CompactRISC™

Debugger Manual

Part Number: 424521772-004

August 1998

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.
2.a	September 1997	Alpha release for CR16B.
2.0	January 1998	Beta release.
2.1	August 1998	Product Version.

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

Chapter 1
OVERVIEW

1.1 INTRODUCTION

The CompactRISC Debugger is a GUI debugger for the CompactRISC family. It is available for IBM PC-compatible computers, running Microsoft 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 installing and configuring the debugger, and the remaining CompactRISC tools.

To help you evaluate a CompactRISC microprocessor, or to develop software for it, National Semiconductor provides an Application Development Board (ADB) kit for your particular member of the CompactRISC family. The CompactRISC Debugger communicates with the ADB, as well as with the instruction level simulator running on the host platform, using the Debugger Communication Interface (DBGCOM) software, which is supplied with the release package.

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 Toolset - C Compiler Reference Manual*)

1.3 DEVELOPMENT ENVIRONMENT

The debugger, communicating with its instruction-level simulator, provides a complete, software-based, evaluation or development environment. It supports all the capabilities necessary for effective and efficient development of your target program.

However you will eventually need to debug your application on real hardware. In this case, the application runs on an Application Development Board (ADB). The host computer communicates with the ADB through an RS-232, RS-422 or JTAG connection. For RS-422 or JTAG connections you will need a PC add-on card installed in your PC, and connected to the ADB.

As a user, you will find no noticeable difference between debugging with the Simulator and hardware-assisted debugging with an ADB. The tools and interface are identical. In both cases, the debugger communicates with the target (either the simulator running on the host platform, or an ADB) with a standard API using the Debugger Communication Interface (DBGCOM) software.

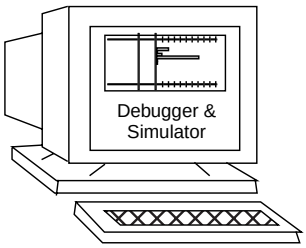

Figure 1-1. Debugging Using a Functional/Performance Simulator

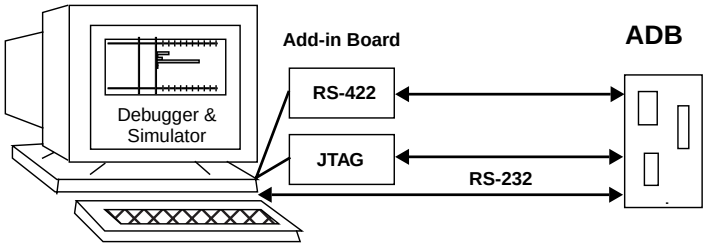

Figure 1-2. Debugging the Program on a Target ADB Board

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.

Limitations

The Simulator has the following limitations:

- ILS does not simulate hardware aspects of instruction execution, such as caching or pipelining. However, it does include factors in its performance estimation to allow for caching, etc.
- 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 Application Development Board (ADB) is product-specific, supporting a particular member of the CompactRISC family. The CompactRISC Debugger communicates with an ADB using the DBGCOM standard interface. Before you can communicate with any ADB, you must first install the DBGCOM software.

Board/host connection

You can connect the ADB to your host system via one of the communication channels supported by the DBGCOM software (e.g., RS-232).

See the *Debugger Communication Interface (DBGCOM) Installation Guide* for details of how to define a communication channel between the CompactRISC Debugger and an ADB.

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

ADBs have a monitor, Target MONitor (TMON), to aid debugging. It provides the following features:

- Control of program execution and debugging.
  The debugger can download your program to on-board memory. It can then execute commands for starting, stopping, single-stepping, and setting breakpoints within your program.

- Data exchange.
  The debugger enables you to display, and change, data located in the on-board memory, and CPU general-purpose and special-purpose registers.

- Run-time environment.
  TMON supports library 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.
  On reset, the monitor initializes itself and some of the board and chip registers. It may perform a small set of diagnostic tests to ensure that the board is operational.

1.4 MANUAL ORGANIZATION

Chapter 1 *Overview* (this chapter).

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.

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,

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

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

Appendix E *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. *Debugger Communication Interface (DBGCOM) Installation Guide*.

