the output of I/O functions to standard files, a host disk file, the Output Window, or to the terminal from which you invoked the debugger.

2.2.11 Performance Estimation

The ability to measure the performance of various parts of the software is an important requirement of embedded system programming. For example, we need to know the time required to perform a particular task, or execute a particular function.

Profiling

You can profile the performance of your program. Profiling provides an indication of where bottlenecks occur, and where the program spends most of its execution time. The Simulator samples the program counter at a predetermined rate, and then estimates performance based on the frequency distribution of the program counter values.

To profile all or a portion of your program, open ***CONFIG*** and select ***SIM ➨ PERFORMANCE ON***.

On completion of program execution, open ***SHOW*** and select ***PERFORMANCE*** to display the Performance Window, and see a bar graph showing an execution profile of either the files or functions of your program. Open ***CUSTOMIZE*** and select **File,** or open ***CUSTOMIZE*** and select ***FUNCTION***, in the Performance Window, to select files or functions, respectively.

The bar graph reflects the percentage of execution time your program spent in each of the displayed portions (modules or functions) relative to the time spent in all the code that was executed with performance mode on. The numbers shown by the bar elements indicate the time spent in that section of code only; they do not include any time spent in any section called from this section of the code. This display includes only portions of code containing symbolics.

The simulator slows down when collecting data for program profiling. If you do not need to profile a particular part of your program, open ***CONFIG*** and select ***SIM ➨ PERFORMANCE OFF*** to disable the function. When startup code debugging is disabled, performance data collection is also disabled.

Note

We recommend that you execute the measured part of the program continuously, without interruptions such as breakpoints or single steps. Breakpoints and single-steps each add a few cycles to the total cycle count, and thus adversely affect the performance measurement.

Chapter 3 describes the graphical aspects of the Performance Window.

Tracing

With performance simulation you can get a cycle-level trace of your program, or part of it, with full information about instructions being executed and the processor pipeline status. The trace information is directed to a log file, which you specify. Select ***CONFIG ➨ SIM AND*** click the ***SETUP*** button. Mark the ***LOG ON/OFF*** box in the dialog box, and select an output file name. In addition, you can select short or long log file format.
For details of the output format, refer to Appendix G.

Config

You can specify configuration parameters for the performance simulation. These parameters deal mainly with memory wait-states. By default, all parts of the memory are assumed to be accessed with zero wait-states. If you want to override this configuration, specify a configuration file name. Select ***CONFIG ➨ SIM AND*** click the ***SETUP*** button. In the dialog box. next to ***SIM CONFIG FILE***, specify the configuration file name. For convenience, you can also generate a conguration file template by clicking ***GEN TEMPLATE***, and specifying the template file name next to it.
For details of the configuration file format refer to Appendix F.

2.2.12 Working with Debugger Command Scripts

To make command entry and debugging more convenient, the debugger provides two commands: **alias** (Section 4.2) and **set** (Section 4.28). In addition to the standard command buttons, the debugger provides a set of custom buttons. To use this feature, open ***CONFIG*** and select ***CUSTOM BUTTONS***.

Command aliasing and macro variables

alias defines short forms, or more meaningful names, for single commands, and defines substitution macros for frequently-used sequences of commands.

set assigns short, meaningful, names to long, complex, strings e.g., an addressing expression for a nested structure.

alias and **set** definitions may conveniently be placed in the debugger initialization files, or in input (command) files. You can specify more than one command on a line, separated by semicolons (;). If a command is interactive, or results in an error, the debugger ignores commands which follow on the same line.

This capability can be used to separate commands within strings for the **alias** and **set** commands.

Log files (crdb.log)

You can record the commands which the debugger receives during a debugging session, and optionally, the responses to the commands. In addition, you can request the debugger either to expand aliased names, or to record them as input. Recording the commands is particularly useful if you want to rerun the debugging session later, either because you are chasing a specific bug which requires some setup of the environment, or you want to use the session as a regression test for your program. Recording the debugger responses allows you to analyze the output at your convenience.

To record a log file, either specify **-l =** *log_filename* on the invocation line, or if you are already in a debugging session, open ***FILE*** and select ***LOG FILE*** or use the **LOG** command (see Section 4.17).

Commands are recorded in a log file just as they were typed. Result lines are displayed with a leading pound sign (**#**), making them comments so that the log file can later be used as a command file. If the original input was from a command file, comments in that file are preceded by two pound signs.

Comments

If you need to place annotations/comments into your log file for later reference, precede them (each individual line) with a pound character (**#**), the comment character. If **#** is the first non-whitespace character, the rest of the line is ignored. To place a comment on a command line you must precede the comment with both a semicolon (**;**) and a pound sign (**#**):

Example

```
break main;# set a breakpoint at main()
```

Replaying scripts

You can use a recorded file as a debugger command file either by specifying its name as an **executable_file** on the command line (Section 2.2.3) or by using **INPUT** (Section 4.15) during a debugging session.

You can also create a command file using a text editor, and run it with this facility. Such files are useful for debugger initialization, setting up particular target system states as a prelude to further debugging, or setting up a particular set of commands, possibly with expected responses, to be executed as a regression test for your program. The debugger initialization files (Section 2.2.2) are examples of such files.

To pause, during command file execution, to read output, or execute one or more external debugger commands, insert a **pause** command (Section 4.21) in your command stream. To continue execution, use **resume** (Section 4.25).

2.2.13 Saving and Restoring the Debugging Context

The following three functions make the debugger convenient to use.

Debugger initialization

When you initially invoke the debugger, you have only rudimentary initialization files. After you have become familiar with the debugger commands, you will have discovered a set of commands that configure the debugger for your use. These include setting communication parameters, and the target chip name. When you are satisfied with your environment, you can save it in a specified file, possibly one of the initialization files, using **SAVECONFIG** (Section 4.26). For succeeding debugging sessions, the debugger sets this environment automatically. Refer to the command description to see what environmental characteristics are saved by the command. Alternatively, open ***FILE*** and select ***SAVESETUP*** to save the current window configuration.

Saving debugging setup

You can stop your debugging session in the middle, and continue it later. End a session with **SAVESTATE** (Section 4.27) to save data such as the break/softbreak lists and current filename. Specify the name of the file into which the state is saved. The default name is **crdb.ctx**.

Restoring debugging setup

In a later session, when you are ready to continue debugging, use the **SETSTATE** command (Section 4.29) to restore the previous debugging setup.

Open ***FILE*** and select ***SAVESTATE*** to save the current state, select ***LOADSTATE*** to restore it.

2.2.14 Working with the Monitor Commands

Debugger-monitor traffic

The debugger sends a series of monitor commands to the monitor to accomplish each command. Use **VERBOSE** (Section 4.39) to monitor the traffic between the monitor and the debugger. The Output Window displays traffic in both directions. This information may be useful in understanding communication, monitor, or debugger problems.

Directly using monitor

To send a command directly to the monitor, with no other processing, type it on the command line preceded by an exclamation point (**!**). The entire command is passed to the appropriate backend (the ILS for SIM mode, and the DEX monitor, for EDB mode) without the **!**. Any subsequent input needed for successfully completing the Monitor command must be prefixed with **!**.

See Appendix D for a description of the Monitor command interface.

2.3 USING THE SIMULATION ENVIRONMENT

One of the tools provided by the debugger to aid you in your program development is the Instruction-Level Simulator (ILS), which accurately interprets and executes your program, one machine instruction at a time. As the Simulator executes your program, it maintains an up-to-date copy of your registers and data memory, so that at any time during your work with the Simulator, you may use the standard debugger features to control the execution of your program and manipulate your registers and data memory.

You can invoke the Simulator either from the command line (Section 2.2), or with the **TARGET** command (Section 4.38). See Chapter 5 for detailed information about the Simulator commands.

Configuring the Simulator

Three simulator variables control the operation of the Simulator. Set these variables with the **SIM CONFIG** *parameter* command, where *parameter* is one of:

clock=*frequency*

Sets the frequency of the Simulation Clock. To use the Simulator for performance estimation, you must set the correct clock frequency.

refresh=*refresh_frequency*

Specifies the time delay between updates of the debugger Status Window (Section 3.2). The time delay can be specified either in terms of time units, or in terms of the number of instructions executed.

sim config perf = *string*

Specifies wait states.

Control during simulation

When the Simulator is executing a target program, it is in full control of the system, preventing any user-interface action, such as window updating, from taking place. The debugger can not service any requests via the keyboard or mouse. To allow you to request such services, the simulator periodically returns control to the debugger. **SIM CONFIG REFRESH** (Section 5.2) specifies the period between such updates as either a time period in ticks, or a number of instructions. The default refresh period is 1000 instructions.

The Simulator provides four sets of basic functions:

1. Instruction-level simulation of the target CPU, executing your program, one machine instruction at a time, and maintaining its own up-to-date copy of the CPU registers and memory.

2. Primitive simulation of I/O functions, allowing you to perform basic debugging of your I/O routines.

3. Primitive simulation of interrupts and traps, for the same purpose.

4. Estimation and display of data and profiles (bar charts) reflecting the theoretical performance of all, or portions, of your program, to aid you in determination of program performance, location of bottle-necks, etc.

The following paragraphs explain the basics of each of these functions.

In executing a target program, the Simulator accurately simulates the external functionality of machine instructions, but does not simulate the hardware functionality (pipelining, cache simulation, etc.), thus performance measurements are best estimates. When Simulator Performance Measurement is turned on, the Simulator maintains a clock giving simulation time. This clock is set to zero at the beginning of the simulation, after a **RESET** command (Section 4.24), and if Performance Measurement is turned off and then back on.

2.3.1 Peripheral Stimulus System

To debug your embedded system software completely, you must simulate the peripheral devices required by your program. The Simulator provides input, output, and interrupt simulation through the Peripheral Stimulus System (PSS).

To use PSS, turn on the Simulator program profiling function, followed by **SIM PSS ON** (Section 5.4).

If you want certain parts of your program to execute faster, use **SIM PSS OFF** (Section 5.4) to turn off PSS.

To rerun a test, use **RESET**, turn the performance simulator off, and use **SIM PSS CLEAR**. After **PSS CLEAR**, you must re-map your PSS output files and reload your PSS input file (Chapter 5).

PSS commands

PSS commands are divided into two groups: external and internal. The detailed operation of PSS is controlled by internal commands, which must reside in PSS input files. Some of the basic operations of PSS can be controlled by external PSS commands, which can be executed in the same manner as any other debugger command.

To use PSS, first build a PSS input file containing the internal commands for the I/O, or interrupt, actions you want to perform. The timing of the various commands to be executed is extremely important. Before you try to use PSS, we recommend that you read carefully all of the information on PSS in this chapter, and in Chapter 5, paying particular attention to the information on both absolute and relative simulation time.

Before you begin using PSS in a simulation, you must use turn performance simulation on, to start the simulation clock and timing mechanism.

All keywords used by PSS must be lower-case. All external PSS commands must be preceded by the keywords **SIM PSS**. All tokens in PSS commands must be separated from each other by white space (spaces, tabs, or newline characters).

PSS input files may be read into memory either from the command line by **LOAD** (Section 5.4), or from within another PSS input file by **READFILE** (Section 5.3).

Appendix B contains listings for an example C program which makes use of the Peripheral Stimulus System to read and write a character stream.

PSS external commands

PSS supports the following external commands, which may be executed from the command line, or from the debugger input files (Section 5.4):
on, **off**, **load**, **unload**, **pss**, **clear**, **files**, and **output**.

ON and **OFF** turn on and off PSS internal command processing for optimizing program execution speed.

LOAD loads and processes a PSS file containing internal commands which carry out the detailed operations required.

UNLOAD unloads a previously loaded file.

PSS reports the state of **PSS** (on or off).

CLEAR clears PSS data structures, closes all PSS files, sets PSS state to off, and sets the time of the last executed **SET** command to current simulation time.

FILES displays the names of the currently open PSS files.

OUTPUT maps a target memory address to a particular output file for recording.

PSS internal commands

PSS supports the **SET**, **OUTPUT**, and **READFILE** internal commands (Section 5.3), which must be executed from PSS input files. Each command must be on a separate line. Comments may be included in PSS input files by preceding them with a pound-sign (**#**).

Before executing PSS internal commands, turn on internal command processing by specifying **SIM PSS ON**. If you later want to increase the speed of execution of your program, and you do not require PSS processing for awhile, you may turn off internal command processing with **SIM PSS OFF**, and then use **SIM PSS ON** to turn on internal command processing later. At any time, you can determine the state of PSS by with the **SIM PSS** command (Section 5.4).

A PSS input file may be loaded from the command line with **LOAD** (Section 5.3) or from another PSS input file with **READFILE**.

SET **SET** sets a target memory address to a specified value each time specified time delays have elapsed, or other conditions have been met.

OUTPUT **OUTPUT**, like the external command, maps a specific target memory address or range to a particular output file for recording.

READFILE **READFILE** loads a PSS file containing internal PSS commands which carry out the detailed PSS operations required.

Using data files To provide either a large number of data points, or a series of repetitive data points, for a **SET** command, you may specify the pathname of a data file in the **SET**. Thus, you can write a sequence of values for a **SET**. One practical application of this feature is the generation of patterns such as square waves, sine waves, or analog-to-digital outputs, using a single command.

The data file format is one value per line (minimum of one value per data file). Comments may be included in the file and must start with a pound sign (#). Data is read from the file cyclically.

If an address range is specified in the **SET** command, several values at a time are read from the file, otherwise, a single value is read on each iteration.

Creating PSS symbols Symbols can replace any entity within a PSS command except reserved words. Symbols can replace *address*, *time*, *time value*, and *path_name*. Symbols must be defined before use and must be unique. They may, however, be defined more than once in a file. The last value of the symbol read from a file is the value used for substitution. To create a symbol (see Section 5.3):

symbol = **[** *value***]**

2.3.2 Simulating I/O Operations

In general, CompactRISC peripheral I/O devices are memory-mapped. To simulate input, use the **SET** command (Section 5.3) to set various memory addresses either by a single command, or from an external file, after a specified delay from some event, and repeated with a specified period. **SET** operations may also be keyed on previous reads or writes of a specific memory address by your program.

For simulated output, use the **OUTPUT** command (Section 5.4) to capture data written either to a specific memory address or to a range of addresses.

2.3.3 Simulating Interrupts

The Peripheral Stimulus System can simulate the non-maskable interrupt, and any of the other maskable traps and interrupts up to vector number 32. The vector numbers for a particular processor are given in the relevant Chip User's Guide.

To simulate interrupt operation in your program, follow these steps:

1. Reserve a block of 32 double words as an Interrupt Vector Table in user memory.

2. Set the INTBASE register to point to this block.

3. Set all of the entries in this block to zero, so that by default, all interrupts are ignored.

4. Choose vector numbers for the interrupts which you want to simulate. The vector number for the non-maskable interrupt is normally one (1) and the vector numbers for traps may be found in the chip User's Guide for the target processor.

5. For each interrupt to be simulated, set the entry in the Interrupt Vector Table corresponding to the vector number of the interrupt to the memory address of its interrupt handler.

6. For each interrupt you want to simulate, in your PSS input file, specify a **SET** command with a special target address of **nmi** for a non-maskable interrupt, or **int** for a maskable interrupt. The value for the **nmi** may be one (1), but the value for the **int** must be the actual vector number of the interrupt to be simulated. The same time delays and conditions may be specified on the **SET** command as for any other **SET** command.

Chapter 3
DEBUGGER USER INTERFACE

3.1 DEBUGGER GUI INTERFACE

When you invoke the debugger, it displays the Main Window (Figure 3-1). This window contains the Main Menu bar, programmable buttons, the command line, source window and output window.

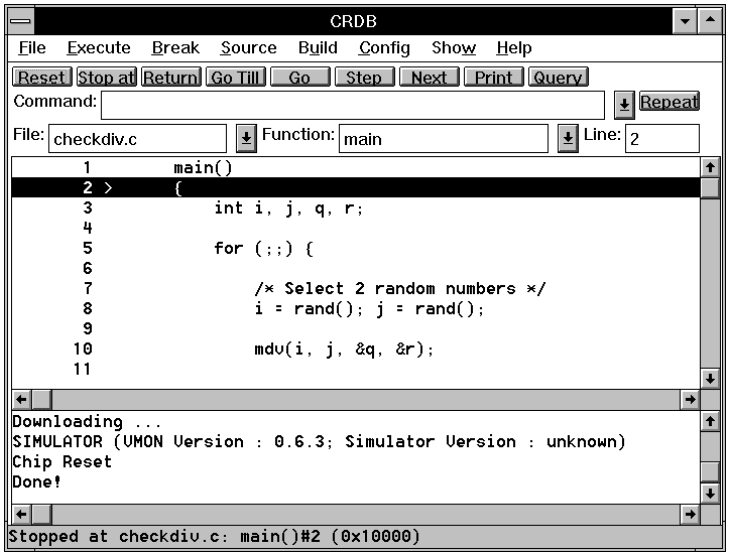

Figure 3-1. Main Window

3.1.1 GUI Operations

Using Pull-down menus

To access a menu with the mouse, move the mouse cursor to the menu and hold down the Left button. When the menu is displayed, move the mouse cursor up or down the list until the desired command is highlighted; then release the mouse button. Section 3.3 lists the selections available on each menu.

If after you select a menu, you decide not to use it, click outside the menu to de-select it.

File Menu	deals with file-related commands, e.g., user COFF file, text files, command files, save and restoring of the debugger's state or environment.
Execute Menu	controls program execution.
Break Menu	sets breakpoints, autocommands to be executed upon reaching a breakpoint, and performs time measurements.
Source Menu	manipulates source file characteristics.
Config Menu	configures your debugging environment (target, color, button mappings etc.).
Show Menu	makes any debugger window active.
Help Menu	invokes the Help system.

Using configurable buttons

The second row contains configurable buttons. To configure these buttons, open ***CONFIG*** and select ***CUSTOM BUTTONS***. The debugger is shipped with the following default settings, most of which can be re-configured:

Reset	issues a reset command.
Stop at	sets a Software Breakpoint at the selected source line.
Return	executes till control returns to the caller of the current function.
Go Till	begins execution of the target program at the address indicated by the current contents of the PC, and continues until the currently selected source line is reached.
Step	executes a single source statement of the target program.
Next	steps the target program to the next statement, executing any called functions, i.e., it steps over the next source statement.
Go	begins execution of the target program at the address indicated by the current PC.
Print	prints the contents of a highlighted variable. *This button is not configurable.*
Query	opens a dialog box which displays variables which are visible under current scope. Variables which are structures can be expanded or contracted. Double clicking on an expanded variable opens a dialog box for modifying the contents of the variable. *This button is not configurable.*

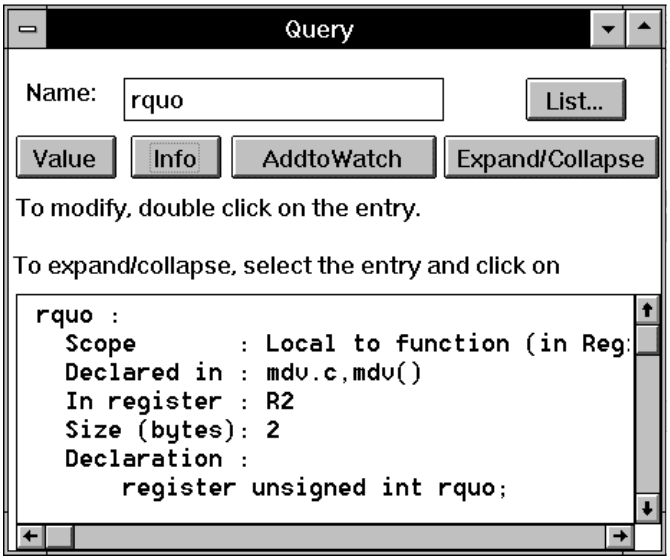

Figure 3-2. Query Window

Interactive commands

To enter commands, click on the command line (the third row) and type the commands. Chapter 4 describes the syntax of the command. You can retrieve, edit and re-execute previous commands with the up and down arrow keys.

