



MetaWare® Debugger

User's Guide for ARC®

4135-063

MetaWare® Debugger User's Guide for ARC®
© 1995-2007 ARC® International. All rights reserved

ARC International

<p>North America 3590 N. First Street, Suite 200 San Jose, CA 95134 USA Tel. +1-408-437-3400 Fax +1-408-437-3401</p>	<p>Europe Verulam Point Station Way St Albans, AL1 5HE UK Tel.+44 (0) 20-8236-2800 Fax +44 (0) 20-8236-2801</p>
--	---

www.ARC.com

ARC Confidential and Proprietary Information

Notice

This document, material and/or software contains confidential and proprietary information of ARC International and is protected by copyright, trade secret, and other state, federal, and international laws, and may be embodied in patents issued or pending. Its receipt or possession does not convey any rights to use, reproduce, disclose its contents, or to manufacture, or sell anything it may describe. Reverse engineering is prohibited, and reproduction, disclosure, or use without specific written authorization of ARC International is strictly forbidden. ARC and the ARC logotype are trademarks of ARC International.

The product described in this manual is licensed, not sold, and may be used only in accordance with the terms of a License Agreement applicable to it. Use without a License Agreement, in violation of the License Agreement, or without paying the license fee is unlawful.

Every effort is made to make this manual as accurate as possible. However, ARC International shall have no liability or responsibility to any person or entity with respect to any liability, loss, or damage caused or alleged to be caused directly or indirectly by this manual, including but not limited to any interruption of service, loss of business or anticipated profits, and all direct, indirect, and consequential damages resulting from the use of this manual. ARC International's entire warranty and liability in respect of use of the product are set forth in the License Agreement.

ARC International reserves the right to change the specifications and characteristics of the product described in this manual, from time to time, without notice to users. For current information on changes to the product, users should read the "readme" and/or "release notes" that are contained in the distribution media. Use of the product is subject to the warranty provisions contained in the License Agreement.

Licensee acknowledges that ARC International is the owner of all Intellectual Property rights in such documents and will ensure that an appropriate notice to that effect appears on all documents used by Licensee incorporating all or portions of this Documentation.

The manual may only be disclosed by Licensee as set forth below.

- Manuals marked "ARC Confidential & Proprietary" may be provided to Licensee's subcontractors under NDA. The manual may not be provided to any other third parties, including manufacturers. Examples—source code software, programmer guide, documentation.
- Manuals marked "ARC Confidential" may be provided to subcontractors or manufacturers for use in Licensed Products. Examples—product presentations, masks, non-RTL or non-source format.
- Manuals marked "Publicly Available" may be incorporated into Licensee's documentation with appropriate ARC permission. Examples—presentations and documentation that do not embody confidential or proprietary information.

U.S. Government Restricted Rights Legend

Use, duplication or disclosure by the U.S. Government is subject to restrictions as set forth in FAR 52.227.19(c)(2) or subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 and/or in similar or successor clauses in the FAR, or the DOD or NASA FAR Supplement.

CONTRACTOR/MANUFACTURER IS ARC International I. P., Inc., 3590 N. First Street, Suite 200, San Jose, California 95134.

Trademark Acknowledgments

ARC-Based, ARC-OS Changer, ARCanGeL, ARCform, ARChitect, ARCompact, ARCTangent, BlueForm, CASSEIA, High C/C++, High C++, iCon186, IPShield, MetaDeveloper, the MQX Embedded logo, Precise Solution, Precise/BlazeNet, Precise/EDS, Precise/MFS, Precise/MQX, Precise/MQX Test Suites, Precise/MQXsim, Precise/RTCS, Precise/RTCSsim, RTCS, SeeCode, TotalCore, Turbo186, Turbo86, V8 µ-RISC, V8 microRISC, and VAutomation are trademarks of ARC International. ARC, the ARC logo, High C, MetaWare, MQX and MQX Embedded are registered under ARC International. All other trademarks are the property of their respective owners.

Contents

Chapter 1 — Before You Begin	9
Technical Support	10
About This Book	10
About the Screen Shots	11
Where to Go for More Information	11
Document Conventions	11
Chapter 2 — Getting Started	13
Starting the Debugger	14
Starting the Windows-Hosted Debugger	14
Starting the Debugger Using a Command Prompt	14
Starting the Command-Line-Only Debugger	15
About the Driver Configuration File	16
Starting Your Debug Session	16
Debugger Options	16
Specifying GUI Options for ARC Targets	19
Using the GUI	24
Title Bar	25
Main Menu Bar	25
Using the Debug Console	27
Error and Condition Reporting	28
Viewing and Saving a Command Log	28
Setting Preferences	28
Stepping Keys	29
Command and Macro Keys	31
Command Input Fields	35
Status Bar	35
Running/Stopped Indicator	35
Drag-and-Drop Data Tabs	35
Docking Windows	37
Snap-to-Grid Windows	38
Saving Window Layouts	38
Look-and-Feel	38
One-Touch Expandable Dividers	39
Common Window Components	39
Working with Debug Projects	41
Creating a New Project	42
Duplicating a Project	42
Removing a Project	42
Changing the Current Project	43
Command Line	43
Getting Help	43
Help Using the Debugger	43
Help Using Controls and Commands	43
Preparing Programs for Debugging	43
Compiling Programs with Debug Information	43

Locating Source Files	44
Exiting the Debugger	45
Chapter 3 — Debugging a Program	47
Controlling Program Execution	48
Running a Program	48
Animating a Process	49
Restarting a Program	50
Hostlink Services	50
Viewing Source Code	51
Viewing Source Files	53
Viewing Source for a Function on the Call Stack	53
Listing All Executable Modules	54
Viewing Machine Instructions	54
Customizing the Disassembly Window	55
Viewing Disassembly Code	56
Working with Instruction History	57
Debugging with Breakpoints	60
Hardware and Software Breakpoints	61
Setting Breakpoints	61
Types of Complex Breakpoints	63
Working with the Breakpoints Window	64
Debugging with Watchpoints	66
Hardware and Software Watchpoints	66
Setting Watchpoints	67
Viewing a list of Current Watchpoints	69
Disabling and Enabling Watchpoints	70
Deleting Watchpoints	70
Viewing and Modifying Data	70
Working with Global Variables	70
Working with Local Variables	71
Examining Data	73
Modifying the Value of a Variable	75
Viewing the Function-Call Stack	76
Working with Functions	77
Evaluating Expressions	78
Debugging Memory	79
Viewing Memory at a Specific Address	80
Modifying the Memory Window	80
Copying or Filling Memory to or from a File	81
Working with a Hardware Stack	83
Examining Memory in Various Formats	84
Viewing an Image from Memory	84
Modifying the Contents of Memory	87
Searching for Patterns in Memory	88
Simulating Instruction and Data Caches	91
Enabling Cache Simulation	91
Command Line	91
Viewing Simulated Caches	92
Cache Algorithms	93
Analyzing Cache Performance	94
Execution Profiling	94
Two Ways to Profile	94

Viewing Source-Code Coverage	102
Enabling Source-Code-Coverage Analysis	102
Viewing Source-Code Coverage	102
Viewing Statement Counts	103
Viewing and Modifying Registers and Flags	104
Setting a Watchpoint on a Register	105
Changing the Registers Window	105
Changing Register Contents	106
Viewing and Modifying Auxiliary Registers	106
Debugging Multi-Tasked/Threaded Applications	107
Enabling Multi-Task/Multi-Thread Debugger Features	108
Viewing All the Tasks or Threads in Your Program	108
Viewing the State of a Specific Thread or Task	108
Viewing the State of Several Tasks at the Same Time	109
Using Task- or Thread-Specific Breakpoints	109
Using Thread-Specific Watchpoints	110
Non-Symbolic Debugging	112
Debugging Files without Debug Information	112
Debugging Executable Files	112
Debugging Stripped Object Files	112
Taking Snapshots of a Window	113
Example of Using the Snapshot Feature	113

Chapter 4 — Debugging a Target 115

Overview	116
Determining the Target System	116
Changing the Default System	116
Overriding the Default System	116
Using the Instruction Set Simulator	117
Stopping Execution by Instruction Count	117
Tracing Instructions	117
Viewing the Last Instructions Executed	117
Cloning the Simulator	117
Using the sysclone Command	117
Adding Simulator Extensions	118
Debugging Remotely with SCIT	118
SCIT Usage	119
Specific Connection Instructions	124
Debugging ARC Systems	126
Specifying Memory Size and Location	127
Simulating Load/Store RAM	129
Using a Memory Map to Indicate an Offset	129
Debugging with Overlays	130
Debugging Closely Coupled Memory (a Harvard Architecture)	132
Simulating DSP Extensions	132
Debugging on ARCanel Hardware	134
Using SmaRT Real-Time Trace (ARC 600 Only)	139
Debugging ARC Media Extensions and SIMD Instructions (ARC 700 Only)	143
Viewing Hardware Data and Instruction Caches	144
Viewing Build-Configuration Registers	145
Debugging an ARC Processor in an SoC	145
Viewing the ARC 600 MPU Registers	146
Displaying MMU and Exception Registers (ARC 700)	146

Debugging with ARC_DLL or BRC_DLL	148
Setting up an Ashling Opella Emulator	149
Using the Cycle-Estimating Simulator (CES)	150
Using the Cycle Accurate Simulator (CAS)	155
Using ARC xCAM (ARC 700 and ARC FPX)	155
Using the ARC xISS Fast ISS (ARC 600 and ARC 700)	155
Flash Programming	155
Chapter 5 — CMPD Extensions	157
Overview	158
About Multiprocess Debugging	158
Debugging Multiple Processes As a Group	158
Debugging Multiple Processes on Multiple Processors	158
Process Sets	158
Single-Process Debugging	158
Multiprocess Debugging	159
Setting up a CMPD Session	159
Sample Demonstration Programs	159
Creating the Multi-Process Debug Session	160
Editing a Multi-Process Configuration	163
Running a CMPD Session	163
Running a Single-Threaded Demo	163
Running a Multi-Threaded Demo	166
Viewing and Controlling Processes	167
Viewing Different Processes Using One Window	167
Controlling Processes Using the Processes Window	167
Changing the Focus	168
Showing the Same Process in all Windows	168
Running and Stopping Processes	168
Adding a Stepping Toolbar to the Processes Window	168
Working with Threads in the Processes Window	168
Using Breakpoints and Watchpoints	169
Compound Action Points	169
Action-Point Stop Lists	169
Using Barriers to Synchronize Processes	169
Creating a Barrier	170
Understanding How Barriers Work	170
Setting Different Action Points for Different Processes	170
Disabling and Enabling Barriers	171
Deleting a Barrier	171
Using Commands with CMPD	171
Combining Command Output	171
Completing Commands	172
Adding Processes During a Debug Session	173
Configuring a CMPD Session on the Command Line	173
Configuring a Session with the Debugger GUI	178
Chapter 6 — Command Features	181
Specifying Controls	182
About Command-Line Options	182
About Toggles	183
Using Debugger Commands	183

Rules and Guidelines for Using Commands	183
Using Command Prefixes and Modifiers	184
Entering Commands at the Debugger Command Prompt	186
Entering Commands in the GUI Command Field	186
Entering Commands in a Script File	187
Using Debugger Variables	188
Printing to stdout	190
Creating while Loops	190
Executing Commands Conditionally	191
Reading and Modifying Processor Data	193
Using Breakpoints and Watchpoints	200
Processor Status and Run Control	203
Using Commands to Generate Interrupts	207
Using Commands to Read and Write Files	216
Using Commands to View Profiling Information	218
Using Commands to Set the Debugger Return Code	221
Conventions for Debugger Input	221
Using Regular Expressions	221
Using Environment Variables in Path Names	222
Using Expressions	222
Using Identifiers That Match Register Names	224
Creating and Using Debugger Macros	224
Declaring and Defining a Macro	224
Invoking a Macro	226
Referencing Macro Arguments	226
Undefining a Macro	227
Listing Currently Defined Macros	227
Modifying the Driver Configuration File	227
Specifying the Driver Location	228
Specifying Default Values for Command-Line Options	228
Changing the Default Start-up to Command-Line Only	228
Reducing Debugger GUI Start-up Time	228
Specifying the Location of Java	228
Chapter 7 — Commands Reference	231
Overview	232
Alphabetical Listing of Commands	232
Chapter 8 — Options and Toggles	273
Command-Line Option Reference	274
Toggle Reference	298
Appendix A — Multi-Tasked RTOS Applications	313
About Threads and Tasks	314
Debugging Multiple Tasks/Threads	314
Specifying the Operating System	314
Debugging Tasks/Threads	314
Debugging Tasks with the Command-Line Debugger	315
Debugging Precise/MQX RTOS Applications	316
Task-Aware Debug Features	316
Configuring for Task Aware Debugging	317
Enabling Task-Aware Debugging in the Debugger	317

Updating TAD Windows	317
Viewing the Context of Blocked Tasks	317
MQX Menu	317
Obtaining Help	320
Debugging Nucleus PLUS Applications	320
Preparing to Debug a Nucleus PLUS Program	320
The Nucleus State Window	321
The Nucleus PLUS Memory: Queue Window	323
The Nucleus PLUS History Log	323
Viewing OS Resources in the Nucleus State Window	323
Debugging Nucleus PLUS in the Command-Line Debugger	324
Running the Nucleus PLUS Demo	327
Debugging ThreadX Applications	331
Preparing to Debug a ThreadX Program	331
The ThreadX State Window	332
The ThreadX Memory: Queue Window	334
Viewing OS Resources in the ThreadX State Window	334
Debugging ThreadX in the Command-Line Debugger	335
Running the ThreadX Demo	337
Appendix B — S-Record Files	341
What Is an S-Record File?	342
Why Use an S-Record File?	342
S-Record Format	342
Calculating the Checksum	342
Example S-Record	342
Appendix C — Quick Commands List	345
Appendix D — Quick Options List	349
Overview	350
Command-Line Options	350
Appendix E — Quick Toggles List	353
Overview	354
Debugger Toggles	354
Index	357

Chapter 1 — Before You Begin

In This Chapter

- [Technical Support](#)
- [About This Book](#)

Technical Support

We encourage customers to visit our Technical Support site for technical information before submitting support requests via telephone. The Technical Support site is richly populated with the following:

ARCSolve — Frequently asked questions

ARCSpeak — Glossary of terms

White papers — Technical documents

Datasheets — ARC product flyers

Known Issues — Known bugs and workarounds.

Training — Future ARC Training sessions

Support — Entry point to sending in a query

Downloads — Access to useful scripts and tools

You can access this site via our corporate website at <http://www.ARC.com>, or you can access the site directly: <http://support.ARC.com>.

Note that you must register before you can submit a support request via our Extranet site.

About This Book

This *MetaWare Debugger User's Guide* shows how to operate the MetaWare debugger targeting ARC processors.

This guide is intended for professional programmers who are familiar with debugging concepts and have some experience working with GUI-based tools. This manual assumes you are familiar with the following:

- C/C++ and/or assembly programming
- Basic debugging concepts
- Standard GUI (graphical user interface) elements

This book contains the following topics:

- [Getting Started](#)
- [Debugging a Program](#)
- [Debugging a Target](#)
- [CMPD Extensions](#)
- [Command Features](#)
- [Options and Toggles](#)
- [Commands Reference](#)
- [Multi-Tasked RTOS Applications](#)
- [S-Record Files](#)

About the Screen Shots

The screen shots for this manual are from the MetaWare debugger hosted on Microsoft Windows. If you are hosting the debugger on a platform other than Windows, window title bars and other system-dependent elements might look slightly different from those in this manual. However, MetaWare debugger windows have the same MetaWare debugger features, in the same order, regardless of the host platform.

If you are debugging code for a target other than the one shown in the screen shots, some windows (such as the **Registers** window) look different from those in this manual. However, the windows on your screen will be appropriate for your target system.

If your installation includes Eclipse-based IDE from ARC, the debugger displays are integrated into the IDE. The information they contain is the same. For information on working with the IDE, see the online help for the IDE.

Where to Go for More Information

Check the release note or readme file for information that was not available at the time this guide was published.

The online help system contains information about using debugger windows, setting options, and completing debugging tasks. It also provides access to manuals for the other tools in the toolkit. You can access the online help by selecting **Help | Help** from the main menu.

Look in the *MetaWare C/C++ Programmer's Guide* section "Where to go for more information" for references to the following:

- Other toolset documentation
- C and C++ programming documents
- Processor-specific documents
- Specifications and ABI documents

Document Conventions

Notes

Notes point out important information.

NOTE Names of command-line options are case-sensitive.

Cautions

Cautions tell you about commands or procedures that could have unexpected or undesirable side effects or could be dangerous to your files or your hardware.

CAUTION Comments in assembly code can cause the preprocessor to fail if they contain C preprocessing tokens such as **#if** or **#end**, C comment delimiters, or invalid C tokens.

Tips

Tips give you information to help you use the debugger more effectively.

TIP You can use CTRL+F or CTRL+B to page forward or backward in many windows.

Toolbar Buttons

You can choose (in **Tools | Debugger Preferences**) whether you want toolbar buttons displayed as icons, text, or both (the default). This guide uses the button text to refer to toolbar buttons. For example, instructions might say “Click **Src fwd**,” even though you have the **Src fwd** button displayed as an icon only.

Path Names

In path names, the term *install_dir* represents the location of your main toolset installation directory, typically C:\ARC\MetaWare\arc.

A forward slash (/) is used in path names. If you are using the debugger on Windows, substitute a backslash (\) in path names.

Chapter 2 — Getting Started

In This Chapter

- [Starting the Debugger](#)
- [Starting Your Debug Session](#)
- [Using the GUI](#)
- [Working with Debug Projects](#)
- [Getting Help](#)
- [Preparing Programs for Debugging](#)
- [Exiting the Debugger](#)

Starting the Debugger

This section describes how to start the debugger using Windows or the command line.

Starting the Windows-Hosted Debugger

If you are using the debugger on Microsoft Windows, you can start the debugger from the Windows **Start** menu, or using the IDE.

After the GUI launches, you can select the application to debug and specify other options. See [Starting Your Debug Session](#) on page 16.

NOTE Launching the debugger using the **Start** menu depends on the path specified during installation.

Starting the Debugger Using a Command Prompt

You can also start the debugger from a command prompt on Windows, UNIX, or Linux. Use the following command syntax to start the debugger:

The debugger executable for ARC is `mdb.exe` on Windows and `mdb` on UNIX or Linux. The command line is as follows:

`mdb [arguments] [-- myprog_args]`

The debugger for ARC defaults to ARC 600. For other ARC targets, you must specify the [-core*](#) option and the [-arc*](#) option (ARCompact targets) or [-a4](#) option (ARCtangent-A4) for your specific ARC processor.

- *arguments* can be any of the following:
 - the name of the program you want to debug; the default is `a.out`.
 - the command-line options and toggles for the debugger. See [Command-Line Option Reference](#) and [Toggle Reference](#) in [Options and Toggles](#) on page 273.

TIP The easiest way to specify options is using the `iss_config.arg` file that ARChitect places in the `/build` directory; see [Using Argument Files](#) on page 15.

- arguments to the program you are debugging. Any arguments that the debugger does not recognize as debugger controls are passed to the program you are debugging.

CAUTION If you specify an argument to the program you are debugging that is identical to a debugger control, the debugger does not pass that argument to the program you are debugging. Be particularly careful when using arguments that start with a dash (-). Consider using the `-- myprog_args` form to pass arguments to your program.

- *-- myprog_args* are arguments you designate as arguments to the program you are debugging by preceding them with two dashes (--) and a space. The debugger discards both dashes and treats all subsequent arguments on that command line as arguments to the program you are debugging.

Using Argument Files

You can package debugger options and file names in an argument file, an ASCII file that contains a list of arguments. This is sometimes called a response file or an @ file. In the argument file, line boundaries are interpreted as whitespace.

You specify the argument-file name, prefixed by @, when invoking the debugger on the command line. The arguments in the argument file are interpreted as if they appeared on the command line.

Example Command Line Using an Argument File

For example, the following command compiles `my_prog.exe` with the options and arguments in defined during the ARChitect build in `iss_config.arg` (assuming the files are located in the same directory):

```
mdb my_prog.exe @iss_config.arg
```

Example 1 Debugger Command-Line Invocations

The following command lines invoke GUI debugger for compiled application `my_prog`. To invoke the command-line debugger, add option `-cl`.

<code>mdb my_prog</code>	//ARC 600
<code>mdb -a4 my_prog</code>	//ARCTangent-A4, core version 8
<code>mdb -a5 my_prog</code>	//ARCTangent-A5, core version 3
<code>mdb -a5 -core2 my_prog</code>	//ARCTangent-A5, core version 2
<code>mdb -a6 my_prog</code>	//ARC 600
<code>mdb -a7 my_prog</code>	//ARC 700, core version 1
<code>mdb -a7 my_prog -core2</code>	//ARC 700, core version 2

Example 2 Debugger Command-Line Invocation with Extension Option

The following command line invokes the GUI debugger for ARCTangent-A5, core version 3, with compiled program `my_prog` and the barrel-shifter extension option.

```
mdb my_prog -Xbarrel_shifter
```

Example 3 Invocation with Extension Option and Arguments to the Application Being Debugged

The following command line invokes the GUI debugger for ARCTangent-A5, core version 3, with compiled program `my_prog` and the barrel shifter extension option; here the first `-Xbarrel_shifter` is consumed by the debugger as a debugger option, and the second is passed as an argument to `my_prog`.

```
mdb my_prog -Xbarrel_shifter -- -abc -xyz -Xbarrel_shifter
```

The program `my_prog` receives the arguments `-abc`, `-xyz`, and `-Xbarrel_shifter`.

Starting the Command-Line-Only Debugger

You can also use the debugger in command-line-only mode. You can use the command-line-only debugger for preparing scripts to perform automated testing, or on computers that do not support a GUI.

To start the debugger in command-line-only mode, launch the debugger from a command prompt with option `-cl`.

For example, to debug program `my_prog`, change to the directory where `my_prog` is located, and run one of the following commands:

```
mdb -cl my_prog
```

The debugger starts, creates and initializes a process for *my_prog*, displays information about options and about the process, and presents an `mdb>` command prompt. Here you can enter debugging commands from the MetaWare debugger command language. For a list of commands, see [Commands Reference](#) on page 231.

About the Driver Configuration File

The debugger has a driver configuration file, *install_dir/bin/mdb.cnf*, where you can change the debugger's default behavior. See [Modifying the Driver Configuration File](#) on page 227 for more information.

Starting Your Debug Session

When you launch the GUI debugger, it displays the **Debug a process or processes** dialog:

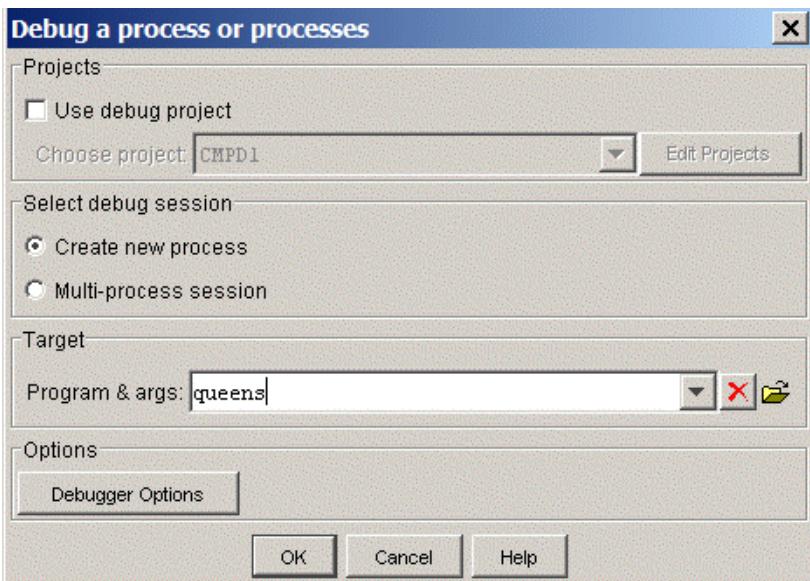


Figure 1 Debug a Process or Processes

- Under **Target**, enter the application you want to debug or browse to it. You can type program arguments and additional debugger command-line arguments after the path and filename.
- Select **Use debug project** to select or create your own project for the debug session. To create a new project, see [Working with Debug Projects](#) on page 41. If you leave **Use debug project** deselected, the debugger creates a directory named `.sc.project` in the directory where you launch the debugger. All debugger configuration files are placed in the `.sc.project` directory.
- To select options for your debug session, click **Debugger Options**. If you are starting a multi-process session, select **Multi-process session** and click the **CMPD Options** button.

When you finish making your selections, click **OK** to launch your debug session.

Debugger Options

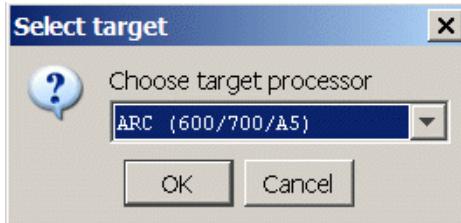
When you use the debugger GUI, the most important options for your target processor are set by default. This section describes how to review or change the options.

If you prefer to specify GUI options in two tabbed dialogs instead of a single tree view, specify the Java flag: **-joff=tree_structured_options** when launching the debugger.

This section lists some of the types of options available, which vary by target processor. Note that you must exit and restart the debugger for options to take effect.

Select Target

If the debugger cannot determine the target processor for which the application specified in **Program & args**: was compiled, the **Select target** dialog appears when you click **Debugger Options**. Choose your target processor from the drop-down list and click **OK**.



Program Options

Depending on your target processor, the **Program Options** tree node offers options such as the following.

Program & args

Enter the full path to the program you want to debug in the **Program & args** field or click the **Browse** button to locate the program you want to debug. You can type program arguments and additional debugger command-line arguments after the path and filename.

Source path

Set the directory where your source file is located in the **Source Path** field or click **Browse** to browse to the source directory.

Directory translation

Set the directory translation if your source tree is not the same as when the program was compiled. Enter a sequence of *old,new* pairs where *new* replaces *old* if *old* is a prefix of the directory of the file when compiled. Separate multiple pairs with a semicolon (Windows) or colon (UNIX and Linux).

Automatically execute to main if function main exists

If this option is checked, the debugger runs to the **main()** function on startup and waits there.

Include local symbols from ELF symbol table

If this option is checked, the debugger uses local symbol information from the ELF symbol table; otherwise it uses only global symbols. If you compiled with **-g** (for debugging information) you do not need to change this option.

Program is already present; don't download

This option prevents the program from being downloaded to the target. Check this option if your program is already present on the target, for example, if you are debugging a program in ROM. If the processor is running, the debugger attempts to stop it before downloading.

Verify program downloaded successfully

If this option is checked, the debugger re-reads memory after download to verify that the download succeeded. You can use this option if you suspect memory problems. Note that re-reading memory from the target takes time.

Show register differences while instruction stepping

When instruction single-stepping, the debugger inspects the value of all registers before and after stepping, and shows you registers that have changed. This is useful but increases register traffic with the target. If the I/O to and from the target is slow, you can uncheck this option to reduce traffic.

Cache target memory

If this option is checked, the debugger does not read the same target memory twice between execution events. This caching reduces traffic to the target. Uncheck this option if you have volatile memory that you expect to change each time it is read.

Read readonly from executable

If you check this option, the debugger does not read code or read-only data from target memory; instead, it reads it from the executable file. This reduces traffic with the target. If your program is overwriting read-only memory, debugger displays such as the disassembly window might be inaccurate. Leave this option unchecked if you are concerned that your program might be overwriting read-only memory.

Prefer software breakpoints

When this option is checked, the debugger tries to set a software breakpoint unless it is unable to write memory (e.g., ROM), in which case it tries to set a hardware breakpoint. Deselecting this box causes the debugger to try hardware breakpoints first. Hardware typically has a limited number of hardware breakpoints, and by default the debugger tries to set a software breakpoint.

Show possible breakpoints in source

If you check this option, the **Source** window shows a dot next to every line of source where you can set a breakpoint. Deselect to hide the dots.

Restore breakpoints from last run of same program

If you check this option, the debugger attempts to restore breakpoints from the last time you ran your program using the debugger. Breakpoint locations might appear to shift if you modify and reload the program.

RTOS Selection

If your application runs on an embedded RTOS, select the correct RTOS so that the debugger is RTOS-aware.

Command-Line Options

Depending on your target processor, the **Command-Line Options** tree node or tab might offer the following options.

- In the **Toggles to turn on** and **Toggles to turn off** fields, enter any toggles to enable or disable. Enter only the toggle name, without **-on** or **-off**. See [Toggle Reference](#) on page 298 for a listing of all toggles.
- In the **Command-line options** field, enter command-line options. See [Command-Line Option Reference](#) on page 274 for a listing of command-line options.
- Select **Enable command logging** and type or browse to the path and file name where you want the debugger to save the log file.
- Check **Enable profiling window** to enable profiling in debugger windows.

Note This option has effect only if the program was compiled with profiling information enabled.

GUI Options

Depending on your target processor, the **GUI Options** tree node offers options such as the following.

- Choose a font size for debugger windows using the **Font size** drop-down menu.
- Enter a **Maximum aggregate display size**. The debugger displays the contents of aggregates (arrays and **structs**) up to a certain size. This parameter specifies that size; the default is 64 K. If you have a slow communications link to the target, you might wish to reduce the size.
- **Reuse existing displays**: Some windows open other windows; for example, double-clicking in the **Call Stack** window can open other windows to show source or disassembly for a given stack position. Select **Reuse existing displays** to reuse an existing **Source** or **Disassembly** window instead of opening a new one. This helps prevent excessive windows in the debugger.

Semantic Inspection

Semantic inspection is a way of inspecting the state of your program and generating information the debugger can use. Semantic-inspection DLLs might be supplied with the debugger or your target processor; see the *MetaWare Debugger Extensions Guide* for information on adding your own semantic-inspection DLL using the debugger's semantic-inspection API.

To have the debugger load one or more semantic-inspection DLLs using the GUI, specify their

filenames and paths or click the **Browse**  button to locate them.

Specifying GUI Options for ARC Targets

In addition to the generic debugger options described in the previous topics, you can set options specific to ARC processors.

Target Selection

Click **Target selection** in the tree view to display the target-selection dialog. Select one of the following from the **Select the target:** drop-down list.

- **MetaWare ARC simulator** (Instruction Set Simulator)
- **Hardware**
- **ARC600 cycle-accurate simulator (CAS)**
- **ARC700 cycle-accurate simulator (CAS)**

The options you can select in the dialogs change depending on the target you select.

Target Selection, MetaWare ARC Simulator

The following settings are mandatory, unless marked optional.

- Select your ARC processor and make processor-specific settings:
 - For ARC 700, optionally enable exceptions, MMU, SIMD media extensions, and recursive stepping of delay slots (execute delay slot when single-stepping an instruction with a delay slot).
 - For ARC 600, optionally enable an MPU.
 - For ARCTangent-A5, choose a core version from the drop-down list.
- (Optional) **Gather stats on: Instruction counts**

If this option is checked, the simulator counts the number of times an instruction at each address is executed. This allows useful profiling, but consumes memory and simulator execution time.

- (Optional) **Gather stats on: Killed instructions**

If this option is checked, the simulator counts the number of times an instruction at each address is not executed due to the settings of flags. This allows useful profiling, but consumes memory and simulator execution time.

Target Selection, Hardware

Target hardware can be an ARCan gel board or custom hardware.

Hardware Connection

Choose a hardware connection type. The available options change depending on your choice.

- **General hardware connection** lets you connect using a parallel port or DLL.
- **Ashling Opella JTAG for ARC** lets you connect using an Ashling Opella emulator over JTAG. For more information, see [Setting up an Ashling Opella Emulator](#) on page 149.
- **Remote ARC server via TCP/IP (SCIT)** lets you connect to a remote ARC processor using MetaWare debugger Interface Transport (SCIT). For more information, see [Debugging Remotely with SCIT](#) on page 118.

General Hardware Connection

Optional: If you have written a DLL interface to your target, select **Execute programs using your DLL interface to ARC** and type or browse to the location of your DLL. For information on writing a target DLL using the debugger's Target Access Interface, see the *MetaWare Debugger Extensions Guide*.

You can set the following parallel port options:

- Specify the correct parallel port for communicating to the ARC processor in the **Parallel port address for I/O** field.
- Specify the **Number of times to retry on the parallel port** if the connection fails.
- (Optional) If you are using a JTAG connection, select **Parallel port is being used in JTAG mode**.
- (Optional) Select **Use Fujitsu fast serial host interface** to connect using the host communications protocol defined for Fujitsu's 5-wire serial interface communications module.

Ashling Opella JTAG for ARC

Specify the following to set up communications using Opella:

- In the **Opella ARC communication DLL** field, enter or browse to the Opella DLL.
- Under **Opella ARC JTAG Frequency**, click a radio button to select your Opella driver version and select the **JTAG frequency** from the drop-down list.

Remote ARC server via TCP/IP (SCIT)

- Specify the **TCP/IP address of the SCIT server** (`localhost` is permissible)
- Specify the **TCP/IP port of the SCIT server**

Blast the FPGA

When you connect to an FPGA such as ARCan gel using any connection type, you can specify a "blast" file to configure the FPGA. The blast file (typically ending in `.xbf`) contains the ARC configuration. This file is downloaded to the FPGA to turn it into your ARC processor. To specify your blast file, select **Blast the FPGA** and enter or browse to your blast file.

Target Selection, ARC600 Cycle-Accurate Simulator (CAS)

Use the radio buttons to select the way you want to connect to and use the CAS. After you select a radio button, the remaining options in the dialog change to offer settings for the way you have chosen.

For more information about the selections you can make here, see the *ARC 600 CAS Cycle Accurate Simulator User's Guide*.

Target Selection, ARC700 Cycle-Accurate Simulator (CAS)

Use the radio buttons to select the way you want to connect to and use the CAS. After you select a radio button, the remaining options in the dialog change to offer settings for the way you have chosen.

For more information about the selections you can make here, see the *ARC 700 CAS Cycle Accurate Simulator User's Guide*.

Simulator Extensions

Click the + next to **Simulator Extensions** to expand the nodes for extensions to the instruction-set simulator (for the cycle-accurate simulator, use **Target Selection**). The options available in the associated dialogs depend on the target processor selected. If the options you want are greyed out, make sure you have selected the **MetaWare ARC Simulator** in **Target selection**.

Simulator Extensions, Instruction Extensions

Optionally select any number of extension instructions supported by your processor.

Simulator Extensions, DSP Instructions

Optionally select any number of DSP extension instructions supported by your processor.

Simulator Extensions, DSP Memory

- (Optional) Select the **Scratch RAM extension** radio button if your processor has scratch RAM
 - (Optional) Check the **Sliding pointers** box if your processor supports sliding pointers.
- (Optional) Select the **XY memory** radio button if your processor has XY memory.
 - Select values for **Size**, **Banks**, and **Bus width**.
 - (Optional) Map X and Y memory to any specific addresses you require.