6. *CompactRISC Toolset - CR16A Programmer's Reference Manual*.

7. *CompactRISC Toolset - CR16B Programmer's Reference Manual*.

8. *CompactRISC Toolset - CR32A Programmer's Reference Manual*.

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

Chapter 2
DEBUGGER FEATURES

2.1 INTRODUCTION

The CompactRISC Debugger is used to debug programs, either by executing them on a National Semiconductor Application Development Board (ADB), or with an Instruction-Level Simulator running on the host platform. 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.

Before using the debugger you must install the DBGCOM software, included in the release package.

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 2.2.14.

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. To run the debugger with the Run option of Windows Start menu, use the following invocation syntax:

crdb [*exec_file*] [**-c=***core*] [**-e=***command_file*] [**-l=***log_file*]

Executable file

exec_file is the file name (including the path, if necessary) of the executable COFF file to be debugged. This file must have been generated using the CompactRISC C Compiler or the CompactRISC assembler, preferably with the debug option. Refer to the *CompactRISC Toolset - C Compiler Reference Manual* and to the *CompactRISC Toolset - Assembler 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, *CompactRISC Toolset - Introduction* describes the procedure for compiling and linking your program.

Core specification

In the invocation line you can specify the name of the CompactRISC CPU core, which is the target of the debugging session. To do this use the **-c=core** option. Possible values of **core** are **CR16A**, **CR16BS**, **CR16BL**. You can also set the core from the debugger command line after the invocation. If no core is specified, the default is **CR16BS**.

Command file

if you invoke the debugger with the **-e=***command_file* option, 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 4-18). 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 can select the target environment in which you want to debug the program. This can be either simulator or development board. The default target is simulator.

Simulator

Use the **COMM -S** command (Section 2.2.5), 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 one of the communication channels supported by the DBGCOM software, and define an appropriate communication channel using the DBGCOM setup procedure.

Use the command:

```
COMM -[s|f] <communication channel name>
```

where **<communication channel name>** is the name you chose for the communication channel to the ADB during DBGCOM setup. Note that **<communication channel name>** can contain only letters and digits.

Alternatively, open ***CONFIG*** and select ***TARGET BOARD***.

For detailed information, and installation instructions, for your DBGCOM, see the *Debugger Communication Interface (DBGCOM) Installation Guide* supplied with this package.

2.2.6 Selecting the Core

You can select the CPU core using the **CORE** command (Section 4.8). The default, if no core is specified, is CR16BS. On downloading code (see below), the debugger automatically selects the core according to the Magic Number which appears in the executable file header. This Magic Number is set during the compilation process according to the user-specified target core.

2.2.7 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 a mixture of 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.

2.2.8 Working with Breakpoints

There are two types of breakpoints: hardware and software. 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.

Hardware breakpoints are provided by either the chip or the simulator. They are available for the CR16B and CR32A.

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.

CR16B with bit manipulation

Note that CR16B bit-manipulation instructions that refer to a specific bit in a word, only access the byte in the word in which the bit is located. Therefore, when trying to detect access to a specific bit, always use the address of its byte.

For example:

```
cbitw $9,0x1000
```

The address that is actually accessed is 0x1001, where bit 9 of the word is located. To generate an address match in this case, define the breakpoint at address 0x1001.

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 4.40) 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.

Attaching commands to break

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

2.2.9 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 DBGCOM software. By default, the debugger executes up to **main**.

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, 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.

2.2.10 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 4.41) 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 4.39) 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.

2.2.11 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.

2.2.12 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.

To get any kind of performance measurement for your program, open ***CONFIG***, select ***SIMULATOR*** and check the ***PERFORMANCE ON*** checkbox.

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***, select ***SIMULATOR*** and uncheck the ***PERFORMANCE ON*** checkbox.

By default, after **RESET**, if the performance simulator is on, the debugger turns the performance simulator off, and turns it on when it reaches `main`. If you want performance information on the startup code, change the debug mode, as explained in Section 2.2.9.

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.