You can copy and paste to the command line from other windows. The repeat button, on the right side, repeats the last command you typed.

Using a dialog box

Dialog boxes pop up when additional information is required from you e.g., file and conditional break point selection. Dialog boxes are indicated by three dots following the menu entry. Use the conventions of the host system to enter information in a dialog box.

Many dialog boxes are closed with a ***DONE*** button. Operations performed in such dialog boxes can not be undone automatically. You must explicitly undo the operations using the appropriate commands.

Selecting windows

To use a window, you must first select it. Selecting a window brings it to the front, and highlights its title bar to indicate that it is active.

To select a window with the mouse, move the mouse cursor to the visible portion of the window, and click the left button.

Alternatively, you can select a window by clicking on its name in the ***SHOW*** menu.

The Menu bar is overlaid by all windows. Thus, when you select a window, the menu bar shows the current window's menus.

When you install the debugger in a program group, make sure that you select the current directory appropriately. The debugger reads, by default, files like **crdb.env**, **crdb.ini** or **crdb.ctx**. from this directory.

To select an iconized window, double-click on it with the left button.

Programming function keys

To scroll with the keyboard, use the up-arrow, down-arrow, ***PGUP***, ***PGDN***, ***HOME***, and ***END*** buttons for vertical scrolling, and the left-arrow and right-arrow for horizontal scrolling.

A *hot key* is a single key on your keyboard which, by itself or in combination with a control key such as ALT, executes a debugger function. Some menu items are mapped to pre-defined hot keys, as indicated in the Menu Item.

3.2 THE DEBUGGER WINDOWS

Source window

This window (Figure 3-2) is part of the Main Window and displays the contents of source files and text files. It is centered on the current display line, which is marked with a '>'. The current execution line (where the program counter is positioned) is marked with reverse video.

Use the scroll bar, or the arrow keys, to scroll through a file. If you scroll down with the mouse, the text scrolls until the last line is brought into view, and no further. In this case, the last line is not the current line. You can use the mouse to reposition the current line to the last line.

The format of a source line is:

BS *lineno hex-addr source line*

where:

B	denotes a hardware breakpoint.
S	denotes a software breakpoint.
>	denotes the current display line.
lineno	is the line number starting at the beginning of the source file.
hex-addr	is the hex-format code address for executable lines. ***hex-addr*** appears only in mixed mode.

Example

```
B 233 i = i + j;
```

To show the code in C followed by the assembly lines open ***SOURCE*** and select ***DISPLAYMODE*** ➨ ***MIXED*** option (Section 4.31)

If you use **debug** (Section 4.9) to specify another COFF file, this window is updated to the appropriate source file.

The fourth row on the Main window is associated with this window and the three entries in that row show the current module, current function and the current line number corresponding to current line display line. To bring another file or function or line in to source window, just type in the new name on the corresponding position in this row and press return. Use **LIST** (Section 4.16) to bring any module/function/line belonging to the COFF file into this window. The filename field is updated appropriately.

If you bring in a file that does not belong to the COFF file, it is displayed as plain text. You can not set breakpoints, etc.

In this window, the mouse keys have special meaning:

Left button
- single-click: repositions the current display line.
- double-click: toggles a soft breakpoint at the current display line.

Right button
- single-click: Executes from the line addressed by current PC up to the selected line.

Status window

The Status Window (Figure 3-3) shows the address in the program counter, and the source file reference corresponding to the current program counter. It is updated whenever there is a change of state in the debugging environment.

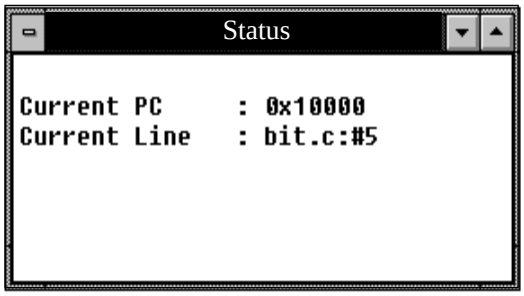

Figure 3-3. Status Window

The individual fields are:

Current PC	displays the current contents of the program counter, and the corresponding source-file line number.
Current Line	shows symbolic information, or the instruction being executed if no symbolic information is available, for the current program counter.
Chip Status	shows whether the chip is stopped, running, or reset.
Execution Time	shows the time taken for this run in microseconds. Applicable only when the target is simulator.

The number of instructions executed, and the simulation time, are normally cumulative, but are initialized to zero by **RESET** (Section 4.24) or **SIM PERF CLEAR** (Section 5.4). After **SIM PERF ON** is executed, they are updated periodically, as controlled by the **SIM CONFIG REFRESH** command (Section 5.2).

Register Window

The Register Window (Figure 3-4) shows the contents of the PC, SP, PSW and other processor registers in hexadecimal format. You can double click on any register name to open a dialog box for modifying the value in that register.

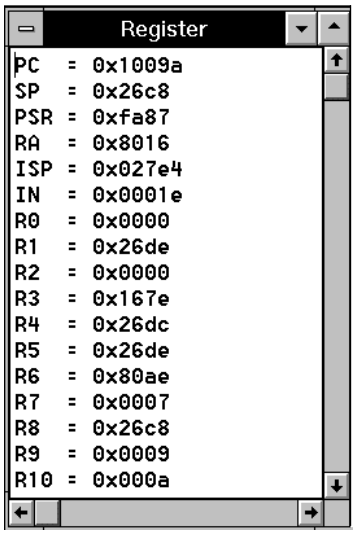

Figure 3-4. Register Window

Output Window

The Output Window (Figure 3-1) allows scrolling, and can show the last few hundred lines of output from the debugger.

Stack Window

The Stack Window (Figure 3-5) displays the current caller stack. Open ***SHOW-STACK***, on the main menu bar, to display the window. The top line is the currently executing function.

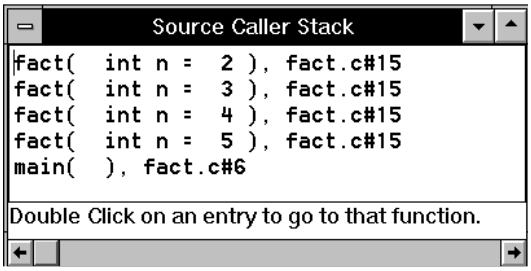

Figure 3-5. Caller Stack Window

The format of each line in this window is:

```
function-nn (arg1, ..., argn)
        ....
        ....
function-2 (arg1, ..., argn)
function-1 (arg1, ..., argn)
main (argc,argv)
```

To bring the source of the calling function (mentioned in that line) into the Source Window, position the cursor on one of these lines, double click on that line. To return to the current execution line, open ***SOURCE*** and select ***DISPLAY PC***.

Watch Variables Window

The Watch Variables Window (Figure 3-6) displays variables selected with the **WATCH** command. It is updated every time there is a change of state in the debugger or target environment. This window does not let the debugger modify the variables. To modify variables, use the Query window.

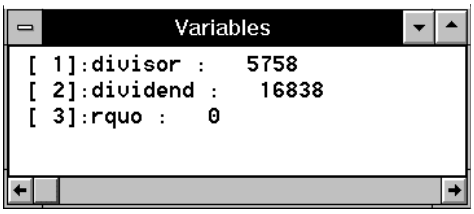

Figure 3-6. Watch Variables Window

Local Variables Window

The Local Variables Window displays the values of the local variables of the currently executing function and its arguments. The name of the function is displayed in the title bar of the window. This window does not let the debugger modify the variables. To modify variables, use the Query window.

Memory Window

The Memory Window (Figure 3-7) displays a memory region in the selected format. You can specify the address, the format and the number of units to be displayed from that address in that format. The address can be an expression, (see **`list -m`**, Section 4.16). Change any one of these, and press return, to bring in the new values. Double click on any item to open a dialog box for modifying the contents of the selected address.

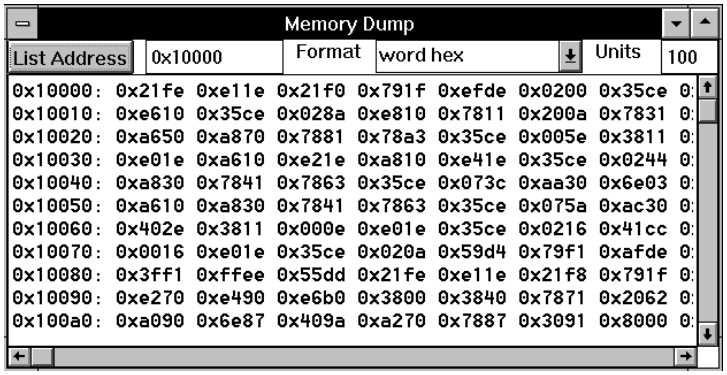

Figure 3-7. Memory Window

Performance Window

After you have collected performance data on all, or a portion, of your program, you may display the data, both numerically and in a chart, in the Performance Window (Figure 3-8).

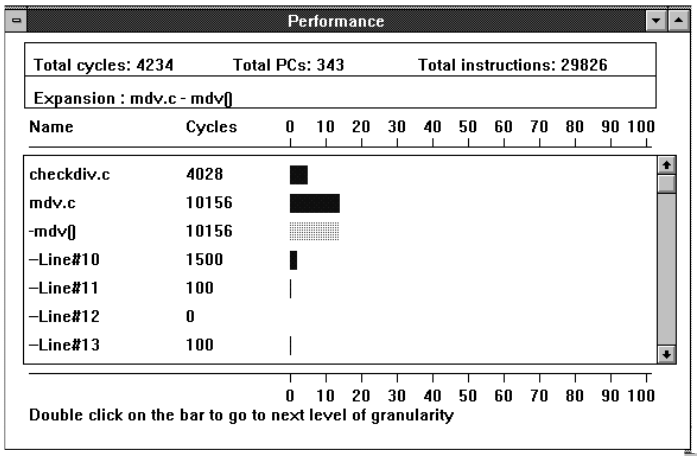

Figure 3-8. Performance Window

This windows menu bar containing the following menus (left-to-right):

Customize — Select the items to be displayed in the bar graph:

- Granularity: select top level display as files or functions.
- Entries: filter entries to be displayed on the graph instead of all the files or functions.
- Performance On: enable or disable performance simulation, if needed. Same functionality as **`SIM PERF ON`** (Section 5.4) command.

Graph — Select various update options:

- Redisplay: refresh the bar graphs.
- Track with source: update the source window corresponding to the current file or function or line selection using the bar graph.
- Track with execution: update these bar graphs whenever the program stops.

Quit: Remove the Performance Window from the display.

On row two of the Performance Window, the debugger displays the total number of cycles simulated (**Total Cycles**), the number of PC samples taken during data collection (**Total PCs**), and the total number of instructions simulated (**Total instructions**).

The dominant feature of the Performance Window is a bar chart showing execution time and percentage by program units, for either program files or C functions. If files are listed, you can double-click on a particular file name to see the execution time of that file broken down by function. If functions are displayed you can double-click on a function name to see the execution time of that function broken down by line. If you request the breakdown of a second file, the currently displayed breakdown is removed. The third row of the window indicates the current **Expansion level,** by file and function name.

Help Window

The debugger provides on-line help on general topics and individual commands (see Figure 3-9). The on-line help is intended for users who do not want to read manuals.

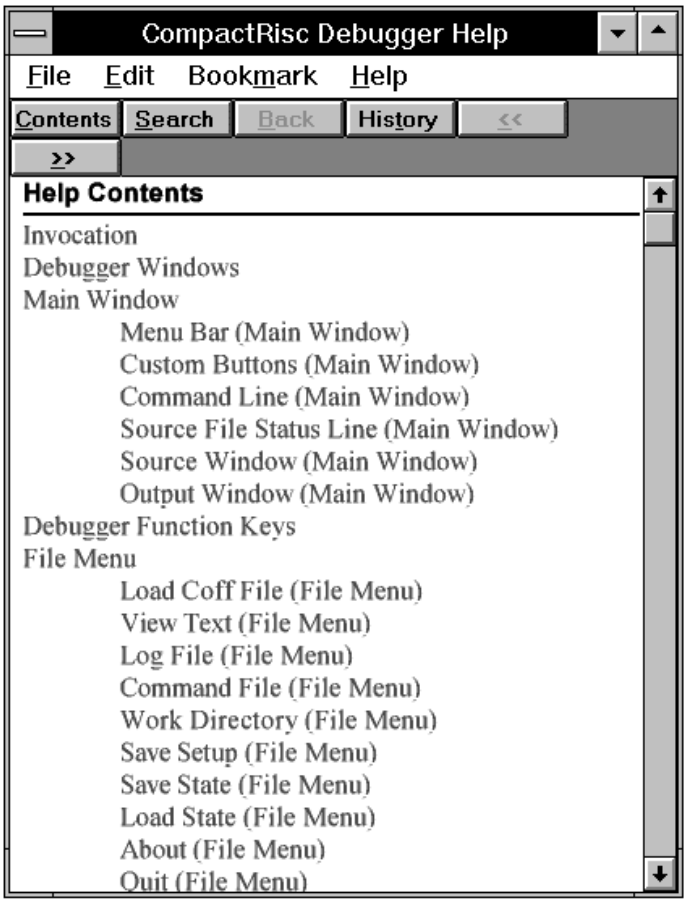

Figure 3-9. Help Window

3.3 MENU DESCRIPTIONS

File menu

The entries on the ***FILE*** menu are:

- **`Load COFF File`**: selects an executable file for loading (Section 4.9)
- **`View Text`**: selects any text file for display. (Section 4.16)
- **`Log File`**: selects a file for display. (Section 4.17)
- **`Command File`**: executes a file containing debugger commands. (Section 4.15)
- **`Work Directory`**: changes the working directory. (Section 4.6)

- **Save Setup**: saves the current environment, window layout and color of the debugger to **crdb.env** file. (Section 4.26
- **Save State**: saves settings related to debugging in **crdb.ctx**. (Section 4.27)
- **Load State**: restores the settings related to debugging from a **crdb.ctx**. (Section 4.29)
- **About**: displays information about the current version of the debugger
- **Quit**: exits the debugger

Execute menu The entries on the ***EXECUTE*** menu are:

- **Go**: begins execution of your program. (Section 4.13)
- **Reset**: reset the system (Section 4.24)
- **Rerun**: executes your program from the beginning
- **Go Till**: executes your program until the selected source line is reached
- **Step source line**: executes the machine instructions contained in the current source line. If the instruction contains a function call, execution stops in that function. (Section 4.34)
- **Next source line**: executes the machine instructions generated by the current source line. If the instructions contain a function call(s), executes the entire function and returns. (Section 4.19)
- **Step instruction**: executes one machine instruction. (Section 4.35)
- **Next instruction**: executes one machine instruction. If the current source-level instruction contains a function call, execute the entire function. (Section 4.20)
- **Debug mode**: enables or disables debugging of startup or exit code. This is a toggle switch.
- **Advanced**: selects options for **Run**, **StepSource** or **StepIns**, **NextIns** or **Step Source**
- **Abort Execution**: aborts execution of your program by sending a signal to the target. There may be some delay before the target stops and responds. (Section 2.2.8)

Break menu The entries on the ***BREAK*** menu are:

- **Soft Break**: manages software breakpoints. (Section 4.30)
- **Break**: manages hardware breakpoints. (Section 4.4)
- **Cmds on Break**: manages the command list to be executed whenever the debugger stops (Section 4.3)

Source menu The entries on the ***SOURCE*** menu are:

- **Show PC Line**: re-centers the Source Window display on the source line addressed by the program counter. (Section 4.37)
- **DisplayMode**: sets the display mode for the Source Window display to Source or Mixed. (Section 4.31)
- **Source Path**: adds a new path to the list of pathnames for finding source files (Section 4.32)
- **Search String**: searches for a string in source window (Section 4.12)

Config menu The entries on the ***CONFIG*** menu are:

- **EDB:** configures the debugger to work with a DEX-ADB development board pair. (Section 4.38)
- **SIM**: configures the debugger to use the Instruction-Level Simulator (Section 4.38)
- **Radix**: sets the global radix for debugger output displays (Section 4.23)
- **Verbose**: displays all communications between the debugger and the monitor. (Section 4.39)
- **Color**: configures the color for all the windows

Show menu The ***SHOW*** menu allows you to activate any debugger window. The windows are:

- Registers
- Status
- Local Variables
- Watch Variables
- Memory
- Performance

Help menu The *Help* menu opens the help sub-system, and allows you to look at the various sections of the on-line help.

Chapter 4
THE DEBUGGER COMMANDS

4.1 INTRODUCTION

This chapter describes the debugger commands in alphabetical order.

Commands which relate directly to the Instruction-Level Simulator are described in Chapter 5. Simulator commands must be prefixed by the keyword **sim**. Commands for the Peripheral Stimulus System must be prefixed by the keywords, **sim pss**.

Arguments, modifiers, and options for all commands are defined in Appendix A.3.

4.1.1 Symbol References in Commands

In general, symbol references are the same as in a C program. The debugger applies the scope rules of C to resolve the symbol references. In addition, there are the following special notations and modifiers for certain symbols;

- To specify a function within a particular file, use:
  file_name@func_name
- The function names also take the following modifiers:

  $b Address of the absolute beginning of a function (the prologue).

  $c Address of the first instruction in the body of the function (after the prologue).

  $e Address of the last instruction in the body of the function (before the epilogue).

  $x Address of the **RETURN** instruction at the end of the function.

These modifiers can be used in any command where *function name* may be used, for example, **arcsin$e**.

- To reference a line number within a file, use:
  file_name#line_no
  Line numbers are counted from the beginning of the file.
- An address range is specified as:
  start_address//end_address
- The current PC can be specified as **.** (period):
  break .

- The address of return PC for the current C function can be specified as `..` (period period):
  `break ..`
- The current source line can be specified as #(pound symbol):
  `break #`
- The source line corresponding to the current PC can be specified as **`&`**:
  `break &`

4.1.2 Command Entry and Formats

You enter the debugger commands in the command-line editing area. Select the command line by clicking on the line (Section 3.1.1). Inputs must be lowercase. You do not have to type the complete command name, only sufficient characters to differentiate it from other commands or aliases.

4.1.3 Command Descriptions

The descriptions of the commands are in the following format:

- The command name, and a short description of its functionality.
- The syntax of each the command's formats, together with a detailed description of the operation of the command.
- How to execute the function with the hot keys, or the mouse.
- Any cautions which need to be observed when using the command.
- Examples of command usage, where necessary.

Most commands have a syntax of the following format:

`command-name` **`[`***`-option`***`]`** **`[`***`-option`***`]`** *`arg1,..,argn`*

Some commands do not have options or arguments. Whenever an asterisk (*) is specified as an argument, it denotes wildcard operation. The debugger provides limited wildcard support.

If you do not specify an operator/argument for a command, which requires one, the debugger usually responds with the current values. For example:

```
Command> comm
         RS-232: Baud = 9600, Port = Com2
```

Inputs are specified in standard C language constant format (0x as a prefix for hexadecimal, 0 as a prefix for octal, no prefix for decimal). Generally, outputs are displayed in the decimal format. Some commands have options to change the display format.

4.1.4 Common Command Modifiers

The following modifiers are common to many commands, and are not repeated for each individual command

List control modifiers:

-r removes an entry from a list.

-d disables an entry from a list.

-e enables an entry from a list.

4.2 ALIAS — DEFINE MACRO

Maps a command, or list of commands to a name which you can use instead of the command(s). An alias can consist of up to eight alphanumeric characters; the first character is alphabetic.

Aliases are not expanded within other aliases, and thus recursion is not allowed. During command validation, the **alias** table is searched before the command table.

Aliases may be expanded in the log file (Section 4.17).

alias *name* = *"command; command ..."*
sets an alias. Specify double quotes if the command contains arguments, or commas, or command separators.

alias *name* = "*command* $$, $$"
defines an alias with arguments, where each "$$" is a placeholder for an argument. Arguments are substituted one-at-a-time, in order, into the macro definition.