Simulator Extensions, Memory Extensions

NOTE Memory extension options are not available for the Cycle-Accurate Simulator or hardware.

- (Optional) Add **load/store RAM** to the simulator by specifying the size and base address of your load/store RAM in the respective fields.
- (Optional) **Add additional memory for the simulator** by specifying the low and high addresses of memory to the **First addr** and **Last addr** fields. Memory need not be contiguous.
- (Optional) **Initialize memory to 0xdead_beef**

By default, the simulator initializes its memory to 0xdead_beef to help trap errors. You might wish to uncheck this option if have a large amount of memory, to avoid initialization and host memory-page allocation.

Simulator Extensions, User Extensions

If you have implemented ISS extensions in your own DLL, type or browse to the path to each DLL in a **Simulator extension** field. For more information on implementing ISS extensions, see the *MetaWare Debugger Extensions Guide*.

Simulator Extensions, Cache Simulation

- (Optional) Check **Instruction cache** to enable the ARC **Instruction cache** window and include the **ichit** and **icmiss** registers in the **Registers** window. Specify the size, line size, ways, and repeating algorithm (**Rep alg**) for your cache (see your processor documentation for permissible values).
- (Optional) Check **Data cache** to enable the ARC **Data cache** window and include the **dchit** and **dcmiss** registers in the **Registers** window. Specify the size, line size, ways, and repeating algorithm (**Rep alg**) for your cache (see your processor documentation for permissible values).
- (Optional) Check Implement cache RAMs if your processor uses cache RAMs, where cache data can be different from memory data.

Simulator Extensions, Floating Point

For ARC 600 or ARC 700, optionally enable single-precision or double-precision floating point and enter or browse to the location of the EIA DLLs.

Simulator Extensions, Interrupts

- (Optional) **Allow interrupts 16-31**

If this option is checked, the simulator allows interrupts 16 through 31 in addition to the default 0 through 15.

- (Optional) **Interrupt on Bad instruction**

Upon encountering an invalid instruction, the simulator either halts, or, if this box is checked, takes the standard ARC interrupt.

- (Optional) **Interrupt on Bad memory access**

Upon encountering a bad memory access, the simulator either halts, or, if this box is checked, takes the standard ARC interrupt.

Simulator Extensions, Terminal/COMM Simulation

(Optional) Use the radio button to use a **Terminal Simulator** or **COMM Port Simulator**.

Terminal Simulator adds a terminal window for console I/O to and from your application.

COMM Port Simulator (Windows hosts only) connects the debugger to one of the COMM ports on the host computer so that the debugger's ISS can access external hardware.

- If you select **Terminal Simulator**, use the check boxes and drop-down lists to specify the following for your terminal simulator:
 - Base address. Default is 0xfc1000. ARC VUARTs are located at $0xfc1000 + 256*N$ for the N th device.
 - COMM port number
 - Accept the **default interrupt vector** or select your own.
 - Check **Use default receive timeout** to use the default timeout (200 ms) when waiting to receive a character, or select your own.
 - Check **Use default send timeout** to use the default timeout (200 ms) when waiting to send a character, or select your own.
 - (Optional) Log I/O. Select **Write log file** to log all outbound I/O. By default, the log goes to the terminal-simulator window (windows hosts) or to **stdout** (UNIX and Linux hosts). You can enter a filename to send the log to a file or enter **stdout** to send the log to **stdout** (the

- default on UNIX and Linux hosts). On Windows, check **Omit generating window** to send the I/O directly to the log file instead of generating a terminal-simulator window.
- (Optional) Specify input file. Check **Read input file** and type or navigate to a file to read as input rather than the terminal window.
 - If you select **COMM Port Simulator**, use the check boxes and drop-down lists to specify the following for your COMM-port simulator:
 - (Optional) Specify the **Receive buffer size** (in bytes).
 - (Optional) Specify **Baud rate**. Select a value at or below the maximum supported by your hardware.
 - (Optional) Specify the **Stop bits**.
 - (Optional) Specify the **Word length** (data bits: ASCII is 7; binary is 8).
 - (Optional) Specify the **Transmit buffer size** (in bytes).
 - (Optional) Specify the **Parity**.
 - (Optional) Specify the **Flow control**.

To specify more than one terminal simulator or COMM-port simulator, specify the properties of each additional simulator on the command line using **-simextp**. You can use the **-simextp=install_dir/bin/termsim** or **-simextp=install_dir/bin/commsim** entry in the **Debugger Options:** field at the bottom of the dialog as an example.

Initialization

- (Optional) To use a chip-initialization file to set up your processor, check the box and enter or browse to your file.
- (ARC 700 only) Optionally check **Pre-initialize low memory exception vectors** to pre-initialize exception vectors in low memory. Then if you do not provide a handler for an exception, the exception stops the processor and the debugger informs you that you hit an exception unintentionally.
- (Optional) Enter the total **Memory Size** or select it from the drop-down list.

Breakpoints/Stepping

- (Optional) For ARC 700, check **Enable breakpoints in user mode** to enable setting of breakpoints when the processor is in user mode.
- (Optional) When **Prefer software breakpoints** is checked, the debugger tries to set a software breakpoint unless it is unable to write memory (e.g., ROM), in which case it tries to set a hardware breakpoint. Deselecting this box causes the debugger to try hardware breakpoints first.

NOTE This option is the same as **Prefer software breakpoints** in **Program Options**.

- (Optional) Uncheck **Single step instructions, rather than cycles** to use cycle stepping instead of instruction stepping.

NOTE This option is not available for the MetaWare debugger ARC Simulator (ISS).

Peripheral Displays

Optionally select the appropriate box to display each UART and VMAC on your system.

AUX Registers

Optionally check the box to map auxiliary registers to memory locations. Then enter the base address of auxiliary-register memory and how many auxiliary registers.

Side-Effect Registers

Optionally enter core and auxiliary register numbers to indicate that those registers have side effects and the debugger should not read them.

Target Reset

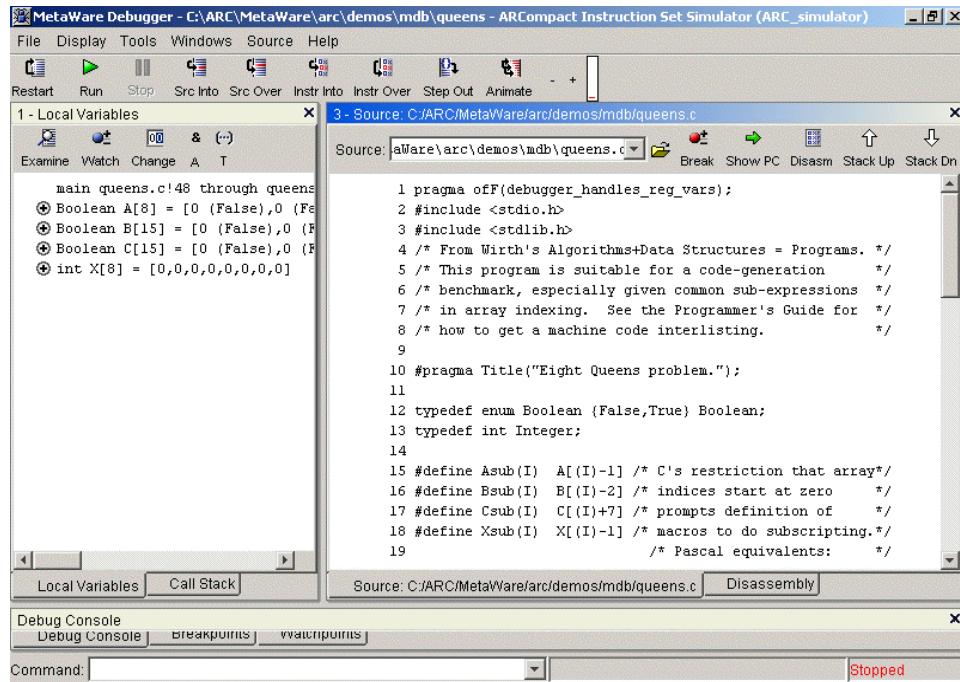
NOTE This option is not available for the MetaWare debugger ARC Simulator (ISS).

Select **Reset target upon restart** to reset the target every time you restart the debugger (full download of code and new run of any chipinit file).

Using the GUI

After you make your selections in the **Debug a process or processes** dialog and click **OK**, the complete GUI functionality becomes available.

[Figure 2 The MetaWare Debugger GUI Desktop](#) presents an overview of the GUI.



[Figure 2 The MetaWare Debugger GUI Desktop](#)

Title Bar

The title bar shows the location of the program being debugged and the target system name.



Figure 3 Debugger Title Bar

Main Menu Bar

The main menu provides access to the features of the debugger. This section lists the menu items and their functions.



Figure 4 Main Menu Bar

File

Restart restarts the program and resets the target. Same function as **Restart** toolbar button.

New Process or Project opens the **Debug a process or processes** dialog to open a different program for debugging in a new process.

Clone Desktop creates a new debugger frame in which you can open windows. This feature is helpful for debugging programs with multiple tasks/threads or processes.

Clone Desktop And Its Frames creates a new debugger frame with the current windows already open. This feature is helpful for debugging programs with multiple tasks/threads or processes.

Print prints the contents of the window currently in focus.

Print Preview shows a preview of printed output from the window currently in focus.

Exit Debugger shuts down all processes and closes the debugger.

Display

Note **Display** menu items vary by target processor and compilation and debugging options.
The items presented here are examples of commonly found items.

Registers opens the **Registers** window.

Aux registers opens the **Auxiliary registers** window for ARC targets.

Disassembly opens the **Disassembly** window.

Instruction history opens the **Instruction history -- Simulator trace** window.

Call Stack opens the **Call Stack** window.

Hardware stack opens the **Memory: Hardware stack** window.

Global Variables opens the **Global Variables** window.

Global Functions opens the **Functions** window.

Locals opens the **Local Variables** window.

Source opens the **Source** window.

Source files opens the **Source files** window.

Breakpoints opens the **Breakpoints** window.

Watchpoints opens the **Watchpoints** window.

Memory opens the **Memory** window.

Examine opens the **Examine** window.

Modules opens the **Modules** window.

Profiling opens the **Profiling** window.

Program output opens the **Program output** window.

Log opens the **Log** window.

Tools

Set Breakpoint opens the **Set Breakpoint** dialog.

Set Watchpoint opens the **Set Watchpoint** dialog.

Search Memory opens the **Memory Search** dialog.

File-Memory-Fill Operation opens the **File/Memory/Fill operation** dialog.

Stepping Toolbar opens a new **Stepping** toolbar in a separate floating window.

Window Toolbar opens a toolbar that contains a button for each window. type. The toolbar appears as a floating window.

Debugger Options opens the **Debugger Options** dialog.

Multi-Process Debug Options opens the **CMPD Setup** dialog if you are using multi-process debugging. The menu item is disabled in single-process debugging.

Edit Projects opens the **Edit Projects** dialog.

Debugger Preferences opens the **Debugger Preferences** dialog.

Windows

Debug Console opens the **Debug Console**.

Show All restores all minimized debugger windows.

Hide All minimizes all debugger windows.

Refresh All refreshes the content of all open and visible windows.

Cascade orders windows from largest to smallest.

Tile Horizontal fits windows by stretching them horizontally.

Tile Vertical fits windows by stretching them vertically.

Open Window Layout browses the file system for a MetaWare debugger window layout file (.windows).

Open Recent Window Layout provides a list of the recently used window layouts. When you select a window layout from the list, the debugger changes appearance and opens windows based on the layout in the file. The current window layout is shown with a check mark.

Save Window Layout saves the current window layout.

Save Window Layout As saves the current window layout to a new name or location.

Dynamic Menu

The dynamic menu (**Source** in [Figure 4 Main Menu Bar](#) on page 25) appears between the **Windows** and **Help** menus and has the name of the window currently in focus. The dynamic menu shows the menu items on the popup menu that you can obtain by right-clicking on the window currently in focus.

Help

Help opens the online help system in a new window.

Readme Opens the readme file in a text editor or the **Debug Console**.

About opens the **About** window containing the version information. Click anywhere in the **About** window to close it.

Depending on the application being debugged, other menu items may be added for features such as threads or RTOSs.

Using the Debug Console

The **Debug Console** is a special window that displays the command output from the debugger. You can scroll through a running list of commands and view output. The **Debug Console** displays the following information:

- A list of commands that have been executed with their command-line syntax
- A breakpoint or watchpoint that has been set or reached
- Errors that are encountered
- Current location of program counter and status
- Current thread if your application is multi-threaded
- If the program has ended

By default, the **Debug Console** is docked at the bottom of the screen. To open or close the **Debug Console**, select **Windows | Debug Console**.

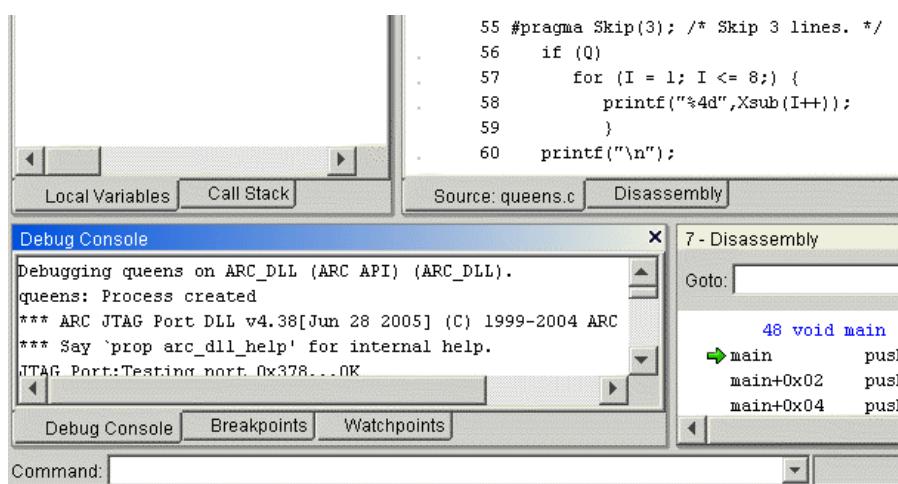


Figure 5 Debug Console

Error and Condition Reporting

Program errors that your program sends to `stderr` appear in the [Program Output Window](#).

Some debugger errors appear in a text box, and some in the **Debug Console**. To send all errors to the **Debug Console**, turn off toggle [error_box](#).

Some conditions of interest (such as an inability to set a hardware watchpoint) are reported in a text box in addition to the **Debug Console**. Turn off toggle [note_box](#) to send the reports only to the **Debug Console**.

Viewing and Saving a Command Log

Select **Display | Log** to view a log of all executed commands (including internal commands) and any output.

Saving a log may be useful for technical-support requests. To save a log, do the following:

1. Right-click in the **Log** window and select **Save to File**.
2. Specify a directory and filename, and click **Save**.

Setting Preferences

Use the **Debugger Preferences** dialog to configure user interface settings, edit the keyboard shortcuts for stepping actions, and create and edit keyboard shortcuts for debugger commands and user-defined macros. To access **Debugger Preferences**, select **Tools | Debugger Preferences**.

You can use the **Debugger Preferences** dialog to set the following properties:

- Desktop: Choose the type of desktop model you prefer and set options specific to the desktop model. Choices are MDI, SDI, and sash panel.

Desktop Model	Description	Available Options
MDI	<p>Displays appear as internal windows to the debugger desktop window. Internal windows can be minimized, maximized, and restored. You can move a display to another internal window by dragging and dropping the tab to a new location.</p> <p>Menu items for minimizing, restoring, cascading, and tiling are added to the Windows menu. Windows can also be docked to stick in a position. Docked windows cannot be minimized.</p>	<ul style="list-style-type: none">• Save window layout on exit• Show close buttons on tabs (roll mouse over to show buttons)• Snap-to-Grid Windows• Docking Windows

Desktop Model	Description	Available Options
Sash panel	Displays appear as panels inside the debugger main window. You can resize a panel by dragging its divider. You can move a panel to another location by dragging the title bar or tab to a new location and dropping it.	<ul style="list-style-type: none"> • Save window layout on exit • Show close buttons on tabs (roll mouse over to show buttons) • One-Touch Expandable Dividers
SDI	Displays appear as individual windows on the desktop of the operating system. You can move a display to another window by dragging the tab to a new location and dropping it.	<ul style="list-style-type: none"> • Save window layout on exit • Show close buttons on tabs (roll mouse over to show buttons)

- Appearance: Set font size, toolbar button style (text, icons, or both), and look and feel.
 - To change font size for all debugger fonts, select a font size from the **Font Size:** list and click **OK** or **Apply**.
 - To change toolbar style to icon only, text only, or both, select one of the radio buttons and click **OK** or **Apply**.
 - To change look and feel, select the radio button for the GUI style you prefer and click **OK** or **Apply**.
- [Stepping Keys](#): Change the keyboard shortcuts for stepping actions.
- [Command and Macro Keys](#): Add, change, or remove keyboard shortcuts for user-defined macros and debugger commands.

When you click **OK** or **Apply**, the preferences take effect immediately; you do not need to restart the debugger. The preferences persist for every project for the current user.

Stepping Keys

The **Debugger Preferences** dialog allows you to modify the keyboard shortcuts, or key bindings, for the standard debugger stepping actions. This feature allows you to optionally use a Visual Studio-style key layout, or define your own custom keyboard shortcuts. You can modify key combinations for stepping actions using one or more command keys (CTRL, SHIFT, or ALT) and one function key (F4 through F12).

The shortcut keys currently assigned to the stepping commands appear as tool tips when place your mouse cursor over the buttons on the stepping toolbar. When you make changes to the keyboard shortcuts for the stepping commands, the tool tips for the buttons update automatically to reflect your changes.



Using Visual Studio Debugger Keyboard Shortcuts

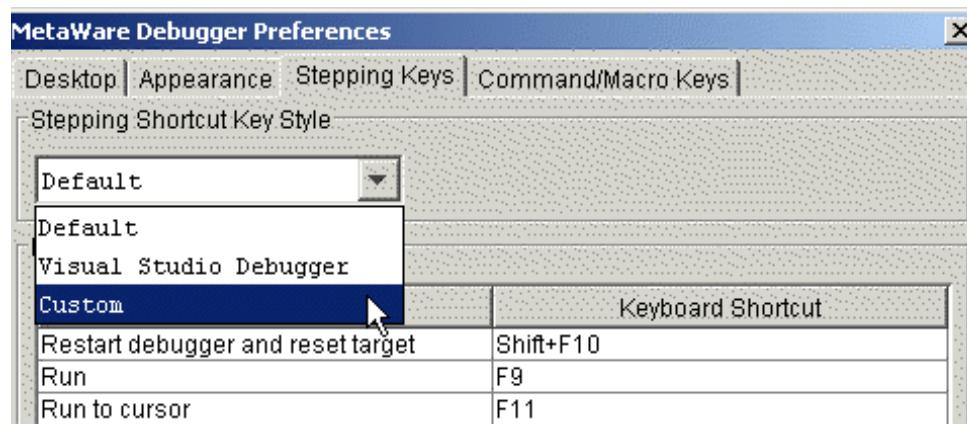
To use a shortcut-key mapping that emulates the stepping shortcuts of the Visual Studio debugger, proceed as follows:

1. Select **Tools | Debugger Preferences** and select the **Stepping Keys** tab.
2. From the **Stepping Shortcut Key Style** drop-down list, select **Visual Studio Debugger**.
3. Click **OK** to save or click **Apply** to commit the changes without closing the dialog.

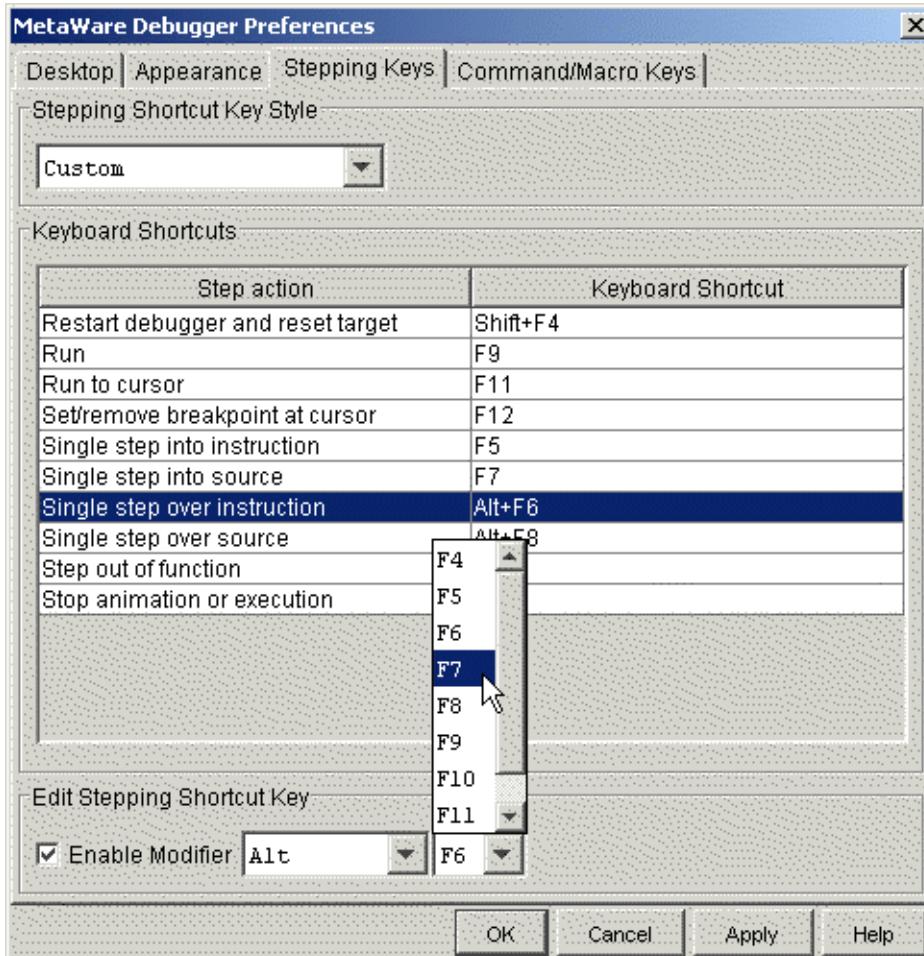
The preferences take effect immediately and persist for every project for the current user.

Editing Keyboard Shortcuts for Stepping

1. Select **Tools | Debugger Preferences** and click the **Stepping Keys** tab.
2. From the **Stepping Shortcut Key Style** drop-down list, select **Custom**.



3. Select a stepping action from the table and select a function key for it from the function-key list under **Edit Stepping Shortcut Key**.



4. To use a command-key combination (CTRL, SHIFT, or ALT), select **Enable Modifier** and select a command key from the drop-down list. Alternately, if you want to use a function key without a command-key combination, deselect **Enable Modifier**.
5. Click **OK** to save or click **Apply** to commit the changes without closing the dialog.

Note No two keyboard shortcuts may share the same key sequence.

The preferences take effect immediately and persist for every project for the current user.

Command and Macro Keys

The **Debugger Preferences** dialog allows you to create, change, and remove keyboard shortcuts, or key bindings, for debugger commands and macros. Using this feature, you can assign keys to easily access macros and debugger commands. You can assign a combination consisting of one or more command keys (CTRL, ALT, or SHIFT) and a numeric key (0-9).

- For a list of available debugger commands, see [Chapter 7 — Commands Reference](#) on page 231. Or type **help all** in the **Command** field.
- For information on creating debugger macros, see [Creating and Using Debugger Macros](#) on page 224.

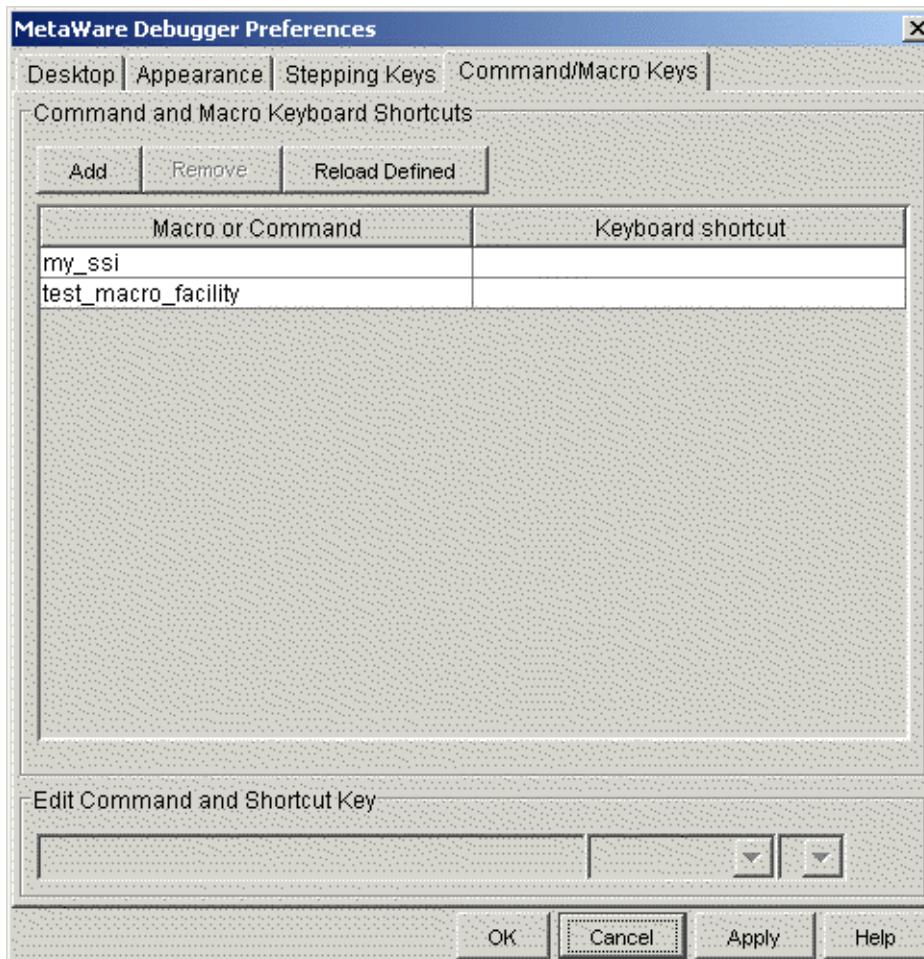
Adding a Keyboard Shortcut for a Macro

1. Define or load your macro definitions into the debugger (see [Creating and Using Debugger Macros](#) on page 224 for more information).
2. Select **Tools | Debugger Preferences**.
3. Click the **Command/Macro Keys** tab.

The macros you defined or loaded into the debugger appear in the table.

NOTE Macros can only be defined and undefined by following the instructions in [Creating and Using Debugger Macros](#) on page 224.

4. Select a macro from the table.



Optional: You can modify the command string in the **Edit Command and Shortcut Key** field. This is useful if your macro accepts arguments and you want to have keyboard shortcuts for

different variants. After you modify the command string, you can reload the original version in the display by clicking the **Reload Defined** button (same as closing and opening the display).

5. Select a command key or combination from the drop-down list of command keys.
6. Select a numeric key from the drop-down list of numeric keys.

NOTE You must assign one or more command keys (CTRL, ALT, or SHIFT) plus a numeric key (0-9) to save the shortcut. No two shortcuts may share the same key sequence.

7. Click **OK** to save or click **Apply** to commit the changes without closing the dialog.

The preferences take effect immediately and persist for every project for the current user.

Adding a Keyboard Shortcut for a Command

You can assign a keyboard shortcut to a debugger command. For a list of commands you can use, see [Chapter 7 — Commands Reference](#) on page 231 or type **help all** in the **Command** field.

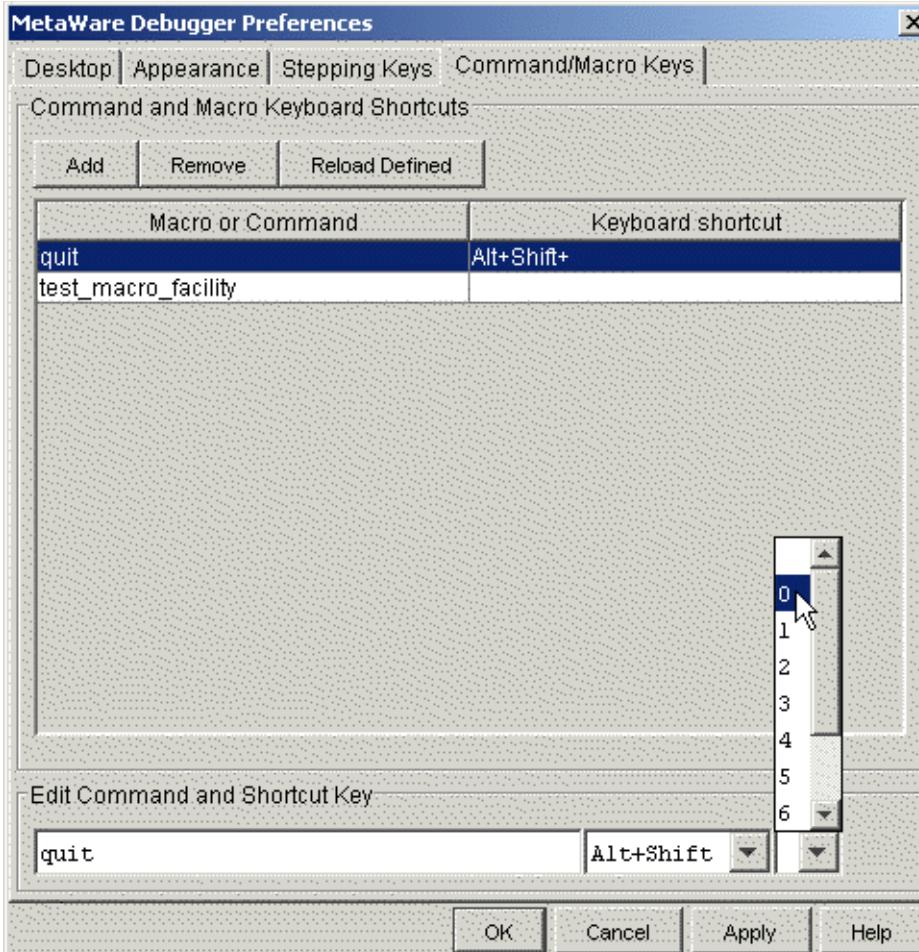
1. Select **Tools | Debugger Preferences**.
2. Click the **Command/Macro Keys** tab.
3. Click **Add**.
4. Type the name of a command in the text field.

NOTE For a list of commands you can enter, see [Chapter 7 — Commands Reference](#) on page 231 or type **help all** in the **Command** field.

CAUTION You must enter the name of a command. If you leave the default <undefined>, the changes are lost when you close the **Debugger Preferences** dialog.

5. Select a command key or combination from the drop-down list of command keys.
6. Select a numeric key from the drop-down list of numeric keys.

NOTE You must assign one or more command keys (CTRL, ALT, or SHIFT) plus a numeric key (0-9) to save the shortcut. No two shortcuts may share the same key sequence.



7. Click **OK** to save or click **Apply** to commit the changes without closing the dialog.

The preferences take effect immediately and persist for every project for the current user.

Editing a Keyboard Shortcut

1. Select **Tools | Debugger Preferences**.
2. Click the **Command/Macro Keys** tab.
3. Select an entry from the table.
4. Change the name or keys as desired.

NOTE You must assign one or more command keys (CTRL, ALT, or SHIFT) plus a numeric key (0-9) to save the shortcut. No two shortcuts may share the same key sequence.

5. Click **OK** to save or click **Apply** to commit the changes without closing the dialog.

The preferences take effect immediately and persist for every project for the current user.

Removing a Keyboard Shortcut

1. Select Tools | Debugger Preferences.
2. Click the Command/Macro Keys tab.
3. Select an entry from the table.
4. Click Remove.

The preferences take effect immediately and persist for every project for the current user.

Command Input Fields

The debugger GUI has a **Command:** input field at the bottom of its desktop.



You can also add a **Command:** input field to any window by right-clicking within the window and selecting Toolbars | Command line from the menu that appears. For a list of the commands you can use, enter [help](#) or see [Commands Reference](#) on page 231.

Note Many input fields offer a list of choices to complete a command. You can select one of the elements from the list. Click **OK** when finished to commit your selection, or click **Cancel** to abort.

Status Bar

The status bar displays messages and events from all debugger windows.

Running/Stopped Indicator

The lower right of the main debugger window shows the following status:

- **Stopped** when the debugger is idle.
- **Run #** when the process is executing
- **Animating** when the process is in animation mode

Drag-and-Drop Data Tabs

Data you display using the **Display** menu appears initially in a new window. Tabs are used if more than one display is present in a panel or window. You can drag a tab into other windows. Each window can contain multiple tabs.

In [Figure 6 Window Containing Several Tabs](#), the user has dragged and dropped tabs to arrange a **Source** tab, a **Disassembly** tab, and a **Registers** tab in one window, with the **Source** tab currently in focus.

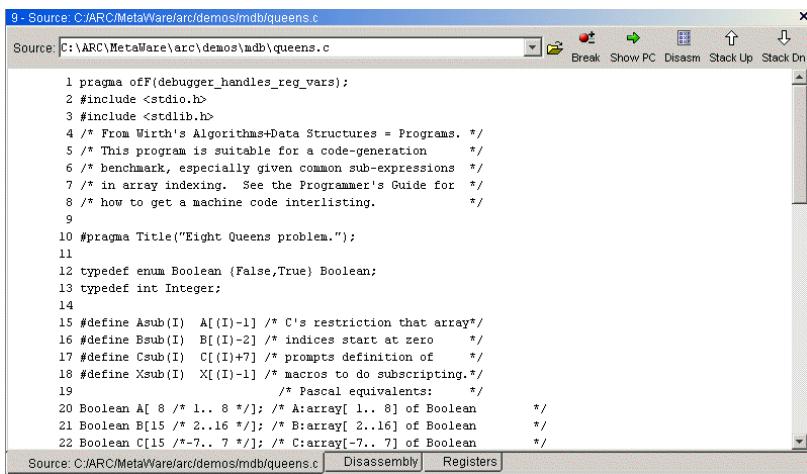


Figure 6 Window Containing Several Tabs

To move a data tab out into its own separate window, click and drag the tab to an empty area in the debugger desktop. You can also right-click within the data tab and select **Tear Off Display**.

Moving a Display Using Its Tab

You can drag and drop a tab in all desktop models. You can also drag and drop a panel using its title bar in the sash-panel desktop model.

To move a display:

1. Left-click on the tab, but do not release the mouse button yet.
2. Drag the tab to the desired window or location.

You can add it as another tab, or split a panel horizontally or vertically. One of the following icons appears in place of your mouse cursor where you drag the tab:



Move the mouse cursor to the center of a panel or window and release. The display appears tabbed inside the panel or window.



You cannot drop the display. Releasing the left mouse button has no effect.



Move the mouse cursor to the top of the panel or window and release. The display creates a new horizontal split above the panel.



Move the mouse cursor to the bottom of the panel or window and release. The display creates a new horizontal split below the panel.