On completion of the portion of your program you want to profile, open ***SHOW*** and select ***PERFORMANCE*** to display the Performance Window. A bar graph shows an execution profile of either files, or functions, for your program. Double click on the bar to go to the next level of granularity. The levels of granularity are: files, functions, lines and instructions.

The bar graph reflects the percentage of execution time your program spent in each of the displayed portions 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 of function, lines and instructions includes only portions of code containing symbolics.

Note

Turning the performance simulation off, does not delete the data base. To reset the data base, either select ***RESET*** from the ***PERFORMANCE WINDOW*** menu or push the ***RESET*** button in the **Simulator Configuration** dialog box.

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*** ➨ ***SIMULATOR*** ➨ ***PERFORMANCE ON***. Click the ***SETUP*** button and mark the ***LOG ON/OFF*** box in the dialog box, and select an output file name.

Pushing the ***RESET*** button in the **Simulator Configuration** dialog box resets the simulation timer and the instruction counter.

For details of the output format, refer to Appendix E.

Config

You can specify configuration parameters for the performance simulation. These parameters deal 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*** ➨ ***SIMULATOR*** ➨ ***PERFORMANCE ON***. Click the ***SETUP*** button a***ND*** in the dialog box, next to ***SIM CONFIG FILE***, specify the configuration file name.

For convenience, you can also generate a configuration file template from the **Setup** dialog box. Type a name for the template file in the field next to the ***GEN TEMPLATE*** button, then click this button. For a detailed explanation of the configuration file format, see Appendix D.

Note

You can clear the existing configuration by pushing the ***CLEAR*** button in the ***SETUP*** dialog box. Passing an empty string as the name of the configuration file causes the default configuration to be valid.

2.2.13 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

To put 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).

2.2.14 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.

2.2.15 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 4.38) 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.

2.3 USING THE SIMULATION ENVIRONMENT

One of the tools provided by the debugger to aid you in your program development is the simulator. The simulator 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 with the **COMM** command (Section 4.7).

In addition, the simulator can estimate performance, and display data and profiles (bar charts) reflecting the theoretical performance of all, or portions, of your program. This can be very useful in determining program performance, locating bottlenecks, etc.

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 and after pressing ***RESET*** in the Simulator Configuration dialog box.

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.

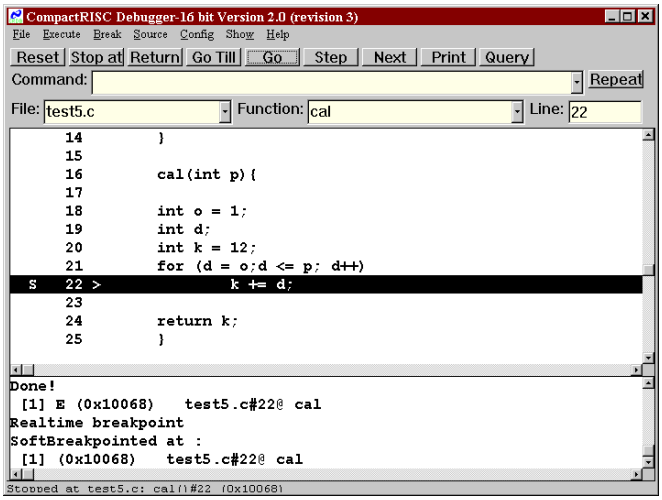

Figure 3-1. Main Window

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.*

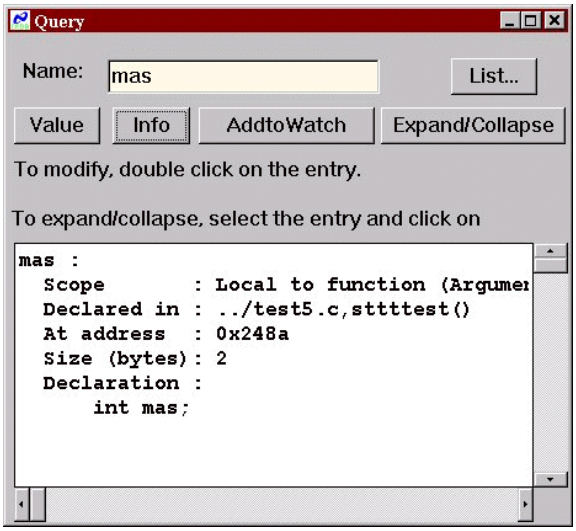

Figure 3-2. Query Window

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.