When using the alias, you must provide the exact number of arguments. otherwise, an error is issued.

alias *name* prints the alias for the name.

alias prints all aliases.

alias -r {*name* **|** *****}
removes a name from the alias table.

Examples To imitate the command names of another popular debugger, define them as aliases of the debugger commands:

```
>alias bd=break -d
>alias bc=break -r
>alias be=break -e
```

To execute **break -r** and **go** as a sequence of commands, create an alias, e.g., `freerun`. When you execute `freerun`:

```
>alias freerun="break -r *; go"
```

To create a short notation for **softbreak**, create the following alias. When you enter **bp 0x200, 0x300**, **softbreak** is executed with 0x200 and 0x300 as arguments.

```
>alias bp="softbreak $$,$$"
```

4.3 AUTOCOMMAND — SPECIFY COMMANDS FOR AUTO EXECUTION

Specifies a list of commands that are executed following an execution command such as **step**, **next**, **reset** or **go**. The specified commands are not validated until they are executed.

autocommand lists the current entries in the **autocommand** list.

autocommand *command*
adds a command to the list. Commands can be: **view**, **list**, **find**, or **where**. Do not specify execution commands (e.g., **step**, **next**, **go**).

autocommand {-r | -d | -e} %*id* | *
A command is identified by its ordinal number, as shown by the output of **autocommand** without an argument.

Mouse Open ***BREAK*** and select ***CMDS ON BREAK.*** Fill in the dialog box.

Examples To display the stack history after each breakpoint, **step**, or **next**:

```
>autocommand where -c
```

To display the value of the PC after each breakpoint, **step**, or **next**:

```
>autocommand view %PC
    [2] - view %PC
```

To display the current list:

```
>autocommand
    [1] where -c
    [2] - view %PC
```

To remove command number 2:

```
>autocommand -r %2
```

To display the value of the structure member, *salary*, addressed by the pointer, *e_list*:

```
>autocommand  view e_list->salary
```

To display the value of the string contained in structure member *name*, of the structure in entry 0 of the array of structures *p_tab*:

```
>autocommand view  p_tab[0].name
```

4.4 BREAK — SET HARDWARE BREAKPOINT

Manages the hardware breakpoint list. A breakpoint may be specified on a PC-match, a read reference, on a write reference, or on either read or write. You may also attach an occurrence count and or logical conditions to a given *address*.

For a general discussion of how to use breakpoints, see Section 2.2.7. The debugger assumes the qualifier as PC-MATCH, if the address is a code address, and as ACCESS, if the address is data address.

If a break address starts on a line, the source line corresponding to that line is marked with a B in the first column. This mark disappears when the breakpoint is removed.

Some options of this command may affect real-time operation. You are notified whenever a real-time breakpoint occurs.

Caution The number of hardware breakpoints that can be specified at a time is a function of your target chip and development system. The particular breakpoint functions provided are also a function of your target development system. Hardware limitations may permit only one breakpoint to operate at any one time.

Mouse To set a complex breakpoint, open ***BREAK*** and select ***BREAK***. Fill in the dialog box.

break [-t] *brkaddr_list*[,**c**=*RLexp*][,**o**=*occ_cnt*][,**q**=*qualifier*][,**s**=*size*]

adds entries as hardware break.

-t adds temporary breakpoints. These breakpoints are deleted after they occur.

brkaddr_list list of break points, code address or data address depending on the target, to be added. See Appendix A for the syntax.

c=RLexp Specifies the relational, or logical, expression which must be evaluated as TRUE to satisfy the break condition. This option may affect real-time operation, because the condition is being evaluated by the debugger.

o=occ_cnt Sets the number of times the given *address* must be referenced before execution is actually interrupted. Specifying this option along with a condition may result in non real-time operation.

q=qualifier The qualifier specifies the type of access which makes the break become effective.

E to stop whenever code is executed at this address. It is also known as PC-MATCH. This is the default when the address specified is a code address.

A to stop whenever the address is accessed, i.e., read or written to. This is the default when the address is data address.

R to stop whenever the address is read.

W to stop whenever the address is written to.

s=size Sets the number of bytes for which the data break is applicable. This field is meaningful when the qualifier is A, R or W.

By default, the debugger sets the size based on the data type of the symbol (subject to hardware limitations), otherwise it sets the size to the size of the target integer (2 for a 16-bit core and 4 for a 32-bit core).

After you set a breakpoint, the debugger responds with the message:

```
[id] : command string
```

The id is used with a percent sign (*%id*) to disable, enable, or remove the breakpoint (see below).

break {-r | -d | -e} *%id* **|** *

break lists the break points in the following format:

```
[id1]  -  command1
[id2]  -  command2
```

Examples

To set the breakpoint at the current source display line:

```
>break #
```

To set the breakpoint at the current PC:

```
>break .
```

To set the breakpoint at the beginning of the line containing the PC:

```
>break &
```

To set the breakpoint at line 36 in the file **main.c**:

```
>break main.c#36
```

To delete a breakpoint after its first occurrence, use the temporary breakpoint feature:

```
>break -t main.c#36
```

To set the breakpoint at the start of the first executable line of the function **initerm**:

```
>break initerm$c
```

To set the breakpoint at the last line of the function **initport**:

```
>break isdn1.c@initport$e
```

To set the breakpoint at the final return of the function:

```
>break main$x
    [4] E (0x7DAF) main.c#19@main
```

To set the breakpoint upon a reference to *initflag* if the current value is not equal to 2. Note that condition option affects real-time performance

```
>break initflag,c= (initflag!= 2)
```

To set the breakpoint upon a reference to *initflag* if the current values of *initflag* and *fileflag* are non-zero:

```
>break initflag,c= (initflag && fileflag)
```

To set the breakpoint at line number 258 in the current source file

```
>break #258
```

To set the breakpoint at the beginning of the prologue of function *is_prime1*, in module *stmts1.c*

```
>break stmts1.c@is_prime1$b
```

To set the breakpoint at line number 164 of the current source file if the value of j is zero and the value addressed by temp is equal to the value addressed by str.

```
>break #164,c= (!j && *temp == *str)
```

To set the breakpoint at the second occurrence when the program is at line 236 of module *stmts3.c* if the value of str[k] equals the value of ch and the value of j is greater than the value of k.

```
>break stmts3.c#236,c= (str[k] == ch && j > k),o=2
```

To disable breakpoint number 4:

```
>break -d %4
    [4] - disabled "main$x"
```

To list the current breakpoint

```
>break
    [1] E (0xE800) crdb_ch2.c#15@ main
```

To set a breakpoint on a local variable of a function.
This differs from other breakpoints because each local variable in a function (which is allocated space on the stack) is created only when the function is called, and disappears when the function is exited. Thus, if you set a breakpoint on a local variable, you should remove it when the local variable goes out of the scope. Otherwise, it will continue to stop whenever the breakpoint condition is met for that stack location.

```
>break i1
    [1] E (0x3E0023C4)  Var : i1   Q: A; S: 4
```

4.5 CALL — EXECUTE USER FUNCTION

`call` *`func_symbol`***`(arg1,arg2,...,argn)`**

Executes any C function in your program. After execution, control usually returns to the debugger with the environment unchanged, except for any side-effects of the function. The type and number of arguments must match.

One use of this command is to execute a previously written debugging function, e.g., to print out a complex program structure in a more customized manner than can be done by the debugger. You may also use a function to set up inputs, such as arrays or structures, to be processed by later stages of your program.

`call` prints the return value (if any) of the called function.

Caution When executing an arbitrary set of functions, it is possible to lose control if you are not aware of function flow. This command saves all the registers before it calls the function, and restores them on return from the function. Hence, global variables used as registers may have their values restored.

There is no support for functions with variable number of arguments. The debugger does not accept literal strings as arguments.

Example

```
>call toint('9')
```

Return value:

```
    9
```

The function **`toint`** has a prototype **`int toint( char c )`**, hence when it is called with a character **`'9'`**, it converts into integer and prints **`9`**.

4.6 CD — CHANGE WORKING DIRECTORY

`cd [`*`path`***`]`** sets the current working directory for creating/reading log and other files. The **`quit`** command returns you to the directory from which you invoked the debugger.

`cd` displays the current working directory.

Mouse Open ***FILE*** and select ***WORK DIRECTORY***

Examples To change the current working directory to `/usr/ISDN/SRC`:

```
>cd /usr/ISDN/SRC
```

To display the current working directory:

```
>cd
    /usr/ISDN/SRC
```

4.7 CHIP — SELECT CHIP FOR EMULATION OR SIMULATION

chip [*chipname*]
Simulates, or emulates, a particular chip. If no argument is specified, the debugger displays the name of the currently selected chip. **chip** is an ideal command to specify in the **.env** or **.ini** files.

Mouse Open ***CONFIG*** and select ***EDB*** or ***SIM***.

4.8 COMM — SET COMMUNICATIONS PARAMETERS

Sets the communication parameters required to communicate with your target board. See the appropriate board manual for the supported baud rate.

comm -b *baud* sets the baud rate for serial communication.
baud 19200

comm -l *line* sets the comm port: com1 or com2 for serial communication.
line 1 | 2

comm -e *hostname* | *IP addr*
sets the ETHERNET communication parameters.
hostname A name in the *hosts* file.
IP addr IP address in *aa.bb.cc.dd* format.

comm displays the current settings.

Caution The **comm** command does not change the baud rates of the target. You must change the speed settings on the board. Before using the ETHERNET feature of the debugger, you must configure the target to work in ETHERNET mode. See the appropriate target manual. Currently, DEX-APB works at 19200 baud.

Mouse Open ***CONFIG*** and select ***EDB*** ➨ ***COMMUNICATIONS***, or open ***CONFIG*** and select ***ISE*** ➨ ***COMMUNICATIONS***. Set up the communication parameters in the dialog box.

Examples To set the baud rate:

```
>comm -b 19200;
    RS232: Baud = 19200, Port = 2
```

To select serial port:

```
>comm -l 2
    RS232: Baud = 19200, Port = 2
```

To list current settings:

```
>comm
    RS232: Baud = 19200, Port = 2
```

To select ETHERNET of the development board using IP address:

```
>comm -e 192.23.37.69
    ETHERNET: host = 192.23.37.69
```

To select ETHERNET of the development board using host address:

```
>comm -e target1
    ETHERNET: host = target1
```

If target1 is an alias for address 192.23.37.69, communication is established to the target at that IP address.

4.9 DEBUG — SELECT THE EXECUTABLE FILE FOR DEBUGGING

debug [-x | -n | -xn] *executable*

specifies a COFF file, downloads its code, and reads its symbol tables to the target board or Simulator.

-n do not download the executable file to be downloaded.

-x do not read symbol table.

Mouse Open ***FILE*** and select ***LOAD***. Enter or select the filename for down loading.

Examples To select *browser.cof*

```
>debug browser.cof
```

To select **fax.exe**, and download to target, without reading its symbol tables.

```
>debug -x fax.exe
```

To select the executable file **prog**, and read the symbol tables without downloading code from the COFF file to the target.

```
>debug -n prog
```

4.10 DEBUGMODE — SELECT DEBUGGING MODE

debugmode {-e | -d} [*startup* | *exitcode* **]**

enables or disables the debugging of C startup code, non-symbolically linked object modules, or C exit code. The default settings are:

- Debugging of startup code is disabled. i.e., the debugger executes up to **main()** and stops at the start of the main function.
- Debugging of non-symbolically linked object modules is disabled.
- Debugging of C exit code is disabled.

exitcode refers to the exit code provided by the C compiler. Disabling the debugging of startup code, disables performance data collection, even if you have selected performance mode on, when using the simulator.

Caution If the C startup code is not assembled with symbols for debugging, and the code is in ROM, the **reset** command may be slow.

Mouse Open ***EXECUTE*** and ***SELECT*** debugmode.

Examples To allow debugging of *startup* code that has been modified:

```
>debugmode -e startup
```

4.11 FIND — FIND VALUE IN MEMORY

find {-a | -b | -c | -w | -f | -p | -l | -i} *value, addr_range*

Finds a pattern in memory. The options are:

-a ASCII string
-b byte
-c char
-w word
-f float
-p pointer
-l long
-i assembly instruction (Not yet supported)

If you do not specify a qualifier, the debugger finds the pattern in memory with a qualifier based on the size of the value.

Examples To search memory, beginning with the address pointed to by *stepte*, for the string "uphill." Displays the memory address where found.

```
>find -a "uphill", stepte
```

To search the memory range 0xc000//0xcfff for "string2."

```
>find -a "string2", 0xc000//0xcfff
```

To search for 345.67 in memory address range 0xd800//0xd8ff.

```
>find -f 345.67, 0xd800//0xd8ff
```

To report if the instruction **"stor r0, flag"** is in memory in the address range 0xe800//0xe8ff.

```
>find -i "stor r0, flag", 0xe800//0xe8ff
```

4.12 FINDSRC — FIND STRING IN A SOURCE FILE

findsrc [-f | -b | -n] [*string*] [,*file_name*]

finds a specified string in the current source file, or the file specified by *file_name*, and updates the Source Window to the selected line. If the string is not found, the Source Window is not changed. If the current source file is used, **findsrc** begins the search at the currently displayed line. The options are:

-f for forward search
-b for backward search
-n for next

string is the specified search string (NULL on first reference). Use double quotes to enclose words separated by blanks or commas. If *string* is not specified for the **-n** option, the last specified search string is used. For option **-f** or **-b**, you are prompted for input.

The file name can contain wildcard characters. The default is the currently displayed file in the source window.

Mouse

Open ***SOURCE*** and select ***SEARCH STRING***.

Examples

To search for "getnum" in all files of the current working directory with the extension c:

```
>findsrc "getnum", *.c
```

To search for "backhandle" in the current source file, from the current display line to the end of the file, and then from the beginning of the file to the current source line.

```
>findsrc -f "backhandle"
```

To search for string "malloc" in the current source file, from the current display line to the beginning of the file, and then from the end of the file to the current display line.

```
>findsrc -b "malloc"
```

To search for the string specified in the previous **findsrc** command in the same direction as for the previous command.

```
>findsrc -n
```

4.13 GO — EXECUTION OF USER PROGRAM

go [-c] [*from_addr***][//***end_addr***]**

issues a **go** command to the target.

If you specify the **-c** option (continue), the debugger does not stop at a break condition, but updates the windows, issues **autocommand** (Section 4.3), and continues execution. The Status Window shows the trigger condition.

from-addr and *stop-addr* are any valid code addresses.

If the debugger encounters a breakpointed state, and **-c** is not specified, it stops and updates the Source Window. The source line corresponding to the address contained in the PC is highlighted.

After a **GO** command, the debugger waits for a response from the target. To regain control, open ***EXECUTE*** and select ***ABORT***.

When the **-c** option is specified, execution is handled asynchronously, and you retain some control over the debugger and can use the menus. To abort the target, Open ***EXECUTE*** and select ***ABORT***. The debugger updates the Status and Source Windows upon receiving a response from the target.

from_addr must be a valid code address for the execution starting points. *end_addr* must be a valid code address for the stopping point. Otherwise, results are unpredictable.

Mouse

To go from the current PC, open ***EXECUTE*** and select ***GO***, or select the ***GO*** button from the configurable buttons.

To go from the current PC until you reach the current source line, open ***EXECUTE*** and select ***GOTILL***.

To re-execute the code from the beginning, open ***EXECUTE*** select ***RUN***.

For continuous execution, or to specify a range, open ***EXECUTE*** and select ***ADVANCED*** to select your options.

To restart from the beginning of the program, open ***EXECUTE*** and select ***RERUN***.

Examples

To start debugging, for breakpoint address 0xE809, current module *crdb_ch2.c*, current line number 20, and current function *main*:

```
> go
    Realtime breakpoint
    Breakpointed at :
    [1] (0xE809) crdb_ch2.c#20@ main
```

To execute from line 10 to line 12 of the current display file:

```
> go #10//#12
```

To execute from address 0xd800 to address 0xd810 without stopping at breakpoints. Control returns to you immediately.

```
> go -c 0xd800//0xd810
```

To execute from the current PC to the return address of the current 'C' function:

```
> go -c ..
```

4.14 INFO — DISPLAY DEBUGGER INFORMATION

info prints the current settings of the following parameters:

- Debugger name and version.
- Target emulator (See **target** command, Section 4.38).
- Target CompactRISC chip (See **chip** command, Section 4.7).
- Environment variable CRDBENV.
- Current working directory (See **cd**, Section 4.6).
- Source directory path (See **srcpath**, Section 4.32).
- Environment file name.
- Initialization file name.
- Current program name (See **debug**, Section 4.9).
- Log file name (See **log**, Section 4.17).

4.15 INPUT — EXECUTE COMMAND SCRIPT FILE

input *file_name*

executes the commands from an input (command) file (see Section 2.2.12). An input recursion of up to four levels is allowed.

Specify **pause** (Section 4.21) to suspend the execution of commands from the input file. Use **resume** (Section 4.25) to continue the execution of commands from the input file after **pause**.

Section 2.2.12 describes the debugger facilities provided for use with input files and explains how to use them.

If an abort (Section 2.2.8) is issued while an input file is being executed, execution of this file, as well as its parent files are aborted.

Mouse Open ***FILE*** and select ***COMMAND FILE***.

4.16 LIST — LIST MEMORY OR FILE

list -m[h|o|d] [b | c | w | f | p | l | i] *address | addr_range*

lists the contents a memory range. The options are:

-h print the values in hexadecimal
-o print the valued in octal
-d print the value in decimal
-b byte
-c char
-w word
-f float
-p pointer
-l long
-i assembly instructions

radix command (Section 4.23) affects the output this command format.

When you specify the address as a numeric expression or in register notation (e.g., **%PC**) or in expression format (e.g., **errno+5**), the debugger evaluates the expression and uses the result as the address.

When you specify the symbolic name (e.g., **errno**), the debugger displays the contents of the variable.

Mouse To list a particular section of memory, open ***SHOW*** and select ***MEMORY*** window.

list *qualified_lineno*

brings the specified source or text file into the Source Window. Use **srcmode** (Section 4.31) to set the display mode. The file is displayed in the Source Window.

You can also view files that do not belong to the current COFF file (e.g., header files). The debugger searches for these files in the directories specified by **srcpath** (Section 4.32).

qualified_line_no can be described as follows:

[*filename***][***@func_symbol***]** *const_lineno*

The requested line, and as many of the following lines as possible, are displayed in the Source Window.

Mouse Position the cursor on the Line box on the fourth row of the Main Window. Highlight the currently displayed line number, enter the new line number, and press ***RETURN***. Similarly, you can use the ***FILE*** and ***FUNCTION*** entries on the row to display a particular file or function.

Examples To list memory within the range sCStr1//sCStr1+60, in floating-point format:

```
> list -mf sCStr1//sCStr1+60
```

To list the contents of the array slice, a_3i[1][0][0]//a_3i[2][0][0]:

```
> list -mw a_3i[1][0][0]//a_3i[2][0][0]
```

To disassemble the code in the function *TestUnion* within the module **vars2.c**:

```
> list -mi vars2.c@TestUnion$b//vars2.c@TestUnion$e
```

To list the file **test1.c** in the Source Window:

```
> list test1.c
```

To list the file **tmp.c** in the Source Window, and place the current display at the function **form_list**:

```
> list tmp.c@form_list
```

4.17 LOG — RECORD DEBUGGER COMMAND SESSION

Records the sequence of commands issued to the debugger and, optionally, the responses returned by the debugger.

log [-a] [*file_name***]**

-a appends the recording to an existing file.

file_name
specifies the log-file name. The default is **crdb.log**.

Invoking the debugger with **-l** option is the same as issuing the **log** command with no arguments or options. (See Section 2.2.3.)

log [-d | -e]

-d disables the log.