- ◆ Move the mouse cursor to the right edge of the panel or window and release. The display creates a new vertical split on the right side of the panel.
- ◆ Move the mouse cursor to the left edge of the panel or window and release. The display creates a new vertical split on the left side of the panel.

Closing a Tab

To remove a tab from a window without closing the entire window, right click on the tab and select **Close**, or click the close button on the tab (move your cursor over the tab name to reveal the close button, as shown in [Figure 7 Tab Close Button](#)).

Note Close buttons on tabs are optional. To disable, select **Tools | Debugger Preferences**. Select the **Desktop** tab, then uncheck **Show close button on tabs**.

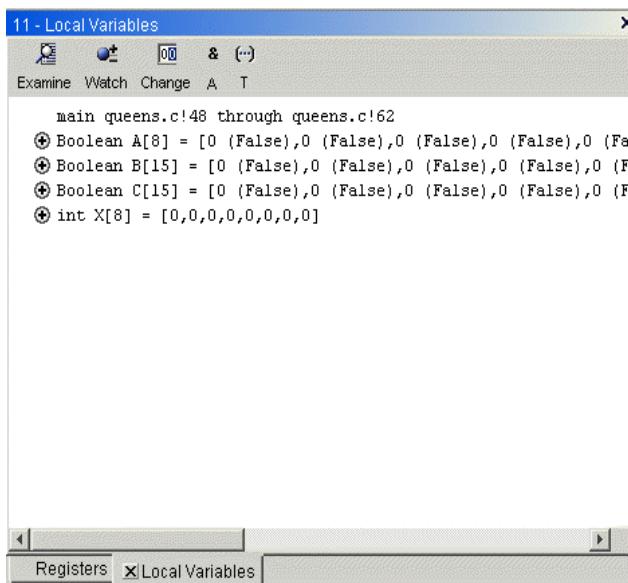


Figure 7 Tab Close Button

Docking Windows

(MDI Desktop Model Only)

When using the MDI desktop model, you can dock windows by dragging them to any edge of the debugger desktop. To remove the window from the dock, click the title bar and drag the window away.

Note You can move a window outside the desktop by right-clicking within the window and choosing **Detach Frame** from the popup menu that appears.

You can choose which edges of the debugger desktop are available for docking by selecting **Tools | Debugger Preferences** and checking the docking regions you want enabled. The preferences take effect immediately and persist for every project for the current user.

NOTE Docked windows cannot be minimized.

To change desktop model, select **Tools | Debugger Preferences**, and select the **Desktop** tab.

Snap-to-Grid Windows

(MDI Desktop Model Only)

When you are using the MDI desktop model, internal windows snap to an invisible grid to make them easy to arrange. To unsnap, select **Tools | Debugger Preferences** and uncheck **Snap to grid**. The preferences take effect immediately and persist for every project for the current user.

To change desktop model, select **Tools | Debugger Preferences**, and select the **Desktop** tab.

Saving Window Layouts

By default, the debugger saves the window layout to the current project when it closes. You can disable autosave by selecting **Tools | Debugger Preferences** and unchecking **Save window layout on exit**.

If you have configured the windows in an arrangement you like but plan to make changes before exiting the debugger, you can save the window layout to load again later. To save a window layout separately from the current project, select **Windows | Save Window Layout As....**

To load a saved window layout, select **Windows | Open Recent Layout | <layout.windows>** or **Windows | Open Layout**.

Look-and-Feel

By default, the debugger starts with the “look-and-feel” most appropriate for the host operating system. You can select different “skins” that radically alter the appearance of the Debugger to suit your personal preference. The skins that are available in the **Look-and-feel** area of the **Appearance** tab (**Tools | Debugger Preferences**) are as follows:

- **Metal**
- **CDE/Motif**
- **Windows**
- **Plastic**
- **PlasticXP**

When you select **Plastic** or **PlasticXP**, you can choose from several color options. Select the option you prefer, click **Apply**, then **OK**. The preferences take effect immediately and persist for every project for the current user.

One-Touch Expandable Dividers

(Sash-Panel Desktop Model Only)

When you use the **Sash-panel Style** as your desktop model, you can choose **One-touch-expandable dividers**. This option shows buttons on each panel divider that allow you to minimize and maximize the panel:

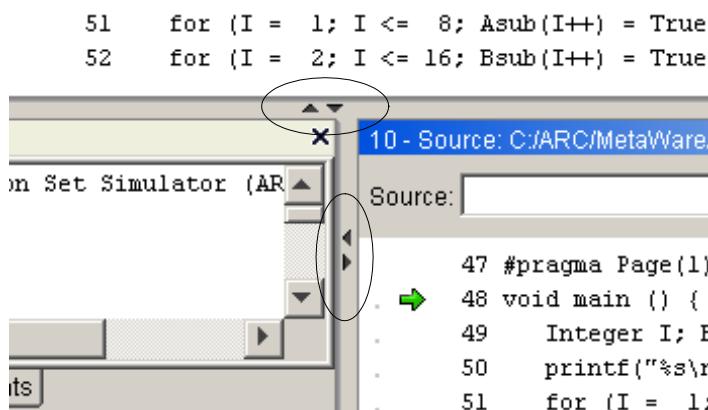


Figure 8 One-Touch Expandable Dividers

To change desktop model, or choose **One-touch-expandable dividers**, select **Tools | Debugger Preferences**, and select the **Desktop** tab.

Common Window Components

In all desktop models, most windows share certain common components depending on their functionality.

Lock/Snap Toolbar

Locking prevents a window from being updated. You can lock the display of a window at a specific location, or compare states by locking a window and comparing it to an unlocked one as you run or step your application. The lock feature works on all debugger windows.

For example, you can open a **Source** or **Disassembly** window and lock it at a location so you can view profiling statistics for specific statements or addresses. A locked **Source** or **Disassembly** window does not update to the most current program counter.

To lock a window:

1. Right-click in a window and select **Toolbars | Lock/Snap toolbar**. The **Lock/Snap** toolbar appears at the top of the window.
2. Click **Lock**. Click this button again to unlock the window.

For information on using the **Snapshot** button, see [Taking Snapshots of a Window](#) on page 113.

Right-Click Menus

Right-click menus are available in most windows and differ by content. Some windows offer profiling by right-click; see [Adding Profiling by Right-Clicking in a Window](#) on page 94.

Most right-click menus offer a choice to save the window data to a file.

Options Toolbar

The **Options** toolbar provides functions for the specific window.

To enable an options toolbar for a window:

1. Right-click in the window.
2. Select **Toolbars | Options**, or **Toolbars | window options**, where *window* is the name of the window, such as **Source** or **Registers**.

The **Options** toolbar appears at the top of the window.

To remove the options toolbar, right-click and select **Toolbars | Options**, or **Toolbars | window options** again.

For example, the **Registers** window contains buttons to perform the following operations:



- **Name:** Specify one or more specific registers to view
- **Watch:** Set a watchpoint on a register
- **Change:** Change a register's value
- **Format:** Change the radix and format (data type) of the display

TIP You can use a [-prop](#) option or [set](#) command to change the way **floats** and **doubles** are displayed. See the listings in [Chapter 8 — Options and Toggles](#) on page 273 or [Chapter 7 — Commands Reference](#) on page 231.

- **Registers per line:** Change the number of registers displayed per line
- **Register bank choice:** Select the register banks to display
- **Stack Up/Stack Dn:** Move up and down the stack

Break Buttons

After selecting a line in a display, click the **Break** button to set a breakpoint on that line:



Watch Buttons

After selecting a line in a display, click the **Watch** button to set a watchpoint on that line:



Change Buttons

Windows that let you change values (such as the **Registers**, **Memory**, and **Local** and **Global** variables

windows) have a **Change** button  that displays a **Change** dialog for an item you have selected (also available by right-clicking and selecting **Change** from the pop-up menu after selecting an item).

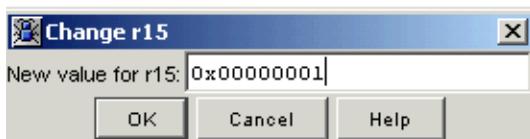


Figure 9 The Change Dialog

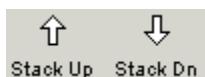
To change a value in the **Change** dialog:

1. Select the value or a portion of the value in the **New value for** field.
2. Enter the new value.
3. Click **OK**.

The **Change** dialog closes, and the new value appears in the original window (highlighted, because it has changed).

Stack Up and Down Buttons

Click **Stack Up** to move up the stack or **Stack Dn** to move down the stack.



Base Selection Boxes

Select the numeric base for the display:



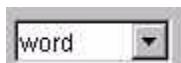
Bytes per Line Selection Boxes

Select the number of bytes to display per line:



Size Selection Boxes

Use the **Size** selection box to view your data in different-sized chunks (such as word, halfword, **long**):



Working with Debug Projects

You can create a debug project to save different settings and window arrangements for different projects. If you change settings while you are working on a project, the debugger saves your changes in the project file on exit.

NOTE You can not switch projects while debugging without exiting the debug session. However, you can copy the current project, exit the debugger without restarting, and select the project you saved.

You can also create, duplicate, and delete projects when launching the debugger by checking **Use debug project** and clicking **Edit Projects** in the **Debug a process or processes** dialog.

Creating a New Project

1. Click **Use debug project** in the **Debug a process or processes** dialog:
If the debugger is already running, select **Tools | Edit Projects**.
2. Click **Create**.
3. In the **Project name:** field, enter a name for the project. Project names cannot contain spaces.
4. Click **OK**.
The project appears in the project list.
5. Click **OK**.

The project is added to the drop-down list of projects in the **Debug a process or processes** dialog the next time you launch the debugger. All debug-project information is stored in the following directory:

Windows

C:/Documents and Settings/*username*/mdb/projects

UNIX or Linux

\$HOME/.mdb/projects

Duplicating a Project

When you duplicate a project, the project window layout and program set options are copied to the new project.

1. Click **Use debug project** in the **Debug a process or processes** dialog:
If the debugger is already running, select **Tools | Edit Projects**.
2. In the **Create/Edit Debug Projects** dialog, select a project from the list.
3. Click **Copy**.
4. In the **Destination project name:** field, enter a name for the new project. Project names cannot contain spaces.
5. Click **OK**.
6. The project appears in the project list.
7. Click **OK**.

Removing a Project

When you remove a project, all window layout files and program settings (including CMPD options) in that project directory are deleted.

1. Click **Use debug project** in the **Debug a process or processes** dialog:
If the debugger is already running, select **Tools | Edit Projects**.
2. In the **Create/Edit Projects** dialog, click the project you want to delete from the list on the right.

3. Click **Delete**. The project disappears from the project list on the right.
4. Click **OK**.

Changing the Current Project

1. Exit the debugger and launch it again
2. In the **Debug a process or processes** window, click select a project from the **Choose project:** list.
3. Click **OK** to launch the debug session.

Command Line

You can use the following command-line options to manage debug projects from the command line:

- [-noproject — Run the debugger without a debug project](#)
- [-OK — Skip the “Debug a process or processes” dialog](#)
- [-project=project_name — Specify debug project](#)

Getting Help

The debugger provides online help systems for using the debugger and for the debugger controls and commands.

Help Using the Debugger

You can access online help in the debugger GUI by selecting **Help | Help** from the main menu. To view help for a specific window, click within the window and press F1, or right-click in the window and select **Help** from the pop-up menu that appears.

Help Using Controls and Commands

If you are using the command-line-only debugger, you can view an on-screen listing of common MetaWare debugger options by invoking the debugger with option **-h**:

```
mdb -h
```

For a list of MetaWare debugger commands, enter [help](#) at the `mdb>` prompt or the **Command:** input field in the GUI. See [Commands Reference](#) on page 231. for more information about the [help](#) command.

Preparing Programs for Debugging

Compiling Programs with Debug Information

You can debug a program compiled without debug information, but the debugging operations you can perform will be limited. See [Non-Symbolic Debugging](#) on page 112 for more information.

Source-Level Debugging

To debug your program at the source level, compile it with option **-g**. Compiling with option **-g** places debug information within the file being compiled. This option also provides the linker with information needed to link the file for debugging and to place debugging information in the executable file.

Debug information, stored in the object or executable file, describes the data type of each variable and function; it also describes the correspondence between source-line numbers and executable-code addresses.

Unless you specify compiler option **-O** to enforce a certain optimization level, option **-g** turns off many of the optimizations the compiler otherwise performs.

NOTE Early variants of the ARCTangent-A5 processor have a restriction that does not allow breakpoint instructions to be placed adjacent to compare-and-branch instructions. This can cause problems when statement stepping with source code, as the debugger may attempt to place breakpoint instructions in the illegal locations. This restriction was removed in later versions of the processor.

If you are using a version of the processor that has this restriction, compile code for debugging with `toggle branch_reductions` turned off in addition to **-g**. Turning off this toggle is necessary only when compiling with **-g** for source-level debugging.

Assembly-Level Debugging

To debug your program at the assembly level, you do not need to compile with option **-g** and you do not need the source-code files. However, because of the non-linear nature of optimized code, do not compile using **-O** optimizations.

Locating Source Files

To debug your program at the source level, make sure the source-code files are accessible. They must be in the same directory location relative to the executable as they were when you (or someone) built the executable. For simplicity, we recommend building the executable in the same directory as the source-code files.

If the debugger cannot find your source-code files, you must do one of the following:

- Set `SOURCE_PATH` at the debugger command prompt or enter the following in the **Command:** input field:

```
set source_path=\\some\\directory
```

NOTE Initial GUI option dialogs can be set for source path and direct translation mechanisms, thus eliminating the necessity for sole reliance upon the command prompt.

- Replace the initial string of the source build location with the initial string of a new source location by setting the `dir_xlation` setting at the debugger command prompt or entering the following in the **Command:** input field:

```
set dir_xlation=old,new
```

Where *old* is the build location and *new* is the new location. See the listing for `set` in [Commands Reference](#) on page 231 for details on `dir_xlation`, including how to change from UNIX to Windows path specification.

- Set the environment variable `SOURCE_PATH` to point to your source-file directory before you start the debugger, or use the `set` command in the debugger.

To set `SOURCE_PATH` before invoking the debugger, type one of the following:

Windows:

```
set SOURCE_PATH=\\some\\directory
```

C shell:

```
setenv SOURCE_PATH /some\directory
```

Bourne shell:

```
SOURCE_PATH=/some\directory
```

Exiting the Debugger

Exiting the debugger terminates all processes started by the debugger, terminates the debugger, and returns to the command prompt from which you originally invoked the debugger.

Note To exit the debugger in the event of an unpredictable situation, switch to the command prompt where you originally invoked the debugger, then press CTRL+C four times.

To exit the debugger GUI, select **File | Exit Debugger** or type [exit](#) in the **Command:** input field of any debugger window and press ENTER.

To exit the command-line debugger, enter command [exit](#) or [quit](#).

Chapter 3 — Debugging a Program

In This Chapter

- [Controlling Program Execution](#)
- [Viewing Source Code](#)
- [Viewing Machine Instructions](#)
- [Debugging with Breakpoints](#)
- [Debugging with Watchpoints](#)
- [Viewing and Modifying Data](#)
- [Debugging Memory](#)
- [Simulating Instruction and Data Caches](#)
- [Execution Profiling](#)
- [Viewing Source-Code Coverage](#)
- [Viewing and Modifying Registers and Flags](#)
- [Debugging Multi-Tasked/Threaded Applications](#)
- [Non-Symbolic Debugging](#)
- [Taking Snapshots of a Window](#)

Controlling Program Execution

You can step through a program using any of the following methods:

- Use one of the stepping-related buttons on the toolbar. Place your cursor over a button for a pop-up explanation.



TIP You can change the keyboard shortcuts for the toolbar. See [Stepping Keys](#) on page 29.

- Enter a command in a **Command:** input field:



The debugger GUI has a **Command:** input field at the bottom of its desktop, and you can add a **Command:** input field to any window by right-clicking and selecting **Toolbars | Command line** from the pop-up menu. For a list of MetaWare debugger commands, enter the command [help](#) or see [Commands Reference](#) on page 231.

TIP The **Command:** input field remembers the previous commands entered. You can repeat a command by selecting it from the drop-down box, optionally editing it, and then pressing ENTER.

- In the command-line debugger, enter a command at the mdb> prompt.

CAUTION **File | Restart** terminates and reloads your program; therefore, you should not use it to restart a program that has not run to termination if terminating the program will produce disastrous results. For example, it is not a good idea to restart a program when you have attached the kernel of your operating system.

Running a Program

You can run your program using the debugger or using the simulator without the GUI.

Running a Program in the Debugger

After you start the debugger and specify a program to load, you can direct the debugger to initialize the program and do one of the following:

- Run the program at full speed.
- Run the program start-up code, then stop at the first executable source statement (the default).
- Stop in start-up code.

By default, the debugger runs the start-up code and stops at function **main()**.

Program output to `stdout` or `stderr` appears in the **Program output** window.

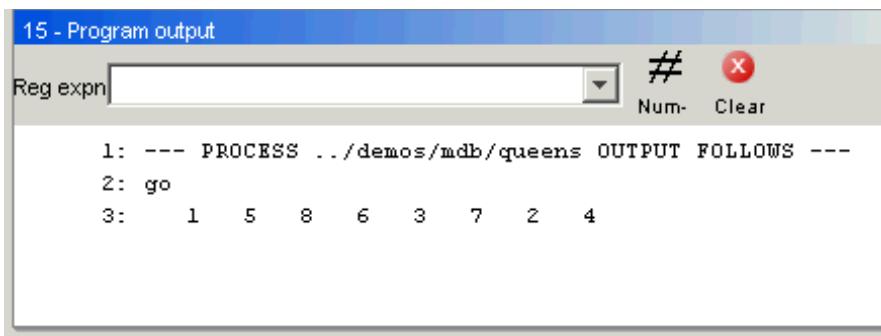


Figure 10 Program Output Window

Running a Program in Silent Mode

You can run your program with command **runarc** (ARCTangent-A4) or **runac** (ARCTangent-A5 and later). This runs an application on the debugger's instruction-set simulator without invoking the GUI or requiring user interaction, which is useful for running your application on the simulator using a script.

Example

The command line shown below runs `test.elf` on the ARC 700 ISS and passes argument `123` to the hypothetical application `test.elf`. A response from the application is also shown.

```
runac -arc700 test.elf 123
SUCCESS on test 28 seed value: 123
```

Stopping in Start-up Code

To debug start-up code, invoke the debugger with command-line option [-nogoifmain](#). To step into start-up code, Click [Instr Into].

Animating a Process

Animation is a way to execute your program continuously, one statement or instruction at a time, pausing between individual source statements or machine instructions. Animation allows you to control the speed at which the command or step is executed.

Animation stops automatically at these points:

- when execution encounters a breakpoint
- when a watchpoint's value changes
- when you click [Stop]
- when the process terminates

NOTE Animation is the only method of execution that stops automatically when the value of a software watchpoint changes.

Starting Animation

To start animating a process, click [Animate]. The [Animate] button now reads **Select...**, reminding you to click one of the stepping buttons. The stepping buttons and **Command:** input field blink until you have told the debugger how to step through your program.

- To animate the process one machine instruction at a time, click [**Instr Into**] (to step into functions) or [**Instr Over**] (to step over function calls).
- To animate the process one source-code statement at a time, click [**Src Into**] (to step into functions) or [**Src Over**] (to step over function calls).
- To animate a command, enter a debugger command in the **Command:** input field, then press ENTER to animate that command.

A counter keeps track of the number of times a machine instruction or source-code statement is animated.

Stopping Animation

To stop animation at any time, click [**Stop**] on any **Stepping** toolbar.

Adjusting Animation Speed

Click [+] or [-] on the **Stepping** toolbar to adjust animation speed: [+] to increase, [-] to decrease speed. A gauge next to the [+] and [-] buttons shows a relative approximation of the speed.

Setting Maximum Speed

You can set a maximum animation speed by specifying command-line option [-anim_scale](#) when you invoke the debugger. For more information, see [Command-Line Option Reference](#) on page 274.

Restarting a Program

To restart a program, choose **File | Restart**, or enter [restart](#) in the **Command:** input field.

CAUTION **File | Restart** terminates and reloads your program; therefore, you should not use it to restart a program that has not run to termination if terminating the program will produce disastrous results. For example, it is not a good idea to restart a program when you have attached the kernel of your operating system.

When you restart a program, the debugger retains the breakpoints and watchpoints that existed when the program last terminated — even those you set in a dynamically linked library.

Hostlink Services

A target program can send a request to a user-written service provider that is attached to the debugger. This request can take practically any form, as long as the target program and the service provider agree upon the format of the request. The request is then managed by the hostlink facility.

NOTE hostlink works no matter where the target program is being executed: hardware, the debugger's simulator, or an RTL simulator.

This service works only with programs that are built with the MetaWare C run-time library, and in which hostlink is resident.

This service allows a program to communicate arbitrary events or to send or receive information to or from the debugger host. For example, your target program could send bitmaps to a custom display generator. It also can allow the program to interrogate the value of hardware signals in an RTL simulator that is running at the same time as the program.

This interface is supplied to the debugger via a semantic-inspection interface. For more information, see the following:

- `Hostlink_service` in *install_dir*/mdb/inc/dbgaccess.h
- *MetaWare Debugger Extensions Guide*

If your MetaWare C run-time library does not contain `_user_hostlink`, you can request a source update for the hostlink library and replace your current library with the update.

An example implementation of a user hostlink service may be found in the directory *install_dir*/etc/semint/hostlink.

To view some standard user hostlink services provided by the debugger, see the header file: *install_dir*/mdb/inc/dbgh1.h.

Viewing Source Code

You can open a source window by selecting **Display | Source** if both of the following are true:

- The executable was compiled with option `-g` (debugging information).
- The debugger can locate the source-code files.

See also [Compiling Programs with Debug Information](#) on page 43 and [Locating Source Files](#) on page 44.

An indicator shows where you are in the source as you run or step your program. When the program stops at a place (such as an instruction) that is not in the source code, the **Source** window contains the message `Not stopped in known source`. This can happen, for example, when program execution is still in the start-up code, or in a library routine, or if the source was not compiled with option `-g`.

Setting or Removing Breakpoints

To set or remove a breakpoint, do any one of the following in the **Source** window:

- Double-click a line of source.
- Right-click a line of source, and then select **Break** from the pop-up menu.
- Select a line of source, and then click the **Break** button.

The left margin indicates breakpoint status:

- A solid red indicator means a breakpoint is set on the line.
- A grey indicator means the breakpoint is disabled.

Evaluating Variables

To evaluate the value of a variable: Right-click on or within one character of a variable, and then select **Evaluate** from the pop-up menu.

View Profiling Information

If the code was compiled with profiling information and your target supports profiling, you can view profiling data in the **Source** window.

1. Right-click, and then select **Profiling** from the pop-up menu.
2. Select any number of menu items from the top half of the resulting menu.

The **Source** window adds a column that contains a profile for the type of profiling you chose.

TIP Reselecting the item removes the column.

For more information, see [Adding Profiling by Right-Clicking in a Window](#) on page 94.

To view profiling information on the command line, use the **disassemble** command with the **-ctr=** option; for more information, see [Using Commands to Dump Counters and Cache Analysis](#) on page 219.

Enabling Source-Code Coverage

You enable analysis of code coverage in a **Source** window; for more information on this feature, see [Viewing Source-Code Coverage](#) on page 102.

Viewing Source at the Program Counter

To view the source line associated with the current location of the program counter, click [**Show PC**] in the **Source** window. If the source line associated with the program counter is in a file different from the one currently displayed in the **Source** window, the debugger replaces the contents of the window with the appropriate source file.

Viewing Source in a Different File

To view a source file other than the current one, enter path and file name in the **Source:** input field. You can use environment variables such as \$HOME and ~.

The debugger cannot operate on header files, source files not associated with object modules of your program, or source files compiled without debug information. You can open such files in a **Source** window, but you can do nothing with them except look at them and scroll through them.

To view a particular source-code line, in the **Source:** input field enter one of the following:

!line_num to go to a line in the current file; for example, !57.

file_name!line_num to go to a line in a different file; for example,
queens.c!44.

Using the **Source:** input field takes you only to actual lines of executable code. If the line you specify is not executable (for example, a declaration or comment), the search stops on the next line of executable code.

Troubleshooting the Source Window

This section lists common problems and possible solutions.

Problem: Source Window Not Showing Source Code

If the Source window displays `Not stopped in known source`, your source file might have been moved, renamed, or deleted from the directory in which it was compiled.

Solution

In the **Command:** field, type the following

```
set source_path=path
```

where *path* is the full path to the source file.

Example

```
set source_path=c:\ARC\MetaWare\arc\demos\
```

Problem: Cannot Set a Breakpoint

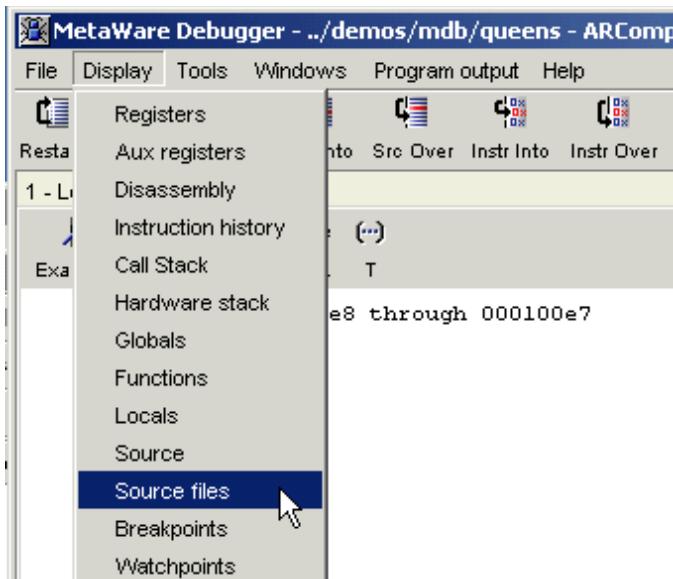
To set a breakpoint, you must first compile with debugging information (option **-g**).

Viewing Source Files

The sections below list your options for display of source files.

Listing All Source Files

To see a list of all source files for which line information exists in the currently executing module (main program or DLL), choose **Display | Source files**.



The **Source files** window shows all source files for which the debugger has line information. Files are sorted alphabetically.

Listing Selected Source Files

To list only those files whose names match a regular expression, enter the regular expression in the **Reg expn** input field.

Opening a New Source Window

To open a new **Source** window and view a file listed in the **Source Files** window, do one of the following:

- Double-click the file entry.
- Click the file entry, and then click the **[Source]** button.

Viewing Functions

To view a list of the functions contained in the source file click on the + next to the source file name.

Viewing Source for a Function on the Call Stack

To view the source code for a function on the function call stack:

1. Choose **Display | Call stack** to open the **Call Stack** window. (See [Figure 19 Call Stack Window](#) on page 76.)
2. In the **Call Stack** window, click the stack frame to select it.
3. Click **[Source]** to open a **Source** window with the source code for the function.

Unless you selected the current function (the uppermost function on the call stack), the **Source** window shows the program counter at the point where control will return to the function. The color of the program counter is different, to show that it is not at the current program-counter position.

Listing All Executable Modules

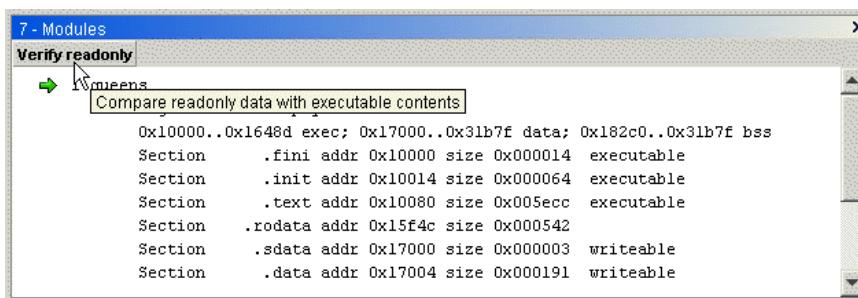
To view a list of all currently loaded software modules, including any DLLs (dynamic link libraries) or SOs (shared objects on UNIX), select **Display | Modules**.

An arrow indicates which module is currently active.

The modules' ELF sections are listed below each module.

Verifying against Memory

Click the **Verify readonly** button to check whether the code in memory matches the code in the ELF file, to ensure no corruption has occurred. The results are displayed in the **Debug Console**.



Viewing Machine Instructions

You view machine instructions in a **Disassembly** window. This window shows disassembled machine instructions ("disassembly code") for a program in order by address, starting at the current location of the program counter. Each line of text represents a single disassembled machine instruction. You can open more than one **Disassembly** window.

To open a **Disassembly** window, do either of the following:

- Choose **Display | Disassembly**.
- In the **Source** window, click [**Disassembly**] on the **Source Options** toolbar.

When you open the **Disassembly** window, it automatically shows disassembled machine instructions at the current location of the program counter.

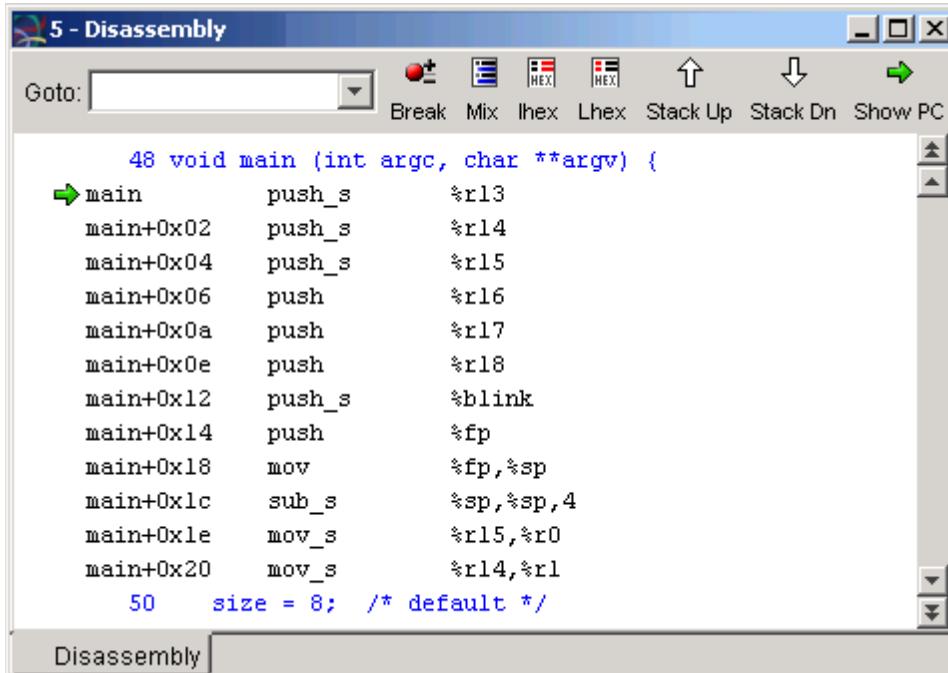
Labels of the form .0, .1, .2, ... are displayed in the left margin. Each label corresponds to a symbolic label and may be used to represent a symbol. For example, in the following sequence, the first occurrence of .0 corresponds to the `main` symbol.

```
main
main+0x04
main+0x08
main+0x0c
main+0x10
main+0x14
main+0x1c
main+0x24
main+0x28
```

```
main+0x2c
main+0x30
```

The labels change their associations when the **Disassembly** window is updated. You can use a label in a simple expression (for example, to specify the address of a breakpoint) for as long as the same values are in the window. After you close the **Disassembly** window, all associations are discarded. Indicators in the left margin show the status of the program, stack frame, and breakpoints.

[Figure 11 Disassembly Window \(ARCtangent-A5 Shown\)](#) shows the disassembly window for ARCtangent-A5. Disassembly windows for other processors differ only in the assembly language displayed.



[Figure 11 Disassembly Window \(ARCtangent-A5 Shown\)](#)

TIP You can use CTRL+F or CTRL+B to page forward or backward.

Customizing the Disassembly Window

You can alter some aspects of text display in the **Disassembly** window.

Displaying Hexadecimal Instruction Codes

By default, disassembly instructions appear as assembly mnemonics. To display hexadecimal instruction codes also, click [**Ihex**] in the **Disassembly** window.

Displaying Hexadecimal Addresses

By default, addresses in disassembly code appear as symbolic labels, usually a function name plus an offset:

```
main+0x0c
```

If you want disassembly addresses to display as absolute hexadecimal addresses instead, click [**Lhex**] in the **Disassembly** window to toggle between symbolic labels and hexadecimal addresses.

TIP To change the minimum width of hexadecimal addresses displayed, use the command `set addr_width=value`, where value is the width.

Displaying Source and Assembly Code

By default, assembly code appears intermixed with source code. Click [**Mix**] in the **Disassembly** window to toggle between intermixed source and assembly code and straight assembly code. When straight assembly code is displayed, the number of the source-code line in the source file appears to the left of the first machine instruction for that source-code line.

Viewing Disassembly Code

In the **Disassembly** window, click [**Show PC**] on the **Disassembly Options** toolbar to view code at the program counter.

To view disassembled instructions at an address different from the one currently displayed, enter the address you want to view in the **Goto:** input field. The address can be any of the following:

- a number (hexadecimal, decimal, or octal) that is the address of a machine instruction
- the name of a function
- the name of a register
- any valid C/C++ expression that indicates a location in program code

The debugger replaces the current contents of the **Disassembly** window with the disassembly code for machine instructions starting at the new address, and adds the new address to the **Goto:** drop-down list.

Navigating the Stack Frame

To scroll up or down the call stack, click the **Stack Up** or **Stack Dn** button.



The **Stack Dn** button moves to the function that called the currently displayed function. The **Stack Up** button moves to the function that was called by the currently displayed function (click **Stack Up** only when the currently displayed function is not at the top of the stack frame).

Viewing Disassembly for a Source Code Line

You can examine disassembly code for any source-code file, provided the debugger can find it, and provided it was compiled with debugging information turned on (option **-g**).

To view the disassembled machine instructions associated with a particular source-code line:

1. Open a **Disassembly** window and make sure intermixed source is turned on (click [**Mix**] if necessary).
2. In the **Goto:** input field, enter one of the following:

<code>!line_num</code>	to go to a line in the current file
<code>file_name!line_num</code>	to go to a line in a different file

Using the **Goto:** input field takes you only to actual lines of executable code. If the line you specify is not executable (for example, a declaration or comment), the search stops on the next line of executable code.

Working with Instruction History

You can use the **Instruction history -- Simulator trace** window to view a history of instructions, step forwards or backwards through the instruction history list, view disassembly, and view registers for the instruction.

Note To view a history of instructions, you must have a simulator or hardware with trace capability (not available on all targets).

To open an **Instruction history -- Simulator trace** window, choose **Display | Instruction history**.