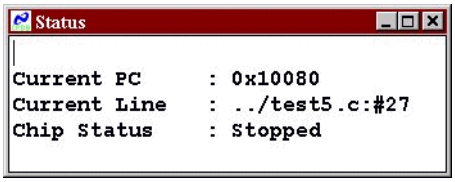

Figure 3-3. Status Window

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.

Register Window

The Register Window (Figure 3-4) shows the contents of the PC, SP, PSR and other general-purpose and 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.

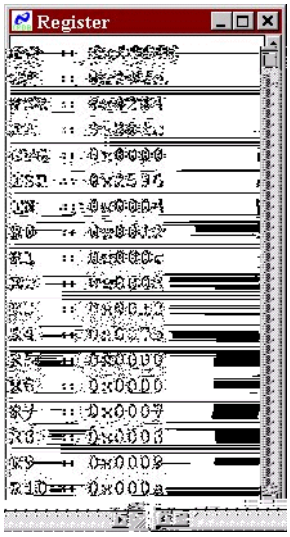

Figure 3-4. Register Window

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.

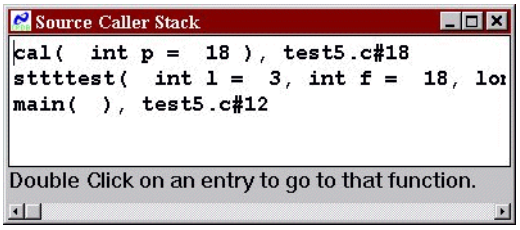

Figure 3-5. Caller Stack Window

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.

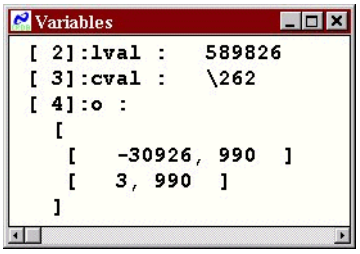

Figure 3-6. Watch Variables Window

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.

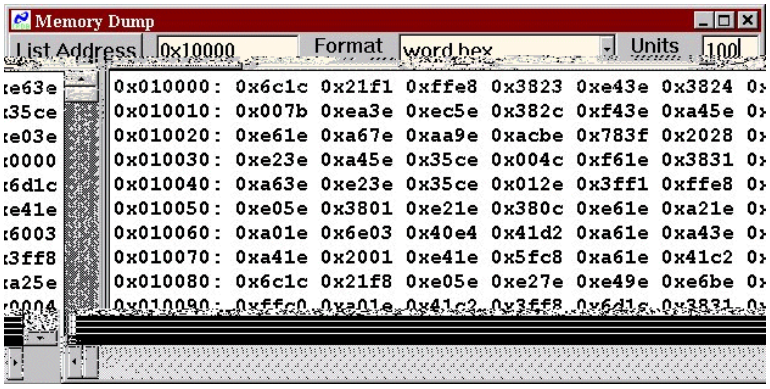

Figure 3-7. Memory Window

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).

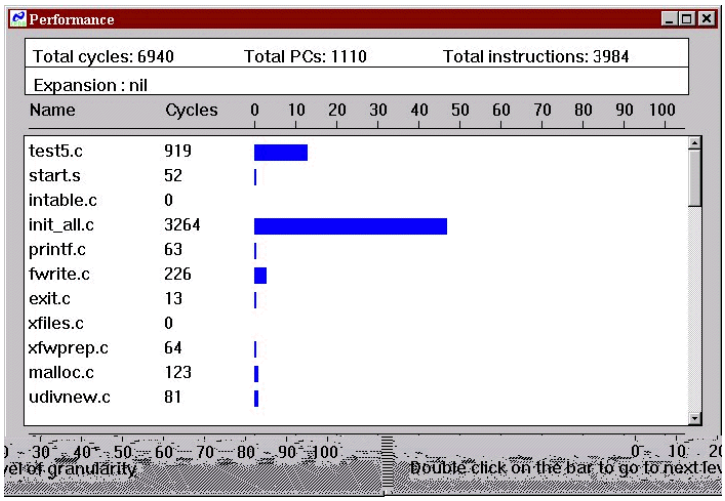

Figure 3-8. Performance Window

The menu bar in this window contains the following (from 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. Same as selecting ***Config -> Simulator -> Performance On***.