-e enables (default) the log.

log [-i | -o]

-i signifies input only. This is a sticky option

-o signifies output also (default).

log [-f | -u]

-f specifies full form (expanding alias, set, etc.). This is a sticky option.

-u specifies unexpanded form (default).This is a sticky option.

If no arguments are given, the debugger records both the commands and their responses in a file named **crdb.log**. There is no default for the file name's extension.

Specify **-a** to append the commands, and possible responses, to an existing file. Otherwise, the debugger creates a new file for logging.

A sticky option apply to all the log operation until it is explicitly disabled. For example, if you open another *file_name* the sticky options do not reset to their default values.

log -s displays the current status of the log-options.

For information on the log file, see Section 2.2.12.

Mouse Open ***FILE*** and select ***LOGFILE***. Fill in the dialogue box.

Example To log all subsequent commands, and output into the **commands.log** file in the current working directory:

```
> log commands.log
```

4.18 MODIFY — MODIFY CONTENTS OF MEMORY OR SYMBOLS

modify [-b | -c | -w | -f | -p | -l | -i] *address* | *addr_range* **[,***value* **[,***value***]]**

-b byte
-c char
-w word
-f float
-p pointer
-l long

Modifies memory locations in various formats. *addr_range* can be any valid text/data address, or a symbol reference, or an address range. The default format for *address* is byte, and for a symbol is based on the type of variable. The value should conform to the format; otherwise, a conversion is applied whenever possible.

If an *addr_range* is specified, and the number of values specified is less than the number of locations in the address range, the values are written into memory repeatedly.

modify -a *string_pointer*,*string*

-a string copy

If the **-a** option is specified, the debugger puts a string copy of *string* into the location pointed to by the value of *string_pointer*. If *string_pointer* is defined as a character, you may specify **&***string_pointer* in the command. See example below.

modify %*reg_name*, *value*

If the **%** option is specified, the debugger sets the specified register to the new value. Register names are target-specific, and are specified in the appropriate appendix.

If `value` is omitted, **modify** becomes an interactive command. If an `array` or `structure` is specified, the debugger displays an element at a time and accepts a new value for each element, or press ***RETURN*** if the current value is not to be modified. The addresses displayed by the debugger are spaced in memory according to the length of the values to be specified. If a single `address` is specified, the debugger queries one time for the new value.

Caution The debugger does not keep track of the proper memory alignments for length values greater than 1; you must do this yourself.

Mouse Open ***SHOW*** and select the ***MEMORY***, ***REGISTERS*** or ***SYMBOL*** windows to modify the programs.

Examples To store the characters 'A', 'B', 'C', and 'D' into the array elements a[0]//a[3]:

```
>modify &a[0]//&a[3],'A','B','C','D'
```

To store the string into the location addressed by **str**, where **str** is defined as char *:

```
>modify -a str,"ABCD"
```

To store the characters, 'A', 'B', 'C', and 'D', into the memory locations addressed by **str** and the following three locations

```
>modify str//str+3,'A','B','C','D'
```

To store the string into memory, beginning at the address of **str[2]**, where **str** is an array of char.

```
>modify -a &str[2],"ABCDF"
```

To store the value 385 into the variable iSNum:

```
>modify iSNum, 385
```

To store the value 25.642 into the variable ifNum:

```
>modify ifNum, 25.642
```

To set the value of general register r4 to the value, 0xff3e1415:

```
>modify %r4, 0xff3e1415
```

4.19 NEXT — EXECUTE NEXT SOURCE LINE (STEP OVER)

The debugger executes the next source line (i.e., C language source line) as a whole. If the statement is a function call, the entire function is executed and the debugger stops on the source line following the call. In some debuggers, this function is also known as step-over.

next -c causes continuous execution of source lines until the end of program.

next -n *number*
causes execution of the given number (n) of source lines.

next [*from_addr* **[//***end_addr***]]**
executes the source lines in the given range. If only *from_addr* is specified, the single statement beginning at that address is executed. If no range is specified, the single statement addressed by the current contents of the PC is executed.
from_addr must be a valid code address for the execution starting points. *end_addr* must be a valid code address for the stopping point. Otherwise, results are unpredictable.
After each source line completes executing, the commands specified in the **autocommand** list (Section 4.3) are executed.
After the debugger executes a **next** command, it displays, for example:

```
Next to 0xE8B0 : crdb_ch2.c : plus_ab #52
```

where 0xE8B0 is the current address, crdb_ch2.c is the current module, **plus_ab** is the current function, and 52 is the current line number.
After **next**, the debugger waits for a response from the target. To regain control, open ***EXECUTE*** and select ***ABORT*** to abort the command.

Mouse To execute a single statement (executing through a function), select ***NEXT*** on the configurable menu, or open ***EXECUTE*** and select ***NEXT SOURCE LINE***. For continuous execution, or to specify a statement count or range, first open ***EXECUTE*** and select ***ADVANCED***.

Caution Attempting a **next** over a complex source line may cause the debugger to use software breakpoints internally.

Examples To execute the next source lines, starting at line 10 up to line 12 of the displayed source file:

```
>next #10//#12
```

To execute the next source lines, starting at line 10 up to line 12 of the displayed source file. If any of your breakpoints are detected, they are reported and execution continues.

```
>next -c #10//#12
```

To execute the next 10 source lines, one at a time:

```
>next -n 10
```

4.20 NEXTINS — EXECUTE NEXT ASSEMBLY INSTRUCTION (STEP OVER)

The debugger executes the next assembly instruction. If the instruction is a Jump to Subroutine, the entire subroutine is executed and the debugger stops on the instruction following the Jump to Subroutine. This is similar to the **next** command but at instruction level.

nextins -c executes instructions until the end of the program.

nextins -n *number*
executes the given number (*n*) of instructions.

nextins [*from_addr* **[***//end_addr***]]**
executes the instructions in the given range. If only *from_addr* is specified, the single instruction beginning at that address is executed. If no range is specified, the single instruction addressed by the current contents of the PC is executed.
from_addr must be a valid code address for the execution starting points. *end_addr* must be a valid code address for the stopping point. Otherwise, results are unpredictable.
After each instruction has been executed, the commands specified in the **autocommand** list (Section 4.3) are executed.
After a **nextins** command, the debugger waits for a response from the target. To regain control, press CTRL-C. If you specify the **-c** option, control returns to you before the command is completed. In this case, open ***FILE*** and select ***ABORT*** to abort the command.

Mouse To execute a single instruction, executing through a function, open ***EXECUTE*** and select ***NEXT INSTRUCTION***. For continuous execution, or to specify an instruction count or range, first open ***EXECUTE*** and select ***ADVANCED***.

Open ***EXECUTE*** and select ***ABORT EXECUTION*** to abort the command.

Examples To execute the next 10 instructions, starting from the current PC:

```
>nextins -n 10
```

To execute the instructions between line 10 and line 12 in the displayed source file:

```
>nextins -c #10//#12
```

To execute the machine-language instructions from address 0xd800 through address 0xd810:

```
>nextins 0xd800//0xd810
```

4.21 PAUSE — SUSPEND INPUT FILE EXECUTION

pause suspends execution of commands from the input file, and prompt for input at the Command Window.

When a **pause** is executed, the debugger displays the following message:

```
*** PAUSED for input, type 'resume' to continue
```

You can execute several commands via the Command Window. To resume execution of the input file commands, use **resume** (Section 4.25).

This command is only useful in an input file.

4.22 QUIT — EXIT FROM THE DEBUGGER

quit quits debugging session, without waiting for confirmation.

Mouse Open ***FILE*** and select ***QUIT***.

Caution To continue your debugging session at a later stage, use **savestate** (Section 4.27). To resume at the same point, use **setstate** (Section 4.29).

4.23 RADIX — SET RADIX FOR OUTPUT DISPLAY

radix [8 | 10 | 16]

sets the radix for displaying output values to octal (8), decimal (10), or hexadecimal (16). The default radix is decimal.

When no argument is specified, the debugger displays the current radix.

The command affects the default behavior of the **watch**, **view**, **where**, and **list** commands with **-m** option.

Mouse Open ***CONFIG*** and select ***RADIX***.

Caution This command sets the radix for outputs only. Specify input values using the standard syntax for C language constants. **radix** does not affect the output of the memory window.

Examples To display the current radix:

```
>radix
```

To set the radix to 16 (hexadecimal):

```
>radix 16
```

4.24 RESET — RESET THE DEBUGGER AND THE TARGET BOARD

reset The debugger issues a **reset** command to the target board. The source file display is synchronized to the beginning of the program.
By default, the debugger executes your program up to the first instruction in the main program (*main$b*). See **debugmode** (Section 4.10) if you want to change this behavior.

Mouse Select ***RESET*** on the main menu, or open ***EXECUTE*** and select ***RESET.***

Caution To debug program initialization code, which has been modified while developing a C application for CompactRISC, invoke debug with the **-e** option, and specify the directory containing the source file as your source path (Section 4.32). If the debugger does not find the file in the source path, it prompts for an alternative directory.

The target system's response to the **reset** command is hardware-dependent.

4.25 RESUME — RESUME EXECUTION OF INPUT FILE

resume Resumes execution of an input file that was suspended with **pause** (Section 4.21). Other commands can be executed before **resume**.

4.26 SAVECONFIG — SAVE CURRENT DEBUGGER CONFIGURATION

saveconfig [*file_name*]

saves current configuration setting in *file_name*, (default **crdb.env**), in the directory pointed to by **CRDBENV**, or in the startup directory if **CRDBENV** is not set.

- target chip name
- communication parameters
- display colors
- window sizing information

Since parameters are saved in the form of debugger commands, you can execute these commands with **INPUT**.

Mouse Open ***FILE*** and select ***SAVESETUP***.

Caution If you do not specify *file_name*, **crdb.env**, if it already exists, is copied into the file **crdbenv.old**, and **crdb.env** is overwritten.

Examples To save the configuration in the default file **crdb.env**, in the default location:

```
> saveconfig
```

To save the configuration in *config.in*, in the current working directory:

```
> saveconfig config.in
```

4.27 SAVESTATE — SAVE CURRENT DEBUGGING STATE

savestate [*file_name*]

Saves the current debugging settings in *file_name* in the current working directory, or in the default file, **crdb.ctx**. The file contains information about the current state of the debugger, including break/softbreak lists, current COFF file, **cd**, and **autocommand**, **watch**, **alias**, **set**, and **srcpath** lists. Use **savestate** and **setstate** (Section 4.29) to **quit** the debugger, and later restore the saved settings.

Mouse Open ***FILE*** and select ***SAVE STATE***.

Caution The environment depends on the state of the debugger, the target, and the target chip. **savestate** and **setstate** only save and restore the state of the debugger. If you are unsure about the state of the target, restart your program.

If you specify no arguments and **crdb.ctx** already exists, it is overwritten.

Examples To save the current state of the debugger in **crdb.ctx**, in the current working directory.

```
> savestate
```

To save the current state of the debugger in **save.fil**, in the current working directory.

```
> savestate save.fil
```

4.28 SET — DEFINE DEBUGGER VARIABLES AND STRINGS

Defines debugger variables and function keys.

set *name=string*
defines *name* to have the value *string*. Whenever you specify **$***name*, in a command *string* is substituted.

set -r *name* | *
removes *name* from the list of defined names. Specify an asterisk (*) to remove every name from the list.

set *name* displays the current value of *name*.

set displays all of the values currently in the list of defined names

Examples To set a symbol **uart** to 100:

```
> set uart = 0x100
```

You can make use of the symbol **uart** in a command by specifying **$uart**:

```
> list -mw $uart
```

4.29 SETSTATE — RESTORE DEBUGGING STATE

setstate [*file_name*]
restores the debugging state as saved by **savestate** (Section 4.27). The debugger automatically downloads your previous COFF file.
The debugger reads *file_name* from the current working directory. The default filename is **crdb.ctx**.

Mouse Open ***FILE*** and select ***LOAD STATE***.

Caution The debugger assumes that the state of the target has not changed since **savestate** was executed.

The debugging environment depends on the states of the debugger, target, and target chip. **savestate** and **setstate** (Section 4.29) only save and restore the state of the debugger.

Examples

To set the state of the debugger to the state captured in the file **crdb.ctx** in the current working directory:

```
>setstate
```

To set the state of the debugger to the state captured in the file **save.fil**:

```
>setstate save.fil
```

4.30 SOFTBREAK — SET SOFTWARE BREAKPOINT

Similar to **break** (Section 4.4), but the breakpoint is implemented by software; hence, the program does not operate in real-time mode if an occurrence count or conditional expression is specified. This mode does not have the global break occurrence count or break qualifier. The break occurs only on an opcode fetch.

For a general explanation of breakpoints, see Section 2.2.7.

softbreak [-t] *softbreak_list* **[,c**=*RLexp***] [,o**=*occ_cnt***]**
adds a soft break.

c=*RLexp*	specifies a condition
o=*occ_cnt*	sets the occurrence count. During operation, the occurrence count is decremented only if the breakpoint condition is met.
-t	adds temporary software breakpoints. These breakpoints are deleted after they occur.
softbreak_list	lists the code addresses at which the breakpoints are set.

softbreak { -r | -d | -e } *%id* **|** *

-r	removes breakpoint
-d	disables breakpoint
-e	enables breakpoint

softbreak lists the current softbreak items.

Mouse

To set a complex breakpoint, open ***BREAK*** and select ***SOFTBREAK***. Fill in the dialog box.

To set a simple execution breakpoint on a particular line of your source file, use the fourth row editable fields like File, Module, Line in the Main Window (Section 3.2) to select and display the appropriate line in the Source Window, and then click the left mouse twice to set **softbreak**. The debugger acknowledges the setting of the breakpoint by displaying an S at the left end of the source line. Double-click again to remove the breakpoint.

You can also set a software breakpoint by pressing the corresponding button in the second row of the Main Window (Section 3.2) when the cursor is on the desired source line.

Caution These breakpoints are implemented by software, and may result in non real-time operation. The total number of soft breakpoints that can be set at a time is limited, and depends on the target. Refer to the appropriate manual for the target.

Examples To set a software breakpoint at the final return of the function *TestVars* in module *vars3.c*:

```
>softbreak vars3.c@TestVars1$e
```

To set a software breakpoint at the current display line:

```
>softbreak #
```

To set a temporary software breakpoint at line number 10 in the current module:

```
>softbreak -t #10
```

4.31 SRCMODE — SET SOURCE FILE DISPLAY MODE

Sets the Source Window display mode. By default, source-only is enabled

srcmode [-s | -m]
sets the display mode for the Source Window.
-s enables source-only display.
-m enables mixed mode display.
(Displays both the source line and lines of generated assembly code.)

Mouse Open ***SOURCE*** and select ***DISPLAYMODE***.

Note A source line may be associated with two separate blocks of assembly code (e.g., a **for** loop line for which the compiler generates code both before and after the loop body). In this case, if you select mixed mode display, you may find the results confusing; there are two non-contiguous blocks of assembly code after the source line. Although this might be confusing at first sight, it is correct and reflects the reality.

The disassembled code shown in the source window is obtained by disassembling the instructions from the COFF file. Hence, this does not reflect code changes which are made to the target memory.

4.32 SRCPATH — SET DIRECTORY PATH FOR SOURCE FILES

Sets the directory pathname for the source files search. The last entry added, or enabled, is the first directory to be searched.

srcpath *pathname_list*
adds a pathname or list of pathnames.

srcpath -r *pathname* | *
removes a pathname. Specify an asterisk, to remove all pathnames.

srcpath displays the current source search path. If it does not find a source file for display, the debugger asks for the name of the directory to be searched.

Mouse Open ***SOURCE*** and select ***SOURCE PATH***. You have more control here to delete or add a directory.

Examples To add ..*test* to the set of source paths:

```
>srcpath..\test
```

To remove ..*data* from the set of source paths:

```
>srcpath -r ..\data
```

To display the current set of source paths:

```
>srcpath
```

4.33 STDIO - REDIRECT VIRTUAL I/O STANDARD FILES

STDIO directs the output of standard I/O functions (Section 2.2.10) to standard files, to a host disk file, to the Output Window. By default, **stdin**, **stderr** and **stdout** are redirected to the Output Window..

stdio -i *<filename>*
maps the **stdin** operations to the specified file.

stdio -o[a]*<filename>*
maps the **stdout** operations to the specified file. If you specify the **-a** option, new output is appended to the current file.

stdio -e[a] *<filename>*
maps the **stderr** operations to the specified file. If you specify the **-a** option, new output is appended to the current file.

stdio -oe[a] *<filename>*
maps the **stdout** operations and **stderr** operations to the file specified. If you specify the **-a** option, new output is appended to the file.

stdio [-i|-e|-o]w
maps the specified operations, **stdin**, **stdout**, or **stderr**, to the Program Output Window.

stdio [-i|-e|-o]r
maps the specified operations, **stdin**, **stdout**, or **stderr**, to the program invocation terminal.

stdio lists the current mappings of I/O operations.

4.34 STEP — STEP ONE SOURCE LINE

The debugger executes the current, or given source line, i.e., C-language line, as a whole. After the execution of source line(s), the commands specified in the **autocommand** list (Section 4.3) are executed.

To regain control, during a **step** command, open ***EXECUTE*** and select ***ABORT EXECUTION*** to abort the command.

step -c Executes source lines continuously until the end of program. If a softbreak or break occurs in this process, the debugger issues a report and continues stepping.

step -n *number*
executes the given number (n) of source lines.

step [*from_addr* **[//***end_addr***]]**
executes the source lines in the given range. If only *from_addr* is specified, the single source statement beginning at that address is executed. If no range is specified, the single source line addressed by the current contents of the PC is executed.
from_addr must be a valid code address for the execution starting points. *end_addr* must be a valid code address for the stopping point. Otherwise, results are unpredictable.

Mouse To step a single source line, press function key F4, or click on the ***STEP*** button, or open ***EXECUTE*** and select ***STEP SOURCE LINE***. For continuous execution, or to specify a source line count or range, first open ***EXECUTE*** and select ***ADVANCED***.

Open ***EXECUTE*** and select ***ABORT EXECUTION*** to abort this command.

Caution Attempting a **step** over a complex source line may cause the debugger to use either hardware or software breakpoints, and may result in non real-time operation.

Examples To execute source lines from line 10 through line 12 of the source file displayed in the source window. If a breakpoint is detected, it is reported and execution continues until line 12 is executed.

```
>step -c #10//#12
```

To step 10 source lines, starting from the current PC:

```
>step -n 10
```

4.35 STEPINS — STEP ONE ASSEMBLY INSTRUCTION

Executes the current assembly instruction or given instruction.
Upon completion of each instruction, the commands specified in the **autocommand** list (Section 4.3) are executed.

After a **stepins** command, the debugger waits for a response from the target. To regain control, open ***EXECUTE*** and select ***ABORT EXECUTION*** to abort the command.

stepins -c executes instructions continuously until the end of program.

stepins -n *number*
executes the given number (*n*) of instructions.

stepins [*from_addr* **[***//end_addr***]]**
executes the instructions in the given range. If only *from_addr* is specified, the single instruction beginning at that address is executed. If no range is specified, the single instruction addressed by the current contents of the PC is executed.
from_addr must be a valid code address for the execution starting points. *end_addr* must be a valid code address for the stopping point. Otherwise, results are unpredictable.

Mouse

To step a single instruction in an assembly program, open ***EXECUTE*** and select ***STEPINSTRUCTION***.
For continuous execution, or to specify an instruction count or range, first open ***EXECUTE*** and select ***ADVANCED***.

Open ***EXECUTE*** and select ***ABORT EXECUTION*** to abort this command.

Examples

To execute instructions from address 0xd800 to address 0xd810. If a breakpoint is detected during execution, it is reported and execution continues.

```
>stepins -c 0xd800//0xd810
```

To execute 10 instructions, starting with the instruction addressed by the current PC:

```
>stepins -n 10
```

Notes

1. A limitation of the CompactRISC architecture makes it impossible to perform a single step if the next instruction modifies the entire contents of the **PSR** register. Do not use the **stepins** command if the next instruction is **RETX**, or **LPR** with **PSR** as the second operand.