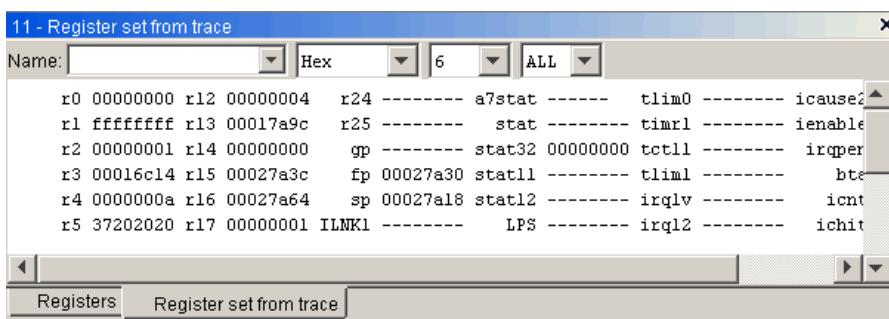
Note Menu choices vary by target processor and other options.

The **Instruction history -- Simulator trace** window lists events that occurred during program execution. The events typically include executed instructions, memory reads and writes, and register writes. The exact type of events and maximum number of events are target-dependent.

Stepping through a History of Instructions

The step buttons work the same way as the stepping toolbar buttons (see [Controlling Program Execution](#) on page 48).

Double click an instruction to view an **Instruction history disassembly** window and a **Register set from trace** window that shows the value of the registers.



Register entries marked with “-----” are registers for which no history exists.

NOTE The **Register set from trace** window appears only if the history of instructions includes register write events.

Filtering and Buffers

The **Filter** menu (right-click) allows you to include or exclude certain trace items. Generally, all are included by default, although the provider of the trace items may choose to have some excluded by default. Press the **NoFil** button to ignore all filtering; press it again to restore filtering.

MetaWare debugger simulators maintain trace buffers. For most targets, you can configure the size of the buffer and the nature of its contents with the **Capture menu** selection (right-click).

Restricting Displayed Instructions

You can restrict the instructions displayed in the **Instruction history -- Simulator trace** window using a regular expression or a Boolean expression.

To restrict the displayed instructions using a regular expression, do the following:

1. Click the **Find** drop-down list.
2. Select **Reg expn**.
3. Enter a regular expression (such as `*r3*` or `*add*`) in the **Reg expn** input field.
The expression is added to the **Reg expn** drop-down list so you can use it again. See [Using Regular Expressions](#) on page 221.
4. Click the down or up arrow (**Next** or **Prev**) or the **Match** icon.

NOTE We recommend that you remove the regular expression when you are no longer using this window.

Regular expression search might take additional time due to increased number of instruction history entries.

To restrict displayed instructions by fields using a Boolean expression, do the following:

1. Click the **Show Field Values** button [...] to show the available fields, such as `[addr=0x10084 pc=0x10084]`.
2. Click the **Find** drop-down list.
3. Select **Bool expn**.
4. Enter your Boolean expression.

A Boolean expression is a C expression whose constituents include the fields in the trace. Each field has a 32-bit value you can use in the expression. You name a field in the Boolean expression by prefixing it with a '\$'; for example `$addr` names the field `addr`. An instruction matches if the Boolean expression evaluates to non-zero for that item *and* the expression contains at least one field present in that item. For example, if you look for items satisfying `$addr >= 0x20000`, an item must contain an `addr` field to match; register writes, for example, would be excluded.

Here are some examples of Boolean expressions:

```
$addr >= 0x2000  
$reg = 13 && $value < 10  
$reg == 13 ? $value == 10 : $value = 1
```

5. Click the down or up arrow (**Next** or **Prev**) or the **Match** icon.

The display is restricted to the instructions that match your criteria.

Marking Lines to Return to

To mark lines with a name and return to them later:

1. Type the name in the **Marks** text field and press ENTER.
2. Select a line and either right-click **Mark** or press the **Mark** button. That line is marked with the given name and an indicator appears to its left.

Later you can return to the marked line by name. Select the **Marks** drop-down and pick the name you want. The debugger returns to that line automatically.

You can refer to a field of a marked line in a Boolean expression with the syntax $\$M\F where M is your mark name and F the field name. Suppose you have marked some line as `init`, and that line has the fields `reg` and `addr`. You can then search on Boolean expressions such as the following:

```
$addr == $init$addr  
$reg == $init$reg || $addr > $init$addr
```

Saving to a File

To save the window contents to a file:

1. Right-click within the window and select **Save to File** from the popup menu.
2. Save to the file and directory of your choice in the resulting dialog.

Configuring Trace Capture

To configure or apply trace-capture settings, right-click within the **Instruction history -- Simulator trace** window and select **Capture | Configure capture**.

The **Trace capture** dialog appears.

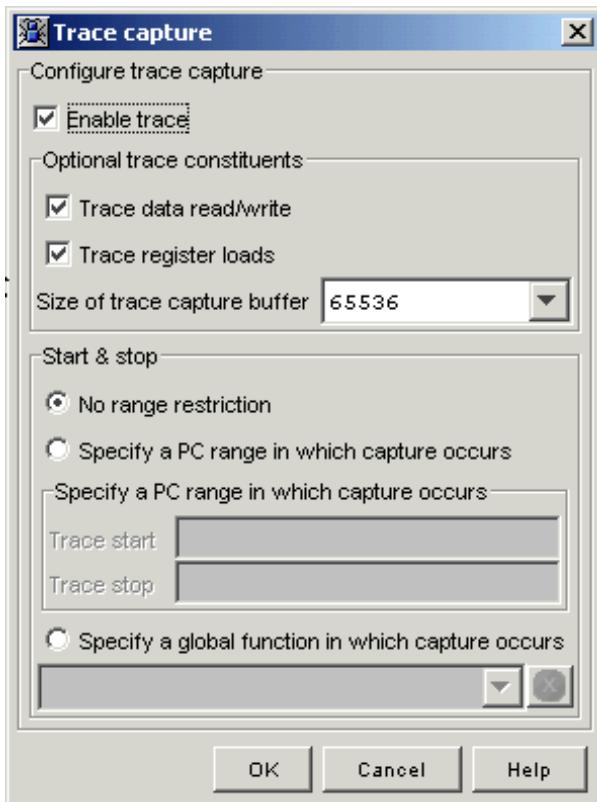


Figure 12 Trace Configuration

Here you can configure when and what you want to capture. You can save space by selecting for trace capture only those constituents that are of interest to you. You can also set the size of the trace-capture buffer using the drop-down list.

You can specify whether you want to capture the trace of a certain PC range, capture the trace of a specific global function, or capture the trace with no range restriction (the default).

After configuring the trace, save your settings by clicking **OK**. The results appear in the **Instruction history -- Simulator trace** window the next time you run or step your program. The **Instruction history -- Simulator trace** window lets you save the trace to a file.

To capture a trace using settings configured in a previous session, choose **Apply capture settings**.

To run the process again without trace capture:

1. Click **Configure capture**
2. Deselect **Enable trace**
3. Click **Save**.
4. Click **File | Restart**.

Debugging with Breakpoints

A breakpoint stops execution of a process so you can perform any of the following:

- determine the execution flow
- check the values of variables
- modify the execution

If you step over a function that contains a breakpoint, execution stops on the breakpoint, even if this means stepping into the function.

Debugging with breakpoints during multi-process debugging requires additional considerations; see [Using Breakpoints and Watchpoints](#) on page 169.

Hardware and Software Breakpoints

The debugger can set hardware or software breakpoints for most targets. Hardware breakpoints are indicated with an ‘H’ in the **Breakpoints** window or command-line listing. Note that resources for hardware breakpoints might be limited.

Whether hardware breakpoints or software breakpoints are the default is target-specific, but software breakpoints are generally the default. You can change the default in **Tools | Debugger Options** or **Tools | Target-Specific Options**, or change the setting of toggle `prefer_soft_bp`.

Related Topics

- [break](#) command
- toggle [prefer_soft_bp](#)

Setting Breakpoints

To be able to set a breakpoint in a source-code file on a line or on a non-global function, you must compile the source with debug information (option `-g`) turned on. You can set breakpoints on global functions without compiling with `-g`.

TIP Compiling with option `-g0` provides line numbers for breakpoints without affecting optimizations like option `-g` does.

You can set a breakpoint:

- on a line in a source file
- on a single disassembled machine instruction
- at an address
- on a function
- at the current location of the program counter

You can set breakpoints in a **Source** or **Disassembly** window, or by using the **Set Breakpoint** dialog.

Setting a Breakpoint in a Source or Disassembly Window

You can set a simple breakpoint any of the following ways:

- Double-click a line of code in a **Source** or **Disassembly** window. (Double-click again to remove the breakpoint.)
- Right-click a line of code and select **Break F12** from the popup menu.
- Click a line of code to select it, then click **[Break]** on the window’s toolbar. (Click **[Break]** again to remove the breakpoint.)

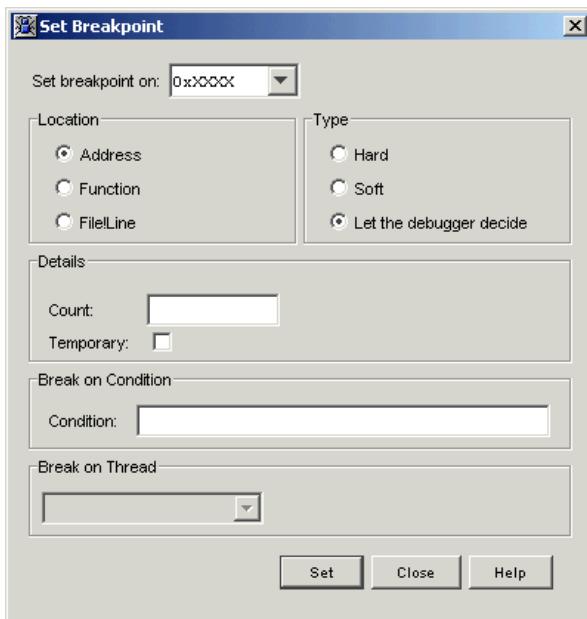
- Enter a **break** command. See [Commands Reference](#) on page 231.

When you set a breakpoint on a line of code, a breakpoint symbol appears to the left of the line.

Setting a Breakpoint Using the Set Breakpoint Dialog

You can use the **Set Breakpoint** dialog to set both simple and complex breakpoints. To open the **Set Breakpoint** dialog, do one of the following:

- Choose **Tools | Set Breakpoint**.
- Click **[Set]** in the **Breakpoints** window.



To set a breakpoint in the dialog, do the following:

- Click a radio button to choose how you want to specify the breakpoint location: by address, by function, or by line number.
- Enter the location in the **Set Breakpoint On:** input field:

Radio Button	Type of Breakpoint Location	Example
address	Hexadecimal address Symbolic label	0x34 main+0x57
function	Function name (enter without parentheses)	printf
file!line	Current source file: "!" followed by a line number Other source file: file name followed by "!" followed by a line number	!44 queens.c! 44

- Choose a hardware or software breakpoint, or let the debugger set the default based on the availability of hardware breakpoints and the default setting in the **Tools | Debugger Options** or **Tools | Target-specific options** (on the command line, toggle [prefer_soft_bp](#)).

4. To make the breakpoint complex, do one or more of the following (see [Behavior of Mixed Breakpoints](#) on page 64 to learn the effects of combining breakpoint types):

- Counted: In the **Count:** input field, enter the number of times you want the breakpoint to skip before it stops execution.
- Temporary: Click the **temporary** check box to break only once.
- Conditional: In the **Condition:** input field, enter a conditional expression.

5. Specify a thread if you want a thread-specific breakpoint.

6. Click [OK].

Setting Task- or Thread-Specific Breakpoints

In a multi-tasked application or application with multiple threads, you can set a breakpoint on one or more specific tasks, meaning execution stops only if the task encounters that breakpoint. See [Using Task- or Thread-Specific Breakpoints](#) on page 109.

For steps to set a CMPD task-specific breakpoint, see [Setting Thread-Specific Breakpoints for a Process](#) on page 167.

Types of Complex Breakpoints

A simple breakpoint just stops execution; the process remains stopped until you explicitly start it again. Complex breakpoints do more. [Table 1 Complex Breakpoints](#) lists the complex breakpoints you can set and describes their effects.

Table 1 Complex Breakpoints

Type of Breakpoint	Effect
Temporary	Stops execution once, and then is deleted.
Counted	Allows control to run past the tagged address a specified number of times before it stops execution.
Conditional	Becomes enabled when a conditional expression associated with it evaluates to true.
Command-execution	Triggers the execution of a specified MetaWare debugger command each time control passes it, whether or not execution stops on it.
task-specific	Stops execution only when encountered by a specified task/thread in a multi-tasked or multi-threaded application.
Mixed	Combines two or more breakpoint types.

Debugging with Breakpoints

A *mixed breakpoint* has any combination of temporary, counted, conditional, and command-execution attributes. [Table 2 Behavior of Mixed Breakpoints](#) shows how mixed breakpoints behave.

Table 2 Behavior of Mixed Breakpoints

If the Breakpoint Is . . .				This is What Happens . . .
Temporary	Counted	Conditional	Command Execution	
Yes	Yes	Yes	No	The condition is tested when the count is reached. As soon as the condition is met, execution stops and the breakpoint is deleted.
			No	As soon as the count is reached, execution stops, the command executes, and the breakpoint is deleted.
			No	Execution stops when the count is reached, then the breakpoint is deleted.
	Yes	Yes	Yes	As soon as the condition is met, execution stops, the command executes, and the breakpoint is deleted.
			No	Execution stops when the condition is met, then the breakpoint is deleted.
			Yes	The command executes when execution stops on the breakpoint, then the breakpoint is deleted.
	No	Yes	Yes	The command is executed every time control passes the breakpoint, whether or not execution stops.
			No	The condition is tested when the count is reached. As soon as the condition is met, execution stops.
			Yes	The command is executed every time control passes the breakpoint, whether or not execution stops.

Working with the Breakpoints Window

You can use the **Breakpoints** window to view breakpoints, enable and disable breakpoints, delete breakpoints, and view source code at a breakpoint location.

Viewing a List of Current Breakpoints

GUI

To see a list of current breakpoints, open a **Breakpoints** window by choosing **Display | Breakpoints**.



Figure 13 Breakpoints Window

Each line in the listing shows the following information for a particular breakpoint:

- the address of the instruction or source line
- the name and line number of the source file (if available)
- the text of the instruction or source line (if available)
- where appropriate, the type (temporary, counted, conditional, command-execution, task-specific)
- where appropriate, ‘H’ for hardware breakpoints

Note If the debugger cannot find a source file associated with the breakpoint, or if the breakpoint is set on a machine instruction that exists outside the realm of known source files, the **Breakpoints** window lists the breakpoint’s address and symbolic label, and the disassembly code for the instruction.

If a breakpoint is task-specific, additional task-related information appears in the breakpoint line. For more information, see [Using Task- or Thread-Specific Breakpoints](#) on page 109.

Command Line

Use the **break** command. See [Commands Reference](#) on page 231 for details.

Enabling and Disabling Breakpoints

You can turn a breakpoint off and on without deleting it. To disable or enable a breakpoint in the **Breakpoints** window:

1. Click the breakpoint to select it.
2. Click [**Enable/disable**].

When a breakpoint is disabled, a “breakpoint disabled” symbol appears to the left of it.

To toggle the enabled/disabled status of all currently set breakpoints, click [**All**] to the right of [**Enable/disable**].

You can also use the [**enable break**](#) command to enable some or all of the currently disabled breakpoints, or the [**disable break**](#) command to disable some or all currently enabled breakpoints. See [Commands Reference](#) on page 231 for details.

Deleting Breakpoints

To delete a breakpoint, do one of the following:

- In the **Source** or **Disassembly** window, double-click the line on which the breakpoint is set. (Double-click again to reset the breakpoint.)
- In the **Breakpoints** window, click a breakpoint to select it, then click **[Remove]** in the **Breakpoint Options** toolbar. You can also double-click the line in the **Breakpoints** window.

To delete all currently set breakpoints from within the **Breakpoints** window, click **[All]** to the right of **[Remove]**.

You can also use the [delete break](#) command to delete either a single breakpoint or all current breakpoints. See [Commands Reference](#) on page 231 for details.

Viewing Source Code at a Breakpoint

To open a **Source** window and view source code starting with the line on which the breakpoint is set:

1. Click a line in the **Breakpoints** window to select the breakpoint.
2. Click **[Source]** on the **Breakpoint Options** toolbar.

Debugging with Watchpoints

Watchpoints monitor a region of memory and pause execution if any changes occur in the contents of that region.

Use watchpoints to find statements that unexpectedly change your program's code or data (for example, an array index that addresses elements beyond the array's allocated space). If the value of a variable, array, structure, or other data item in your program is changing unexpectedly, you can set a watchpoint on the region of memory containing that data item, then animate your program. When a watchpoint's value changes, the program stops and the debugger updates all values in open windows.

Each watchpoint is assigned an index number when you create it; these numbers are successive integers, starting with 1 (one).

To speed up the debugger's response time when you are watching a region of memory, close any open windows that update frequently (for example, the **Disassembly**, **Memory**, **Registers**, and **Source** windows). This saves the overhead of updating these windows as your program executes.

Hardware and Software Watchpoints

The debugger can set hardware watchpoints on systems that support hardware watchpoints, and can set software watchpoints on any system; the instruction-set simulator can simulate software watchpoints even if your hardware build does not include them. Software breakpoints are not as reliable as hardware watchpoints, but can be very useful for resolving problems not visible elsewhere.

If your target system supports hardware watchpoints, the debugger uses those instead of software watchpoints wherever it can. Usually there are certain system constraints on what a hardware watchpoint can watch, and the debugger does not set a hardware watchpoint on any variable that passes into and out of scope.

The advantage of hardware watchpoints is that they halt your process whether you are animating it or running it, whereas software watchpoints halt the process only if you animate it.

An 'H' appears to left of all hardware watchpoints in the **Watchpoints** window.

You can use watchpoints with the debugger's animation feature, described in [Animating a Process](#) on page 49.

For example, first set a watchpoint on a register, and then set animation for single stepping, either SSI, SSO, ISI, or ISO.

Setting Watchpoints

You can set a watchpoint on a global or local variable, a register, an addressable expression, or any region of memory specified by a start address and a length (in bytes).

Setting a Watchpoint on a Variable

To set a watchpoint on a variable in the **Local Variables** window, the **Global Variables** window, or the **Examine** window:

1. Click the line containing the variable to select it.

2. Click the  icon or right-click and select **Watch** from the pop-up menu.

Note No watchpoint symbol appears next to variables that have watchpoints set on them. To determine whether a variable is being watched, open the **Watchpoints** window.

You can also set a watchpoint on a variable using the [watch](#) command. See [Commands Reference](#) on page 231 for details.

Setting a Watchpoint on a Register

1. Choose **Display | Registers** to open a **Registers** window.
2. Click a register to select it. The line containing the register is highlighted.
3. Click **[Watch]**. A watchpoint indicator appears to the left of the highlighted line.

Note The simulator does not allow watchpoints to be set on the `icnt` register. When using a simulator, set a property to tell the simulator how many instructions to execute before stopping, as described in [Stopping Execution by Instruction Count](#) on page 117.

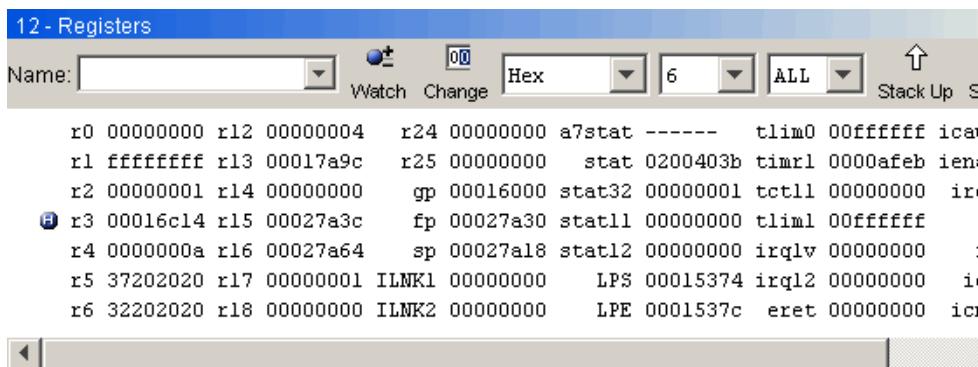


Figure 14 Register Watchpoint

If multiple registers are displayed on the line with the watchpoint indicator, you cannot tell which register(s) are being watched. For example, when viewing [Figure 14 Register Watchpoint](#), it is impossible to determine whether the watchpoint is actually set on `r3`, `r15`, `fp`, or `sem`. To verify the

watchpoint is where you want it, change **Registers per line** to 1 (one), so only one register is listed on each line.

You can also set a watchpoint on a register by using the [watchreg](#) command. See [Commands Reference](#) on page 231 for details.

Removing a Watchpoint on a Register

To remove a register watchpoint, click the register then click **[Watch]**.

Setting a Watchpoint in the Set Watchpoint Dialog

Use the **Set Watchpoint** dialog to set a watchpoint on a range of memory not necessarily associated with an identifier, or to set conditional watchpoints. A conditional watchpoint becomes enabled when a conditional expression associated with it evaluates to true.

To open the **Set Watchpoint** dialog, do one of the following:

- Choose **Tools | Set Watchpoint**.
- Click **[Set]** in the **Watchpoints** window.

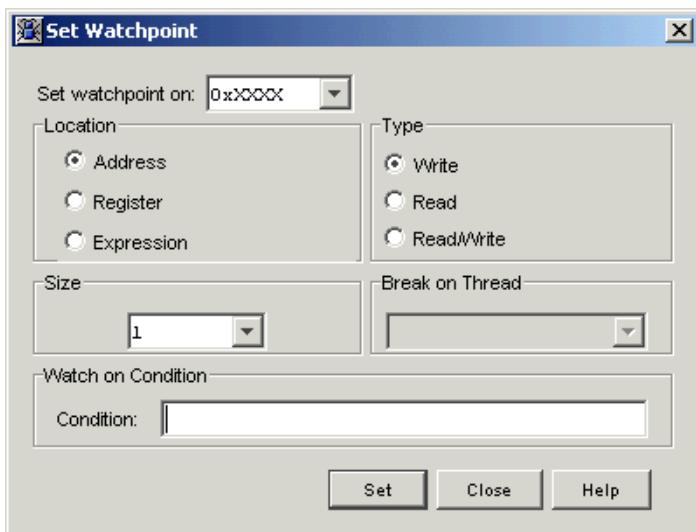


Figure 15 Set Watchpoint Dialog

- Click a radio button to choose how you want to specify the watchpoint location: by address, by register, or with an expression.
- Enter the location and access type in the **Set watchpoint: Location** and **Type** input fields:

Radio Button	Watchpoint Location and Type	Example
Address	Hexadecimal address	0x34
	Symbolic label	main+0x57
Register	Register name	r12
Expression	Expression that evaluates to an address in memory; must be a modifiable value (one that can appear on the left side of an assignment expression)	
Write	Sets write permission	
Read	Sets a read-only permission	
Read/Write	Sets a read/write permission	

3. To make the watchpoint conditional, enter a conditional expression in the **Condition:** input field.
4. Click [OK] or press [ENTER]. The watchpoint location you specified is added to a **Set watchpoint:** drop-down list for future use. A different drop-down list appears for each radio button: **address**, **register**, and **expression**.

Setting a Watchpoint on a Function Argument

To set a watchpoint on an argument in a function on the call stack:

1. Open the **Call Stack** window (**Display | Call stack**).
2. Step through or run your program until the function of interest appears in the call stack.
3. In the **Call Stack** window, do one of the following:
 - Double-click the line displaying the function call.
 - Click the line displaying the function call, then click [**Locals**] on the **Stack Options** toolbar.
 Either action opens a *frame-relative Local Variables* window; that is, a **Local Variables** window for a function other than the current one (see [Viewing Variables Local to a Call-Stack Function](#) on page 73).
4. In the frame-relative **Local Variables** window, locate and click an argument of the called function. The variables local to the function (including function arguments) appear in the **Local Variables** window in declaration order. Their values are the values they had when the called function was in scope.
5. In the frame-relative **Local Variables** window, click [**Watch**] on the **Locals Options** toolbar.

Viewing a list of Current Watchpoints

To view a list of currently set watchpoints, choose **Display | Watchpoints**.

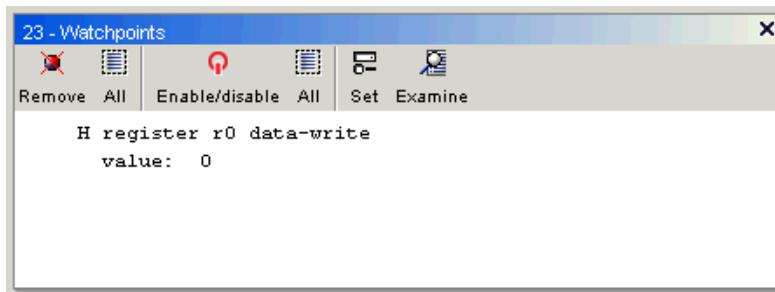


Figure 16 Watchpoints Window

Each line in the listing shows the following information for a particular watchpoint:

- Whether the watchpoint is a hardware ('H') watchpoint (see [Hardware and Software Watchpoints](#) on page 66 for more information).
- Whether the watchpoint is disabled ('-').
- The name of the watched variable or register.
- The value of the watched variable or register. In the case of an aggregate data structure, such as an array or a **struct**, the value of each element is displayed; for example, X [1,3,5,7,6,2,4,8]

If the value of the variable has changed, the previous value (followed by => and the current value) appears to the right of the variable name. Changed values are highlighted in red.

You can also display current watchpoints by entering the [watch](#) command with no arguments. See [Commands Reference](#) on page 231 for details.

If a watchpoint is task-specific, additional task-related information appears in the watchpoint line. For more information, see [Using Thread-Specific Watchpoints](#) on page 110.

Disabling and Enabling Watchpoints

You can turn a watchpoint off and on without deleting it. To disable or enable a watchpoint in the **Watchpoints** window:

1. Click the watchpoint to select it.
2. Click [**Enable/disable**] in the **Watchpoint Options** toolbar. When you disable a watchpoint, a dash ('-') appears to the left of the line.

To toggle the enabled/disabled status of all currently set watchpoints, click [**All**] to the right of [**Enable/disable**].

You can also use the [enable watch](#) command to enable some or all currently disabled watchpoints, or the [disable watch](#) command to disable some or all currently enabled watchpoints. See [Commands Reference](#) on page 231 for details.

Deleting Watchpoints

To delete a watchpoint on a variable or register in the **Watchpoints** window:

1. Click the line containing the variable or register to select it.
2. Click [**Remove**] on the **Watchpoint Options** toolbar.

To delete a watchpoint in the **Registers** window:

1. Select the line where the watchpoint appears (click the register).
2. Click [**Watch**] on the **Register Options** toolbar.

To delete all current watchpoints, click [**All**] to the right of [**Remove**] on the **Watchpoint Options** toolbar.

You can also use the [delete watch](#) command to delete either a single watchpoint or all current watchpoints. See [Commands Reference](#) on page 231 for details.

Viewing and Modifying Data

Working with Global Variables

The **Global Variables** window displays your program's global variables by name and value, in alphabetical order; each line represents one variable.

To open the **Global Variables** window, choose **Display | Global Variables**. As the process executes, the debugger updates the **Global Variables** window to show the current value of each global variable.

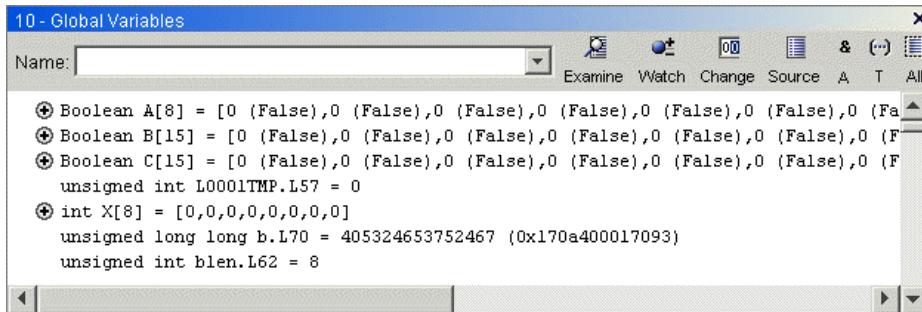


Figure 17 Global Variables Window

TIP You can use CTRL+F or CTRL+B to page forward or backward.

- To toggle the display of variable type, click on the toolbar.
- To show or hide addresses, click .
- To toggle between displaying all global variables and only those compiled with debug information, click .
- To display a specific variable, enter the variable name in the **Name:** field and press ENTER.
To return to viewing all global variables, delete the text in the **Name:** field and press ENTER.

Working with Variables in the Global Variables Window

You can use the **Global Variables** window to select a variable, then perform the following debugging operations on it:

- Click to add the variable to an **Examine** window.
- Click to set a watchpoint on the variable.
- Click to change the value of the variable.

To save the data to a file, right click and select **Save to File**.

Working with Local Variables

The **Local Variables** window shows the names and values of local variables currently defined, in declaration order. Each line represents one variable.

To open the **Local Variables** window, choose **Display | Locals**

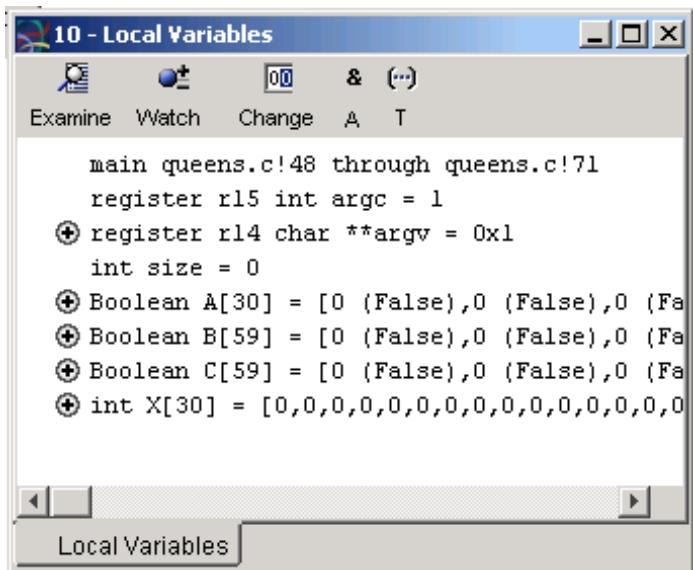


Figure 18 Local Variables Window

NOTE Local variables are displayed only for functions that were compiled with debugging information (option **-g**).

Viewing Local Variables Currently in Scope

When you open the **Local Variables** window, all non-global variables currently in scope are displayed. Each line of text represents either the name and scope of a function, or one variable defined within the scope of that function. As the process executes, the debugger updates the **Local Variables** window to show the current value of each variable currently in scope.

All variables defined within a function's scope are listed below the line representing that function. In the case of nested scopes, the top-most scope displayed is the innermost one.

To toggle the display of variable types on and off, click (...) on the **Locals Options** toolbar.

To show or hide addresses, click &.

You can use the **Local Variables** window to select a variable, then perform the following debugging operations on the variable:

- Click to add the variable to an **Examine** window.
- Click to set a watchpoint on the variable.
- Click to change the value of the variable.

To save the data to a file, right click and select **Save to File**.

Viewing Variables Local to a Call-Stack Function

You can use a frame-relative **Local Variables** window to view local variables defined within the scope of a specific function's stack frame. To open a **Local Variables** window that shows the variables local to a selected function call:

1. Open the **Call Stack** window (**Display | Call stack**). See [Figure 19 Call Stack Window](#) on page 76.
2. Step or run your program until the function call of interest is in the call stack.
3. Do one of the following:
 - Double-click the line displaying the function call.
 - Click the line displaying the function call, then click [**Locals**] on the **Stack Options** toolbar.

Opening a frame-relative **Local Variables** window allows you to view local variables specific to several function calls at once: Click the **Call Stack** window's [**Locals**] button to open a different **Local Variables** window for each set of local variables you want to monitor.

As the contents of a function-specific **Local Variables** window go out of scope, the window clears and then updates to show the variables defined in the current scope.

Examining Data

Use an **Examine** window to inspect the value of a variable or the value of an element of an aggregate data structure (array element, structure member, and so on). You can also toggle on and off the display of the variable type.

To open an **Examine** window, choose **Display | Examine**, or click [**Examine...**] in a **Global Variables** or **Local Variables** window.

Note The **Examine** window is available only for applications compiled with debugging information (option **-g**).

You can examine additional items by typing them in the **Examine:** input field and pressing ENTER. To remove an item, right-click it and choose **Remove** from the pop-up menu that appears.

To follow an item, double-click it, or select the item and then click the **Follow** button. Following an item shows the component parts of an aggregate or dereferences a pointer.

To save the data to a file, right click and select **Save to File**.

Examining the Value of a Variable

To examine the value of a variable in the **Global Variables** window or the **Local Variables** window, do either of the following:

- Double-click the line containing the variable.
- Click the line containing the variable to select it, then click .

Changing the Data Format

The **Examine** window does not allow you to change the format of the values displayed. The **memory** command (see [Commands Reference](#) on page 231) accepts prefixed format modifiers to specify the numeric base and data type of values displayed, including fixed-point "Q" integer types. The output of a **memory** command appears in the **Debug Console** ([Windows | Debug Console](#)).

Examining an Expression

To examine an expression defined in the scope of the current stack frame:

1. Open an **Examine** window (**Display | Examine**).
2. In the **Examine**: input field, enter any addressable expression (that is, an L-value such as a variable, a pointer, or a register) defined within the scope of the top-most stack frame (the program counter's current position).
3. Press [ENTER]. The debugger evaluates the expression and displays the result in the **Examine** window.

If the expression is a data structure, you can examine each component in the current **Examine** window or in a separate **Examine** window. The **Examine** window's tree structure allows you to navigate through many levels of nested data structures. If the expression is a pointer, you can follow it by way of the tree structure to determine its value.

All expressions you enter in the **Examine** window are evaluated relative to the top-most stack frame.

Examining Deeper Levels in an Aggregate Data Structure

When you open an **Examine** window on an aggregate data structure, the **Examine** window's default configuration is *tree-structured*; that is, all components of the data structure are listed below it, joined by a tree diagram (see [Figure 18 Local Variables Window](#) on page 72). Components that are themselves data structures are marked with a data-structure symbol.

Generally, you can learn all you need to about a simple data element by looking at it in the original **Examine** window. There are several ways you can examine the components of a data structure that are themselves nested structures.

- Click the nested-data symbol ([+]) next to the structure (or click the line to select it, then press [ENTER]) to expand the tree structure and display the structure's elements in the current **Examine** window.

You can navigate through many levels of nested structures in this way.