Graph — Select various update options:

- Redisplay: refresh the bar graphs.
- Print to log file: Print the displayed data to the debugger's log file.
- 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.

Reset: — Reset the data base for the Performance Simulator.

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 any 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 line numbers are displayed, you can double-click on a line number to see the execution time of that line, broken down by instruction.
If you request the breakdown of a second file, function or line, 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.

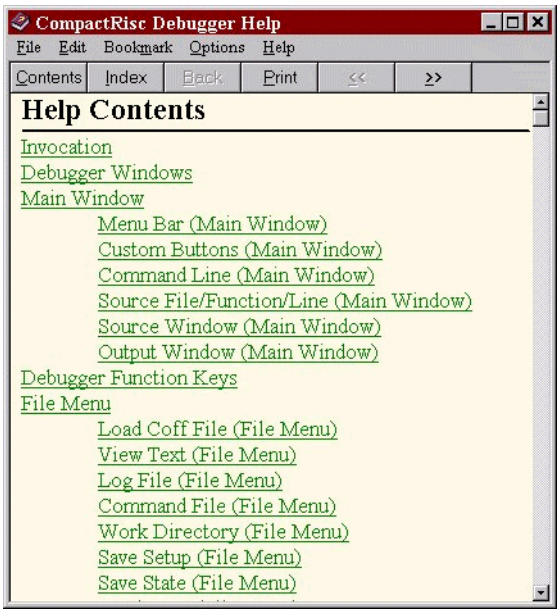

Figure 3-9. Help Window

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 2.2.9)

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:

- Core: configures the member of the CompactRISC CPU core family. (Section 4.8)
- **`Target Board`**: configures the debugger to use the required communication channel (Section 4.7)
- **`Simulator`**: configures the debugger to use the Instruction-Level Simulator (Section 4.7)
- **`Radix`**: sets the global radix for debugger output displays (Section 4.23)
- **`Verbose`**: displays all communications between the debugger and the monitor. (Section 4.38)
- Custom Buttons: Configure the configurable buttons (Section 3.1.1)
- **`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.

Arguments, modifiers, and options for all commands are defined in Section A.2.

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).
  - **`$e`** Address of the first instruction in the epilogue.
  - **`$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>        radix
                Radix : 10
```

You can see the syntax of any command by entering the command name followd by a question mark. For example:

```
Command>        radix ?
                radix
                radix 8|10|16
```

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 2.2.8. 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 Hardware breakpoints are provided by the CR16B and CR32A chips, and their simulators. Hardware limitations 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
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 first line of the epilogue of 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 continues 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 c:\test
```

To display the current working directory:

```
>cd
     c:\test
```

4.7 COMM — SET COMMUNICATIONS CHANNELS

Sets the communication channel required to communicate with an Application Development Board (ADB), or a simulator running on the host platform.

To communicate with an ADB, you must first define a communication channel, using the Debugger Communication Interface (DBGCOM) setup procedure. For more details of how to define a communication channel see the *Debugger Communication Interface (DBGCOM) Installation Guide*.

comm -s sets the debugger to communicate with a simulator running on the host platform.

comm -s *<communication_channel_name> [,Pid]*

sets the debugger to communicate with an ADB using the required communication channel.

<communication_channel_name>
one of the available communication channels, which you defined using the DBGCOM setup procedure.

Pid The id of the processor on the selected ADB. Supported only for communication channels using the JTAG protocol.

comm -f *<communication_channel_name> [,Pid]*

Sets the debugger to communicate with an ADB, using the specified communication channel, while forcing the DBGCOM to open this channel. Required only if DBGCOM refuses to open a communication channel used by other debugger, but no other debugger is currently functional (e.g., it was killed before closing the communication channel).

comm -c close the current communication channel.

comm -l displays all available communication channels names as defined with the DBGCOM setup procedure.

comm displays the current settings.