2. In simulation mode, you can only perform single steps after the point in the start-up routine where both the following conditions are fulfilled:
   - the **INTBASE** register points to a dispatch table, with the appropriate **TRC**, **BPT** and **ISE** trap handlers
   - the **ISP** and **SP** registers are initialized.

   For convenience, the debugger initializes these registers to default values, allowing you to perform single steps from the first instruction of your program. The debugger looks in your program for the following global symbols on which to base the default values:
   - **__dispatch_table** (dispatch table - used to initialize **INTBASE**)
   - **__ISTACK_START** (start address of interrupt stack - used to initialize **ISP**)
   - **__STACK_START** (start address of program stack - used to initialize **SP**)

The default startup routine uses all these symbols. To enable single steps from the first instruction, in simulation mode, we recommend that you continue to use these symbol names, even if you modify the start-up routine.

4.36 SYMBOL — DISPLAY SYMBOL CHARACTERISTICS

Displays the characteristics of the symbols.

symbol {* | *pattern**}**
displays all symbols matching the pattern. The pattern can be *symbol_name* or *symbol_qualifier* (any sequence of characters which can begin a valid symbol). The asterisk is a pattern wildcard, standing for zero or more additional characters.

symbol -l{ * | *pattern** **}**
displays only local (automatic variable) symbols from the current function.

symbol -t {*datatype/tagname* **|** *symbol_name* **|** ***}**
displays the tag or type of the structure or symbol name.

symbol -f *qualified_modulename*
displays all the symbols from the specified module. Since global symbols are not attached to any specific module, they are not displayed.

symbol -g [*pattern****]**
lists all the names of the globals beginning with *pattern*. If *pattern* is omitted, the names of all the global symbols are listed.

symbol *address*
attempts to find the symbolic mapping of *address*

Mouse Open ***QUERY*** to get a dialog box. (Section 3.2). Use the info button to get the details on the symbol.

Examples To search for the symbol, *getnum*, in the current scope and display the characteristics:

```
> symbol getnum
```

To search for the symbol *new_int* in the current function and display the characteristics:

```
> symbol -l new_int
```

To display the tag of the symbol *k_struct*:

```
> symbol -t k_struct
```

To display all the symbols defined in the module **temp.c**:

```
> symbol -f temp.c
```

To display all the global symbols:

```
>symbol -g
```

To display the symbolic mapping of address 0x28c:

```
>symbol 0x28c
    iSNum2:
        Scope          : Static to file
        Declared in    : vars1.c
        At address     : 0x28c
        Size (bytes)   : 2
        Declaration    :
            static int iSNum2;
```

4.37 SYNC — SYNCHRONIZE SOURCE FILE DISPLAY

sync brings the source line corresponding to the current PC into the Source Window. This is helpful when you are looking at a source file other than the current file, and want to restore the display to the current execution context.

Mouse Open ***SOURCE*** and select ***SHOW PC LINE***.

4.38 TARGET — SPECIFY EXECUTION ENGINE

target [*emulator_name*]

emulator_name EDB or SIM.
edb - for using National's development system, i.e., DEX-ADB.
sim - for using National's instruction and performance simulator.
Specifies the type of emulation, or simulation, engine, e.g., simulator, development board, ISE or target board.
info (Section 4.14) displays the name of the currently selected target.
When you invoke the debugger, it assumes the target is a development board.

Mouse Open ***CONFIG*** and select ***EDB*** to use the development board.
Open ***CONFIG*** and select ***SIM*** to use the Instruction-Level Simulator

Examples To use the Simulator for program development:

```
>target sim
```

To display the current selection:

```
>target
    Current_target = SIM
```

4.39 VERBOSE — MONITOR COMMUNICATION TRAFFIC TO TARGET

verbose enables or disables the monitoring of traffic between target and the debugger. Each specification toggles the previous state. Upon startup, the mode is disabled. This command is for diagnosis purposes only.

Mouse Open ***CONFIG*** and select ***VERBOSE***.

4.40 VIEW — VIEW VALUE OF STRUCTURE OR SYMBOL

view *expression*[,*print_specifier*]

Computes the value of *expression*. *expression* may be a C language symbol, or structure element reference. When a symbol or structure element is specified, without any *print_specifier*, it displays the symbol based on the symbol type and the current **RADIX** setting (Section 4.23).

view %*reg_name*[,*print_specifier*]

displays the value of the specified register. Register names are target-specific.

expression is a C-language symbolic expression.

print_specifier is one of:

- *d* signed decimal
- *i* signed decimal
- *o* unsigned octal
- *x* unsigned hexadecimal using “a, b, c, d, e, f”
- *X* unsigned hexadecimal using “A, B, C, D, E, F”
- *u* unsigned decimal integer
- *e* floating-point in engineering notation
- *E* floating-point in engineering notation
- *f* floating-point
- *g* double-signed value printed in +/- format
- *G* double-signed value printed in +/- format
- *c* single character
- *s* character string
- *hd* short-signed decimal integer
- *hi* short-signed decimal integer
- *ho* short-unsigned octal integer
- *hX* short-unsigned hexadecimal integer using “ABCDEF”
- *hx* short-unsigned hexadecimal integer using “abcdef”
- *hu* short-unsigned decimal integer
- *ld* long-signed decimal integer
- *li* long-signed decimal integer
- *lo* long-unsigned octal integer
- *lX* long-unsigned hexadecimal integer using “ABCDEF”
- *lx* long-unsigned hexadecimal integer using “abcdef”
- *lu* long-unsigned decimal integer
- *Le* long-double in engineering notation (lowercase e for exponent)
- *LE* long-double in engineering notation (uppercase E for exponent)
- *Lf* long-double floating-point
- *Lg* long-double signed value printed in f or e format, whichever is more compact for the given value and precision. The e format is used only when the exponent of the value is less than −4 or greater than the specified precision. Trailing zeros are truncated and the decimal point appears only if one or more digits follow it.
- *LG* long-double signed value, identical to the g format except that G introduces the exponent (where appropriate) instead of E.

Mouse

You can use any of the following options to view a variable:

- Select ***QUERY*** in the main window.
- Open ***SHOW*** and select ***WATCH,*** to specify the variable names to be watched continuously.
- Open ***SHOW*** and select ***LOCAL VARIABLES***, to print the current values of the local variables and arguments of the currently executing functions.
- Highlight a variable, with the mouse, and select ***PRINT***.

Examples

To view the value of the expression *sStr1+4*:

```
>view sStr1+4
```

To view the value addressed by the pointer *pPtr4*:

```
>view *pPtr4
```

To view the sum of the value addressed by *pPtr2* and 4:

```
>view *pPtr2+4
```

To view the value addressed by the sum of the contents of *sXStr1* and 18, where *sXStr1* is a pointer variable:

```
>view *(sXStr1+18)
```

To view the value of the array element *a_2c[1][1]*:

```
>view a_2c[1][1]
```

To view the value addressed by the array element *days[5]*, where days is an array of pointers

```
>view *days[5]
```

To view the value of the array element *days[5]*:

```
>view days[5]
```

To view the value of member *next*, of the structure addressed by member *next*, of the structure addressed by the contents of *e_list*

```
>view e_list->next->next
```

To view the high byte of the value of *hCNum3* as a hexadecimal number:

```
>view hCNum3,hx
```

To view the high byte of the value of hCNum1 as an octal number:

```
>view hCNum1,ho
```

To view the value 2, shifted left the number of bits indicated by *iCNum1+10*, as a long integer:

```
>view 2<<(iCNum1+10),ld
```

To view the value of *iCNum1* modulo 3:

```
>view iCNum1%3
```

To view the product of the sizes of *union2* and *union4,* multiplied by 2:

```
>view sizeof(union2)*sizeof(union4)*2
```

To view the value of *iCNum2*, if the value of *iCNum1* is greater than 4, otherwise, view the value of *iCNum3:*

```
>view (iCNum1>4)?iCNum2:iCNum3
```

To view the value of the Stack Pointer:

```
>view %sp
```

4.41 WATCH — SELECT VARIABLES FOR AUTO DISPLAY

Manages a list of expressions to be displayed in the Watch Window (Section 3.2). The Variable Window is updated whenever there is a change of state of the debugger, or the target environment, such as the occurrence of a breakpoint.

watch *expression*[,*print_specifier*]

adds an entry to the watch list.

expression	Refer to Appendix A.
print_specifier	See the **VIEW** command (Section 4.40).

watch {**-r** | **-d** | **-e**} %*id* | *

watch lists the current entries on the **watch** list.

Mouse To look at the Variables Window, open ***SHOW*** and select ***VARIABLES***.

Examples To add *namebuf* to the display list. The watch window is updated with its value:

```
>watch namebuf
```

To display the variable *namebuf* from the watch list:

```
>watch -d namebuf
    [1] - Disabled
```

To remove the variable *namebuf* from the watch list:

```
>watch -r namebuf
```

To disable display of variables on the watch list:

```
>watch -d *
```

To add *iCNum1* to the display list, and display its value in hexadecimal format:

```
>watch iCNum1, x
```

4.42 WHERE — DISPLAY CURRENT CONTEXT

where [-c | -v] [*func_symbol***[***@symbol***]]**
shows the program context at any point.

where [-v] displays the current source line and function with arguments. If **-v** is specified, it also displays the total variables and their values.

where -c displays current function call stack history with arguments

where -cv [*func_symbol***[***@symbol***]]**
with no arguments specified, displays the current stack history with the local variables for each function.
If only *func_symbol* is specified, **where** displays the current stack history with local variables for only the specified function.
If *func_symbol* and *symbol* are specified, **where** displays the current stack history and the value of the local variable, *symbol*, defined in function *func_symbol*.
The format is based on current radix setting (Section 4.23).

Mouse Open ***SHOW*** and select ***STACK*** to bring a window containing the function call stack. Double click on any of the output lines to bring the source for this line into the source window. Correspondingly, the local variables window is updated if the local variable window is already open.

Examples To display the current function with arguments:

```
>where -c
```

To display the current stack history with local variables for the called functions:

```
>where -v
```

To display the stack history and the local variables of function *getnum*:

```
>where -cv getnum
```

4.43 ! — SINGLE LINE COMMAND TO MONITOR

! *command to target*
The string that follows ! (the bang) character is passed to the monitor. The string should be a valid monitor command. It is not validated by the debugger. The response is echoed on the output window. This command is useful in executing functions not directly supported by the Debugger.

Caution The string is passed as it is; there is no attempt to substitute any of the symbols specified in this string. If, while debugging, you issue monitor commands that alter the state of the monitor, the debugger may enter an undefined state. The debugger does not process the output from these commands.

Chapter 5
INSTRUCTION-LEVEL SIMULATOR COMMANDS

5.1 INTRODUCTION

The PSS helps you to simulate the behavior of the specific peripherals on your board. You can thus simulate your application's hardware environment, and investigate its implications on program execution.

This chapter describes, in detail, all the Instruction-Level Simulator commands. These commands all have the prefix SIM.

Section 5.2 describes how to configure the simulator. Sections 5.3 and 5.4 detail the peripheral simulator commands.

For a detailed explanation of the PSS, see Chapter 2.

5.2 CONFIGURATION COMMANDS

CONFIG - Configure Simulator for Operation

sim config *param[= value]*

This command configures the simulator. If *value* is not specified, the current value is displayed.

param is one of:

clock[= *frequency* **KHz** | **MHz]**

sets the simulation clock frequency. *frequency* is an integer.

perf = *string*

string is a set of performance simulator parameters of the form:
-sa *split_address*
-fh *waits*
-fl *waits*
-lh *waits*
-ll *waits*
-sh *waits*
-sl *waits*
Where: fh is fetch high, fl is fetch low, lh is load high, ll is load low, sh is store high and sl is store low.
wait is the number of wait states.

refresh[= *interval***]**

sets the interval between updates of the Status Window. *interval* can be specified either as *integer_i*, indicating the number of instructions, or *integer_t*, indicating the number of clock ticks (where a tick represents one clock period).

Examples

To set the frequency of the simulation clock to 25 MHz:

```
> sim config clock=25 MHz
```

To display the current configuration:

```
> sim config clock
    25000
```

To split memory at address 0x9000, and set the fetch wait states on all addresses higher than 0x9000 to 2:

```
> sim config perf -sa 0x9000 -fh 2
```

5.3 PSS INTERNAL COMMANDS

OUTPUT - Map Output File to Address

output *qual_address path* **[-a] [-n]**

Records writes to address *qual_address*, into the file designated by *path*.

-a, appends new data to an existing file; otherwise, if the specified file exists, it is first erased.

-n, intercepts writes to memory, where the write is necessary to invoke an action, not to change the contents of the memory address. The write to the address activates all the PSS activity, but does not modify the memory.

If more than one **output** command is activated at a given address, and if one of the **output** commands uses the **-n** flag, memory is not modified.

The format of the output file is:

set *address value time*

Example To direct the writes to address 0x8000 to /pss/file:

```
>output 0x8000 /pss/file
```

READFILE - Read Input File

readfile *full_path* **[-c]**

reads an input file specified by *full_path*. The specified file must contain internal PSS commands. If multiple files are specified, they are all read prior to the start of simulation.

If the initial **set** command of the new file is a relative command, i.e., if it contains a relative delay time, the time of execution of the last set command is used as baseline time.

If no **se**t command has been previously executed, the baseline time is the beginning of the simulation.

-c, executes the **set** commands in the file relative to current time.

Example To execute the commands in the PSS input file **pss.int** from the current PSS input file, and execute the set commands in it relative to the current time:

```
>readfile pss.int -c
```

SET - Set address

Sets a value to a specified memory address at particular times, or on the occurrence of certain conditions.

set *address1 value1 time1* **[<condition>]**
OR
set *interrupt vector time1* **[<condition>]**

The directive may take one of three forms.

SYNTAX1: set *address1 value1 time1* **[every** *time2* **[***times1***]]**

This directive sets the value *value1* to the address or address-range *address1*, after a delay of *time1*. The delay may either be absolute or relative. If **[every** *time2* **[***times***]]** is included, *value1* is set repeatedly in periodic intervals of size *time2*.
If *times1* is specified, the value is only set *times1* times

SYNTAX2: **set** *address1 value1 time1* onread *address2* [*value2*]

value1 is set to *address1* after a delay of *time1* from the time that *address2* is read.
If *value2* is specified, *value1* is set to *address1* only if *value2* was read from *address2*.

SYNTAX3: **set** *address1 value1 time1* onwrite *address2* [*value2*]

value1 is set to *address1* after a delay of *time1* from the time that *address2* is writen to.
If *value2* is specified, *value1* is set to *address1* only if *value2* was writen to *address2*.

*address***[**:*qualifier***][**:*range***]**

address A positive integer representing an address in the simulated address space. It may be in decimal, octal (begins with 0), or hexadecimal (begins with 0x).

qualifier is the type of item addressed by *address*.

b Byte (8-bit byte, default)

w Word (16-bit, address must be word-aligned, i.e., 0, 2, 4, 6, ...)

d Double (32-bit, Address must be double-word aligned, i.e., 0, 4, 8, 12, ...)

n [*bit_position***]**

bit_position is in the range 0-7 (default is zero). When a bit position is specified, *range* cannot be specified.

range An integer (decimal, octal, or hexadecimal) representing the length of the memory range.

value An integer (decimal, octal, or hexadecimal)

A *path* of a data-file containing integers (decimal, octal, or hexadecimal).

time is defined as *time*[*qualifier*]

time is a positive decimal integer.

qualifier is one of:

r Relative to the previous set statement, if available, or relative to current time if no previous set statement is available, or when reading a PSS file with the **-c** flag. This flag is ignored for *time2*.

a Absolute time from the beginning of the simulation or from the last ***reset*** or ***sim perf on*** command. This flag is ignored for *time2*.

and/or one of

t Simulation clock ticks. Length of one tick is determined by the frequency of the simulation clock.

n Nanoseconds

u Microseconds

m Milliseconds

s Seconds

interrupt is one of:

int (maskable interrupt)
Where *vector* is the interrupt vector number. It is read from the value field or from a file.

nmi (non-maskable interrupt)

For an explanation of how to simulate interrupts, see Section 2.3.3.

Caution

Execution of a **set** command may take place any time between 0 and 100 ticks following its specified execution time. To ensure that one **set** command precedes another in execution, the two commands must have delays which differ by at least 100 ticks.

Examples

To read a total of ten values of data from a data file into *perip_read* every 5 ms after an initial wait of 1 ms from the beginning of the simulation:

```
> set perip_read from_data_file 1m every 5m 10
```

To set *perip_flags* to reset, when your program reads *perip_read*:

```
> set perip_flags reset 0 onread perip_read
```

To write 50 double words (taken from the data file) to memory, starting at address 0xFFFF0000, one ms after you start the DMA:

```
> set 0xFFFF0000:d:50 data_file 1m onwrite DMA_start start_transfers
```

To generate a square wave with a period of 2 ms:

```
> add1 = 0xFFFE0000:w
> set add1 0 0m every 2m
> set add1 1 1m every 2m
```

Creating Symbols

symbol [= value]

Creates a symbol with the value specified in the command. Symbols must be unique, and must be defined prior to their usage.

Symbols can replace any entity except: **set**, **output**, **readfile**, **onread**, **onwrite**, **int**, **nmi**, and **every**. Symbols cannot start with a digit.

5.4 SIM PSS COMMANDS (FROM DEBUGGER COMMAND LINE)

PSS - Report State of PSS System

sim pss reports the state (on or off) of the Peripheral Stimulus System.

PSS CLEAR - Clear PSS Data and Status

sim pss clear clears the PSS data structure, sets PSS to off, recycles all the dynamically allocated memory, and closes all open files (including the verbose file, if it was open).

clear also sets the time of the "last loaded **set** command" to current simulation time (see **sim pss load**).

If you use **PSS** to restart a test, you must issue **reset**, **sim perf on**, **sim pss clear**, and **sim pss on** commands. You must then reload any PSS files which you want to use.

PSS FILES - Display Names of PSS Files

sim pss files

displays the names of the files opened by the Peripheral Stimulus System. The number of references to the file is displayed following the pathname.

PSS LOAD - Load PSS File Into PSS

sim pss load [*path* **[-c]]**

If *path* is specified, PSS loads the specified file into the system. If no *path* is specified, PSS displays a list of the files currently loaded, followed by the *file_id*. **load** can be executed at any time during execution of your program.

A relative **set** command is one which contains a relative delay time. All commands designated relative, within the file being loaded, are executed relative to the baseline time of the load. If the first **set** command in the file being loaded is a relative command, the execution time of the last previous **set** command is used as baseline time. If the first **set** is not relative, the baseline time is time 0 for the simulation.

-c, sets baseline time to the current simulation time.

A file can be "unloaded." To remove files from the system, you can also execute a **clear** command, which removes all of the files, and then perform **load**s of all the desired files.

Caution In execution of **set** commands at load time, the behavior is specified by the **-e** flag used in the previous **on** command.

PSS OFF - Turn Off Execution of Internal PSS Commands

sim pss off turns off the execution of internal PSS commands. If you are not using PSS during the debugging of a portion of your program, you can turn off PSS to improve the speed of execution of your program, and later turn PSS on again.

PSS ON - Turn On Execution of Internal PSS Commands

sim pss on [-e] [-v [*path***]]**

turns on the execution of internal PSS commands.

By default, all of the loaded **set** commands, with execution times less than the simulation time at which the **on** command is executed, are not executed.

-e flag executes all **set** commands with a time earlier than the current time. **set-every** commands are set to a new full interval and executed, and then the timing of the intervals is restarted.

The **-e** flag is "sticky," i.e., **load** commands executed after the **on** command behave according to the specification or non-specification of **-e** on the last previous **on** command.