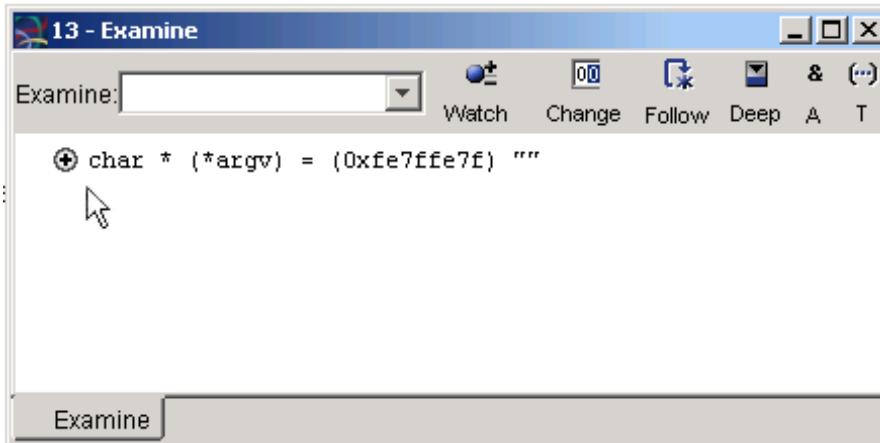
- Click the structure to select it, then click **Follow** on the **Examine Options** toolbar to display the structure by itself in the current **Examine** window.

This method is useful if you have a very large data structure or one that contains nested components with many levels; examining all the levels of such a structure at once in an **Examine** window can result in a very large, awkward tree structure.

Reevaluating the Value of Data Structures

When examining a data structure, you can choose between *deep* or *shallow binding*. In deep binding (the default), the address is computed just once. In shallow binding, the expression containing the data structure is re-evaluated each time the **Examine** window is refreshed (typically every time execution stops).

To reevaluate a data structure, click the button to its left.



Dereferencing a Pointer

CAUTION When you dereference a pointer, be aware of possible side effects. See [Evaluating Expressions](#) on page 78 for important details.

Dereferencing a Pointer from a Variables Window

To dereference or follow a pointer from a **Local Variables** (or **Global Variables**) window:

1. Open a **Global Variables** or **Local Variables** window that shows the pointer you want to dereference (choose **Display | Global Variables** or **Display | Locals**). If the pointer is a local variable, it must be defined in the current scope.
2. Click the line displaying the pointer.
3. Click [**Examine...**]. This opens an **Examine** window that shows the pointed-to item by name and value.
4. If the pointed-to item displayed in the **Examine** window is also a pointer, click the nested symbol next to the pointer; or click the item to select it, then click the **Examine** window's **Follow** button to further dereference the pointer.

Following a Pointer in an Examine Window

To dereference a pointer from an **Examine** window:

1. Open an **Examine** window (choose **Display | Examine**).
2. In the **Examine:** input field, enter any pointer that is defined within the current scope of your program.
3. Press [**ENTER**] to direct the debugger to evaluate the pointer expression.
4. Click the nested-data symbol next to the pointer; or click the item to select it, then click the **Follow** button to display the value of the dereferenced pointer.

Modifying the Value of a Variable

You can modify process execution by changing the values of variables or of elements in aggregate data structures. Use a **Global Variables** or **Local Variables change...** dialog, or use the **Examine** window.

Modifying the Value of a Variable Using the change... Dialog

1. Open a **Global Variables**, **Local Variables**, or **Examine** window.
2. Click the variable or data-structure element whose value you want to change.

3. Click the  icon to open the **change...** dialog.
4. Enter the new value for the variable or data-structure element in the **New value for...** input field.
5. Click **[OK]** or press **[ENTER]**.

Modifying a Value Using the Examine Window

1. Open an **Examine** window (**Display | Examine**).
2. In the **Examine:** input field, enter a valid expression that assigns a new value to the addressable expression (“Lvalue”) that you want to modify.
3. Press **[ENTER]**. The debugger evaluates the expression and displays the result in the **Examine** window.

Examples

To assign the decimal value 12 to **int** variable **my_count**, enter this expression in the input field:

```
my_count = 12
```

To store the address of variable **my_count** in pointer **my_ptr**, enter this expression:

```
my_ptr = &my_count
```

You can also change the value of a variable or data-structure element using the [evaluate](#) command (See [Commands Reference](#) on page 231 for information).

Viewing the Function-Call Stack

The **Call Stack** window displays the *function-call stack* (also known as *function-activation records*) for your program. This is the chain of function calls and the arguments used in each call. The **Call Stack** window shows all current function-activation records, with the most recent function call at the top. Each line of text in the window represents a single activation record.

To open the **Call Stack** window, choose **Display | Call stack**.

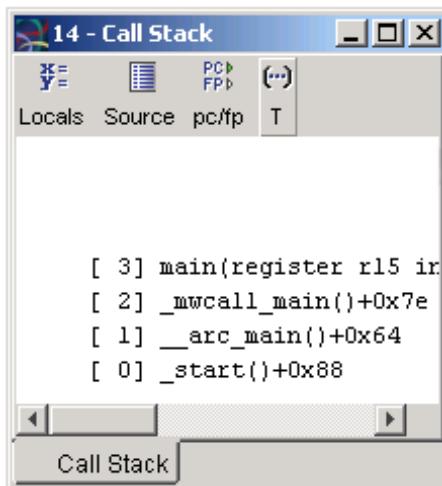


Figure 19 Call Stack Window

In the **Call Stack** window you can select a function call, then do any of the following:

- View the variables local to that stack frame by clicking the **Display locals** button: 
Or right-click and choose **Locals** from the pop-up menu.
See also [Viewing Variables Local to a Call-Stack Function](#) on page 73.
- Evaluate expressions relative to the stack frame of that function.
See also [Evaluating Expressions Currently in Scope](#) on page 79.
- View the source code for the function by clicking the Source button: 
See also [Viewing Source for a Function on the Call Stack](#) on page 53.
- Run your application until it returns to the current frame by right-clicking and selecting **Run to cursor** from the pop-up menu (not reliable for recursive functions).
- Toggle display program-counter and frame-pointer register values by clicking the **pc/fp** button:


NOTE The window shows [EVICTED] instead of the PC value for overlay functions that are evicted from memory. For more information on debugging with overlays, see [Debugging with Overlays](#) on page 130.

- Toggle display of type information by clicking the (...) button: 
- View the contents of the hardware stack, if your target architecture provides one. (See [Working with a Hardware Stack](#) on page 83.)

TIP If you experience crashes when working with the **Call Stack** window, try specifying the maximum number of stack frames to display using option **-max_stack_depth=N**.

Working with Functions

The **Functions** window shows the functions in your program. It finds functions based on the symbols in the symbol table for the executable ELF section. Each line represents one function, and shows the function name and hexadecimal address.

Viewing a Specific Function or a Subset of Functions

To view a specific function or a subset of functions, type a value into the **Name:** field and press ENTER. The named entity only is displayed.

You can also enter a regular expression; as a result, only the entities that match the regular expression are displayed. To restore the display of all functions, empty the **Name:** field and press ENTER.

Setting or Removing a Breakpoint on a Function

To set or remove a breakpoint in the **Functions** window, do any one of the following:

- Double-click a function.
- Right-click a function and then select **Break** from the pop-up menu.



- Select a function and then click the Set or Reset Breakpoint button.

Viewing Disassembly for a Function

To view disassembly source for a specific function:

1. Right-click a function.
2. Select **Source** from the pop-up menu.

A **Disassembly** window opens for the function.

NOTE A new **Disassembly** window does not open if a **Disassembly** window is already open; the first **Disassembly** window is reused.

When the left margin shows...

- a solid red rectangle, a breakpoint is set on the function.
- a grey rectangle, the breakpoint is disabled.

Viewing a Specific Function or a Subset of Functions

Type a value into the **Name:** field to display the named entity only. You can also enter a regular expression; as a result, only the entities that match the regular expression are displayed.

Advancing execution to a specific function:

1. Right-click a function.
2. Select **Run to cursor** from the pop-up menu.

Viewing Only Functions with Debug Information

To toggle between all functions and only those compiled with debug information (option **-g**), click **[All]**.

Saving the Data to a File

Right-click and select **Save to File**.

Evaluating Expressions

The MetaWare debugger expression evaluator can evaluate any expression that is valid in the current evaluator language. By default, the evaluator determines the correct language for the current context, then evaluates the expression in terms of that language. The following expressions can be evaluated:

C Any valid C expression (variable, pointer, **struct**, **enum**, and so on, or an expression containing these).

C++ Any valid C or C++ expression.

See [Using Expressions](#) on page 222 for details.

The expression evaluator can evaluate functions using the [**evaluate**](#) command, and can call functions using the [**call**](#) command.

The expression evaluator cannot evaluate statements (**for**, **while**, **switch**, and so on) or iterators.

CAUTION If you enter an expression with associated side effects, the MetaWare debugger expression evaluator causes those side effects to occur.

For example, if you enter the expression `my_var++` or `my_var=5`, the debugger changes the value of `my_var`.

Evaluating Expressions Currently in Scope

By default, the debugger evaluates expressions relative to the current location of the program counter (that is, relative to the top-most stack frame). Any expression you evaluate can contain only identifiers that are in scope.

To evaluate an expression defined in the current scope, use the **Examine** window.

1. Open an **Examine** window (choose **Display | Examine**).
2. In the **Examine**: input field, enter any valid expression that is defined within the scope of the top-most stack frame (the program counter's current position).
3. Press [ENTER] to evaluate the expression and display the results in the **Examine** window.

All expressions you enter in this window are evaluated relative to the top-most stack frame.

Evaluating Expressions in Scope for a Function on the Call Stack

To evaluate an expression defined in a scope different from the current scope, use the **Call Stack** window (see [Figure 19 Call Stack Window](#) on page 76) to open a frame-specific **Local Variables** window, and from that a frame-specific **Examine** window.

1. Open a **Call Stack** window (choose **Display | Call stack**).
2. In the **Call Stack** window, double-click the activation record of the function call for which you want to evaluate the expression. A **Local Variables** window relative to that function call opens.
3. In the **Local Variables** window, click any line to select it, then click [**Examine...**] to open a frame-relative **Examine** window.
4. In the **Examine**: input field of the **Examine** window, enter an expression that is defined within the scope of the currently selected function.
5. Press [ENTER] to evaluate the expression and display the result in the **Examine** window.

All expressions you enter in this window are evaluated relative to the stack frame of the currently selected function call in the **Call Stack** window.

Debugging Memory

You can view memory at the program counter or any other address using the **Memory** window. The **Memory** window shows “raw” memory on the system where your program is running.

By default, memory is displayed in ascending order by address. If the value of a register (such as the stack pointer) happens to coincide with the address on display (the stack pointer register is pointing to the memory address), that register is displayed to the right of the column.

Note that by default, the debugger avoids reading the same target memory twice between execution events. If you have volatile memory that changes each time it is read, you can disable this behavior using one of the following methods:

GUI

Uncheck **Cache target memory** in **Debugger options | Program Options**.

Command Line

Turn off toggle `cache_target_memory`.

Viewing Memory at a Specific Address

To view the contents of a region of memory associated with your program, use the **Memory** window. By default, the **Memory** window shows the contents of memory in ascending order by address.

NOTE If you are debugging symbolic code, refer to the **Breakpoints**, **Disassembly**, or **Call Stack** window to obtain a meaningful starting address for a region of memory you want to view.

Each line of text in the **Memory** window begins with an address on the left (for example, 0804826c), followed by a string of 32 hexadecimal digits representing the contents of 16 bytes of memory starting at that address.

The initial display starts at an arbitrary address. In the **Goto:** input field, enter the register name, hexadecimal start address, or symbolic label of a memory region you want to view, then press [ENTER].

Changing a Value

To change a value in memory:

1. Click a line to select it.
2. Click the **Change** button  or right-click and select **Change**.
3. Enter the new value and click **OK**.

Examining a Location

To examine a memory location in an **Examine** window, right-click a line, and then select **Examine** from the pop-up menu.

The selected address is displayed with the prefix `*(int*)`, which means that the address has been cast to a pointer-to-integer. You can then change the “int” part of the prefix to any type. Subsequent examine selections from the Memory display change the address only: Your selected type preference is preserved.

For information about symbolic labels, see [Customizing the Disassembly Window](#) on page 55.

Modifying the Memory Window

By default, the **Memory** window shows 16 bytes of memory per line, in hexadecimal, grouped in words, in ascending order from the starting address you specified.

To change the text display, you can do any of the following, using the toolbar:

You can use the drop-down menus to change the display parameters: 

- Number format (base)
- Bytes per line. To display memory usage in a single column, use the drop-down menu to set the number of bytes per line to 4.
- Data type by which you want the bytes of memory to be grouped

- Click the up or down arrow to display memory in ascending or descending order.
- Change the minimum width of hexadecimal addresses: enter the command `set addr_width=value`, where value is the width.

Viewing the Function-Call Stack

Because you can modify the **Memory** window, you can use it to display the function-call stack in stack formation. To display the function-call stack, open a **Memory** window.

1. To display four bytes per line, choose 4 from the **Bytes** field.
2. If the stack grows downward on your target system, set the display to descending order (the **Up** button is displayed).
3. Open the **Call Stack** window (choose **Display | Call stack**), and click **[pc/fp]** to display the frame-pointer address for each stack frame.
4. Enter the frame-pointer address of the top-most stack frame (or whatever function activation record you want to examine) in the **Goto:** input field of the **Memory Options** toolbar.

You can inspect the entire stack by entering the top-most frame-pointer address in the **Goto:** input field to display the high end of the stack, then entering the address at that address in the **Goto:** input field, and so on down the rest of the stack.

Copying or Filling Memory to or from a File

By default, the MetaWare debugger downloads the application being debugged to target memory (simulator or hardware). You can also copy binary files and Motorola S-record files to and from memory using the debugger's file-memory-fill operation. This may be useful, for example, when loading a dataset into memory so your embedded program can access it or when dumping memory to a file.

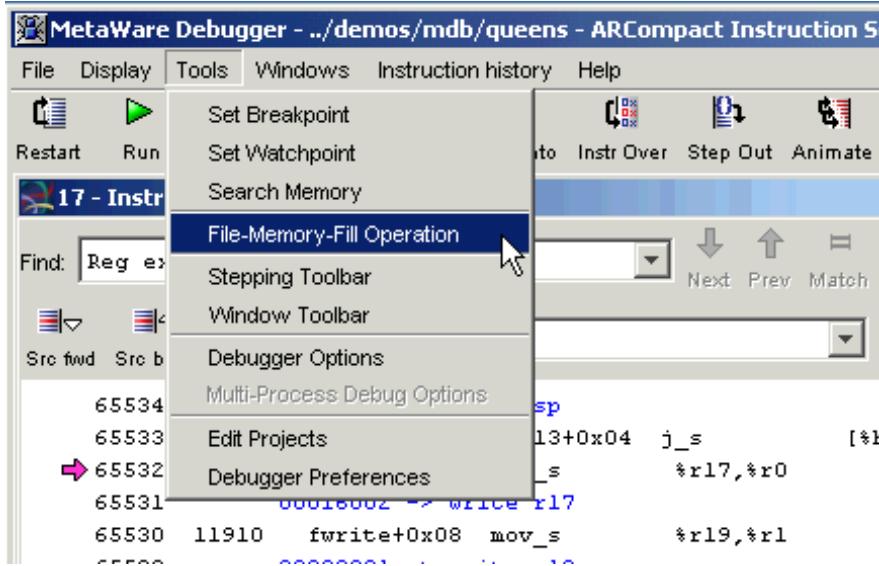
NOTE To disable automatic downloading of the application you are debugging, select **Program is already present; don't download** in **Debugger options | Program Options**.

For more information about Motorola S-record files, see [Appendix B — S-Record Files](#) on page 341.

1. Select **Copy memory to file**.
2. Enter the starting address in the **Address** field.
3. Enter the total size in bytes of memory you want to copy in the **Length** field.
4. Enter or browse to the file **File to write** field.
5. Click **OK** to save memory to the file.

GUI

1. Choose Tools | File-Memory-Fill Operation.



The **File/Memory/Fill operation** dialog appears.



2. In the **File/Memory/Fill operation** dialog, choose one of the following:
 - **Copy a file into memory**
 - **Copy memory to a file**

- **Fill memory**

The lower half of the dialog changes to offer parameters based on your selection:

- Whether **The file is an S-record file** (default is binary)
- **Starting address** to read or write or fill
- **File to write or File to read**, if any
- **Length** of memory to copy to a file or to fill
- **Value** to fill with if filling memory
- **String** (hex or **char**) to fill with if filling memory

3. Click **OK**.

Command Line

Use the following commands:

- [file2mem](#)
- [fillmem](#)
- [mem2file](#)

Working with a Hardware Stack

To view the hardware stack, choose **Display | Hardware Stack** to open the **Memory: Hardware Stack** window. The **Memory: Hardware stack** window shows memory with a display width the natural size of elements on the run-time stack.

NOTE The **Memory: Hardware Stack** window is available only on target architectures that provide a hardware stack.

Displaying a Specific Address

To display the hardware-stack contents at a specific address, enter the address (a hexadecimal value) in the **Goto:** input field.

Modifying the Value at an Address

To modify the value of a memory address, do the following:

1. Click the value that you want to change, and then click the **Change** button, or right-click the value and select **Change**.

The **Change** dialog appears.

2. Modify the value in the **New value for** field of the **Change** dialog, and then click **OK**.

The new value appears in the window (highlighted, because it has changed).

Changing the Display

You can do any of the following using the toolbar:

- Change the value of a memory address using the **Change** button.
- Change the base numbering system: in the **Base** field, choose 16, 10, 8, or 2.

- Change how the contents are grouped: in the **Size** field, choose the data type by which you want the bytes of memory to be grouped.
- Change the order of the display: use the **Up** and **Down** buttons to toggle between ascending and descending address order.

Examining Memory in Various Formats

The default display unit for the **Memory** window is a hexadecimal four-byte word. To examine one of these four-byte blocks in a variety of numeric formats:

1. In the **Bytes per Line** field, set the number of bytes per line to 4. (Otherwise the block examined is not the one you selected but the first block in the row.)
2. Click the block of memory to select it.
3. Being very careful not to move the cursor, right-click to activate the **Memory** window's right-click menu.
4. Choose **Examine** from the right-click menu. This opens an **Examine** window with the block displayed in default **int** format.
5. To display the block in another format, change **int** to the format of your choice in the **Examine** window's **Examine:** input field, then press ENTER.

Displaying Memory in Stack Configuration

The **Memory** window can show memory in stack configuration. To do this, set the **Size** field to 4 to generate a column of stack contents. If the value of a register (such as the stack pointer) coincides with an address being displayed, the register's name is displayed on the right of the column.

Viewing an Image from Memory

The MetaWare debugger can display an image from memory or display a data structure from memory as a two-dimensional image.

Displaying an image from memory is useful for applications that process graphics or video.

Displaying a data structure as an image is useful for all applications; it permits you to check the data structure visually for symmetry and shape and watch it change as your application runs. If the image changes in an unexpected way, you can then examine the data structure to determine the exact values it contains.

Image display of data structures is most effective for structures that are inherently two-dimensional. For example, if you have an array **char a[32][32]**, you can view it as a 32-by-32 grayscale image and easily monitor distortions as you run your application. In the window, the columns are titled with the offset from the address given on the left edge, and contents are shown in the default size of the image element (1, 2, or 4 bytes).

Kinds of Image

Image display supports the following kinds of image:

8-bit grayscale: A sequence of bytes representing an 8-bit grayscale image.

16-bit grayscale: A sequence of halfwords representing a 16-bit grayscale image.

16-bit R5:G6:B5 color: The 16-bit RGB display is a sequence of half-words representing a packed red-green-blue image: 5 bits red, 6 bits green, and 5 bits blue, from most to least significant.

32-bit ARGB color: The 32-bit ARGB display is a sequence of 32-bit words representing an

alpha-red-green-blue image: 8 bits each of transparency, red, green, and blue from least to most significant.

4:2:0 YUV color: The 4:2:0 YUV display requires three arrays. The address is the Y component, which points to an array (width by height) of 8-bit intensity values. The U and V addresses point to subsampled arrays (width/2 by height/2) of 8-bit chrominance values.

Viewing an Image or Data Structure

1. Turn on toggle [image window](#).
2. Select **Display | Image**.
3. Open the **Configure image display** dialog by doing one of the following:
 - Choose **Image | Configure | Configure image parameters** from the main menu bar.
 - Right-click in the text area of the **Image** window and select **Image | Configure | Configure image parameters**
 - (see [Figure 20 Opening the Configure image Display Dialog](#) on page 85).

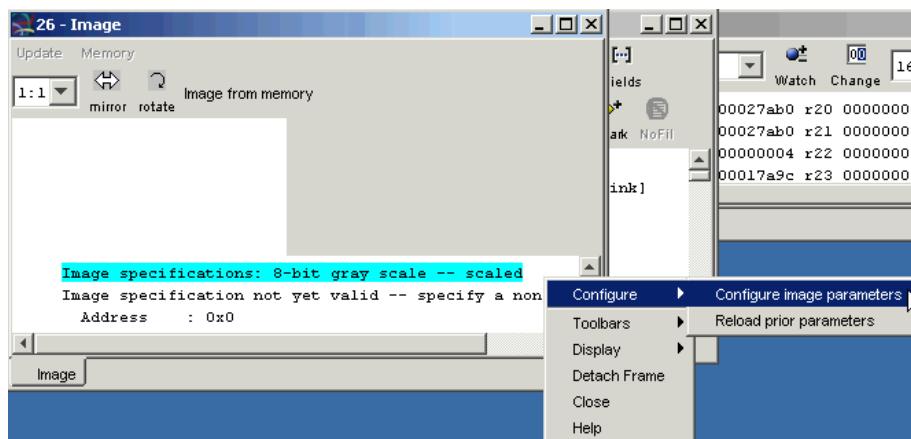


Figure 20 Opening the Configure image Display Dialog

The **Configure image display** dialog appears.

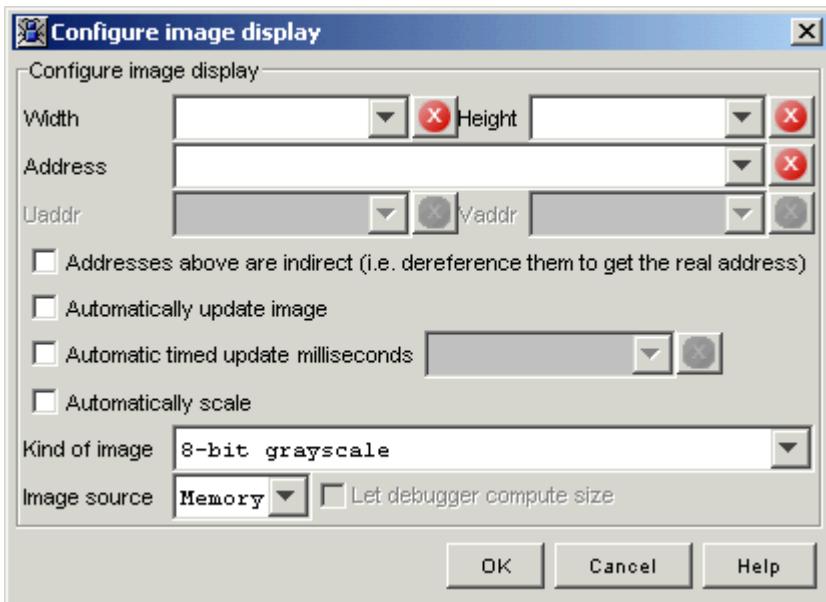


Figure 21 Configure Image Display Dialog

4. In the **Configure image display** dialog, specify the following for your image:

- Width
- Height
- Address
- Updating and scaling options

If the addresses you enter use indirection, check **Addresses above are indirect** to dereference them.

If you select **Automatically update image**, the window image updates every time execution stops. Alternately, you can reduce memory traffic by deselecting **Automatically update image** and using the **Image** window's **Update** button to update only when you want to update.

You can have grayscale image formats automatically scaled to improve the contrast, when the values in the image do not provide enough variation.

- Kind of image (see [Kinds of Image](#) on page 84)
- Image source (memory or registers)

5. Click **OK**.

TIP If you receive an exception such as `java.lang.OutOfMemoryError: Java heap space` you can specify a larger heap size for the JVM using option [`-jvmheap`](#).

Reloading Saved Parameters

1. Turn on toggle [image window](#).
2. Select **Display | Image**.

3. Select **Configure | Reload prior parameters**.

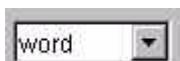
All **Image** windows are reset to the prior parameters.

Modifying the Contents of Memory

To change the value of a block of memory, open a **Memory** window.

1. In the **Goto:** input field, enter the starting address at which you want to change the contents of memory.
2. Choose a word (four bytes) of memory by clicking its left boundary.

The memory-changing feature allows you to process four bytes of memory at a time because word is the default display format. To process memory in a different unit, choose a different unit from the **Size** drop-down menu.



3. Click **[Change]** on the **Memory Options** toolbar to open the **change...** dialog.
4. Enter the new value for the block of memory in the **New value for...** field.
5. Click **[OK]** or press **[ENTER]**.

The new value appears in the block of memory you selected, highlighted in red to show that the value has changed.

Searching for Patterns in Memory

Choose Tools | Search Memory to open the **Memory Search** dialog.

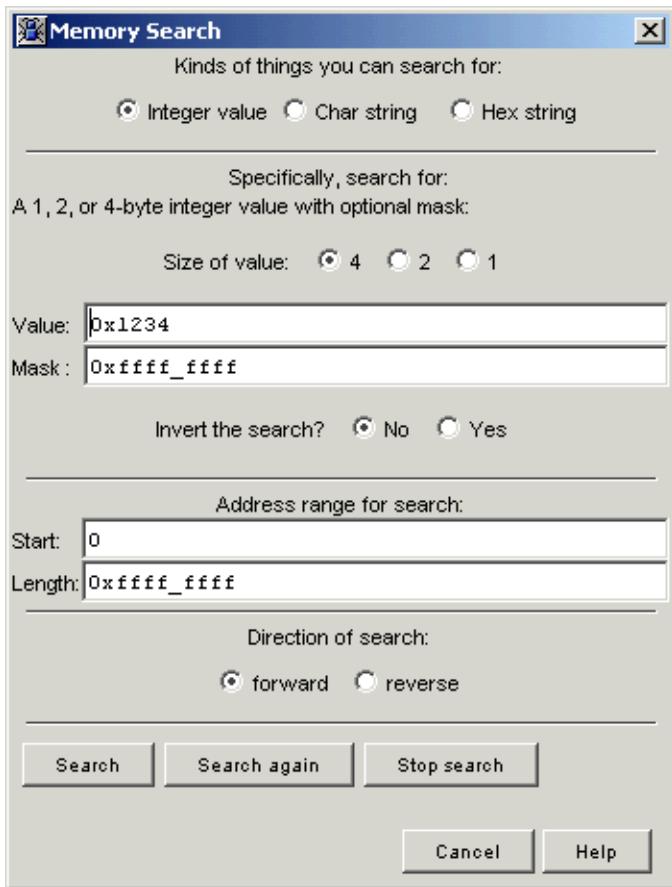


Figure 22 Memory Search Dialog

A search covers a range of memory specified by a start address and a length. A search pattern can be any of the following:

- a character string (including wildcards and escaped characters)
- a hexadecimal byte string (including wildcards)
- a one-byte, two-byte, or four-byte integer

If you search for a string, the debugger searches for the string at all addresses in the range. If you search for an integer value, the debugger searches for it at the start address plus multiples of the size. For example, if you search for a two-byte integer starting at address 0x1000, the debugger searches locations 0x1000, 0x1002, 0x1004, and so on.

After the debugger finds a pattern, it shows a **Memory** window starting at the address where the pattern was found. You can then continue the search from where the pattern was found.

Searching for an Integer Value

Choose **Integer value** and click the appropriate size, then enter a one-byte, two-byte, or four-byte value in the **Value:** field. Examples of valid integer values:

```
r15
p->x
```

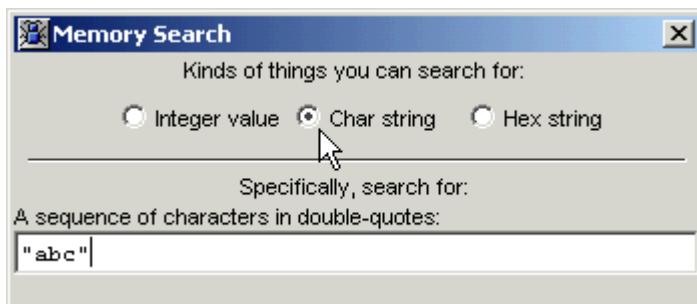
0x1234
a[i+1]

Integers are searched under a mask. Either accept the default mask value, or enter your own hexadecimal value in the **Mask:** input field. For an explanation of how the search for an integer value interacts with the mask value, see [Searching for an Integer under a Mask](#) on page 90.

If you want to find the first occurrence matching the masked value, choose **No** for **Invert the search?**. If you want to find the first occurrence that does *not* match the masked value, choose **Yes**.

Searching for a Character String

If you choose **Char string**, enter a sequence of characters enclosed in double-quote characters ("like this"). You can enter the wildcard character "?" for "any single character," but *not* "*" for "zero or more characters."



You can also enter the two-character escape sequences:

Escape Sequence	Represents This Character
\?	?
\n	Newline (ASCII 10)
\t	Tab (ASCII 9)
\0	0 (ASCII 0)
...	...
\9	9 (ASCII 9)
\"	Double-quote
\\	Backslash
\X	Character X, where X is anything not in this table

Examples

"abc"
"Hello, world.\n"
"Hello, ?????.\n"

Searching for a Hexadecimal String

You can search memory for a hexadecimal string.

You can enter the wildcard character "?" for "any single hex digit," but *not* "*" for "zero or more digits."

Examples

7ff
deadbeef
dea?beef

0x2x1001_1110 for 0x9e (MetaWare C extension: second ‘x’ for radix)

To search for a hexadecimal string, choose **Hex string**, and enter the string in the field:



Specifying a Search Address Range and Direction

- In the **Start**: input field, enter an expression representing the address where you want the search to begin. The default is 0.
- In the **Length**: input field, enter an expression representing the length, in bytes, of the memory region you want to search. The default is 0xffff_ffff (search all of possible memory).

NOTE If you search for a string, the debugger searches for it at all addresses in the range.

If you search for an integer value, the debugger searches for it at the start address plus multiples of the size. For example, if you search for a two-byte integer starting at address 0x1000, the debugger searches the locations 0x1000, 0x1002, 0x1004, and so on.

Specify the direction of the search.

- Choose **forward** to search from the start address to higher-numbered addresses.
- Choose **reverse** to search from the start address to lower-numbered addresses.

Click **[Search]** to begin the search.

After the debugger finds a pattern, it shows a **Memory** window starting at the address where the pattern was found. You can then click **[Search again]** to continue the search from where the last pattern was found.

To stop the search at any time, click **[Stop search]**.

Searching for an Integer under a Mask

The debugger searches for an integer pattern “under a mask.” A match occurs if the following is true:

`(search_int & mask) == (value_in_memory & mask)`

The default mask value is -1 (that is, all bits == 1), meaning the mask has no restrictive effect.

Searching for an integer under a non-default mask can yield multiple non-identical matches. For example, searching for 0x9ac0 under mask 0xffffc means that the differing values 0x9ac0, 0x9ac1, 0x9ac2, and 0x9ac3 are all matches:

```
mask 0xffffc: 1111 1111 1111 1100
search_int 0x9ac0: 1001 1010 1100 0000
-----
matches 0x9ac0: 1001 1010 1100 0000
matches 0x9ac1: 1001 1010 1100 0001
```

```
matches 0x9ac2: 1001 1010 1100 0010
matches 0x9ac3: 1001 1010 1100 0011
```

You can specify an arbitrary expression for the mask. The mask gives the ability to wildcard any individual bit in the searched-for integer.

Endian Reversal in Integers

On a little-endian target, an integer value is extracted from the searched location with appropriate endian reversal. For example, the value 0xaabb is represented in memory as the sequence of bytes bb aa. Thus searching for the little-endian integer 0xaabb is like searching for the hexadecimal string bbaa.

Command Line

See the following commands:

- [memsearch](#)
- [memsb](#)

Simulating Instruction and Data Caches

Note Cache simulation is not available for all targets.

The instruction cache and data cache simulator allows you to see how many cache misses would be happening in the real hardware; the simulator does not actually store data in a cache.

Enabling Cache Simulation

To enable cache simulation, do the following:

1. Start the debugger and click the **Debugger Options** button. If the debugger is already running, select **Tools | Debugger Options**.
2. Select **Target selection** and select the **MetaWare debugger Simulator** as the target.
3. Select **Simulator extensions | Cache simulation**.
4. Check the **Instruction cache** box or the **Data cache** box (or both) and specify values using the drop boxes.
5. Click **OK**. If the debugger was already running, you must exit and restart it for the settings to take effect.

Command Line

When using the command-line debugger, specify the following options to enable cache simulation:

-icache=cache_size, line_size, ways, attribute

-dcache=cache_size, line_size, ways, attribute

Refer to your processor documentation for legal values. For example:

```
mdb queens -icache=512,8,2,o -dcache=512,16,1,o
```

After you specify the cache simulation, you can access debugger windows for icache and dcache. See [Command-Line Option Reference](#) on page 274 for details on the **-icache** and **-dcache** command-line options.

Viewing Simulated Caches

As you step through a program, you can display simulated instruction and data caches. To open an **Instruction cache** window, choose **Display | Instruction cache**; to open a **Data cache** window, choose **Display | Data cache**.

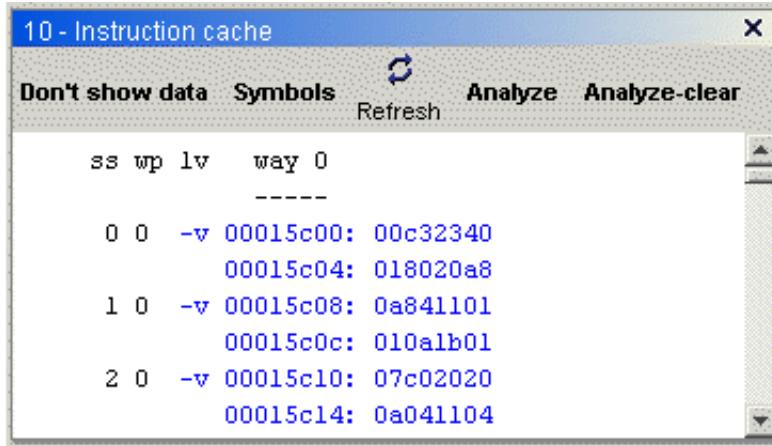


Figure 23 Instruction Cache (ARC Target Shown)

You can use each column in the **Instruction cache** window ([Figure 23 Instruction Cache \(ARC Target Shown\)](#)) as follows:

- ss Set number: sets of cache RAM line locations are numbered sequentially
- wp Way pointer: shows which line in a set is currently the candidate for replacement on the next cache miss
- 1 1 locked
- not locked
- v v valid data
- invalid data
- way 0 The data in each way;
to “...” in a way column indicates invalid data
way n

Invalid data is not shown. If cache simulation is n -way, you see the way pointer change.

To save the information in the window, right-click and select **Save to File**.

Cache Algorithms

Instruction and data caches are simulated using the parameters and algorithms listed in [Table 3 Cache Parameters and Algorithms](#).