Mouse Open ***CONFIG*** and select ***TARGET BOARD*** or ***SIMULATOR***. Set up the communication parameters in the dialog box.

Examples To set the debugger to communicate with a simulator:

```
>comm -s
     Target is Simulator.
```

To select a communication channel called “MY_RS422”:

```
>comm -s MY_RS422
     Target is MY_RS422.
```

To list available communication channels:

```
>comm -l
     Communication channel 1 is: MY_RS422
     Communication channel 2 is: JTAG_01
     Target is Simulator.
```

To display the current communication setting:

```
>comm
     Target is Simulator.
```

4.8 CORE - SET THE CURRENT CPU CORE

core [CR16A|CR16BL|CR16BS]

specifies the member of the CompactRISC CPU core family which is the target of the debugging session. For the CR16B, the small, or large, programming model is also specified.
This command is not relevant for the 32-bit CompactRISC core debugger.

core displays the current CPU core.

Mouse Open ***CONFIG*** and select ***CORE***. Select the required CPU core.

Examples To select CR16BL CPU core small programming model:

```
> core CR16BL
```

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 routine that is executed after returning from main.

In simulator mode, disabling the debugging of startup code, disables performance data collection on the startup code, even if performance mode on is selected.

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 **f**or 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*** and select ***GO***.

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 information (See **comm**, Section 4.7).
- Target core family name (See **core**, Section 4.8).
- 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 2.2.13). 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 2.2.13 describes the debugger facilities provided for use with input files and explains how to use them.

If an abort (Section 2.2.9) 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 2.2.13.

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*** or ***REGISTERS*** window 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’
```

. st5

To store the string into the locathe 3edressed by. st5

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 core family 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 op-code fetch.

For a general explanation of breakpoints, see Section 2.2.8.

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$x
```

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 2.2.11) 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 soft-break 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
```

Note 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.

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 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***.

4.39 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 VARIABLES,*** 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
```

4.40 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 4.39).

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
```

4.41 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
```

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
comm	Switch to simulator mode	-s	4.7
	Set communication channel name and processor id	-{s **|** f} <*communication_channel_name*> [,Pid]	
	List all available communication channels	-l	
core	Set cpu core	<*core_name*>	
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

Command	Definition	Syntax	Section
go	Go	[-c] [<*from_addr*>][//<*end_addr*>]	4.13
info	Debug info		4.14
input	Execute input file	<file_name>	4.15
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 also output	[-i | -o]	
	Expand aliases, or not	[-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

Command	Definition	Syntax	Section
resume	Resume execution from command file after pause		4.25
saveconfig	Save configuration	**[**<*file_name*>**]**	4.26
savestate	Save state	**[**<*file_name*>**]**	4.27
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*>**]]**	

Command	Definition	Syntax	Section
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*>**]]**	
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
verbose	Display Monitor traffic		4.38
view	View data	<*expression* **[**,<*print_specifier*>**]**	4.39
	View register	%<reg_name> **[**,<*print_specifier*>**]**	
watch	Adds entry	<*expression*> **[**,<*print_specifier*>**]**	4.40
	Removes, disables, enables entry	**{** -r **|** -d **|** -e **}** %<*id*> **|** *	
where	Locate self	**[**-v **]** -c **[**-v **[**<*func_symbol*>**[**@<*symbol*>**]]**	4.41

A.2 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
<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>	:=	& (represents 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 | $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 “ABCDEF”
hx	short unsigned hexadecimal integer in lower case “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 in upper case “ABCDEF”
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’.
<qualified_code_address>	:=	any valid *<address>* or *<symbolic_address>* 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 first instruction in 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**
- **Unexpected Communication results on <tmon command>**
  The <tmon command> failed.
  <tmon command> can be one of the following:
  "register read/write", "memory read/write" or "go".
  Check that the core definition matches the core on the ADB, and that the board is functional. Try to reset the target. You may have to restart the debugger.
- **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.
- **Address out of range**
  The address is greater than the maximum address value.
- **Chip is not Running**
- **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)

- **`Bad occurrence count`**
  Incorrect value of occurrence count for the soft break or hard break.
- **`This command is unavailable now`**
  The debugger does not support this command. Please report the problem to National Semiconductor.
- **`Unknown EIM error`**
  The debugger detected an unknown communication error.