PSS OUTPUT - Assign PSS Output File to Address

sim pss output *address* = *path* **[-a] [-n]**

assigns the file specified by *path* to *address*, and opens the file if necessary. All write operations to *address* are recorded in that file. Once opened, the file remains open until explicitly closed (by using **sim pss output** *address* =) or until a **clear** command is issued.

One address can be mapped to many files, and many addresses can be mapped to a single file.

-a appends new data to an existing file. If **-a** is not specified, and the file exists, it is erased before the first write. The file mode, append or erase, is determined by the file-open operation.

-n, intercepts writes to memory. This mode is needed for emulation of devices like UARTs, where a write to a particular memory location causes an action, but is not required to modify the contents of the memory location. A write to the address activates all the PSS activity but does not modify the contents of the memory. If more than one of the **output** commands is activated for a particular address, and one of the **output** commands uses the **-n** flag, the address is not modified.

sim pss output *address* =

removes all mappings of the address to output files, closes the files, and terminates the recording of writes to the address.

sim pss output *address*

displays a list of the names of files mapped to the specified address.

PSS UNLOAD - Unload a Previously Loaded PSS File

SIM PSS UNLOAD *id*

unloads the PSS file specified by *id*. A PSS file (or group of PSS files) can be unloaded at any time during the debugging of your program. To get a list of valid *id*'s, use the **LOAD** command without arguments.

Appendix A
QUICK REFERENCE GUIDE

A.1 CRDB COMMAND SYNTAX

Command	Definition	Syntax	Section
alias	Define macro	<name> = command	4.2
	Define substitution macro	<name> = "command $$, $$"	
	List macro	<*name*>	
	Remove macro	-r {<*name*> | * }	
autocommand	Adds entry	<*command*>	4.3
	Removes, disables, enables entry	{ -r | -d | -e }%<*id*> | *	
	List autocommands		
break	Adds entry	[-t] <*brkaddr_list*>[,c=<*RLexp*>] [,o=<occ_cnt>][,q=<qualifier>][,s=<size>]	4.4
	Removes, disables, enables entry	{ -r | -d | -e }%<*id*> | *	
call	Call function	<*func_name*>(arg1,arg2,..,argn)	4.5
cd	Change working directory	[<*path*>]	4.6
chip	Select chip	<*chip_name*>	4.7
comm	Set baud rate (Note: only 19200 is supported)	-b {4800 | 9600 | 19200 | 38400 | 57600 | 115200}	4.8
	Set comm port	-l {1 | 2 | 3 | 4}	
	Set Ethernet communication parameter	-e <*hostname*> | <*IP address*>	
debug	Select file	[-x| -n| -xn] <*file_name*>	4.9
debugmode	Select debugging modet	[- e | -d] {*startup* | exitcode }	4.10
find	Find value	[-a | -b | -c | -w | -f | -p | -l | -i] <*value*>, <*addr_range*>	4.11
findsrc	Find string	[-f | -b | -n] [<*string*>] [,<*file_name*>]	4.12
go	Go	[-c] [<*from_addr*>][//<*end_addr*>]	4.13
info	Debug info		4.14
input	Execute input file	<file_name>	4.15

Command	Definition	Syntax	Section
`list`	List memory	-m [h| o| d] [b | c | w | f | p | l | i] <*addr_range*>	4.16
	List source	<*qualified_lineno*>	
`log`	Create or append to file	[-a]<*file_name*>	4.17
	Disable or enable log	[-d |-e]	
	Log input only or output also	[-i | -o]	
	Expand aliases or not expand aliasest	[-f | -u]	
	Display logging status	-s	
`modify`	Modify memory	[- b |-c | -w |-f |-p |-l] <address> | <*addr_range*>[,<*value*>[,value]]	4.18
	Modify string	-a <*string_pointer*>,<*string*>	
	Modify register	%<reg_name>,<value>	
`next`	Continuous execution till end of program	-c	4.19
	Stepover <number> of source lines	-n <*number*>	
	Stepover source lines in the address range	[<*from_addr*> [//<*end_addr*>]]	
`nextins`	Continuous execution till end of program	-c	4.20
	Stepover <number> of assembly instructions	-n <*number*>	
	stepover assembly instructions in the address range	[<*from_addr*> [//<*end_addr*>]]	
`pause`	Pause for user input during command file processing		4.21
`quit`	Quit the debugger		4.22
`radix`	Set radix	[8 | 10 | 16]	4.23
`reset`	Reset the target		4.24
`resume`	Resume execution from command file after pause		4.25
`saveconfig`	Save configuration	[<*file_name*>]	4.26
`savestate`	Save state	[<*file_name*>]	4.27

Command	Definition	Syntax	Section
set	Set a variable	<*name*> = <*string*>	4.28
	Undefine a name	-r <*name*> | *	
	Display a variable assignment	<*name*>	
setstate	Set state	<*file_name*>	4.29
softbreak	Adds breakpoint	[-t] <*softbreak_list*> [,<c=<*RLexp*>] [,o=<*occ_cnt*>]	4.30
	Removes, disables, enables	{-r | -d | -e }%<*id*> | *	
srcmode	Set source mode	[-s | -m]	4.31
srcpath	Set source path	<*pathname_list*>	4.32
	Remove source path	-r {<*path*> | *}	
stdio	Map stdin to a file	-i <filename	4.33
	Map stdout to a file	-o[a] <file name>	
	Map stderr to a file	-e[a] <file name>	
	Map stdout, stderr to a file	-oe[a] <file name>	
	Map stdin, stdout, stderr to output Window	-[ieo]w	
	Map stdin, stdout, stderr to debugger output Window	-[ieo]r	
	List current mappings		
step	Countinuous execution till end of program	-c	4.34
	Step <number> of source lines	-n <*number*>	
	Step source lines in the address range	[<*from_addr*> [//<*end_addr*>]]	
stepins	Continuous execution till end of program	-c	4.35
	Step <number> of source instructions	-n <*number*>	
	Step source instructions in the address range	[<*from_addr*> [//<*end_addr*>]]	

Command	Definition	Syntax	Section
symbol	Display symbol info	{* | <*pattern*>*}	4.36
	Display watch symbols	-l {* | <*pattern*>*}	
	Display symbol tag	-t{<*datatype*>/<*tagname*>|<*symbolname*> | *}	
	Display module symbols	-f <*qualified_modulename*>	
	Display global symbol	-g <*pattern*>*	
sync	Synchronize source window		4.37
target	Set target	edb|sim|	4.38
verbose	Display Monitor traffic		4.39
view	View data	<*expression* [,<*print_specifier*>]	4.40
	View register	%<reg_name> [,<*print_specifier*>]	
watch	Adds entry	<*expression*> [,<*print_specifier*>]	4.41
	Removes, disables, enables entry	{ -r | -d | -e } %<*id*>| *	
where	Locate self	[-v] -c [-v [<*func_symbol*>[@<*symbol*>]]	4.42

A.2 SIMULATOR COMMAND SYNTAX

Command	Definition	Syntax
`output`	Map address to output file	<*qual_address*> <*full_path*> [-a] [-n]
`readfile`	Read input file	<*full_path*> [-c]
`set`	Set address to value	<*qual_address*> <*set_value*> <*qual_time*> [every <*qual_time*> [<*times*>]]
	Set address to value on read	<*qual_address*> <*set_value*> <*qual_time*> onread <*qual_address*> [<set_value>]
	Set address to value on write	<*qual_address*> <set_value> <*qual_time*> onwrite <*qual_address*> [<set_value>]
`sim config`	Configure simulator	clock = <*frequency*> perf = <*parameters*> refresh = <*refresh_frequency*>
`sim pss`	Report state of PSS	
	Clear PSS data and status	CLEAR
	Display names of PSS files	FILES
	Load input file	LOAD <*full_path*> [-c]
	Turn off PSS	OFF
	Turn on PSS	ON
	Map address to output file	OUTPUT <*address*> = <*full_path*> [-a] [-n]
	Unmap address	OUTPUT <*address*> =
	Display mapping of address	OUTPUT <*address*>
	Unload PSS input file	UNLOAD <*file_id*>
`sim perf`	Turn on performance estimation	ON
	Turn off performance estimation	OFF
	Set range for performance estimation	RANGE <*address*> <*address*>
`sim trace`	Turn on unconditional trace	ON
	Turn off unconditional trace	OFF

A.3 ARGUMENTS

Read the string, ":=", as "is defined as". "null" represents the empty set. .

<addr_range>	:=	*<address>*||*<address>*
<address>	:=	*<numeric_expression>* | *<symbolic_address>* | *<symbolic_expr>*
<brkaddr>	:=	*<addr_range>* | *<cur_pc>* | *<address>* | *<line_for_cur_pc>* | *<cur_display_line>*
<brkaddr_list>	:=	*<brkaddr><brkaddr_list>* | null
<chipname>	:=	refer to official chip name list
<command>	:=	any debugger command
<const>	:=	*<integer>*
<cur_display_line>	:=	#
<cur_pc>	:=	.
<datatype>	:=	any legal C data type
<emulator_name>	:=	EDB |ISE
<end_addr>	:=	any valid code address
<expression>	:=	any C symbolic expression
<frequency>	=	*<decimal_integer>* [Mhz | khz]
<file_id>	=	file identifier given by debug command
<file_name>	:=	file name without the path
<fixed_symbol>	:=	*<global_symbol>* | *<static_symbol>*
<from_addr>	:=	any valid code address
<full_path>	=	[*<path>*]*<file_name>*
<func_line>	:=	[*file_name*]*<func_symbol>*
<func_symbol>	:=	symbolic name of a defined function
<global_symbol>	:=	any valid C global symbol
<id>	:=	<integer>
<length>	=	*<integer>*
<line_for_cur_pc>	:=	& (representing the source line to which the current PC address maps)
<const_lineno>	:=	#*<const>*
<lineno>	:=	*<const_lineno>* | <cur_display_line> | *<line_for_cur_pc>* |*<qualified_lineno>*
<local_addr>	:=	$b | $c | $e | $r | $x
<name>	:=	a sequence of up to eight alphanumeric characters, of which the first is alphabetic
<number>	:=	*<decimal_num>* | *<hex_num>* | *<octal_num>*
<numeric_expression>	:=	any legal C expression consisting of *<number>*s and *<operator>*s

<occ_cnt>	:=	*<integer>*
<operator>	:=	+ | - | * | /
<opr>	:=	any C relational or logical operator
<path>	:=	any legal pathname
<pathname_list>	:=	*<path>* | *<pathname_list>*,*<path>*
<print_specifier>		One of the following:
		d signed decimal
		i signed decimal
		o unsigned octal
		x unsigned hexadecimal in lower case “a,b,c,d,e,f”
		X unsigned hexadecimal in upper case “A,B,C,D,E,F”
		u unsigned decimal integer
		e floating-point in engineering notation
		E floating-point in engineering notation
		f floating-point
		g double signed value printed in ± format
		G double signed value printed in ± format
		c single character
		s character string
		hd short signed decimal integer
		hi short signed decimal integer
		ho short unsigned octal integer
		hX short unsigned hexadecimal integer in uppercase “ABC-DEF”
		hx short unsigned hexadecimal integer in lower case “abc-def”
		hu short unsigned decimal integer
		ld long signed decimal integer
		li long signed decimal integer
		lo long unsigned octal integer
		lX long unsigned hexadecimal integer in upper case “ABC-DEF”
		lx long unsigned hexadecimal integer in lower case “abcdef”
		lu long unsigned decimal integer
		Le long double in engineering notation (lowercase ‘e‘ for exponent)
		LE long double in engineering notation (uppercase ‘E’ for exponent)
		Lf long double floating-point
		Lg long double signed value printed in f or ‘e’ format, whichever is more compact for the given value and precision. The ‘e’ format is used only when the exponent of the value is less than -4 or greater than the specified precision. Trailing zeros are truncated and the decimal point appears only if one or more digits follow it.
		LG long double signed value, identical to the ‘g’ format, except that ‘G’ introduces the exponent (where appropriate) instead of ‘e’.
<qual_address> in simulator	=	*<address>*[:[b | w | d | n]][:*<range_length>*] | int | nmi

<qualified_code_address>	:=	any valid *<address>* or *<symbolic_address>* lying in the code range
<qualified_lineno>	:=	**[***<file_name>***][***@*<func_symbol>***]** **[***<const_lineno>***]**
<qualified_modulename>	:=	*<file_name>*
<qual_time>	=	*<positive_decimal_integer>* **[**r	a**][**t	m	u	n	s**]**
<pattern>	:=	*<symbol>*	*<symbol_qualifier>*
<range_length>	=	*<integer>*
<refresh_frequency>	=	*<integer>***[**i	t**]**
<reg_name>	=	any valid register name for target chip
<RLexp>	:=	any legal C expression consisting of *<const>*, *<symbol>*, and *<opr>*
<set_value>	=	*<integer>*	*<full_path>*
<simple_break>	:=	*<cur_pc>*	*<address>*	*<line_for_cur_pc>*	*<cur_display_line>*
<softbreak_list>	=	*<qualified code address>***[**,*<softbreak_list>***]**	null
<static_symbol>	:=	any valid C static symbol
<string>	:=	a C language string
<string_pointer>	:=	the address of a string or a *<value>* defined as char*.
<symbol>	:=	any valid (defined) C symbol
<symbol_qualifier>	:=	any sequence of characters which can begin a valid symbol
<symbolic_address>	:=	*<fixed_symbol>*	*<func_line> <local_addr>*	*<mod_line><const_lineno>*	*<lineno>*
<symbolic_expr>	:=	Any legal C expression consisting of *<const>*s, *<symbol>*s, *<symbolic_address>*es, and *<operator>*s
<tagname>	:=	tagname of structures and unions
<value>	:=	a legal value of the type implied by other arguments of the command
<value_list>	:=	*<value>*	*<value>*, *<value_list>*

$b	the address of the absolute beginning of a function (the prologue)
$c	the address of the first instruction in the body of the function (after the prologue)
$e	the address of the last instruction in the body of the function (before the epilogue)
$x	the address of the RETURN instruction at the end of the function

Appendix B
TROUBLE-SHOOTING HINTS

B.1 TARGET ERROR MESSAGES

Communication errors are displayed in the following form:

Target Error/Warning: [error message]

where **error message** is one of the following:

- **Emulator Time-out: No response from Emulator**
- **Emulator Line Too Long**
  The emulator's inputs are too large (max. is 0x500)
- **Emulator Output Unexpected**
- **Comm Port Open Failure**
  The debugger failed to open connection with the DEX through serial port or ETHERNET connection.
- **Comm Port Close Failure**
  The debugger failed to close connection with the DEX through serial port or ETHERNET connection.
- **Comm Port Read Failure**
  The debugger cannot read the emulator's inputs.
- **Comm Port Send Failure**
  The debugger is unable to send a message to the emulator.
- **Chip is Running**
  You have tried to send a request to the emulator while the chip is running.
- **Only 1 hard breakpoint allowed**
  This is really a limitation of your development board.
- **Invalid range**
  The address range is invalid.
- **Handshake error**
  The debugger failed to establish communication with the DEX through serial port.
- **Invalid chip specified**
  The name of the chip you specified in the **chip** command is incorrect.
- **Address out of range**
  The address is greater than the maximum address value.

- **Chip is not Running**
- **Bad MON command**
  The command sent to the emulator is invalid.
- **Virtual I/O transmission**
  The debugger gets empty line from the emulator through virtual I/O transmission.
- **Search length too large**
  The search length exceeds the maximum (max is 0x500)
- **Invalid emulator specified**
  The name of the emulator specified in the target command is invalid.
- **Illegal emulator, chip combination**
  The chip name specified in the chip command is unsupported by the current emulator
- **Emulator type cannot be verified**
  You may be attempting to use a development board that is not connected to the system.
- **Bad occurrence count**
  Incorrect value of occurrence count for the soft break or hard break.
- **This command is unavailable now**
  The debugger and the DEX do not support a command. Report the problem to National Semiconductor.
- **Unknown EIM error**
  The debugger and the DEX are unable to communicate. Try to power up, and restart again.
- **BUS error; please re-download file**
  The serial connection is lost, due to the DEX power-off or reset, try to re-download file, or re-establish the communication using the **comm** command.

B.2 SERIAL LINKAGE PROBLEMS

The most common problem with a serial-link connection is a mismatch between the configuration of the RS-232 port on the host side, and that on the target board side.

The target board is configured as a DCE. Therefore, if your host port is configured as a DTE, you should be able to use a standard RS-232 cable for the connection. If your host is configured as a DCE, you need a null modem cable, in addition to a standard RS-232 cable, for the connection. If you are not sure of your host configuration, try to connect both with and without the null modem cable. Set the VERBOSE mode of communication with your monitor. If you get communication, but it looks like garbage, you may have the board and host set to mismatching baud rates.

B.3 ETHERNET PROBLEMS

- **`Can not find usable winsock.dll`**
  The debugger is unable to locate **`winsock.dll`** in the path. The debugger requires a **`winsock.dll`** Ver1.1 API, or higher, library to communicate with the DEX via ETHERNET.
  You should modify the PATH to include a directory containing such a library.

- **`Broken pipe; please re-download file`**
  The ETHERNET connection is lost, due to the DEX power-off or reset. Try to re-download the file, or re-establish communication using the **`comm`** command.

Appendix C
PERIPHERAL SIMULATION SYSTEM - EXAMPLE

This appendix describes an example program which makes use of both the Peripheral Stimulus System, and the virtual I/O capability of the CompactRISC, system to perform I/O functions. The example models a UART-like device, using the PSS to provide a character-string input to the program, and virtual I/O to echo the string to the screen.

To try out the example on your own, first build a COFF file using the command:

```
crcc -g -o example.cof example.c -Wl,"-d" -Wl,"linker.def"
```

```
crdb -e=example.inp
```

As is, the example echoes the string, **`Hello!,`** to the debugger's Output Window.

You may examine the file *test.doc* to determine the actions taken by PSS, and the file *example.log* to see the outputs of the program. For a detailed description of the syntax for each command used in *example.pss* and *example.inp*, refer to Chapter 5.

C.1 SOURCE FILES

C.1.1 example.c

```
#include     "example.h"
#include     <stdio.h>

void main() {
        char   buf;         /* terminal output buffer */

        MReset = 0;         /* reset UART */

        GMode = 0;          /* global mode register */
                            /* one stop bit */
        BRGL = 20;          /* baud rate = 9600 */
        BRGH = 0;

/* initialize receiver */
        RMode = 0xFA;       /* 8-bit, even parity, auto-enable
                               DCD', internal clock */

/* initialize transmitter */
        TMode = 0xF8;       /* 8-bit, even parity, auto-enable
                               CTS', internal clock */
```

```
/* read a character */
rd:
        while ((ModStat & DCD)!=0) ;     /* if DCD' low, */
        Cmnd |= RcvEn;                   /* enable receiver */
rd1:
        while ((RTStat & RxRDY)==0) ;    /* when RxRDY set, */
read:
        while ((buf = RxH)==0) ;         /* get character */
        if (buf == CR)                   /* if CR, */
            goto exit;                   /* quit */

/* transmit a character */
xmt1:
        write(1, &buf, 1);  /* output character to screen *
xmt:                        /* use PSS to simulate peripheral
                               output */
        while ((ModStat & CTS)!=0) ;   /* if CTS' low, */
        TxH = buf;          /* transmit character */
        Cmnd = TxEn;        /* enable transmitter */
        goto rd;          /* continue */
exit:
        while ((ModStat & CTS)!=0) ;   /* if CTS' low, */
        buf = CR;           /* transmit CR character */
        write(1, &buf, 1);  /* output character to screen */
        Cmnd |= TxEn;       /* enable transmitter *
exit1:
        while ((ModStat & CTS)!=0) ;   /* if CTS' low, */
        buf = LF;           /* transmit LF character */
        write(1, &buf, 1);  /* output character to screen */
        Cmnd |= TxEn;       /* enable transmitter */
```

C.1.2 example.h

```
/*                                          */
/*    UART Register Addresses                */
/*                                          */

extern char uart_base[];

#define RxH     uart_base[0]    /* receiver holding register */
#define TxH     uart_base[0]    /* transmitter holding register
*/
#define RMode   uart_base[1]    /* receiver mode */
#define TMode   uart_base[2]    /* transmitter mode */
#define GMode   uart_base[3]    /* global mode */
#define Cmnd    uart_base[4]    /* command */
#define BRGL    uart_base[5]    /* baud rate generator divisor
                                   latch(low) */
#define BRGH    uart_base[6]    /* baud rate generator divisor
                                   latch(high) */
#define RTSMask uart_base[7]    /* r-t status mask */
#define RTStat  uart_base[8]    /* r-t status */
#define ModMask uart_base[9]    /* modem status mask */
#define ModStat uart_base[10]   /* modem status */
#define PwrDn   uart_base[11]   /* power down */
#define MReset  uart_base[12]   /* master reset */
```

```
/* Modem Status Register bits */
#define CTS         0x10        /* Clear to Send */
#define DCD         0x20
#define DSR         0x40        /* Data Set Ready */

/* Command Register bits */
#define RcvEn       0x1         /* receiver enable */
#define TxEn        0x2         /* transmitter enable */

/* R-T Status Register bits */
#define RxRDY       0x1         /* receiver data ready */
#define TxBE        0x2         /* transmitter buffer empty */

#define CR          0x0D
#define LF          0x0A
```

C.1.3 linker.def

Please copy **linker.def** from the CR Tools installation directory and add following statement as the last line in this file

```
_uart_base = 0xe000;
```

This line binds the **uart_base** address to 0xE000. This address is used in **example.inp** and **example.pss**.