Table 3 Cache Parameters and Algorithms

Parameter	Definition
Line size	Length in bytes of each line; must be power of 2 and a multiple of 4 bytes
Ways	Number of ways; must be a power of 2: If ways = 1, this is a direct-mapped cache If ways = 2, this cache is 2-way set-associative If ways = N, this cache is N-way set associative
Cache size	Total size in bytes of the cache
Way size	Size of each way = cache size / ways
Lines per way	Lines in each way = way size / line size
Set	A collection of lines, one for each way

A cache picture looks like this, where N = lines per way:

	Way 0	Way 1	Way 2	Way 3
set 0	line 0	line 0	line 0	line 0
set 1	line 1	line 1	line 1	line 1
...				
set $N-1$	line $N-1$	line $N-1$	line $N-1$	line $N-1$

An address A is mapped into a set as follows:

$$\text{Set index} = (A \text{ / line size}) \% \text{ lines per way}$$

The particular byte within a line is as follows:

$$\text{Byte index} = A \% \text{ line size}$$

After A has been mapped into a set, the caching algorithm determines whether the address is present in any of the ways. If so, the data is returned; if not, the data is cached; the way into which the data is cached depends on the replacement algorithm.

For example, a cache with these parameters:

Cache size	2,048 bytes
Line size	8 bytes
Ways	2
Lines per way	$(2,048 / 2) / 8 = 128 = 0x80$

would look like this:

	Way 0	Way 1	Addresses
set 0	line 0	line 0	0 4 400 404 800 804
set 1	line 1	line 1	8 10 408 410 808 810

	Way 0	Way 1	Addresses
...			
set 7f	line 7f	line 7f	3f8 3fc 7f8 7fc 8f8 8fc

For example, for address 808 (computations in hex):

$$\text{Set index} = (808 / 8) \% 80 = 101 \% 80 = 1$$

Analyzing Cache Performance

Cache performance analysis consists of evaluating estimated access time for memory given the presence of the cache, and thrash analysis to see if competing lines in memory are thrashing each other.

GUI

1. Choose **Display | Instruction cache** or **Display | Data cache** after specifying your cache in **Debugger Options > Simulator Extensions > Cache simulation**.
2. Click **Analyze** in the resulting window.

For more information, see [Cache Analysis](#) on page 153.

Command Line

```
prop dcache_stats=1  
prop icache_stats=1
```

```
display icache analyze  
display dcache analyze
```

Related Topics

- [Using the Cycle-Estimating Simulator \(CES\)](#) on page 150.
- [Using Commands to Dump Counters and Cache Analysis](#) on page 219

Execution Profiling

You can find out where your program is spending its time by profiling it. The simulator is enabled by default to display profiling information; profiling on hardware requires you to build your application with timer and profiling options as described in the sections below.

NOTE The simulator can run your application slightly faster if you disable profiling as described in [Disabling the Profiling Window for Faster Simulation](#) on page 101.

Two Ways to Profile

You can view profiling information by right-clicking in a window, or you can open a **Profiling** window by selecting **Display | Profiling**. The information displayed and the tasks you can perform are different in each case.

Adding Profiling by Right-Clicking in a Window

Depending on your target, when you right-click in certain windows (such as **Source** and **Disassembly**), you can select **Profiling** from the menu that appears, allowing you to annotate the display lines with profiling address counters. Profiling choices vary according to the following:

- whether you run your code on the simulator or hardware
- which other options you specified

- the target processor

When you select **Profiling** from the right-click menu, you can select the columns and displays you want to view.

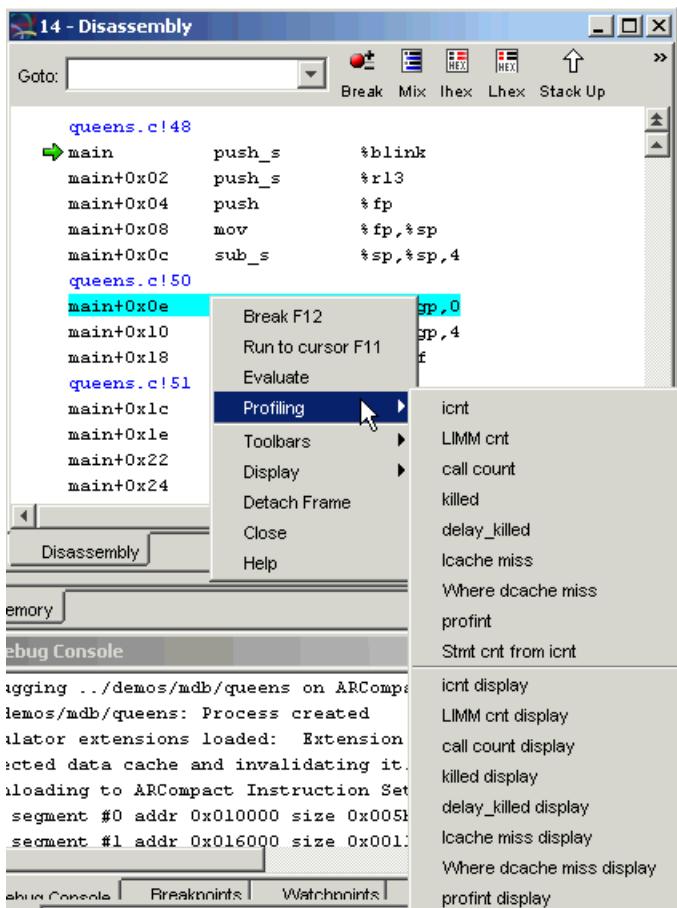


Figure 24 Disassembly Profiling Menu (ARCompact Disassembly Shown)

Adding Columns to the Current Window

The upper section lists profiling data that you can view as a column in the current window. Select an item to display a column for any upper-section menu item.

Columns can include instruction counts (icnt) or cycles (if you have turned on toggle [cycles](#)), but not time. You can see time for hardware targets that have an extension timer by profiling using a **Profiling** window, as long as you have specified **-Xtimer0** or **-Xtimer1** or selected your Enhanced programmable timer in **Debugger Options | Simulator Extensions | Instruction Extensions**.

Explanation of Columns

The columns available vary by the application and processor being debugged and the debugger options you specify.

- icnt:** Instruction count. Counts how many times the instruction was executed.
- LIMM cnt:** Long-immediate count. The long-intermediate instructions executed within the function.

LIMM instructions require another cycle to fetch to complete an instruction. SHIMM (short-immediate) instructions can avoid this added cycle. You can generate SHIMMs using the

small-data optimizations described in the *MetaWare C/C++ Programmer's Guide*

- **call count:** Function-call count. Normally applies only to the entry points of functions (function labels). This count records the number of times a given instruction is the target of a branch and link instruction.
- **killed:** Instruction-killed count. Identifies the number of times an instruction was not executed, normally due to a failed condition-code test.
- **delay_killed:** The number of instructions not executed in a delay slot when a branch is not taken.
- **Icache miss:** The number of instructions responsible for an instruction-cache miss.
- **Where dcache miss:** The number of instructions (typically loads and stores) that resulted in a memory access that was not in the data cache.
- **profint:** statistical-profiling interrupt count. Active only for execution profiling with interrupting timers; counts instructions where timer interrupts occurred.
- **Stmt cnt from icnt:** Statement count derived from instruction counts. This count is available in the **Source** window only. It indicates the number of times a line of code was executed.

Opening Windows

The lower section lists the types of profiling data you can view in a separate window. Profiling windows show sorted displays of address counters, so you can see where in the program you are getting the highest count values. For example, to view instruction counts in a separate window, click **Profiling | icnt display**.

The **Instruction counts** window opens, showing disassembled code for your program.

- Click the **Refresh** button to update the display after running or stepping.
- Click the **Clear** button to clear the display.
- Click the **By addr/By cnt** toggle button to sort by address or by count.

Profiling Using a Profiling Window

Select **Display | Profiling** to open a **Profiling** window where you can configure and view statistical profiling output and summarize address counts.

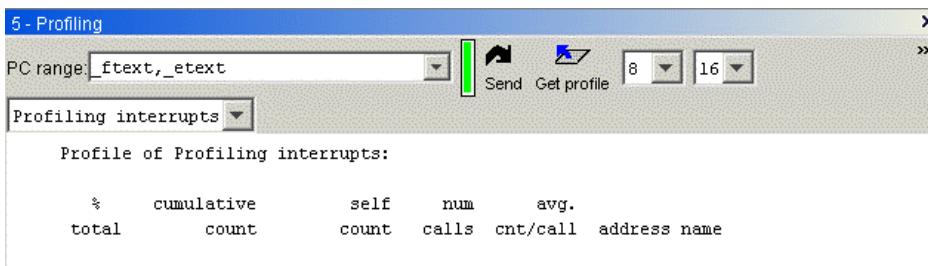
NOTE When you debug on ARCangel hardware, some features require you to compile and link your application with profiling data using option **-p** or **-pg** and a **-Xtimer*** option.

The **Profiling** window provides the same type of data as the MetaWare toolset's command-line utility **prof**. For more information on the **prof** utility, see the *ELF Linker and Utilities User's Guide*.

Displaying the Current Execution Profile

The **Profiling** window might read **Program has no statistical profiling** data when you first open it; this is the normal initial state. Use the drop-down list to choose which data (address counters) to include, such as instruction counts, cycle counts, or cache-miss counts. Then click **Get Profile** to display the current execution profile for the address counters currently selected. On some targets you can do this while the program is running. On other targets you must stop the program to display the profile.

Note For an explanation of the address counters, see [Selecting a Parameter for Call-Graph Profiling](#) on page 98.



When profiling on hardware, you can use the controls at the top of the window to dynamically configure profiling parameters at run time. For details, see [Statistical Profiling and Interval Timers](#) on page 99.

TIP To clear the profiling statistics displayed, use the **Clear** and **Clr All** buttons. The buttons might be initially located beyond the visible area of the window; click the **>>** button to view them:



Configuring the Profiling Window

You can right-click in the **Profiling** window and select **Options** to configure the following options:

- Summarize with time.

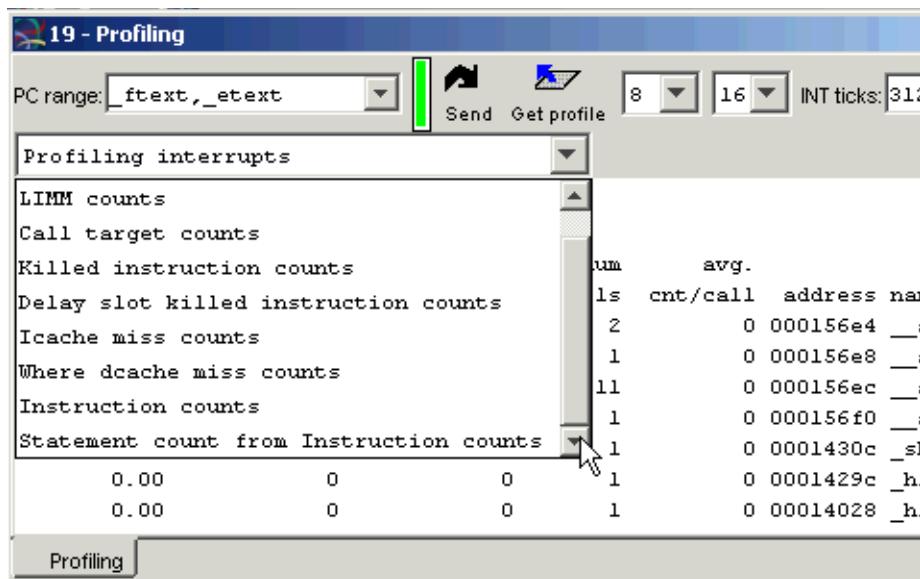
For hardware only: Requires an extension timer (**-Xtimer0** or **-Xtimer1** or **Debugger Options | Simulator Extensions | Instruction Extensions**).

- Sort by symbol address.
- Demangle symbol names.
- Exclude static symbols.
- Suppress call graph.
- Suppress heading.
- Show low-level information.
- Save `mon.out/gmon.out` file.
- Generate dot/lefty call graph (requires compiling with **-pg** when debugging on hardware).
- Generate VCG call graph (requires compiling with **-pg** when debugging on hardware).

Note The **vcg** call-graphing program is supplied; the **dot** and **lefty** programs are available from the following URL: <http://www.research.att.com/sw/tools/graphviz/>

Selecting a Parameter for Call-Graph Profiling

When debugging on the simulator, you can use the drop-down menu in the **Profiling** window to select a profiling parameter (address counter) and click **Get profile** button to obtain a profile of that parameter.



After you click **Get profile**, the **Profiling** window displays the percentage that parameter occurs in the profile, the cumulative count or sum for that function and the ones above it, the self count of the number of occurrences in that function, the number of calls to the function (if applicable), the average count of that parameter per call (if applicable), and the address and name of the function.

The following parameters (address counters) are selectable:

- **Profiling interrupts:** Where profiling interrupts occurred, in descending order.
- **LIMM counts:** The long-intermediate instructions executed within the function.
- **Call target counts:** The number of times the address was the target of a call instruction, i.e., which functions were called the most and how many times they were called, in decreasing order.
- **Killed instruction counts:** Identifies the number of times an instruction was not executed, normally due to a failed condition-code test.
- **Delay slot killed instruction counts:** The number of branch-delay-slot instructions not executed (usually due to an unfilled delay slot).
- **Icache miss counts:** The number of instructions responsible for an instruction-cache miss.
- **Where dcache miss counts:** The number of instructions (typically loads and stores) that resulted in a memory access that was not in the data cache.
- **Instruction counts:** The total number of instructions executed. This parameter is the default address-counter selection, and reveals where your application is spending the most time during execution.
- **Statement count from instruction counts:** The number of high-level-language statements executed as determined from the sum of the instruction counts for the list of instructions associated with each statement.

Statistical Profiling and Interval Timers

Statistical profiling using interval timers is useful when running on hardware, where the simulator's instruction and address counters are not present. To perform statistical profiling using interval timers, you must link your application using option **-p** or **-pg**. For debugging on the simulator, compiling with **-p** or **-pg** is not required and is not recommended.

In statistical profiling, an interval timer interrupts the executing program at a regular frequency. At each interrupt, the interrupted PC is used to increment a profiling counter. Each profiling counter covers a range of code addresses. The resulting table of counters gives an approximate idea of where the program is spending its time. For more information on providing an interval timer, see the *MetaWare C/C++ Programmer's Guide*.

Configuring Statistical Profiling

You can use the controls at the top of the **Profiling** window to change the PC range, counter size, bytes-per-count, and interrupt-frequency controls. Set the parameters, then transmit the data to the application.

- **PC range:**

Enter the starting and ending addresses between which you want your program profiled, separated by a comma. Profiling begins with the starting address (inclusive), and continues up to but not including the ending address. If you do not enter a starting address, profiling begins with the symbol `_ftext` (the start of the `.text` section). If you do not enter an ending address, profiling stops at the symbol `_etext` (the first address beyond the end of the `.text` section). If the starting address is a function name, you can enter `$` as the ending address to indicate the end address of the function. You can also follow the starting and ending addresses with an offset in the form `+offset` or `-offset`, where `offset` is a literal integer.

Examples:

```
0x1000,0x1fff // address range 0x1000 to 0x1fff
0x1000,        // address range 0x1000 to _etext
,0x1000        // address range _ftext to 0x1000
Try,$          // address range from Try to end of Try
Try            // address range from Try to _etext
,              // address range _ftext to _etext
```

- Bytes per count

Bytes per count specifies the size of the region of code assigned to each profiling counter. One profiling counter for each possible PC address is optimal accuracy. For example, in RISC architectures with 4-byte instructions, you can obtain the most accurate profiling by having each counter cover four bytes of code. However, the smaller the region of code covered by each counter, the greater the memory that is required to maintain the counters. Thus, if your system does not have sufficient memory to allow optimal coverage, you can set the counters to cover larger sections of code. You can also reduce the number of counters by restricting the range of code being profiled.

- Profile counter size

Select the size of the profiling counters. By default each counter is 16 bits, to save memory. However, if your application is long-running and the 16-bit counters are overflowing, you can increase the size to 32 bits. Note that 32-bit counters require more memory.

You can also increase the bytes-per-count value and/or restrict the range of the program being profiled to compensate for the extra memory required if you use 32-bit counters.

- Countdown interrupt ticks

Use the text box to adjust the frequency with which profiling interrupts occur. The value shown is the number of timer clock ticks between each profile interrupt. You can increase accuracy by reducing the number of ticks. However, your program then runs slower. The minimum number of ticks that is possible varies.

NOTE This parameter is not supported for all run-time environments.

Click **Send** to send profiling parameters to the application. Any parameters you specify remain local to the **Profiling** window until you click **Send**, at which time the entire set of configuration data is transmitted to the application. After you click **Send**, the debugger displays a summary of the settings you sent.

Checking Statistical Profiling State

A colored indicator next to the **Send** button in the **Profiling** window shows the state of statistical profiling run-time support in the application. Colors mean the following:

Red	The application was not linked with statistical profiling support. Use the -p or -pg driver options when linking to include statistical profiling support.
Magenta	Statistical profiling support is present, but the application is still executing run-time start-up code, and profiling support has not been initialized.
Green	Statistical profiling support is fully initialized and functioning.

Profiling Using an ARCanel Board

If you compile and link your application for profiling using options **-Xtimer0** or **-Xtimer1** and **-p** or **-pg**, you can select profiling to configure and view statistical profiling output.

After you have compiled and linked the executable, you can view the statistical profile using the MetaWare debugger or the GUI-based integrated profiling tool from ARC International. See the *User's Guide* for the tool for details.

Compiling

The following example shows how to compile and link a program with statistical profiling information, saving the executable as `queens.out`:

```
mcc target_processor -p -Xtimer0 queens.c -o queens.out
```

Be sure to specify your *target_processor* (such as **-a5**) as well.

TIP To avoid memory-allocation errors, compile with **-Hheap=N** where *N* is about half the text size (depending on the granularity of time slicing).

For debugging on the simulator, compiling with **-p** or **-pg** is not required and is not recommended.

Debugging

Debugging does not require an option to enable profiling. The following example shows the command-line debugger being invoked on `queens.out` with profiling output enabled for an ARCangel board:

```
mdb -hard queens.out
```

Timer Displays

The debugger provides peripheral displays for the extension timers. To enable the displays on the simulator, do one of the following.

GUI

1. Start the debugger and click the **Debugger Options** button. If the debugger is already running, select **Tools | Debugger Options**.
2. Select **Target selection** and select the target **MetaWare debugger ARC Simulator**.
3. Select **Simulator extensions | Instruction extensions**.
4. Select the **Enhanced programmable timer** you compiled with **(0 or 1)**.

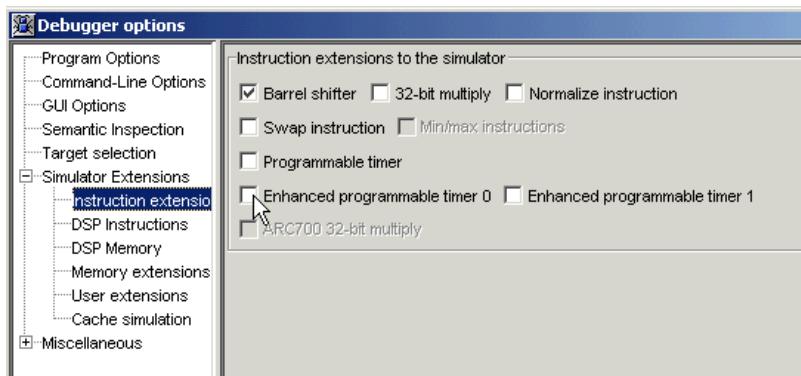


Figure 25 Selecting an Extension Timer

5. Click **OK**. If the debugger is already running, you must exit and restart it for the settings to take effect.

To open the displays, choose **Display | Hardware**, and select your extension timer.

Command Line

Specify option **-Xtimer0** or **-Xtimer1**.

To view the timers on the command line, use the command **display timer0** or **display timer1**.

Disabling the Profiling Window for Faster Simulation

The simulator can run your application slightly faster if you disable profiling

GUI

1. Select **Tools | Debugger Options**.
2. Click the **Command-Line Options** tab.
3. Deselect **Enable profiling window**.

Command Line

Specify option **-noprofile** when invoking the debugger, or in your `.args` file.

Viewing Source-Code Coverage

You can see a list of times a specific line of code was called or if it was not called at all using the source-code-coverage feature. Source-code-coverage analysis involves counting the number of times each instruction is executed in the simulator, which allows the debugger to determine whether a source statement has been executed. Source-code-coverage analysis consists of listing source lines along with execution counts, and showing the lines that have not been executed.

Source-code-coverage analysis lets you view the following:

- A column that shows a count of executions for each statement in a **Source** window
- A list of statement execution counts in a **Statement counts** window

Enabling Source-Code-Coverage Analysis

To enable analysis of source-code coverage, you must compile or link using option **-g** (generate debug information) or toggle `emit_line_records` (see [Enabling Code-Coverage Analysis for Selected Modules](#) on page 102).

Enabling Code-Coverage Analysis for Selected Modules

The MetaWare debugger provides code coverage analysis for modules compiled with source line information. The compiler emits source line information for a module when you compile that module with **-g** or toggle `emit_line_records` (on supported targets). You can limit code coverage to selected modules by compiling only those modules with source line information.

Compiling modules with **-g** may alter code generation significantly to improve source-code debugging. However, compiling using `emit_line_records` provides only line number information and may be a better choice if you want better code optimization.

See documentation for toggle `emit_line_records` and optimizations in the *MetaWare C/C++ Programmer's Guide* for more information.

Viewing Source-Code Coverage

To view statement execution counts for source-code coverage:

1. Open a **Source** window by selecting **Display | Source**.
2. Right-click in the **Source** window, select **Profiling**, and select **Stmt cnt from icnt** to view statement execution counts in a column in the **Source** window.

3. Right-click in the **Source** window, select **Profiling**, and select **Stmt cnt from icnt display** to view a **Statement count from Instruction counts** window.

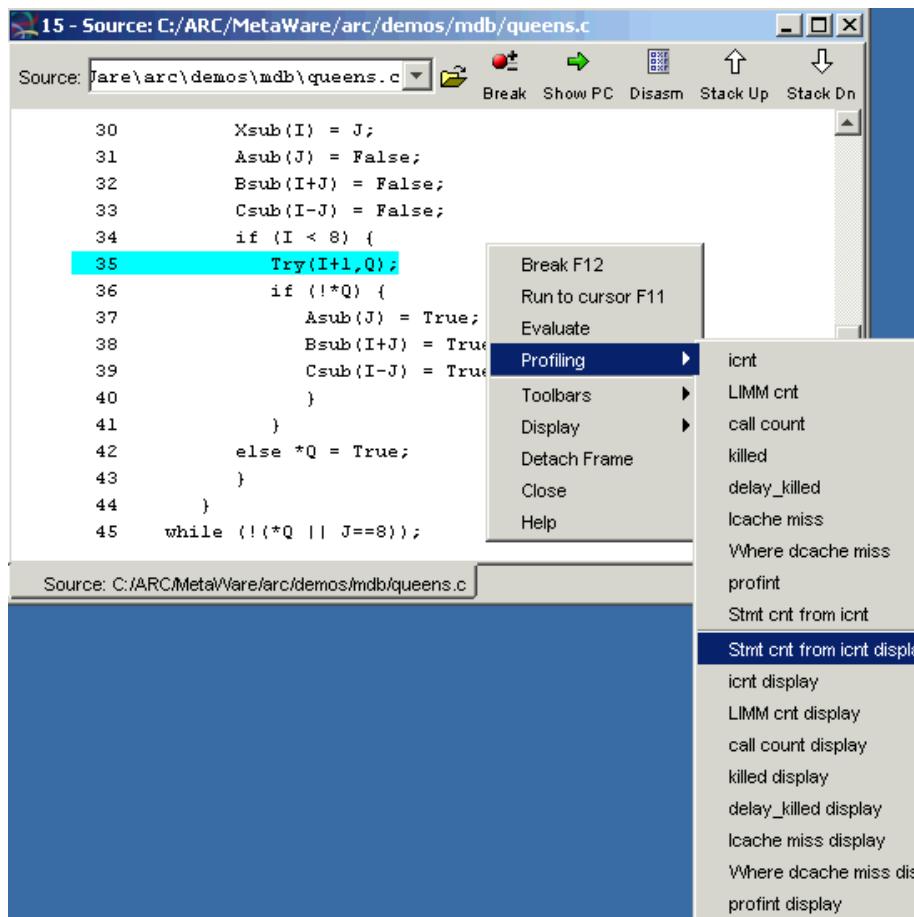


Figure 26 Viewing Code Coverage

Viewing Statement Counts

You can also open a **Statement count from Instruction counts** window to list each statement and the number of times it was executed.

1. Open a **Source** window by selecting **Display | Source**.
2. Right-click in the **Source** window, select **Profiling**, and select **Stmt cnt from icnt display** from the lower half of the menu that appears.

The number of statement counts is derived from the number of times the first instruction of the statement was executed.

[Figure 27 Statement count from Instruction counts Window](#) shows an example, with the number of statement counts appearing on the left. The Simulator records the data derived from the first instruction that was executed.

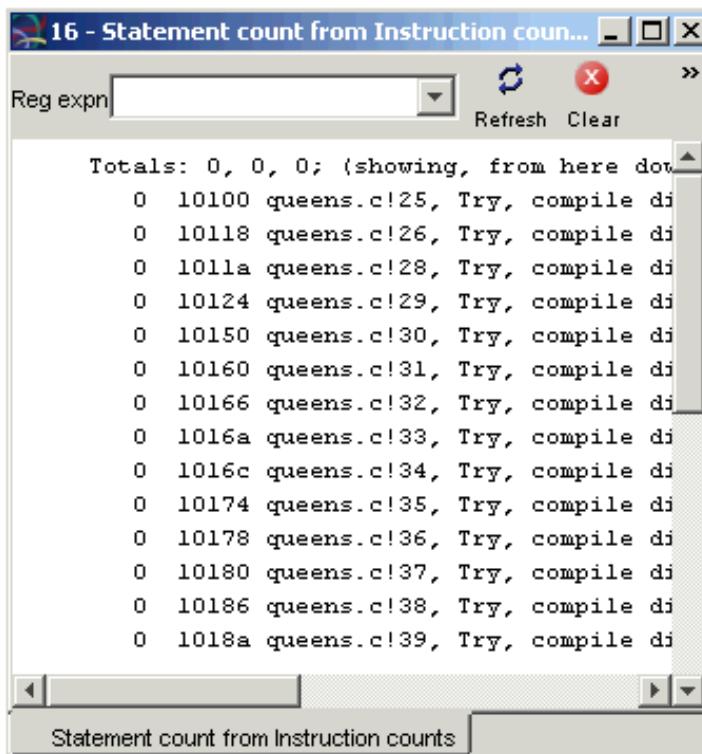


Figure 27 Statement count from Instruction counts Window

TIP You can use CTRL+F or CTRL+B to page forward or backward.

Viewing Source Code for a Statement

Double-click on a line in the statement to view the line of code associated with the statement in the **Source** window.

Sorting Entries

Click the [By file/line] button to sort the entries in the window by file and then by line number, rather than by count.

Save Data to a File

Right click and select **Save to File** to save the data to a file.

Viewing Statements not Executed

If you scroll to the bottom of the **Statement count** window, you can observe the statements that were never executed and thus not “covered.”

Viewing and Modifying Registers and Flags

The **Registers** window lists the names and contents of machine registers, including any flags register, on the system where your program is running. Choose **Display | Registers** to open the **Registers** window.

NOTE The **Registers** window for your target may differ in appearance.

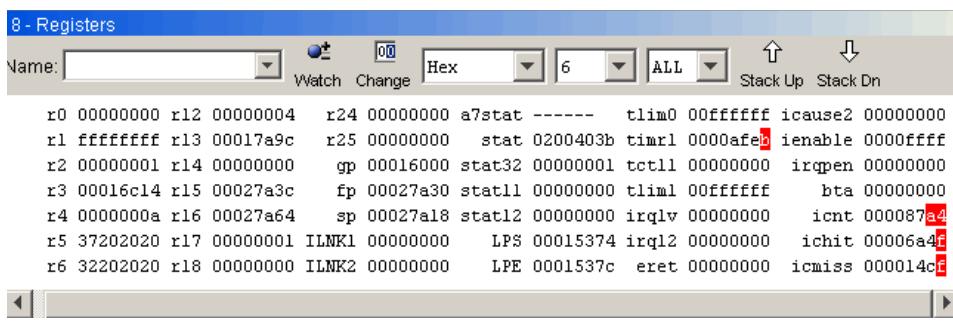


Figure 28 Registers Window

If process execution causes the contents of a register to change, the **Registers** window highlights the changed digits in a different color.

TIP You can use CTRL+F or CTRL+B to page forward or backward.

Setting a Watchpoint on a Register

To set a watchpoint on a register:

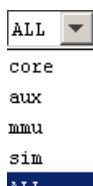
- Click the register name, and then click the **Watch** button.
- Or
- Right-click the register value, and then select **Watch**.

The left margin displays a colored indicator when one or more watchpoints are active on a line of the display, and a gray indicator when all watchpoints on that line of the display are disabled. For more information about watchpoints, see [Debugging with Watchpoints](#) on page 66.

Changing the Registers Window

To display only some registers, do any of the following:

- Choose a subset of registers from the drop-down list in the **Registers** window:

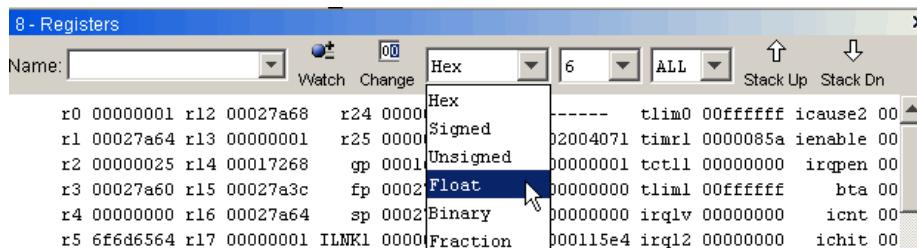


- Enter a register name or a regular expression in the **Name:** field to restrict the display to a single register, or to registers whose names match the regular expression (see [Using Regular Expressions](#) on page 221).

To change the number of register columns, choose a different value from the **Registers per line** list: **1**, **2**, **3**, **4**, or **auto**; **auto** adjusts the number of columns to fit the window.

Changing the Radix and Format of Displayed Values

To change the radix and format (numeric base and data type) of displayed values, select your choice from the drop-down list in the **Format** box:



Changing Register Contents

To modify the contents of a register:

1. Click the register whose value you want to change, to select it.
2. Click the **[Change]** button or right-click and select **Change**.

The **Change** dialog opens.

3. In the **New value for...** input field, enter a new value (decimal or hexadecimal).
4. Click **[OK]** or press ENTER.

The debugger replaces the current contents of the named register with the specified value.

For ARC targets, you can specify the value for the **IDENTITY** register of the instruction-set simulator. This is useful to make the ISS more closely match your hardware, so that your test code can run on the ISS as if it were running on actual hardware. Use the command

-prop=ident=value

where *value* is the value you desire.

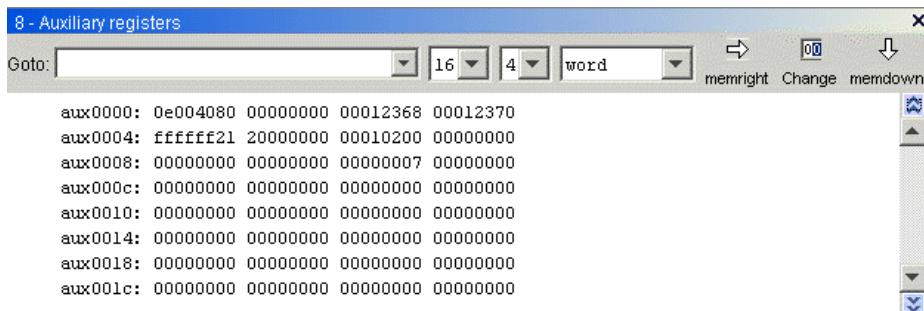
TIP You can add any auxiliary register to the **Registers** window when you launch the debugger by specifying **-prop=use_aux=N,name** where *N* is the auxiliary-register number and *name* is the auxiliary-register name.

Viewing and Modifying Auxiliary Registers

You can view information about auxiliary registers in the **Auxiliary registers** window. Values that have changed are highlighted as you step through your program.

Viewing

To open an **Auxiliary registers** window, select **Display > Aux registers**.



You can use the **Goto:** input field to view auxiliary registers starting at a particular auxiliary-register number. Enter a decimal number *without* “aux.”

You can use the drop-down menus to change the display parameters: [16] [4] [word]

- Number format (base)
- Registers per line
- Size of data displayed

Click the **memright** or **memleft** button to display by decreasing or increasing offsets.

To display increasing or decreasing addresses, click the **Display increasing/decreasing addresses**



button . The button changes direction after you click it.

Modifying

To change the value of a register:

1. Click the register to select it.
2. Click the **Change value** button or right-click and select **Change**.
3. Enter the new value and click **OK**.

Note Not all register values can be changed.

Command Line

See [Reading Registers](#) on page 193 and [Writing Auxiliary Registers](#) on page 197.

Debugging Multi-Tasked/Threaded Applications

This section describes how to use the debugger’s thread and task-specific features. For information about these features as they relate to a particular operating system, and for an introduction to threads and tasks in general, see [Multi-Tasked RTOS Applications](#) on page 313.

Note The words *thread* and *task* are used interchangeably throughout this guide. The debugger itself is RTOS-aware and labels the affected windows “thread” or “task”

depending on the specific RTOS. For example, the debugger refers to *task* for Precise/MQX™ and Nucleus PLUS, and refers to *thread* for ThreadX.

Enabling Multi-Task/Multi-Thread Debugger Features

To enable the multi-task/thread features of the debugger, you must first inform the debugger what operating system your application program is running on. Operating systems differ in the way the debugger obtains information about threads. For information on how to specify the operating system in the debugger, see [Specifying the Operating System](#) on page 314.

Viewing All the Tasks or Threads in Your Program

Generally, when a multi-task/multi-thread application starts, tasks/threads have not yet been created. After the application has run long enough to start up the threads or tasks, the debugger automatically detects the presence of threads or tasks. When the application stops, the debugger finds out from the operating system which task or thread is current.

The debugger then adds these items:

- a **Task Lock** menu (or a **Thread Lock** menu)
- a **Display | Tasks** command for opening the **Tasks** window (or a **Threads** window)
- a **Display | <RTOS> state** command that allows invocation of the RTOS window
- a **Display | <RTOS> state** command for opening an operating-system resources window (the actual name of this command depends on the operating system).

The **Task Lock** menu (or **Thread Lock** menu), lists the created tasks, indicates the current task, and provides a mechanism for locking a set of windows to one task.