Appendix C
DEBUGGER LIMITATIONS

C.1 EDIT FIELD SIZE

The size of the edit fields in the dialog boxes is limited to 64 characters.

C.2 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.

C.3 TARGET ACCESS WHILE PERFORMING VIRTUAL I/O

You may not issue Debugger commands which access the target board, or simulator running on the host platform, while the application is performing virtual I/O.

C.4 OPEN SIMULATOR AS A TARGET BOARD

You may define, using the DBGCOM setup, a target communication channel that communicates with a simulator running on the PC host. This is useful if you are using a debugger other than National's CompactRISC Debugger. Since the type of communication is fully transparent to the debugger, it relates to the simulator as if it were a board, i.e. performance-related features are disabled.

If you are using National's Debugger, and want on-line performance information, you should use the debugger's simulation mode. However, if you want to communicate with the simulator in target mode, you can still get performance information, but this is saved in a log file rather than presented on-line.

For more details of how to define a simulator communication channel, refer to the *Debugger Communication Interface (DBGCOM) Installation Guide*.

Appendix D
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” decimal 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 E
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, and the wait-state configuration, as read from the simulator configuration file. The simulator prints a cycle-by-cycle status of the CPU's pipeline.

Note that ***RESET*** closes the log file, and then opens a file with the same name. Thus the contents of the log file are lost. To save this data, either specify a different log file name to the debugger, or save the contents of the log file to another file before ***RESET***.

RESET also resets the simulation timer. This affects the cycle number displayed on the file. In addition, ***RESET*** resets the instruction number.

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.
- When enough bytes to constitute a full, valid, instruction are resident in the queue, they are passed to the decoder.
- 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.

For more information regarding the execution flow refer to the Programmer's Reference Manual for the core you are using.

The following is the structure of the pipeline status log.

- CYCLE No.
  The number of clock cycles since the later of the following events: ***RESET***, or reset of the simulator data base.

- PROGRAM COUNTER
  The address of the executing instruction.

- Inst. No.
  The number of assembly instructions executed since the later of the following events: ***RESET***, or reset of the simulator data base. Only the three least significant digits are displayed.

- 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

[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| PROGRAM-|Inst.| EXECUTING|Inst.| Q  A  D  I  E  | de-|
No.  | COUNTER| No.| INSTRUCTION|length|   B  B D  X  | lay|
     ||             | s  f  f  i  i  | ca-|
     ||             | i  e  e  n  n  | use|
     ||             | z  t  t  s  s  ||
     ||             | e  c  c  t  t  ||
     ||             |    h  h        ||
=============================================================
====
  1  ||             | 0  b  -  -  -  | IF|
  2  ||             | 4  c  b  -  -  | IF|
  3  ||             | 8  -  c  -  -  | IF|
```

```
 4  ||             | 8  - - 1 - | IF|
 5  | 00000000| 1 movw  $0xb60:l,r06| 2  d - 2 1 ||
 6  | 00000006| 2 lpr   r0,intbase2| 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,r06| 2  f - 4 3 ||
12  | 0000000e| 4 movw   r0,sp2| 4  g f - 4 ||
13  ||             | 8  - g - - | IF|
14  ||             | 8  - - 5 - | IF|
15  | 00000010| 5 movw   $0x3730:l,r06| 2  h - 6 5 ||
16  | 00000016| 6 lpr    r0,isp2| 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:l6| 0  i - 8 7 ||
22  ||             | 4  j i - - | IF|
23  ||             | 8  - j 8 - | IF|
24  | 00000020| 8 addd   $-4:s,sp2| 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,r02| 6  b - 1 0 ||
28  | 00000026| 11 stord  r0,0xc28:l6| 4  M b - 1 ||
29  ||             | 4  c M 2 1 | EX|
```

INDEX

A

B

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W