C.2 COMMAND FILE (EXAMPLE.INP)

```
log example.log
emu sim
target sim
debug example.cof
stdio -ow
sim perf on
sim pss output 0xE001:0x10 = result.doc -a
sim pss load example.pss
sim pss on -v test.doc
pause
go
sim pss clear
quit
```

C.3 PSS INPUT FILE (EXAMPLE.PSS)

```
#
#     UART Register Addresses
#

uart_base = 0xE000
```

```
RxH = 0xE000            # receiver holding register
TxH = 0xE000            # transmitter holding register
Cmnd = 0xE004           # command
Cmnd0 = 0xE004:n0       # command
Cmnd1 = 0xE004:n1       # command
RTSMask = 0xE007        # r-t status mask
RTStat = 0xE008         # r-t status
RTStat0 = 0xE008:n0     # r-t status
RTStat1 = 0xE008:n1     # r-t status
ModStat0 = 0xE00A:n0    # modem status
ModStat1 = 0xE00A:n1    # modem status
ModStat2 = 0xE00A:n2    # modem status
ModStat3 = 0xE00A:n3    # modem status
ModStat4 = 0xE00A:n4    # modem status
ModStat7 = 0xE00A:n7    # modem status
MReset = 0xE00C         # master reset

# Modem Status Register bits
CTS = 4      # Clear to Send
DCD = 5
DSR = 6      # Data Set Ready

# Command Register bits
RcvEn = 0    # receiver enable
TxEn = 1     # transmitter enable

# R-T Status Register bits
TxBE = 1     # transmitter buffer empty

CR = 0x0D
LF = 0x0A
period = 1042u
delay = 200ua

# Reset state for most registers
on_reset = 0

from_uart_data = uart.inp
output TxH result.doc -n

# Chip reset
set Cmnd on_reset 0 onwrite MReset
set RTStat TxBE 0 onwrite MReset
set RTSMask on_reset 0 onwrite MReset
set ModStat0 on_reset 0 onwrite MReset
set ModStat1 on_reset 0 onwrite MReset
set ModStat2 on_reset 0 onwrite MReset
set ModStat3 on_reset 0 onwrite MReset
set ModStat4 on_reset 0 onwrite MReset
set ModStat7 on_reset 0 onwrite MReset



# baud rate - 9600 baud
set RxH from_uart_data delay every period 14
# set RxRDY
# set RTStat0 1 delay every period 14
set RTStat0 1 delay every period 14
# set RTStat0 0 0 onread RxH
```

```
set RTStat0 0 0 onread RxH
# clear receiver enable
set Cmnd0 0 0 onread RxH


# set TxBE
set RTStat1 0 0 onwrite TxH
set RTStat1 TxBE 5tr
# clear transmitter enable
set Cmnd1 0 0 onwrite TxH
# set CTS
set ModStat4 1 5tr
set ModStat4 0 255tr
```

C.4 PSS DATA FILE (UART.INP)

```
0x48        # ‘H’
0x65        # ‘e’
0x6C        # ‘l’
0x6C        # ‘l’
0x6F        # ‘o’
0x21        # ‘!’
0x0D        # CR
0x0A        # LF
```

Appendix D
MONITOR INTERFACE

D.1 INTRODUCTION

You can send commands directly to the DEX, with no other processing, from the debugger (Section 2.2.14) by typing a command on the command line preceded by an exclamation point (**!**). (See the Caution in Section 4.43.)
Any subsequent input needed to complete the DEX command, or any new commands, must also be prefixed with **!**.

The debugger communicates with the DEX via ETHERNET or a serial connection.

Conventions

The following conventions are used in the monitor commands definitions:

size For register is any size of [1/2/3/4] in bytes. For address is any positive size (e.g. 1/2/3/4/5..n) in bytes. Default *size* is 1 byte.

type The type of operand. /s for signed, /u for unsigned. Default *type* is unsigned.

reg Any legal register name. (with *size*)

addr Any valid unsigned absolute address. *addr-range* - two absolute *addr*'s low and high, defining an address range.

const Any constant integer value.

var One of: (with *size* and with *type*)
- *addr* whose content is used as value.
- *reg* whose content is used as value.
- *addr* or *reg* whose content + offset is used as (pointer) to an *addr* whose content is used as value.
- any *const*.

op One of: +, -, *, /, %, |, &, !, ^, ~, <<, >>

eq One of: ==, <>, <, >, <=, >=
- unary: changed, in-range *const*

boolean
One of: ||, &&, !

prd A regular expression in post fix notation with up to 16 vars and the operators: *eq*, *op*, *boolean* terminated with a "," ";" or NL.

Command Table

The following table summarizes the commands used to interface with the DEX:

Command Syntax	Function	Page
Configuration Commands		
CFG *options*	Configures the DEX for the target ADB.	D-5
R [$*number***]** *reg/size...*	Defines the target registers.	D-5
B *bit-reg bit-name/mask...*	Defines the names and masks of bit registers.	D-6
M [$*number*] *name* [$*parts_n*] */addr/size,....name* [$*parts_n***] */addr/size***	Defines special memory spaces.	D-6
H [$*number*] *breakpoint-definition...*	Defines the numbers and definitions of the target hardware breakpoints.	D-7
ECFG [[/TR:H *cable-type***] [/I** *ip-addr***] [/H** *dex-name***] [/G** *gateway***]]**	Configures the ETHERNET to recognize the DEX	D-7
DSYM *name addr size*	Defines a symbol.	D-9
DEX Reset and Initialization Commands		
VINIT	Implements changes to the configuration database	D-9
VRS	Reset the DEX's state to power-up	D-10
BOOT	Boots the DEX hardware	D-10
Register Manipulation Commands		
DR *register... register-range...*	Dump Registers, specify the registers by name, or by range	D-10
C *regname = val*	Assigns the value ***val*** to the register ***regname***	D-10
SR	Saves an image of the ADB CPU registers in the DEX	D-11
RR	Restores the ADB CPU registers from an image saved in the DEX	D-11
Memory Manipulation Commands		
DM *format elem_size address[/size]...*	Dump memory. Memory is specified by address and size (in bytes) and is dumped in the specified order	D-11
LM *format elem_size address[/size] data*	Load Memory. Memory is specified by address and size in bytes.	D-12
M *src-addr dest-addr n [N]*	Moves ***n*** bytes of data starting at ***src-addr***, to the block of memory starting at ***dest-addr***.	D-12

Command Syntax	Function	Page
F ***start-addr end-addr data [elem-size] [N]***	Fills the block of memory starting at ***start-addr*** and ending at ***end-addr*** with the specified ***data.***	D-13
SR ***start-addr end-addr data [size]***	Searches for the first appearance of ***data*** between ***start-addr*** and ***end-addr***, and prints the address where the data has been found	D-13
Breakpoint Commands		
SBP ST ***id addr [count condition action]***	Creates a software breakpoint.	D-14
SBP TR ***id addr [count condition action]***	Creates a software breakpoint that triggers tracing.	D-14
RBP ***{A | id}***	Remove a software breakpoint.	D-15
HBP ST ***number type addr [mask] [times condition action]***	Creates a hardware breakpoint.	D-14
HBP TR ***number type addr [mask] [times condition action]***	Creates a hardware breakpoint that triggers tracing.	D-14
HRBP ***{A | id}***	Removes hardware breakpoints.	D-15
Execution Commands		
G ***regname***	Starts program execution.	D-16
G	Continues program execution.	D-16
ST ***[n] [v]***	Steps ***n*** assembly instructions.	D-16
ST S ***[v]***	Single steps through the program until it hits a stop breakpoint.	D-16
BAL ***reg addr***	Simulates the BAL (branch and link) assembly instruction.	D-17
JS ***addr***	Simulates JS (jump subroutine) assembly instruction.	D-17
B ***reg addr***	Initializes the PC to ***addr***, but does not transfer control to your program.	D-17
Remote Commands		
RS	Issues a RESET signal to the ADB board.	D-18
ISE	Sends an ISE (non-maskable interrupt) to the ADB.	D-18
Time Commands		
CT	Clears the timer.	D-18
GT	Gets the time.	D-18
Simulation Specific Commands		
SIM CONFIG ***config-params***	Sets simulator parameters: frequency, wait-states.	D-19
SIM PSS [on | off]	Turns pss simulation on and off. Without parameters, returns status.	D-18

Command Syntax	Function	Page
SIM PSS LOAD *filename*	Loads ***filename*** to the peripheral simulator.	D-19
SIM PSS CLEAR	Clears the state of the peripheral simulator.	D-18
Miscellaneous Commands		
I *mask*	Sets the value of the monitor interrupt mask.	D-19
EXT *command*	Adds commands to the monitor.	D-19

D.2 MONITOR COMMANDS

D.2.1 Configuration Commands

There are three types of configuration commands:

CFG - Configures the DEX for a specific chip/board.

ECFG - Configures the ETHERNET communication.

DSYM - Defines DEX internal symbols.

Configuration Command

CFG *options*

The configuration command configures the DEX for a specific chip/board environment. It is issued each time the debugger needs to configure the DEX to work with a new TMON e.g., at the start of a debugging session.

options *is* a list of options, each has the format: ***cfg-type cfg-msg.*** A '**;**' is used to separate two options in the same CFG command**.**

cfg-type is one of: **R** registers, **B** bit-register, **E** encoding, **M** memory, **H** hardware breakpoint.

cfg-msg has a different meaning depending on ***cfg-type***

The configuration options can be specified in any order. The debugger may issue as many CFG commands as needed to configure the DEX.

The following is a description of the various options:

Registers

R [$*number*] *reg/size...*

number is the number of all the registers to be defined by all the register directives. This number must appear once only in the first register directive command.

reg is either the name of one register (e.g., PC, R15 etc.) or a range of registers that are instances of the same register type, with a hyphen to distinguish between them (e.g., R1-R14). A register name can contain up to eight characters.

size is the size of register or register range members in bytes.

This options defines the registers on the ADB board; each register is defined by its name and its size.

Example

To configure 0x38 registers, R1, R2,... R14, each two bytes long, PC four bytes long, and PSR one byte long:

```
>CFG R $38 R1-R14/2 PC/4 PSR/1
```

Register bit masks

B *bit-reg bit-name/mask...*

bit-name is the character name of a register bit. A bit-name length is up to two characters.

mask is the mask of this bit in the register.

There are two bit-registers in the CompactRISC architecture: CFG and PSR. The DEX uses the bit registers internally. This command defines the location of each relevant bit in the these registers.

Example

To configure the PSR register with two bits (I in bit 0, T in bit 12) and the CFG register with two bits (A in bit 8, D in bit 11):

```
>CFG B PSR I/0001 T/1000;B CFG A/100 D/800
```

Instruction encoding

E ***name/encoding/size***

name is the name of an assembly instruction. To run a debugging session the following encoding must be configured:

BPT - breakpoint instruction encoding

encoding is the hexadecimal encoding of the instruction.

size is the size in bytes of the instruction encoding.

This option defines the encoding of assembly instructions used by the DEX.

Examples

To configure breakpoint encoding to be 0xf038 in two bytes:

```
>CFG E BPT /38f0/2
```

Special memory types

M [$*number***]** ***name*** [$***parts***] ***/addr/size,....name*** **[$***parts***]** ***/addr/size...***

number is the number of all the memory types to be defined by all the memory directives. This number must appear once only in the first memory directive command.

name is the name of a memory type (e.g., **FLASH**). A name length is up to eight characters

parts is the number of parts in the memory type.

addr is the start address of the memory type part.

size is the size of the memory type part space in bytes.

This option defines special memory spaces. e.g., FLASH.

Memory-mapped registers, that the DEX uses internally, are treated as special memory spaces, and must be defined by this command in order to be recognized by the DEX. The register name is the memory name, and the address and size are the address and the size of the registers. The following list describes all these registers:

- **IMASK**
  Maskable interrupt vector register, used for enabling/disabling interrupts when TMON is running on the ADB (and not the application).

- **DSR**, **DCR**, **CAR**
  Registers used for defining and handling hardware breakpoints (not required if these are CPU registers).

Hardware breakpoints

H [$*number*] *breakpoint-definition...*

number is the number of hardware breakpoints.

breakpoint-definition is:

: {FP|FNP|NULL} {RWP|RWNP|NULL} {M|R} ***dcr***
{M|R} ***car*** {M|R} ***pc*** [{M|R} ***pass***]

bp_number - the hardware breakpoint number.

FP - fetch with pass counter
FNP - fetch without pass counter
NULL - the breakpoint cannot be defined to stop on fetch

RWP - read/write with pass counter
RWNP - read/write without pass counter
NULL - the breakpoint cannot be defined to stop on reads and writes

M - memory
R - register

dcr, car, pc, pass **-** These four virtual registers represent the register, or memory-mapped registers, which act as “debug-control register”, “compare-address register”, “program counter” and “pass counter” which define a hardware breakpoint in the CompactRISC architecture.

This option defines the hardware breakpoints on a specific board. It defines the number of breakpoints, together with their functionality. In the CompactRISC architecture, a hardware breakpoint can be defined either as a fetch breakpoint (which stops execution when a specific address is fetched), or as a read/write breakpoint (which stops execution when a specific address is written to, or read from). A hardware breakpoint has both functions at different times (a specific hardware breakpoint can not be used simultaneously for both fetch and for read/write). In the CompactRISC architecture, the four virtual registers defined above are used to set the hardware breakpoints. The ***pass*** register needs to be defined only if the breakpoint has a pass counter.

Example

To define a board to have one hardware breakpoint. The number of this hardware breakpoint is 0. It is both a fetch breakpoint with pass counter, and a read/write breakpoint with no pass counter. The dcr reg-

ister is a memory-mapped register, **mdcr**, the dsr register is a memory-mapped register, **mdsr**, the car register is a memory-mapped register, **mcar**, **pc** is the pc register, and the pass counter register is a memory-mapped register, **pass**.

```
> CFG H $1 0 FP RWNP M mdcr M mdsr M mcar R pc M pass
```

Notes

1. The registers and memory-mapped registers used to define hardware breakpoints, must have been previously defined by the CFG M and CFG R commands.

2. As the hardware-breakpoints definition relies on the virtual-registers definition, they must be used in a different CFG command line from the one defining these registers.

An Example of a Complete Board Configuration

```
> CFG R $15 pc/4,sp/2,psr/2,ra/2,isp/4,in/4,r0-r13/2, B
psr t/2; E bpt /f07b/2

> CFG M $b dcr/20fb/1, car $3 /22fb/1 /24fb/1 /26fb/1, pass
/28fb/1, pcexe $3 /22fb/1 /24fb/1 /26fb/1, nmis/02fe/2

> CFG H $1 0 FNP RWP M dcr M  dsr M car M pcexe M pass
```

The above example configures the DEX to work with a board. Three CFG commands are used:

- The first command configures the DEX for 0x15 registers: pc - 4 bytes, sp - 2 bytes etc. It defines bit T of the PSR register at bit number 1, the breakpoint instruction is encoded in two bytes its encoding is 0x7bf0.

- The second command describes the special memory spaces on the ADB. It describes 0xb different memory areas.
  dcr resides at address 0xfb20, and is 1 byte long.
  car has one byte at each of the three addresses: 0xfb22, 0xfb24, and 0xfb26.
  pass is 1 byte at address 0xfb28.
  pcexe is defined the same as **car**.
  nmis is 2 bytes long at 0xfe02.

- The last command defines the hardware breakpoints. There is one hardware breakpoint, number 0, which can be defined as a fetch breakpoint with no pass counter, or a read/write breakpoint with pass counter. **dcr**, **dsr**, **car**, **pcexe** and **pass** are all defined in memory.

ETHERNET Configuration (applicable only in board configuration)

ECFG [/TR*:cable_type] [/I ip-addr] [/H dex-name]* [/G *gateway*]]

cable_type is ***tp*** - for twisted pair connector.
caux - for coaxial cable connector.
external - for thick ETHERNET connector.

ip-addr is the IP address allocated for the DEX (e.g., **138.15.36.1**).

dex-name the DEX name in the net.

gateway the default gateway IP address.

Sets the ETHERNET cable configuration and configures the ETHERNET kernel to recognize the DEX.

Example To define the ETHERNET configuration with ip address **138.15.36.1**:

```
>ECFG /I 138.15.36.1
```

Define Symbol

DSYM ***name addr size***

name is the symbols name

addr is the start address

size is the size it takes in memory

Defines a symbol used by the DEX to perform an internal task. Symbols may be defined at any time during a debugging session. The sizes and addresses of symbols defined with this command may be changed at any during the debugging session.

D.2.2 DEX Reset and Initialization Commands

DEX INITialization

VINIT

The debugger must issue this command after every time it finishes updating the configuration database. This command makes sure that all changes to the configuration database take effect in the DEX.

Note This command resets the ADB.

DEX Reset

VRS

Reset the DEX's state to power-up. The configuration is erased.

DEX Boot (Applicable only in board configuration)

BOOT

Boots the DEX hardware.

D.2.3 Registers Manipulation Commands

Dump Registers

DR ***reg... register-range...***

reg is the name of one of the registers defined with the **CFG R** command. (e.g. pc, r1).

register-range is a range of registers defined with the **CFG R** command. (e.g. r0-r7).

The DEX dumps the contents of the specified registers in the order they appear in the **DR** command. The **DR** command with no arguments dumps the content of the arguments of the previous **DR** command.

Examples To dump the pc register followed, by r2, r3 and r4, the sp register and r2:

```
>R pc r2-r4 sp r2
    0100000002000000030000000400000ff00000002000000
```

To run the same command again:

```
>DR
    0100000002000000030000000400000ff00000002000000
```

Change Register

C ***regname*** = ***val***

regname is the name of one of the registers defined with the **CFG R** command.

val is the value to be assigned to this register.

The DEX assigns the value ***val*** to the register ***regname*** according to the register specification in the **CFG R** command.

Example To change the value of sp register to be 0x1234:

```
>C SP=3412
```

Save Registers

SR

The DEX saves an image of the ADB CPU registers in the DEX's memory.

Restore Register Image

RR

The DEX restores the ADB CPU registers from an image saved in the DEX by the **SR** command.