The **Tasks** window (or **Threads** window), a scrollable list of all the tasks in your program, provides an alternate way to access the information and operations found on either the **Tasks** menu or the **Threads** menu.

The **<RTOS> State** (real-time operating-system resources) window lists the operating-system objects relevant to your application with multiple tasks. The actual name and appearance of the **<RTOS> State** window depends on the operating system. For a description of the components and features of the **<RTOS> State** window for your operating system, see the appropriate section in [Multi-Tasked RTOS Applications](#) on page 313.

To open the appropriate window and view all the tasks in your program:

1. Run your program to a point where the tasks are known. At this point all operating-system resources are defined and available to the debugger. [Multi-Tasked RTOS Applications](#) on page 313 provides operating-system-specific details about how to get your program to this point.
2. Display the tasks by selecting **Task Lock**. If your application has so many threads or tasks that the menu no longer fits on a screen, choose **Task Lock | More tasks...** or use **Display | Tasks** to view the list of tasks in the **Tasks** window rather than on a menu.

Viewing the State of a Specific Thread or Task

When the application you are debugging stops, any data windows (such as **Disassembly**, **Source**, and **Local Variables** windows) reflect the state of the current task. To display state information for some other task:

1. Open the **Tasks** window.
2. Double-click the line displaying the task whose state you want to view, or choose the task from the **Tasks** menu.

All windows automatically switch to show the state of the task you selected; in other words, this set of windows is now “locked” to the selected task. Any commands you type in the **Command:** input field (such as **reg** or **stack**) reflect the state of this task.

To switch back to the current task, do one of the following:

- Click the **current task** line in the **Tasks** window.
- Choose **Tasks | Current task**.

Viewing the State of Several Tasks at the Same Time

The debugger provides a mechanism for cloning (duplicating) its GUI desktop and any open windows associated with it. By cloning a set of windows and then locking a different task to each cloned set, you can view the state of several tasks in your application concurrently.

A colored stripe appears at the top of all windows locked to a specific thread. Each cloned window set has a different stripe color.

Each window locked to a given task shows the task ID in its title bar. The word “current” also appears in the title bar of the current task.

Locking a Cloned Set of Windows to a Thread or Task

To lock cloned sets of windows to different threads:

1. Choose one of the following:
 - To duplicate the desktop only, select **File | Clone desktop**.
 - To duplicate the desktop and all its open windows, select **File | Clone desktop and its frames**.
2. Open a **Tasks** window and select **Display | Tasks or Tasks | More tasks...**
3. Select a task by clicking the line displaying that thread, then clicking **[Tasklock]** on the **Tasks Options** toolbar. Or simply double-click the task. Doing this locks all the windows in the cloned window set to the selected task.

NOTE As a shortcut to steps 2. and 3., select the desired task from the **Tasks** menu if the task you want appears there.

Using Task- or Thread-Specific Breakpoints

This section discusses breakpoints for multitasked applications when debugging a single process; for task-specific breakpoints when debugging multiple processes, see [Setting Thread-Specific Breakpoints for a Process](#) on page 167.

Setting a Task-Specific Breakpoint Directly on a Line of Code GUI

To set a task-specific breakpoint on a line of code in the **Source** window or the **Disassembly** window:

1. Lock the window to the desired task, in either of these ways:
 - In the associated desktop, choose the desired task from the **Tasks** menu.
 - In the associated **Tasks** window, double-click the desired task.

2. Right-click a line of code (with or without selecting it first) and select **TS break** from the popup menu.

When you set such a task-specific breakpoint on a line of code, a ‘T’ breakpoint symbol appears to the left of the line.

CAUTION As with other breakpoints, double-clicking the task-specific breakpoint in the **Source** or **Disassembly** window removes the breakpoint.

Command Line

Use the [**break**](#) command with the \$tid pseudo-variable (which is always set to the current thread.). See [Commands Reference](#) on page 231.

Setting a Task-Specific Breakpoint in the Set Breakpoint Dialog

To set a task-specific breakpoint using the **Set Breakpoint** dialog:

1. Open the Set Breakpoint dialog by selecting **Tools | Break/watch**, or by clicking the **[Set]** button on the **Breakpoint Options** toolbar.
 2. Specify the location of the breakpoint in the **Set Breakpoint On:** input field. (See step 2. in [Setting a Breakpoint Using the Set Breakpoint Dialog](#) on page 62 for details.)
 3. In the **Tasks** field (below the **Condition:** input field), click the Down arrow to see a list of your application’s task, then choose a task by clicking it.
 4. If you want the breakpoint to be any of temporary, counted, or conditional:
 - Click the **temporary** check-box.
 - In the **Count:** input field, enter the number of times you want the breakpoint to skip before it stops execution.
 - In the **Condition:** input field, enter a conditional expression.
- See [Setting Task- or Thread-Specific Breakpoints](#) on page 63 for details.
5. Click **[OK]**.

Viewing Task-Specific Breakpoints

You can observe that a breakpoint is task-specific as follows:

- In the **Breakpoints** window, the breakpoint indicator contains the letter ‘T,’ the breakpoint contains the label “task specific,” and the condition shows the task number (because a task-specific breakpoint is a conditional breakpoint).
- In the **Source** and **Disassembly** windows, the breakpoint indicator contains the letter ‘T’ if the breakpoint is task-specific.

Using Thread-Specific Watchpoints

Setting a Task-Specific Watchpoint Directly on a Variable GUI

To set a task-specific watchpoint on a variable in the **Local Variables** window, the **Global Variables** window, or the **Examine** window:

1. Lock the window to the desired task, either of these ways:
 - Select the desired task from the **Tasks** menu
 - In the associated **Tasks** window, double-click the desired task.
2. In the **Local Variables**, **Global Variables**, or **Examine** window, click the line containing the variable to select it.
3. Click [**Watch**].

NOTE No watchpoint symbol appears next to variables that have watchpoints set on them. To determine whether a variable is being watched, you must open the **Watchpoints** window.

As with other watchpoints, double-clicking the line containing the task-specific watchpoint in the **Watchpoints** window removes that watchpoint.

Command Line

Use the [watch](#) command with the \$tid pseudo-variable (which is always set to the current thread.). See [Commands Reference](#) on page 231.

Setting a Task-Specific Watchpoint Directly on a Register GUI

NOTE Before setting any watchpoints on registers in the **Registers** window, it is a good idea to set the **Per Line** field to 1 (one), so that only one column of registers appears in the text area.

If more than one register appears on a line, the watchpoint indicator appears to the left of the entire line, showing that a watchpoint has been set on one or more of the registers in the line. This indicator does not show which register(s) are being watched.

To set a task-specific watchpoint on a register in the **Registers** window:

1. Lock the **Registers** window to the desired task, either of these ways:
 - Select the desired task from the **Thread Lock** menu
 - In the associated **Threads** window, double-click the desired task
2. Click the register to select it.
3. Click [**Watch**] on the **Register Options** toolbar (or right-click in the text area to open the right-click menu, then left-click [**Watch**]).

When a task-specific watchpoint is set on a register, a watchpoint symbol appears to the left of the line containing the register in the **Registers** window.

Command Line

Use the [watchreg](#) command with the \$tid pseudo-variable (which is always set to the current thread.). See [Commands Reference](#) on page 231.

Setting a Task-Specific Watchpoint in the Set Watchpoint Dialog

To set a task-specific watchpoint using the **Set Watchpoint** dialog:

1. Open the Set Watchpoint dialog by choosing Tools | Set Watchpoint or clicking [**Set**] on the **Watchpoint Options** toolbar.
2. Specify the location of the watchpoint in the **Set watchpoint:** input field. (See step 2. in [Setting a Watchpoint in the Set Watchpoint Dialog](#) on page 68 for details.)

3. In the **Tasks** field (below the **Condition:** input field), click the down arrow to see a list of your application's task, then choose a task by clicking it.
4. If you want the watchpoint to also be conditional, enter a conditional expression in the **Condition:** input field.
5. Click **[OK]**.

Viewing Task-Specific Watchpoints

In the **Watchpoints** window, the watchpoint indicator contains the letter 'T,' the watchpoint contains the label "task specific," and the condition shows the thread number (because a thread-specific watchpoint is a conditional watchpoint).

Non-Symbolic Debugging

This section discusses issues to be aware of when debugging programs compiled without debug information.

Debugging Files without Debug Information

The MetaWare debugger can debug an executable compiled without option **-g**. Such a file does not contain explicit debug information, such as the data type of each variable and function, the correspondence between source line number and addresses in the executable code, and so on.

In this case, the debugger reads the executable file symbol table. This table contains some useful information, but not enough to support full source-level, symbolic debugging.

Debugging Executable Files

Executable files compiled without turning on debug information contain no source-file information, so the debugger cannot perform statement stepping, and the **Source** window is always empty.

Executable files do not provide type information, so expressions involving built-in and user-defined types are not available to the debugger. There is no knowledge of scopes or local variables in executable files, so the **Local Variables** window does not list local variables. Similarly, there is no knowledge of local functions in executable files.

Executable files provide enough information for the **Global Variables** window to display values. How a value is displayed depends on its size:

2 bytes:	unsigned short
1 byte:	unsigned char
all others:	unsigned int

The **Disassembly** window lists the names of functions and global variables. You can set breakpoints on global functions.

The **Call Stack** window lists the names of the functions, but the type and number of the arguments is not known.

Debugging Stripped Object Files

If you have stripped the object-file symbol table with a **strip** utility, even minimal symbolic information is not available to the debugger, and it cannot display any global variables or global

functions. The **Global Variables** window is always empty. The only way to set breakpoints in such a file is by address.

TIP If you are debugging a stripped file, you can set the address of __HOSTLINK__ using
-prop=__HOSTLINK__=address.

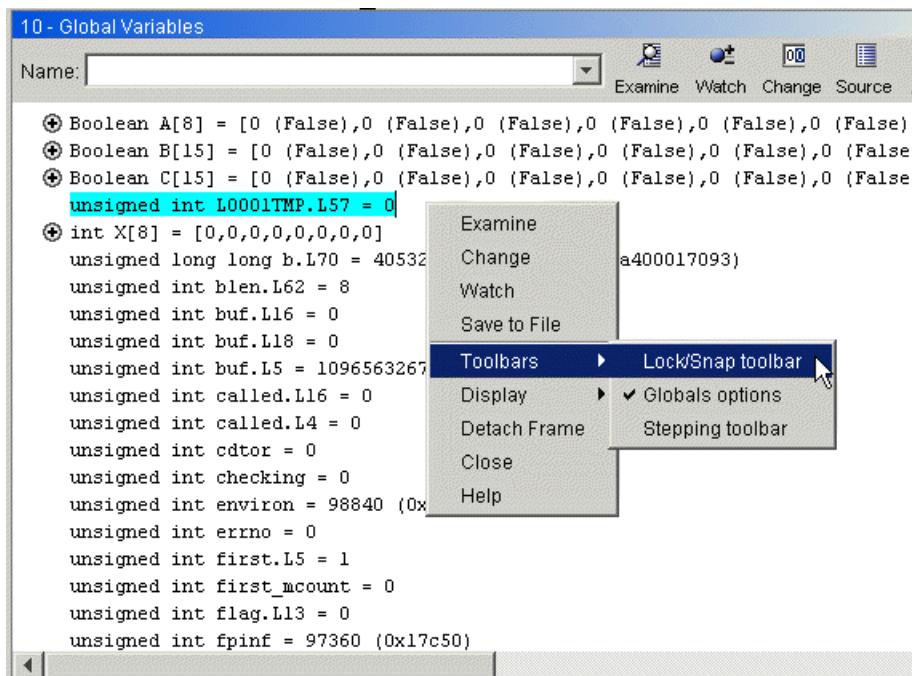
Taking Snapshots of a Window

You can take snapshots of windows at different times as you run or step through your program and use those snapshots to compare the state of items such as variables and register values as execution proceeds.

Example of Using the Snapshot Feature

The following example shows how to take snapshots of variable states in the **Global Variables** window.

1. Start the debugger.
2. Set a breakpoint in your program, See [Debugging with Breakpoints](#) on page 60.
3. Open the **Global Variables** window by selecting **Display | Global Variables**.
4. Right-click in the **Global Variables** window and select **Toolbars | Lock/snap toolbar** from the menu that appears.



The **Lock/Snap** toolbar appears at the top of the window.



Taking Snapshots of a Window

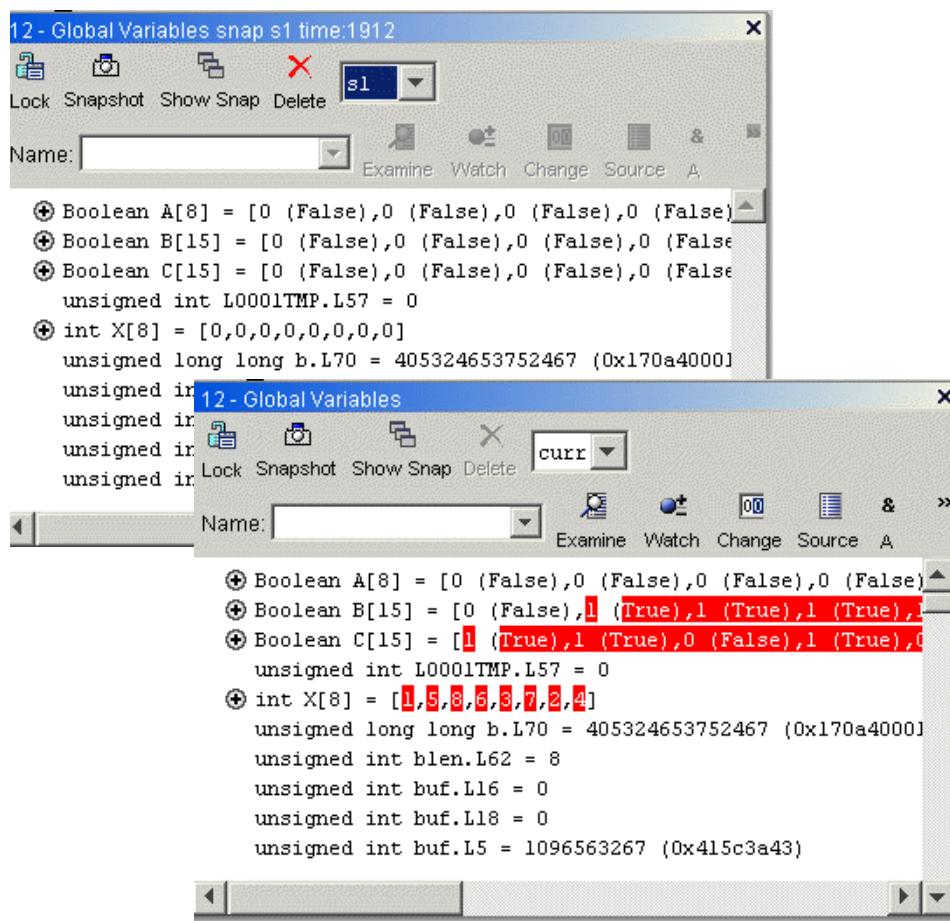
5. Click **Snapshot** to save a snapshot of the current display. Additional buttons and a drop-down menu are added to the toolbar. You can use the **Show snap** button to cycle through the available snapshots, the drop-down list to select a snapshot to view, and the **Delete** button to delete snapshots.



6. Click **Run** to run to the breakpoint you set.
7. Click **Snapshot** to save another snapshot of the current display.

As execution proceeds and the contents of the window change, you can take additional snapshots. Every time you click Snapshot, the debugger adds a snapshot to the drop-down list of snapshots.

8. Scroll through the snapshots using the drop-down menu. You can display a snapshot by choosing it from the snapshots list. The number of seconds since the program started appears in the title bar of the window. The label **curr** appears next to the current display.



Note that the window buttons are dimmed (unavailable) in the snapshots.

Chapter 4 — Debugging a Target

In This Chapter

- [Overview](#)
- [Determining the Target System](#)
- [Using the Instruction Set Simulator](#)
- [Debugging Remotely with SCIT](#)
- [Debugging ARC Systems](#)
- [Flash Programming](#)

Overview

This chapter contains information that pertains to debugging using a simulator or hardware with your specific target processor.

Determining the Target System

You can use the MetaWare debugger to debug programs running on any of the following target systems:

- simulator
- target processor
- multiple simulators or target processors

The debugger inspects your program during startup to determine the system on which to run it. If the debugger supports both a simulator and hardware system for the target, it selects the first system it finds, which is the “preferred” or “default” system. The debugger typically uses the simulator as the default system. You can check what system you are debugging on by scrolling to the top of the output shown in the **Debug Console** (**Windows | Debug Console**) after you launch the debugger.

You can change the default system by instructing the debugger to change or override the default system, as described in [Changing the Default System](#) on page 116.

Changing the Default System

On some targets, when you specify command-line option **-sim** or **-hard**, the debugger places the appropriate systems ahead of the others in the list, so that they are effectively defaults.

For example, to debug on hardware when the default system is the simulator, use the following option:

```
mdb -hard
```

Overriding the Default System

You can also debug on a non-default system by adding a prefix to the name of the executable file when you launch the debugger using a command prompt. When the executable file’s name is prefixed by *NNN*[^], the debugger searches for a system with *NNN* in the system’s name, and uses that system instead of the default system.

For example, if the default is the MetaWare debugger instruction set simulator, you can set the system to hardware by specifying:

```
mdb hard^a.out
```

Conversely, if the default system is hardware, you can debug on the simulator by specifying:

```
mdb simu^a.out
```

NOTE On Windows you must use two ^ symbols on the command line, because the Windows command processor consumes one. For example:

```
mdb hard^^a.out
```

Using the Instruction Set Simulator

The MetaWare debugger provides an instruction-set simulator (ISS) for each target, in big- and little-endian variants where applicable. MetaWare debugger instruction set simulators have additional capabilities that target hardware may not have. You can use the instruction set simulator to set breakpoints and watchpoints, trace instructions, view the last instruction, and add simulator extensions.

Stopping Execution by Instruction Count

The simulator does not allow a watchpoint to be set on the `icnt` register. To stop execution in the simulator after a certain number of instructions, use the following command:

After you enter this command, the next `run` command executes only N instructions and then stops. At that time the `stop_after` property has no further effect on `run` (until you set it again).

Tracing Instructions

When you specify this command, the simulator enters tracing mode, and prints each instruction as it is executed:

```
prop trace=1
```

Viewing the Last Instructions Executed

You can use command `prop lastpc` to view a limited amount of execution history. When you enter the following command, the simulator prints the PC addresses of the last N instructions executed:

```
prop lastpc=N
```

This facility is useful if, for example, your program has stopped and you do not know how it got to that point. You can also enter either of these commands:

```
trace  
trace NNN
```

The first command displays a small number of lines, and the second command displays NNN lines. You can also use the **Instruction history -- Simulator trace** window (see [Working with Instruction History](#) on page 57).

To change the maximum number of PC values saved by the simulator, specify:

```
prop maxlastpc=NNN
```

NNN is adjusted upward to the next power of 2.

Cloning the Simulator

You can clone the little- or big-endian simulator system to run more than one simulation in the same debugger session. Use the command `sysclone` to clone a system.

Using the `sysclone` Command

You can use option `-c` to add the `sysclone` command to every debugging session.

In the following example the debugger executes the `sysclone` command before starting the `queens` program:

```
mdb red^queens -c="sysclone ARC_simulator red"
```

You can specify as many `-c=command` options as you like. If you add them to the `ARGS=` line in driver configuration file `bin/mdb.cnf`, these options will take effect every time you run the debugger. You

can also put the commands in a file, and specify `-c="read filename"`. For example, the `mdb.cnf` file might contain:

```
sysclone BRC_simulator bigsim
sysclone ARC_simulator littlesim
sysprop bigsim mem=8000000
sysprop littlesim mem=50000
```

These commands define two new simulators with different memory sizes. The **sysprop** command provides a way to specify properties to a system. The debugger uses **sysprop** to communicate values to the various systems. See [Commands Reference](#) on page 231 for more information about **sysclone** and **sysprop**.

Adding Simulator Extensions

You can extend the instruction set simulator using the Simulator Extension API:

- Declare ownership of memory regions so you can implement a memory-mapped device such as a peripheral.
- Add an extension that generates interrupts. For example, you can add an extension that implements an interrupt timer.
- Add your own core and auxiliary registers (up to 64), condition codes, and instructions.

You can find more information in the *MetaWare Debugger Extensions Guide* or in the file `simext.h`, which is located in the following directory:

```
install_dir/mdb/inc/simext.h
```

See also option **-simext*** in [Command-Line Option Reference](#) on page 274.

Debugging Remotely with SCIT

SCIT enables the remote debugging of a target processor across a network, in simulation or actual hardware. The debugger normally connects to a target via a well-defined target-specific interface, typically implemented by a target DLL (shared object). SCIT allows the target DLL to reside on another machine on a network, and thus the debugger and its target can be on separate machines potentially miles apart.

NOTE Network access to large programs can be quite slow, as the debugger makes random access to the program file to load the symbol table.

SCIT consists of a client and a server. The server connects to the target DLL and thus the target. The debugger connects to the server via the client and can debug the target as if it were local to the debugger host. SCIT thus creates a transparent communication link between the debugger and a remote target.

[Figure 29 Normal MetaWare Debugger Operation](#) is an illustration of the normal operation of the MetaWare debugger.

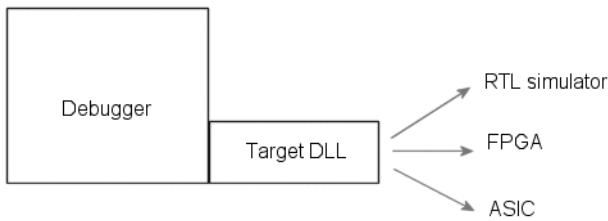


Figure 29 Normal MetaWare Debugger Operation

A DLL implements the interface to the target; the interface is target-specific and is defined by the debugger. The target can be either the CPU in simulation, or actual hardware; the MetaWare debugger is generally unaware of the distinction, which is hidden by the target DLL. SCIT transports the target interface across a network: [Figure 30 MetaWare Debugger with SCIT](#) shows the operation using SCIT.

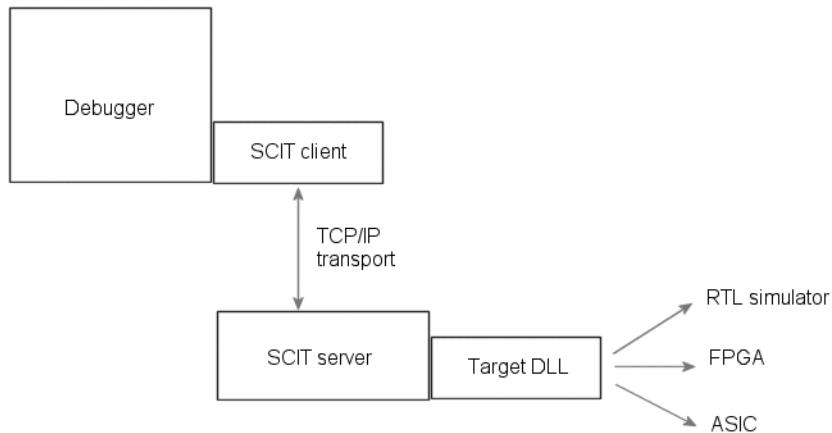


Figure 30 MetaWare Debugger with SCIT

With SCIT, the target DLL and the target it connects to can reside on separate machines. Instead of connecting to the MetaWare debugger, the target DLL is connected to the SCIT server. The SCIT client connects to the MetaWare debugger instead, and transports the target interface via TCP/IP to the SCIT server.

SCIT Usage

SCIT consists of the following files:

- Client: *targetcli.dll* (Windows) or *targetcli.so* (UNIX and Linux)
- Server: *targetsrv.exe* (Windows) or *targetsrv* (UNIX and Linux)
where *target* is *arc* for ARCTangent-A4 or *ac* for ARCTangent-A5 and later.

These two files are in the `bin` directory of the debugger installation.

Both the client and server must be informed which TCP/IP port will be used to communicate; the default port is 10875 and can generally be used without changing it (unless you wish to concurrently run multiple servers on the same machine). The client generally must be told the address of the

machine on which the server resides; the client's default is `localhost`, meaning the server is on the same machine as the client.

Shared Memory

SCIT can also be used with shared memory (instead of TCP/IP) as the communications channel between the client and server. This is not recommended for client-server usage, but it is recommended for client-interface handler usage.

If shared memory is used, the client and server or interface handler must reside on the same computer. To use shared memory you specify (part of) the name of the shared memory conduit using the `shmem` parameter. As of this writing, shared memory doesn't "disconnect" like TCP/IP does when the client terminates. Thus a subsequent client connection does not cause the server to reset the target. If you use the `shmem` parameter (see [Client Arguments](#) on page 122), the communications channel is assumed to be shared memory; otherwise it is assumed to be TCP/IP. Shared memory channels remain after programs exit on UNIX; use the UNIX `ipcs` utility to see active shared memory keys; use `ipcrm` to remove them.

See also [Debugging an ARC Processor in an SoC](#) on page 145.

SCIT Startup

To use SCIT, the server must first be started, and given the target DLL implementing the connection to the target.

Example (ARC Target Shown)

```
arcsvr -DLL=target.dll
```

Optionally, you can specify the TCP/IP port and whether the server repeats. "Repeat," means that after the debugging session is over and the client disconnects, the server automatically restarts. This is useful for maintaining a constant service to a target that can be accessed from time to time from different hosts.

After starting the server, start the MetaWare debugger, specifying the SCIT client as its target connection.

GUI

Specify a SCIT connection in **Debugger Options** or **Target-specific options**, depending on your target. When you select a hardware connection and **Remote server via TCP/IP (SCIT)**, the GUI provides an area to enter the TCP/IP address and port of the SCIT server.

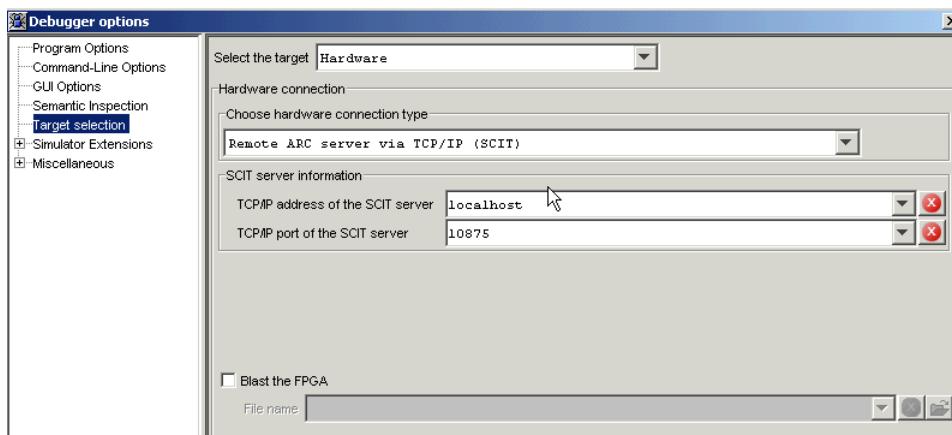


Figure 31 Specifying a SCIT Connection on GUI Startup (ARC Target Shown)

Command Line

```
mdb myprog.out -DLL=targcli,ipaddr=remote_machine
```

where *remote_machine* is the IP address of the machine running the server and, for non-ARC targets, *targ* is a target processor that supports SCIT.

After startup, you can proceed to debug normally. The IP address can either be the 32-bit TCP/IP address (e.g., 10.11.12.13) or the machine name (e.g., varoom). Using the TCP/IP address rather than the machine name is preferable as the client might not be able to map the name into an IP address for one reason or another. The port and address can also be specified via environment variables.

You can learn about the arguments to the server by running it without arguments. It also informs you of the arguments to the client. The arguments are as follows:

Table 4 SCIT Server and Client Arguments

Server Argument	Purpose
-ipport=N	Specify the TCP/IP port to be used. You must use the same for the target and server.
-DLL=N	Name of the target DLL
-repeat	Specify that the server is to allow another connection after a prior client disconnects. The server completely reloads the target DLL from each time a connection is made.
-shmem=id	Specify the identifier to use in constructing the shared-memory name. You must use the same <i>id</i> for the target and server. On UNIX, the <i>id</i> must be a decimal or hex integer; e.g., 123 or 0xabc.
-passwd=P	Specify a password for the server. If a password is specified, the client must specify the same password, or the connection is dropped.
-hold=sec	Specify a maximum hold time <i>sec</i> in seconds (relevant only for TCP/IP). After a connection is terminated, if the client so requests, only connections to the same client are permitted for a number of seconds which is the minimum of either the client request or <i>sec</i> . An unsuccessful connection attempt resets the clock.

Table 5 Server Environment Variable

Server Environment Variable	Purpose
SCIT_PORT	Specify value for -ipport . The argument -ipport takes precedence over the environment variable.
SCIT_SHMEM	Specify value for -shmem . The argument -shmem takes precedence over the environment variable.

Table 6 Client Arguments

Client Argument	Purpose
ipport=N	Specify the TCP/IP port to be used. You must use the same port for the target and server.
ipaddr=A	Specify the address of the machine running the server. Use either a numerical TCP/IP address (e.g., 10.11.12.13) or the machine name, if the client can translate the machine name into its numerical address.

Table 6 Client Arguments

Client Argument	Purpose
ipfile=filename	Each line in <i>filename</i> is of one of the following forms: # comment <i>A</i> <i>A:N</i> Where <i>A</i> and <i>N</i> take the same form as arguments to ipaddr and ipport respectively. Each address is tried in turn until a connection is made. The port used for each address is the one specified by <i>:N</i> ; if <i>:N</i> is not given, the port is the one specified by ipport=N ; if ipport=N is not used, the port is the default port. Example: # List of servers 192.9.200.125:11211 # The next two entries use the default port. bluebird 172.9.117.3
shmem=id	Specify the identifier to use in constructing the shared-memory name. You must use the same <i>id</i> for the target and server. On UNIX, the <i>id</i> must be a decimal or hex integer; e.g., 123 or 0xabc.
-passwd=P	Specify the password that is sent to the server. If the server was not started with a password, this parameter has no effect. Otherwise, the passwords must match. If they do not, or if you do not specify a password to a server started with one, the server drops the connection.
-hold=sec	Specify a maximum hold time <i>sec</i> in seconds. This argument is relevant only for TCP/IP, and for servers started with a -hold argument. See SCIT Server and Client Arguments on page 122 for more information.

Table 7 Client Arguments

Client Environment Variable	Purpose
SCIT_ADDR	Specify default for -ipaddr . The argument -ipport takes precedence over the environment variable.
SCIT_PORT	Specify value for ipport . The argument ipport takes precedence over the environment variable.
SCIT_FILE	Specify value for ipfile . The argument ipfile takes precedence over the environment variable.
SCIT_SHMEM	Specify value for shmem . The argument shmem takes precedence over the environment variable.

Specific Connection Instructions

The following enable you to connect SCIT to specific boards:

ARCangel3 (AA3) Development Board

Present on the server must be the AA3 DLL `apipar.dll` (for parallel port ARC builds) or `apijtag.dll` (for JTAG builds). These DLLs are present in the `install_dir/bin` directory.

If you want to blast the AA3 before you begin debugging, the file you wish to blast with must reside in the directory in which you started the server, and in the directory in which you started the debugger. The reason for this is that the debugger verifies that the blast file is present and exits otherwise. Furthermore, the DLL `aa3blast.dll` must be in the PATH on the server machine, or in the directory in which you started the server.

For example:

```
arcsrv -DLL=apijtag -repeat  
mdb myprog.out -DLL=arccli,ipaddr=server_machine
```

Note that the parallel port IO address is assumed to be 0x378 by the target DLL. If this is not so on the server machine, you must specify the parallel port address when invoking the debugger; for example:

```
mdb myprog.out -port=0x3bc -DLL=arccli,ipaddr=server_machine
```

Remote Blasting an ARCangel3

You can blast an ARCangel3 remotely using a file residing on the host or the SCIT server.

- To blast using a file on the host, use the `-blast=file_name` form of the [blast](#) option. This might take time as the file (in excess of 1 M) must be transmitted.
- To blast using a file on the server, copy the file (for example, `my_remote.xbf`) to the server and use the following [prop](#) specification:
`-prop=rblast=my_remote.xbf`

ARC RASCAL

Present on the server must be the RASCAL interface DLL `rascalint.dll` (.so).

For example:

```
arcdrv -DLL=rascalint -repeat  
mdb myprog.out -DLL=arccli,ipaddr=server_machine
```

If the communication between client and server is not fast, we recommend turning off toggle [show](#) [reg_diffs](#) so that the debugger will not acquire all the registers after each assembly-level instruction step command that you give.

SCIT Interface Handler Usage

This section is for SoC implementors interested in providing MetaWare debugger access to a processor embedded in a system-on-chip (SoC) simulation.

The SCIT server uses a target interface handler to process commands on the target. The handler allows a remote MetaWare debugger to communicate with the target; thus you can use the MetaWare debugger to debug a target embedded in an SoC simulation. For information on launching a remote MetaWare debugger for ARC processors, see [Debugging an ARC Processor in an SoC](#) on page 145.

The MetaWare debugger uses the same SCIT client (such as `arccli`). But instead of using the SCIT server, you connect the interface handler with your embedded target by providing the handler the

interface to your target. [Figure 32 Target Connection in an SoC Simulation Using a SCIT Handler](#) is a diagram of the connection.

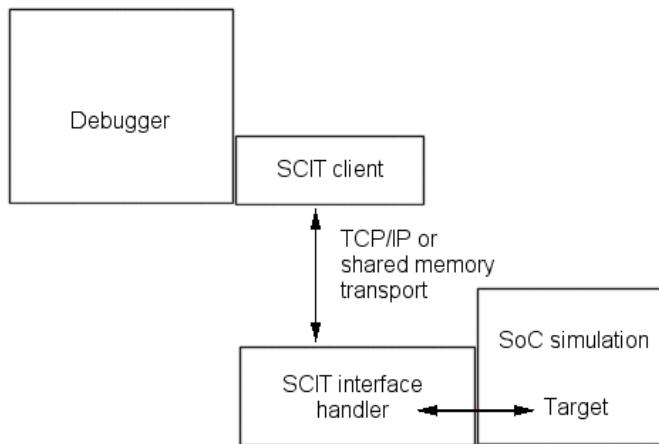


Figure 32 Target Connection in an SoC Simulation Using a SCIT Handler

The interface handler resides in a separate DLL that exports the following two functions:

```
SCIT_handler *get_CPU_handler(Interface_type *CPU, unsigned options)
SCIT_handler *get_multi_CPU_handler(Interface_type **CPUs, int count, unsigned options)
```

Interface_type is *ARC* (*arcint.h*). You provide the target interface when you invoke **get_CPU_handler()**, or an array of such interfaces for **get_multi_CPU_handler()**. Providing the target interface connects the handler with the target (the arrow between the handler and target in [Figure 32 Target Connection in an SoC Simulation Using a SCIT Handler](#)). The SCIT handler uses a dummy callback object unless bit 0 of *options* is 1 (see [Inserting a Processor](#) on page 126).