D.2.4 Memory Manipulation Commands

Dump Memory

DM ***format elem_size address***[***/size***]...

format is A - ASCII
B - binary

elem_size specifies the size in which memory is accessed, where:
B - byte
W - word
D - double word

The DEX dumps ***size*** bytes of memory contents starting at ***address.*** Memory is specified by address and size (in bytes) and is dumped in the specified order. Memory accesses are made in bytes, words, or double-words, according to ***elem_size***. If the response is longer then the maximum packet length of the communication protocol, the data is fragmented into a few response packets each beginning with a full response header. The **DM** command, with no arguments, dumps the content of the arguments of the previous **DM** command using the previously defined format.

Examples To dump four bytes from address 0x2000, and seven bytes from address 0x300:

```
>DM A B 0020/ 4 0003/7
    1122EE4401020304050607
```

To get the value of the same memory address again:

```
>DM
    1122EE4401020304050607
```

Load Memory

LM *format elem_size address*[*/size*] *data*

format is A - ASCII
B - binary

elem_size specifies the size in which memory is accessed, where:
B - byte
W - word
D - double word.

The DEX loads to the target board memory area specified by address size bytes of the data Memory is written in bytes, words, or double-words, according to ***elem_size***. The length of the data should not exceed the maximal data length of the communication protocol.

Example To load 10 bytes to address 0x2000 do:

```
>LM A W 0020/ 10 1122334401020304050E
```

Move Block of Data

M ***src-addr dest-addr n*** **[*N*]**

src-addr start address of original block

dest-addr start address of destination block.

n number of bytes to move.

The DEX moves ***n*** bytes of data, starting at ***src-addr***, to the block of memory starting at ***dest-addr***. The move is performed one byte at a time. The monitor verifies that the moved data matches the original data block unless the optional ***N*** (No verify) appears at the end of the command.

Example To move nine bytes of data from address 0x9000 to address 0xd000:

```
>M 0090 00d0 9
```

Fill Memory with Data

F *start-addr end-addr data* [***elem-size***] **[*N*]**

start-addr start of memory block to be filled.

end-addr last memory address to be filled.

data the data to be filled.

elem_size specifies the size in which memory is accessed, where:

B - byte
W - word
D - double word.

The DEX fills the block of memory starting at ***start-addr*** and ending at ***end-addr*** with the specified ***data***. The ***data*** is specified as bytes, words or double-words, depending on ***elem-size***. The monitor verifies that the moved data matches the original data block unless the optional N (No verify) appears at the end of the command.

Example To fill the memory area starting at 0x9000 ending at 0xd000 with the value 09:

```
>F 0090 00d0 9
```

Search Memory for Data

SR *start-addr end-addr data* [***size***]

start-addr start of memory block to be searched.

end-addr the last byte of the memory block to be searched

data the data to be searched

size specifies the data size, where:

B - byte
W - word
D - double word

The DEX searches for the first appearance of ***data*** between ***start-addr*** and ***end-addr***, and prints the address where the data has been found. Data may be a byte, word or double-word as specified by ***size***. The data is zero-extended or truncated to fit ***size***.

Example To search for the word 0X44, in the range 0X9000 to oXFFFF:

```
>SR 0090 FFFF 44 W
```

D.2.5 Breakpoint Commands

Create Software Stop Breakpoint

SBP *ST id addr* [*count (condition) CM mon-cmd-list*]

id is the breakpoint identification number.

addr is the break address.

condition is: (*prd).*

count is the number of times *addr* is hit before the actual break.

action is: ***CM mon-cmd-list.***
A list of up to five monitor commands separated by ';'. for breakpoints with stop execution.

This command creates a software breakpoint. Execution stops, after ***addr*** has been passed ***count*** times, if ***condition*** is TRUE. If a command list is defined, the commands are executed automatically when the program stops because of the breakpoint.

Examples

To create a software stop breakpoint with id = 2 at address 0x1000. Execution stops the fifth time that address 0x1000 is reached, and r0 equals '0'. Then the DEX dumps R5 and resumes execution of the application.

```
> SBP ST 2 0001 5 (r0=0) CM DR R5; G
```

To create a stop breakpoint with id = 5 at address 0x333. When the application execution is stopped, the DEX sends the message "B 5" and memory address 0x1000 is dumped as four bytes.

```
> SBP ST 5 3303 CM DM 0010/4
```

Create Hardware Stop Breakpoint

HBP ST *id type addr* [*mask*] [*count condition action*]

id is the hardware breakpoint identification number.

type is
R - read
W - write
A - access
E - execution

addr is the break address.

mask a number in the range 1-0xf. For read/write/access breakpoints it specifies which bytes cause a breakpoint if read or written to.

condition is (***prd***).

count is the number of times *a*
a list of up to five monitor commands separated by ';'. for breakpoints with stop execution.

This command creates a hardware breakpoint. Program execution stops after *count* times that the ***addr*** is passed and the ***condition*** is TRUE. If a command list is defined the commands are executed automatically when the program stops because of the breakpoint.

The DEX's response: **H id addr**

id is the hardware breakpoint number.
addr is the address that caused the breakpoint.

Examples To create a hardware breakpoint number 6 in the DEX breakpoint table when reading the least-significant-byte at address 0x100:

```
>HBP ST 6 R 100 1
```

To stop execution when the command at address 0x5510 is passed twice:

```
>HBP ST 7 E 155 2
```

Software Breakpoint Remove

Syntax **RBP {*A* | *id*}**

id is the breakpoint identification number.

A - apply to all breakpoints.

Example To remove all breakpoints.

```
>RBP A
```

Hardware Breakpoint Remove

HRBP {*A* | *id*}

id is the hardware breakpoint identification number.

A - apply to all breakpoints.

Example To remove hardware breakpoint number 0:

```
>HRBP 0
```

Execution Commands

Go

G [***reg***]

reg is the name of a general-purpose register specified in the **CFG R** command.

The monitor starts program execution at the address in the CPU Program Counter (PC); control is passed to the user program. To start at a specific address, first use the **C PC=addr** (change program counter) command.

For normal termination of a program, the first G command issued must specify a register ***reg***. The monitor initializes the register to point to the address of the End Of Program code. Since a typical CompactRISC program ends with the **jump reg** assembly instruction, we recommend that you use the link register of this jump instruction.

Example To put the End-Of-Program address in **ra** and start program execution:

```
>G ra
```

Step n Instructions

ST [*n*] [v**]**

Steps ***n*** assembly instructions. If ***n*** is not specified, one step is performed. If ***v*** is specified, the DEX sends the pc value to the debugger after each instruction execution.

Example To step five assembly instructions, and report the pc to the debugger after each step:

```
>ST 5 v
```

Single Step Until Stopped

ST S [v]

The DEX single steps through the program until a stop breakpoint is hit. If there is no stop breakpoint, other than end-of-program, it steps through the rest of the program. If ***v*** is specified, the DEX sends the pc to the debugger after each instruction execution.

Branch and Link

BAL ***reg addr***

reg is the link register

addr is a subroutine address.

This command simulates the **BAL** instruction. ***reg*** is loaded with the address of an **EXCP BPT** instruction, and then the subroutine is executed. On return from the subroutine, the **B RET** message is issued.

Example To set the PC to 0xe000, load r4 with a return address where the **EXCP BPT** instruction can be found, and then transfers control to the routine at address 0xe000:

```
> BAL r4 00e0
```

Jump Subroutine

JS ***addr***

addr is a subroutine address.

This command is equivalent to **BAL**. **RA** is used as the link register.

Begin

B ***reg addr***

reg is the link register,

addr is a subroutine address.

This command initializes the PC to ***addr***, but does not transfer control to your program. It also initializes ***reg*** to point to the return address where an **EXCP BPT** instruction resides. This halts your program.

Example To set the PC to 0xe000 and load **RA** with a return address where an **EXCP BPT** instruction can be found do:

```
> B ra 00e0
```

D.2.6 Remote Commands

Reset

RS

Issues a **RESET** signal to the ADB board. This restarts the monitor.

In response TMON sends the reset trap message to the DEX:

`0x3c monitor-type monitor-version`

ISE

ISE

Sends an **ISE** (non-maskable interrupt) to the ADB.

The DEX response: **E ISE**

D.2.7 Time Commands

Clear Timer

CT

Clears the DEX timer.

Get Timer

GT

Gets current timer value.

Where number is the timer value.

D.2.8 Simulation Specific Commands

These commands are used only in simulated environment. It is basically an interface to simulation specific functions.

Set Simulator Configuration

SIM config *config-params*

Sets the simulator configuration according to the parameters.Set/Get PSS Status

PSS Status

SIM pss [on|off]

Sets the Peripheral Simulator status on or off. Without parameters returns the status.

Load PSS File

SIM pss load *filename*

Loads *filename* to the Peripheral Simulator.

Clear PSS State

SIM pss clear

Clears the Performance Simulator state.

D.2.9 Miscellaneous Commands

Set Interrupt Mask at Monitor Time

I *mask*

Sets the value of the monitor interrupt mask. It also sets ADB PSR I and E bits. If the mask is zero, ADB PSR I and E bits are set to zero.

Extension Command

EXT *command*

command is a string representing a new monitor command.

In order for the EXT command to work the following routine needs to be supplied:

```
bool exec_extension_commands(char *cmd, char *response)
```

Note In order for the DEX to be able to recognize EXT command, add the extension code to the Makefile and compile using the command:
`make EXTRA_CFLAGS=-D__EXTENSION__`

D.3 TRAP AND ERROR MESSAGES FROM DEX TO THE DEBUGGER

DEX Indications	Meaning
*	Prompt; DEX ready to receive commands.
?	DEX command error message
data	Printed data for DM and DR commands
B TRC	Step ended.
B num	Reached breakpoint number
B END	Reached the end of program execution.
H num addr	Hardware breakpoint “num” at “addr”

DEX Error Messages	Meaning
E SIM	Simulator error message.
E TMON	Communication problems with TMON.
E SRC	Search failed.
E VRF addr	Verify error at address addr.
E ASRT addr	Assertion failed at address addr.
E BPT	Non-debugger breakpoint trap.
E DBG	Debug trap.
E DVZ	Division by zero.
E FLG	Flag trap error.
E HBT	Non-debugger hardware breakpoint.
E ILL	Illegal instruction.
E ISE	ISE response.
E NES	TMON entered nested state.

DEX Error Messages	Meaning
E NMI	Non-maskable interrupt.
E PROF	Error in profiling.
E SVC	Supervisor call.
E TRC	Non-debugger trace trap.
E UND	Undefined instruction.
E VIO	Virtual I/O protocol error.

Appendix E
DEBUGGER LIMITATIONS

E.1 PERIPHERAL STIMULUS SYSTEM (PSS)

The PSS feature, while operable, is not fully supported.

E.2 EDIT FIELD SIZE

The size of the edit fields in the dialog boxes is limited to 64 characters.

E.3 STEP/NEXT COMMANDS WHILE MEASURING PERFORMANCE

Using **step**/**next** commands, while measuring performance with the performance simulator, may increase the number of cycles measured. To get an accurate performance measurement, avoid breakpoints and single steps during performance measurement.

E.4 TARGET BOARD ACCESS WHILE PERFORMING VIRTUAL I/O

Debugger commands which access the target board may not be issued while the application is performing virtual I/O.

E.5 APPLICATION DEBUGGING IN SIMULATED ENVIRONMENT

In simulated environment, application debugging (stopping at breakpoints, stepping etc.) may begin only after the intbase register is loaded with the value of the application interrupt dispatch table. The intbase register is loaded either in the **libadb.a** start routine, or in the application start routine. When you invoke the debugger, set a breakpoint in your application in one of the lines that appears after loading the intbase register. Run the program, using the debugger **Go** command, until you reach this breakpoint, and then start debugging.

Appendix F
PERFORMANCE SIMULATION CONFIGURATION FILE

You can configure the wait-state parameters of the simulated memory through a simulator configuration file. The address space can be partitioned into several sections, each with its own wait-state configuration. You can define up to 10 (non-overlapping) sections. For each section you can define different wait-states for different access types, according to the type of access (load, store, fetch) and the size of access (byte, word, 3 bytes, double-word).

In addition, different delay values can be defined for the “wait-state” delay and for the “hold” delay.
(A “wait-state” delay is the number of cycles between the appearance of an address on the address bus and the appearance of the corresponding data on the data bus.
A “hold” delay is a delay typical of DRAM accesses. It is the number of cycles between the appearance of data on the data bus and the time in which a new address can be written to the address bus.)

Each “memory section declaration” in the simulator configuration file consists of a reserved sequence, followed by a memory range, followed by a list of wait-state parameters. For convenience and readability the wait-state values should be written in a table format. Comments can be added to the configuration file for clarity.

SYNTAX

A memory section declaration consists of the following parts:

- Section header.

  ```
  [Address Range]
  ```

- Address range.
  Two hexadecimal numbers marking the start and end addresses of a memory section.

  ```
  00000000 0000F000
  ```

- Wait-state values.
  A list of “wait-state” and “hold” values. The position in the list determines the access type combination to which the values correspond.

  ```
  1  1  1  1
  0  0  0  0
  0  0  0  1
  0  2  2  2
  0  3  0  1
  0  0  0  0
  ```

- ';' - Comments.
  A comment begins with a semicolon and ends with EOL. The following abbreviations are all comments, and appear in the configuration only for clarity:
  - ws - Memory wait-state cycles.
  - hold - Memory hold cycles.
  - L - Load access.
  - S - Store access.
  - F - Instruction fetch access.

Example

```
[Address Range] 00000000 0000f000
; ACCESS SIZE (in bytes)/
;              /
; 1  2  3  4          / ACCESS TYPE
;---------------;------
 1  1  1  1  ;ws   L
 0  0  0  0  ;hold
;---------------;------
 0  0  0  1  ;ws   S
 0  2  2  2  ;hold
;---------------;------
 0  3  0  1  ;ws   F
 0  0  0  0  ;hold
;---------------;------
```

Appendix G
PERFORMANCE SIMULATION TRACE OUTPUT

The simulator sends detailed information of the program execution to a file. The beginning of the file contains some general information: Simulator version, date and time, executable file name and the wait-state configuration, as read from the simulator-configuration-file. If the default wait-state configuration is used (all 0's) the configuration does not appear in the output file. If the simulator was invoked with the "long format" option, it prints a cycle-by-cycle status of the CPU's pipeline.

PIPELINE STATUS SYNTAX

The CompactRISC CPU pipeline consists of three pipeline stages: Fetch, Decode and Execute. The execution flow of a single instruction consists of the following phases:

- An address is set on the address bus.
- One, or more, cycles later (depending on the wait-states) the corresponding data appears on the data bus and enters the queue.
- Enough bytes to constitute a full, valid, instruction are resident in the queue and are passed to the decoder (CR32 only).
- A single cycle after an instruction entered the decoder it is passed to the execution stage for execution.

Under optimal conditions, all stages of the pipeline might be occupied with instructions in various stages of their execution.

The following is the structure of the pipeline status log.

- CYCLE No.
  The number of clock cycles since the start of program execution.
- PROGRAM COUNTER
  The address of the executing instruction.
- Inst. No.
  The number of assembly instructions executed since the start of program execution.
- EXECUTING INSTRUCTION
  Disassembly of the executing instruction.
- Inst. Length
  The number of bytes in the instruction.

- Q size
  The number of valid bytes in the instruction queue. Valid queue bytes are bytes fetched from memory, but not yet consumed by the decoder.

- AB fetch
  The number of transactions on the Address Bus that were triggered by an Instruction Fetch (as opposed to Load/Store transactions). The letters 'a' - 'j' (corresponding to the digits '0' - '9') represent the least significant digit of the Address Bus Fetch counter.
  - Lower-case letters indicate an instruction fetch bus transaction.
  - 'M' indicates a Load or Store transaction on the bus.
  - A hyphen '-' indicates an idle bus cycle.

- DB fetch
  The number of transactions on the Data Bus that were triggered by an Instruction Fetch (as opposed to Load/Store transactions). The letters 'a' - 'j' (corresponding to the digits '0' - '9') represent the least significant digit of the Data Bus Fetch counter. A Data Bus transaction indicated by a letter in this column corresponds to the Address Bus transaction indicated by the same letter in the 'AB fetch' column.
  - Lower-case letters indicate an instruction fetch bus transaction.
  - 'M' indicates a Load or Store transaction on the bus.
  - A hyphen '-' indicates an idle bus cycle.
  - An asterisk '*' indicates an idle bus cycle caused by a Non-Sequential Fetch.

- ID inst
  The number of instructions decoded by the decoder. Only instructions that have reached the execution stage are counted. The digits '0' - '9' represent the least significant digit of the counter.
  - Digits indicate an instruction decoding.
  - A hyphen '-' indicates an idle cycle.

- EX inst
  The number of instructions executed by the CPU. The digits '0' - '9' represent the least significant digit of the counter. An instruction indicated by a digit in this column corresponds to the decoding of the same instruction indicated by the same digit in the 'ID inst' column.
  - Digits indicate an instruction execution.
  - A hyphen '-' indicates an idle cycle.

- Delay cause
  Under optimal circumstances a single instruction can be executed on each and every cycle. When this is not the case, the cause may be an instruction that takes more than a single cycle to execute, or a full instruction that has not been yet fetched from memory. In either case, this column indicates the cause for lack of execution. 'IF' indicates that the CPU is waiting for a full instruction to be fetched. 'EX' indicates that a long instruction is executing.

Example

```
          CRXX Performance Simulator, version X.X.X
Date: Sun Feb 4 15:23:25 1996
Simulated program: dhry1.x
Output format: LONG

[Address] 1 00000000 - 0000f000
; ACCESS SIZE (in bytes)/
;         /
; 1   2   3   4  / ACCESS TYPE
;-------------------------;------
 1   1   1   2   ;ws   L
 0   0   0   0   ;hold
;-------------------------;------
 0   0   0   1   ;ws   S
 0   2   2   2   ;hold
;-------------------------;------
 0   3   0   1   ;ws   F
 0   0   0   0   ;hold
;-------------------------;------
```

CYCLE No.	PROGRAM-COUNTER	Inst. No.	EXECUTING INSTRUCTION	Inst. length	Q s i z e	A B f e t c h	D B f e t c h	I D i n t s	E X i n s	D S P M	delay case
1					0	b	-	-	-	-	IF
2					4	c	b	-	-	-	IF
3					8	-	c	-	-	-	IF
4					8	-	-	1	-	-	IF
5	00000000	1	movw $0xb60:l,r0	6	2	d	-	2	1	-	
6	00000006	2	lpr r0,intbase	2	4	e	d	-	2	-	
7					0	d	*	-	2	-	EX
8					4	e	d	-	-	-	IF
9					8	-	e	-	-	-	IF
10					8	-	-	3	-	-	IF
11	00000008	3	movw $0x36b0:l,r0	6	2	f	-	4	3	-	
12	0000000e	4	movw r0,sp	2	4	g	f	-	4	-	
13					8	-	g	-	-	-	IF
14					8	-	-	5	-	-	IF
15	00000010	5	movw $0x3730:l,r0	6	2	h	-	6	5	-	
16	00000016	6	lpr r0,isp	2	4	i	h	-	6	-	
17					0	h	*	-	6	-	EX
18					4	i	h	-	-	-	IF
19					8	-	i	-	-	-	IF
20					8	-	-	7	-	-	IF
21	00000018	7	bal ra,*+8:l	6	0	i	-	8	7	-	
22					4	j	i	-	-	-	IF
23					8	-	j	8	-	-	IF
24	00000020	8	addd $-4:s,sp	2	6	a	-	9	8	-	
25	00000022	9	stord ra,0(sp)	2	8	M	a	0	9	-	
26					8	-	M	0	9	-	EX
27	00000024	10	movd $0:s,r0	2	6	b	-	1	0	-	
28	00000026	11	stord r0,0xc28:l	6	4	M	b	-	1	-	
29					4	c	M	2	1	-	EX

```
Total cycles :  201316
Total instructions: 87775
Cycles per instruction: 2.29
Average instruction length: 3.15
Total fetches: 160729
Total store instructions: 20405
Total load instructions: 20028
```

INDEX

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W