The resultant object has two methods of primary significance. The first is as follows:

```
int process_property(const char *key, const char *value)
```

This method modifies the default transport behavior of the handler. The value of *key* can be *shmem* or *ipaddr*, with the semantics the same as for the SCIT server (see [Table 6 Client Arguments](#) on page 122). Like SCIT, the handler uses TCP/IP (*ipaddr*) by default; change the value to shared memory (*shmem*) if using shared-memory access. You can also use the environment variable *SCIT_SHMEM*, just as for the SCIT server.

The second significant method is the following:

```
int process_command_if_present(int options)
```

This method is the mechanism by which the handler checks for and processes commands from a remote MetaWare debugger. You must call this method frequently enough to permit timely processing of MetaWare debugger requests. For details on the *options* parameter and the return value, see file *scit_handler.h*.

For multiprocessor handling, you provide an array *CPUs*[] of processors. The remote MetaWare debugger, when in CMPD mode, indexes the processors with the values 1..*count*, where *count* is the

number of CPUs. Thus your array must have an empty slot at 0. In other words, process N accesses CPUs [N].

Inserting a Processor

One way of inserting a processor into an SoC simulation is to drop in a target loaded by the MetaWare debugger using the debugger engine and execution vehicle access mechanism, where you use the MetaWare debugger engine to load and initialize your target. The target might be the ISS or a cycle-accurate simulator, for example. In this case the debugger engine initializes the target with an appropriate debugger callback object that gives the target access to certain facilities. Specify 1 for the *options* parameter to `get_CPU_handler()` so that the SCIT handler does not replace the callback object with a dummy. The SCIT standalone server supplies a dummy callback as there is no debugger present in the target.

At the time of this writing, the SCIT handler and MetaWare debugger must reside on machines with the same endianness. This precludes running the MetaWare debugger on Windows with a handler on UNIX, for example.

The debugger GUI runs threads that communicate with the target at a priority lower than normal. If your SoC simulation runs on the same host at normal priority it “starves out” the debugger GUI, resulting in erratic behavior. You can either run your SoC simulation at a lower priority, or specify the following Java flag to the MetaWare debugger to prevent the MetaWare debugger lowering priority:

```
-joff=change_run_thread_priority
```

For ARC processors, the SCIT interface handler DLL is `archand.dll` (Windows) or `archand.so` (UNIX or Linux).

Debugging ARC Systems

This section tells how to use the MetaWare debugger with ARC hardware and simulator target systems.

You can debug your program using the MetaWare debugger instruction set simulator, an ARCan gel FPGA development board, or a third-party DLL. The MetaWare debugger works with these ARC target systems:

Type of Target System	Little-Endian	Big-Endian
MetaWare debugger instruction simulator	<code>ARC_simulator</code>	<code>BRC_simulator</code>
ARCan gel FPGA development board	<code>ARC.hardware</code>	<code>BRC.hardware</code>
Customized third-party DLL	<code>ARC_DLL</code>	<code>BRC_DLL</code>

Use the system names when issuing commands to the debugger. For example, issue the following command to clone an ARCTangent-A4 big-endian simulator:

```
mdb -a4 red^queens -c="sysclone BRC_simulator red"
```

See also [Debugging with ARC_DLL or BRC_DLL](#) on page 148.

ARCan gel and simulation tools such as RASCAL are simply different targets that the debugger can attach to using a DLL.

When the debugger reports an inability to connect to the target, the problem may lie in the debugger, the target, or anywhere between them. After attempting several times to connect, the debugger can report only a conclusion such as the following, even if the problem is within the target:

Communications could not be established; aborting.
Download failed (2).

Be sure to check your cabling, target power supply, and target hardware when you receive such an error.

Related Topics

- [Specifying Memory Size and Location](#)
- [Simulating Load/Store RAM](#) on page 129
- [Using a Memory Map to Indicate an Offset](#) on page 129
- [Debugging with Overlays](#) on page 130
- [Debugging Closely Coupled Memory \(a Harvard Architecture\)](#) on page 132
- [Simulating DSP Extensions](#) on page 132
- [Debugging on ARCanel Hardware](#) on page 134
- [Using Smart Real-Time Trace \(ARC 600 Only\)](#) on page 139
- [Debugging ARC Media Extensions and SIMD Instructions \(ARC 700 Only\)](#) on page 143
- [Viewing Hardware Data and Instruction Caches](#) on page 144
- [Viewing Build-Configuration Registers](#) on page 145
- [Debugging an ARC Processor in an SoC](#) on page 145
- [Viewing the ARC 600 MPU Registers](#) on page 146
- [Displaying MMU and Exception Registers \(ARC 700\)](#) on page 146
- [Debugging with ARC_DLL or BRC_DLL](#) on page 148
- [Setting up an Ashling Opella Emulator](#) on page 149
- [Using the Cycle-Estimating Simulator \(CES\)](#) on page 150
- [Using the Cycle Accurate Simulator \(CAS\)](#) on page 155
- [Using ARC xCAM \(ARC 700 and ARC FPX\)](#) on page 155
- [Using the ARC xISS Fast ISS \(ARC 600 and ARC 700\)](#) on page 155

Specifying Memory Size and Location

You can add non-contiguous memory to the simulator using the simulator's user-programmable memory extension facility.

You can add memory by specifying each memory space or by increasing the total size of memory. Total size can be specified for hardware debugging as well.

GUI

To specify each memory space, proceed as follows:

1. Start the debugger and click the **Debugger Options** button. If the debugger is already running, select **Tools | Debugger Options**.
2. Select **Target selection** and select the target **MetaWare debugger ARC Simulator**.
3. Select **Simulator extensions | Memory extensions**.

4. Specify the memory beginning and end addresses in the **First addr** and **Last addr** fields.
5. Click **OK**. If the debugger was already running, you must exit and restart it for the settings to take effect.

To increase the total size of memory for simulator or hardware, proceed as follows:

1. Start the debugger and click the **Debugger Options** button. If the debugger is already running, select **Tools | Debugger Options**.
2. Select **Miscellaneous | Memory size**.
3. Select or enter the memory size in **Specify simulator or hardware memory size**.
4. Click **OK**. If the debugger is already running, you must exit and restart it for the settings to take effect.

Command Line

On the command line, the following example:

```
-memext=0x3_0000,0x3_ffff
```

specifies 0x1_0000 bytes of memory to add to the simulator. You can set up your own memory spaces by specifying **-mem**=0 and then specifying the spaces with option **-memext**. Alternately, you can increase the simulator's internal memory by specifying **-mem=mem_size**, where *mem_size* is the size of your internal memory (generally the same as the highest address).

If you do not specify any **-mem=mem_size**, then the simulator will allow any memory address to be used without generating an error. Use **-mem=mem_size** if you want to restrict the valid memory of the simulator to **0-<size-1>**, in which case the simulator will stop with an error, if your program accesses memory outside that specified region.

You can also use option **-memextinit** to specify the initial value of all added memory, or use **-memext** with a third argument to specify the initial value of the memory it adds (overriding any **-memextinit** value for that **-memext** address range).

Example 4 Options to Define Memory on the ARC Simulator

NOTE In an actual debug session, enter these options on a single command line.

```
-mem=0 \ -- Eliminate internal simulator memory so that we
          \ -- can redefine using -memext below
-memext=0,0x20000 \ -- TEXT
-memext=0x40000,0x80000 \ -- Data RAM
-memext=0x100000,0x120000 \ -- Data ROM
```

Example 5 Options to Add and Initialize Memory on the ARC Simulator

NOTE In an actual debug session, enter these options on a single command line.

```
-memext=0,0x20000 \ -- TEXT
-memext=0x40000,0x80000 \ -- Data RAM
-memextinit=0 \ -- initialize a77 added memory to 0
-memext=0x100000,0x120000 \ -- Data ROM
```

Example 6 Options to Add and Selectively Initialize Memory on the ARC Simulator

NOTE In an actual debug session, enter these options on a single command line.

```
-memext=0,0x20000 \-- TEXT
-memext=0x40000,0x80000, 0 \ -- Data RAM, initialized to 0
-memextinit \ -- Initialize all except data RAM to 0xdeadbeef
-memext=0x100000,0x120000 \ -- Data ROM
```

Simulating Load/Store RAM

To simulate load/store ram, do the following:

1. Start the debugger and click the **Debugger Options** button. If the debugger is already running, select **Tools | Debugger Options**.
2. Select **Target selection** and select the target **MetaWare debugger ARC Simulator**.
3. Select **Simulator extensions | Memory extensions**.
4. In the **load/store RAM** area, fill in the size and base address of your load/store RAM.
5. Specify any other memory values you require (see [Specifying Memory Size and Location](#) on page 127).
6. Click **OK**. If the debugger is already running, you must exit and restart it for the settings to take effect.

Command Line

When using the command-line debugger, specify the following properties to simulate load/store RAM:

```
-prop=ldst_ram_base=M
-prop=ldst_ram_size=N
```

Where *M* is the base address or your load/store RAM area, and *N* is its size in bytes.

NOTE Load/store RAM does not use simulated data cache. This causes differences in cycle estimation.

Using a Memory Map to Indicate an Offset

You can use the *delta* argument to the [**addmap**](#) command to indicate that an area needs to be accessed with an offset. Keep in mind that for ARC debug interfaces the width must be 4.

[**Example 7 Memory Map with Commands to Indicate an Offset**](#) maps all of memory as legal to prevent the debugger halting while things are in test mode.

Example 7 Memory Map with Commands to Indicate an Offset

```
memmap TESTMAP
addmap TESTMAP 0x00000000,0x00040000, width 4 // FLASH
addmap TESTMAP 0x00040000,0x40000000, width 4, delta 0x40000000
// CODE ROM
addmap TESTMAP 0x40000000,0, width 4 //RAM all rest of memory
prop memmap=TESTMAP
```

Entering the command [**memmap**](#) without an argument prints the memory map currently in use. Note that the end of each memory segment is actually one byte past it and 0xffff_ffff is represented with 0.

You might wish to place the commands in a chip-initialization (chipinit) file to be executed prior to program download and after target reset from the debug interface. Use of chipinit files is discussed in

Debugging with Overlays

This section describes the windows for debugging with overlays. The MetaWare debugger automatically understands applications that use the Automated Overlay Manager and takes care of the switching without distractive details, while providing the functionality to display which overlays are currently loaded.

For additional techniques, details about the Automated Overlay Manager, and command-line debugging of overlays, see the *Automated Overlay Manager User's Guide*.

Viewing Overlays

You can view overlays using the GUI or on the command line.

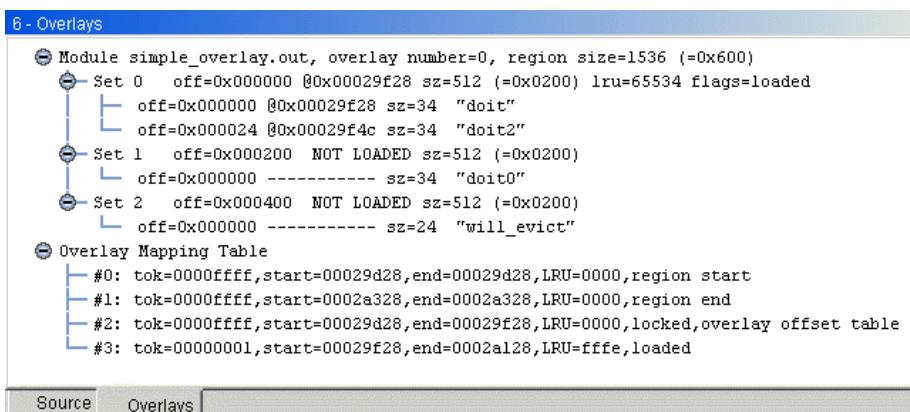
GUI

To open the **Overlays** window, select **Display > Overlays**.

The menu item is not available if the debugger does not detect overlays in the application.

The **Overlays** window is an XML tree view that displays the overlay executable name and memory-region size (as hex and decimal values), with nodes for each overlay set in the application.

NOTE Some items appear only for loaded overlays.



Source Overlays

Display for Each Module

- Filename
- Overlay package number (0 unless you specified one by linking the module with **-Boverlay_num=N**)
- Size of overlay region as a decimal and hex value

Display for Each Overlay Set

- Set number
- Offset within the overlay region in memory
- Address or NOT LOADED (if the overlay set is not yet loaded or is evicted)
- Size in bytes
- LRU value

This value is set to 0xffff and decremented each time the overlay is accessed. The Automated Overlay Manager uses this value to evict the least-recently-used overlay. Only loaded entries (functions) have an LRU. Data entries do not have one.

- Flags (bits in the overlay mapping table)
- Address in memory

Display for Each Function in an Overlay Set

- Offset within the set
- Address in memory
- Size in bytes
- Name

Display for the Overlay Mapping Table

- Index into the mapping table (e.g., #0)
- Linker-created token referring to the overlay set (e.g., tok=0000ffff)
- %pc start address for the entry
- %pc end address for the entry
- least-recently used value

This value is set to 0xffff and decremented each time the overlay is accessed. The Automated Overlay Manager uses this value to evict the least-recently-used overlay. Only loaded entries (functions) have an LRU. Data entries do not have one.

- Type of entry

Possible values are `region start`, `region end`, `loaded`, `locked`, `overlay offset table`, `delete`, or `move`. An entry can have several flags associated with it as well.

Command Line

display overlays

Viewing Evictions in the Call Stack Window

You can view evicted overlays in the **Call Stack** window or command-line display as you step or run your application.

GUI

1. Select **Display >Call Stack**.
2. Click the **pc/fp** button, which displays the program-counter and frame-pointer register values.

If an overlay is evicted, [EVICTED] appears in place of the PC value.

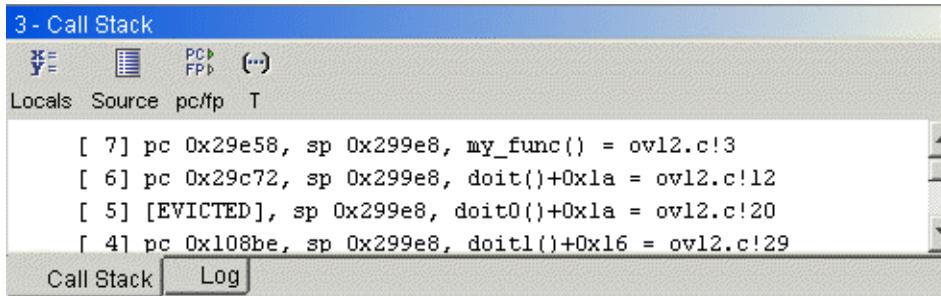


Figure 33 Evicted Function in the Call Stack Window

Command Line

Use the [stack](#) command.

Related Topic

- *Automated Overlay Manager User's Guide*

Debugging Closely Coupled Memory (a Harvard Architecture)

If you have configured your processor to have separate instruction and data spaces (known as closely coupled memory or a Harvard architecture), do the following:

1. Build your application using option **-Hccm**. Be sure to use the option for both compiling and linking so that the correct libraries are linked.
2. Create a special linker command file to ensure that all data sections are linked in the data space and only executable instructions are linked in the instruction space. See the *ELF Linker and Utilities User's Guide* for an example command file.
3. Notify the simulator of the distinct addresses of code and data memory as described in [Specifying Memory Size and Location](#) on page 127.

Simulating DSP Extensions

The debugger provides simulation for ARC DSP extensions, including XMAC and XY Memory. These extensions are supported at the instruction level in the simulator. Most extensions can be selected in the GUI or using command-line options (see [Command-Line Option Reference](#) on page 274). The debugger GUI supports XY and XMAC windows on both hardware and simulator. These windows show the state of the XY memory and XMAC registers.

When you run an application on the simulator, the debugger models the behavior of the extensions. Additional related command-line options are available to configure modeling, more or less along the lines of configuration options frequently used for building hardware.

XMAC Simulation

To enable an XMAC window in the simulator:

1. Start the debugger and click the **Debugger Options** button. If the debugger is already running, select **Tools | Debugger Options**.
2. Select **Target selection** and select the target **MetaWare debugger ARC Simulator**.
3. Select **Simulator Extensions | DSP instructions**.
4. Select your XMAC instruction options.

5. Click **OK**. If the debugger is already running, you must exit and restart it for the settings to take effect.

You can now open an **XMAC** window using the **Display** menu.

If you are using the command line, specify option **-Xxmac** only when you are running on hardware, in order to provide the **XMAC** window. When you run on the simulator, choose the specific **-Xxmac_*** option instead:

```
-Xxmac_16
-Xxmac_d16
-Xxmac_24
```

See the **X*** options in [Command-Line Option Reference](#) on page 274 for more information.

The debugger provides a display of the mode register and accumulator depending on the XMAC configuration selected.

XY Memory Simulation

To enable an **XY Memory** window in the simulator:

1. Start the debugger and click the **Debugger Options** button. If the debugger is already running, select **Tools | Debugger Options**.
2. Select **Target selection** and select the target **MetaWare debugger ARC Simulator**.
3. Select **Simulator Extensions | DSP memory**.
4. Select the **XY memory** radio button.
5. Select values for **Size**, **Banks**, and **Bus width**.
6. Map X and Y memory to any specific addresses you require.
7. Click **OK**. If the debugger is already running, you must exit and restart it for your settings to take effect.

You can now open an **XY Memory** window using the **Display** menu.

If you are using the command line, you can specify XY Memory simulation with option **-Xxy**.

Use this option alone when running on hardware, to provide the **XY Memory** window. When running on the simulator, in addition to this option, which is required, you can specify other **-Xxy*** options to configure the simulator; see **X*** in [Command-Line Option Reference](#) on page 274. For example:

- To specify the number of banks of XY Memory you want in the simulation:

-Xxybanks=value

value ranges from 1 to 4. The default is 4.

- To specify the size of XY Memory for each bank:

-Xxysize=value

value is the size in bytes of each of the X and Y regions: 512, 1024, 2048, 4096, 8192, or 16384.

The default is 16384.

Set the simulation of memory bus width with the option:

-mem_bus_width=value

where *value* is 8, 16, 24, or 32. The default is 32.

Stopping Execution on Loads of a Value

Because the simulator initializes memory to 0xdeadbeef (ARCtangent-A4) or 0x7ffe7ffe (ARCtangent-A5 and later) before running a program, you might be able to catch loads of uninitialized values in your program by stopping execution on loads of this value.

You can use this command to direct the simulator to stop on a full-word load of the specified *value*:

```
prop stoponload=value
```

where *value* is a non-zero integer constant such as 0xdeadbeef, or 77, and so on. For example, this command directs the simulator to stop on loads of the value 0xdeadbeef from memory:

```
prop stoponload=0xdeadbeef
```

If *value* is 0, this facility is turned off.

Debugging on ARCan gel Hardware

This section describes how to debug ARC targets using ARCan gel hardware. The general steps are as follows:

1. Set up the hardware connection.
2. (Optional) Specify a blast file to reinitialize your ARCan gel.
3. Specify additional options.

GUI

1. Set up the hardware connection.

Connect the parallel port to your PC, and ensure the correct parallel port is specified in the **Parallel port address for I/O** field. Use the **detectpp** program to determine the parallel-port address (for more information, see [Communicating with Hardware on the Parallel Port](#) on page 135).

TIP Use the commands **net stop gportio** and **net start gportio** to ensure there are no parallel port conflicts.

TIP Make sure the correct dip switch is set on the ARCan gel 3 board (for best results, set switch 2 to the On position).

2. Start the debugger and select **Debugger Options**. If the debugger is already running, select **Tools | Debugger Options**.
3. Select **Target selection** and select the target **Hardware**.
4. Under **Hardware connection**, select **General hardware connection** (for information on using an Ashling Opella emulator, see [Setting up an Ashling Opella Emulator](#) on page 149).
5. (Optional) Choose **Execute programs using your DLL interface to the ARC** and specify a DLL; otherwise the debugger uses a default DLL.
6. Check **Blast the FPGA** and enter or browse to your aa3 .xbf file in the **File name** field.
7. Click **OK**. If the debugger is already running, you must exit and restart it for the changes to take effect.

Command Line

1. Set up the hardware connection and specify DLLs:
 - For parallel-port communication, use option [-hard](#) to specify hardware and the default parallel-port DLL.
 - For JTAG communication, use option [-jtag](#) to specify the default `apijtag.dll`.
 - To specify your own DLL, use option [-DLL](#) (with the DLL name as an argument).
2. To reinitialize your ARCan gel, use option [-blast](#), specifying a blast file if desired.
3. Specify additional options and toggles if desired, for example:

[-dcache](#)
[-icache](#)
[-off=cycle_step](#)
[-on=reset_upon_restart](#)

Communicating with Hardware on the Parallel Port

Note This section applies only if you are running the MetaWare debugger on a Windows host.

The MetaWare debugger connects to the ARCan gel development boards via a bidirectional parallel port.

The default parallel-port address of 0x378 is almost always correct. If it is not, you must determine the port's input/output range, and use the first address in the range as the value of `ARC_hardware`'s [-port](#) property.

Determining Parallel-Port Addresses

To determine the parallel-port address for versions of the debugger hosted on Windows:

1. Right-click **My Computer** on the Windows desktop.
2. Select **Manage** from the contextual menu. The **Computer Management** window opens.
3. Click **[System Information]**.
4. Click **[Hardware Resources]**.
5. Click **[I/O]**.
6. Look for a line like the one below in the **Computer Management** window:

0x03BC-0x03BF Printer Port (LPT1) OK

The first number (0x3BC) is the port address for the parallel port.

Using the Generalized Parallel-Port Driver

Note This section applies only if you are running the MetaWare debugger on a Windows host.

If you are running the MetaWare debugger on a Linux host, see [Using the Generalized Parallel Port Driver on Linux Hosts](#) on page 137.

If you are running the MetaWare debugger on a UNIX host, see [Debugging Remotely with SCIT](#) on page 118.

To support I/O on different versions of Windows, the debugger uses a generalized I/O port driver, `gportio.dll`. This DLL is installed in `install_dir/bin`, and is accessible if you have `install_dir/bin` in your PATH.

You can use this generalized port I/O driver for other purposes; however, we do not support its use other than with the MetaWare debugger. See file `install_dir/mdb/inc/gportio.h` for more information.

I/O operations on Windows are privileged; a driver is required to access a device. When you install generalized I/O port driver `gportio.dll`, the driver is copied to the `system32/drivers` directory, and an entry is added to the Windows registry.

Installing the Parallel-Port Drivers

To install the driver, enter the following command at the system prompt.

NOTE Enter the command from the directory above `bin`.

`bin\gpioinst`

This command copies the driver to the appropriate directory and sets its mode to *automatic*. *Automatic mode* means the driver starts up when you first need it, so it is available any time you want to begin a debugging session.

You must reboot your system to use this driver the first time.

Changing the Driver's Mode

After you have added `install_dir/bin` to your path, you can enter this command at any time to change the `gportio` driver's mode:

`gpio mode_option`

[Table 8 Mode Values for Generalized Parallel Port Driver `gportio`](#) lists the possible values of `mode_option`, which are defined by Windows driver setting protocols. We recommend either the *automatic* or *manual* mode.

Table 8 Mode Values for Generalized Parallel Port Driver `gportio`

Mode	Meaning
<code>automatic</code>	The Service Control Manager loads the driver after everything in the system is up and running. You can also start and stop the driver manually.
<code>boot</code>	The Windows loader starts the driver. This mode is not recommended for general use because most Windows services are not yet available at load time.
<code>disabled</code>	You can never start the driver.
<code>manual</code>	You can start and stop the driver manually through Control Panel Devices , or through direct calls to service control manager API. The driver is not started by default.
<code>removed</code>	Removes the driver from the registry. The effect is the same as if the driver had never been installed. If you reinstall the driver, you must reboot for the install to take effect.
<code>system</code>	The driver is started after Windows is loaded but while the system is still initializing. This mode is not recommended — it is probably too early to start the driver.

Checking for the Presence of the Driver

To check for the presence of the driver:

1. Open **My Computer | Control Panel | Devices**.
2. Scroll down until you see **GPortIO**, or enter G for a shortcut.
3. Highlight the GPortIO line.

You can then start or stop the driver, if its mode is *automatic* or *manual*.

NOTE If you run **gpio** to change the driver's mode, you do not need to reboot — close the **Devices** window, then re-open it to view a refreshed driver setting.

Using the Generalized Parallel Port Driver on Linux Hosts

NOTE This section applies only if you are running the MetaWare debugger on a Linux host.

If you are running the MetaWare debugger on a Windows host, see [Using the Generalized Parallel-Port Driver](#) on page 135.

If you are running the MetaWare debugger on a UNIX host, see [Debugging Remotely with SCIT](#) on page 118.

The debugger supports fast debugger download and access via the parallel port to the ARCanegel 3. You must install a device driver for this feature to be available.

You can install using one of two different methods.

Method 1: You Assign the Major Device Number

NOTE You must be root to perform the installation.

1. Make `/dev/gpio` using your chosen major number and make it writeable:

```
mknod /dev/gpio c 250 0 # You choose 250, for example.  
chmod 666 /dev/gpio # Make it writeable.
```

2. Install the module and tell it what number it has.

```
insmod ./gpio.o gpio_major=250 # insmod and rmmod are  
# in /sbin.
```

Method 2. Linux Assignes the Major Device Number

NOTE You must log in as root to perform the installation.

1. Install the module:

```
insmod ./gpio.o # insmod and rmmod are in /sbin.
```

2. Check what major number was assigned using one of the following methods.

— Check the messages file:

```
vi /var/log/messages  
May 6 14:48:30 redhat1 kernel:  
gpio: major number 250 has been allocated to this driver.
```

— Check the devices file:

```
vi /proc/devices:  
Character devices:  
1 mem  
...  
250 gpio  
...
```

3. Make /dev/gpio using the major number assigned and make it writeable.

```
mknod /dev/gpio c 250 0      # The major number must agree  
                                # with the one assigned.  
  
chmod 666 /dev/gpio          # Make it writeable.
```

To Verify Installation

```
cat /proc/modules  
gpio           2736    0
```

To Remove the Driver

```
rmmmod gpio      # insmod and rmmmod are in /sbin.  
rm /dev/gpio
```

To Check Installation or Removal

```
vi /var/log/messages      # You must log in as root to do this.
```

To Check the Driver Statistics

```
cat /proc/gpiomem
```

Blasting an ARCan gel Hardware System

The ARCan gel development boards require a special “blasting” operation after power-up to download the ARCan gel CPU image to the board. The debugger must blast the board before it can download and run an executable each time the the board is powered up.

To blast ARCan gel hardware:

1. Start the debugger and click the **Debugger Options** button. If the debugger is already running, select **Tools | Debugger Options**.
2. Select **Target selection** and select the target **Hardware**.
3. Check **Blast the FPGA**.
4. Optionally enter the filename and path of a blast file into the **File name** field.
5. Click **OK**. If the debugger is already running, you must exit and restart it for your settings to take effect.

From the command line, you can blast your hardware system by specifying option [-blast](#).

After the debugger has blasted the development board, programs can be downloaded and executed by the debugger as long as the board is powered up.

To make this blast operation easier, the distribution includes a blast script, which is invoked with no arguments — it simply blasts the default ARCan gel CPU image to the board.

NOTE For information about changing the default ARCan gel CPU image, see option [-blast](#) in [Command-Line Option Reference](#) on page 274.

Using SmaRT Real-Time Trace (ARC 600 Only)

This section describes how to use the ARC 600 small real-time trace (SmaRT) with the debugger. For information about the SmaRT hardware component, see the hardware documentation that accompanies the component.

The debugger examines the SmaRT build configuration register (SMART_BUILD). If the version field is valid, the debugger enables trace-fetch functionality and adds **SmaRT Trace** to the **Display** menu.

NOTE Data are retrieved and displayed only when the processor is stopped.

Displaying SmaRT Data Using the GUI

To display SmaRT data, select **SmaRT Trace** from the **Display** menu. The **SmaRT Trace** menu item appears only after the debugger has detected the presence of SmaRT by checking the SmaRT build configuration register.

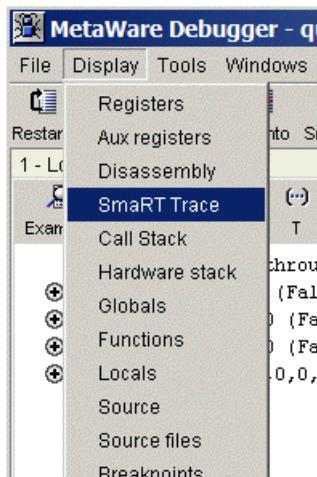


Figure 34 Displaying SmaRT Data

NOTE When you first open the **SmaRT Trace** window, the content is indeterminate.

Enabling Trace Capture

To enable trace capture, right-click in the SmaRT Trace window and select **Enable SmaRT Capture** from the pop-up menu, as shown in [Figure 35 Enabling SmaRT Trace Capture](#).

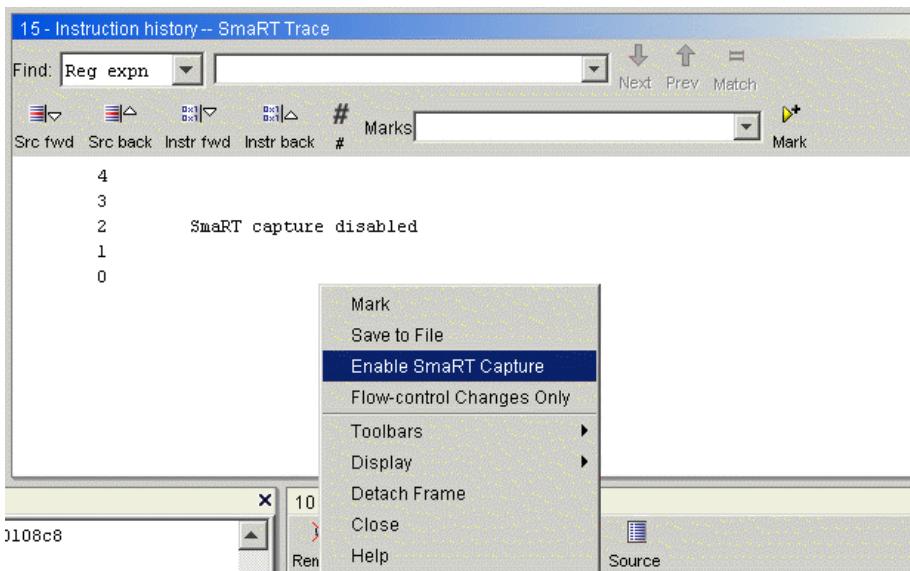


Figure 35 Enabling SmaRT Trace Capture

By default, trace data are displayed in order of execution, similar to the display in the **Instruction history -- Simulator trace** window. This mode displays a list of executed instructions and their addresses in the order they were executed, as shown in [Figure 36 SmaRT Trace Window Default Display](#). By default, the window shows all instructions executed while trace capture is enabled:

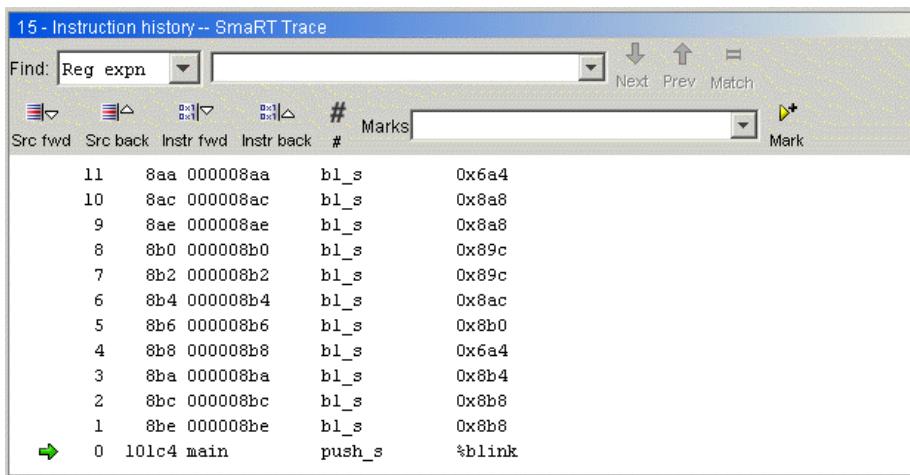


Figure 36 SmaRT Trace Window Default Display

Displaying Data for Flow Control Changes Only

To display only those instructions involved in a flow-control change, right-click in the **Smart Trace** window and select **Flow-control Changes Only** from the pop-up menu, as shown in [Figure 37 Displaying Flow-Control Changes Only](#).

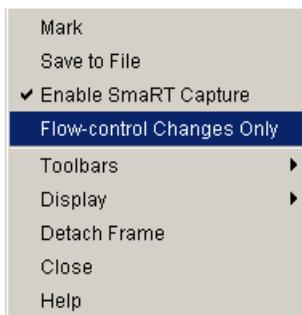


Figure 37 Displaying Flow-Control Changes Only

Displaying only flow-control changes filters out sequences of instructions between changes in flow control unless there is just one instruction. See [Figure 38 Flow-Control Changes Only](#).

	11	10282	printf+0x26	mov_s	%r13,%r0
	10	10284	printf+0x28	bl.d	_mwunlock_file
	9	11880	_mwunlock_file	mov_s	%r1,%r0
	8	...			
	7	11894	_mwunlock_file+0x14	b.d	_mwmutex_unlock
	6	151d8	_mwmutex_unlock	j_s	[%blink]
	5	1028c	printf+0x30	mov_s	%r0,%r13
	4	...			
	3	10296	printf+0x3a	j_s	[%blink]
	2	101e0	main+0x1c	mov_s	%r13,1
	1	...			
	0	101ec	main+0x28	add_s	%r0,%r13,%r12

Figure 38 Flow-Control Changes Only

Accessing Source

You can double-click a line (an instruction) in the **Smart Trace** window to update the **Source** window (if available) and highlight the associated source-code statement. A **Source** window opens if no **Source** window is currently open.

If no source line information is available, the **Disassembly** window is updated or opened instead.

Innermost loops

SmaRT Trace does not show every iteration of innermost loops; instead the **SmaRT Trace** window labels instructions within a loop as [repeated] to indicate that the preceding flow-control changes were repeated at least twice, as shown in [Figure 39 SmaRT Trace Handling of Innermost Loops](#).

```
19 117d4 memmove+0x18 sub_s      %r2,%r2,1
18 117d6 memmove+0x1a cmp        %r2,-1
17          [repeated]
16 117da memmove+0x1e bnz_s     0x1_17cc = memmove+0x10
15 117cc memmove+0x10 ldb.ab    %r3,[%r1,1]
```

Figure 39 SmaRT Trace Handling of Innermost Loops

Displaying SmaRT Data Using the Command Line

On the command line, use the **display smart**. See the listing under [display](#) for the exact syntax. Entered without an argument, **display smart** displays SmaRT Trace data after the debugger has detected the presence of SmaRT Trace.

Examples

These examples use the `queens` demo included on your distribution.

The first example uses **isi** to step a few times so the debugger can detect SmaRT from the `SMART_BUILD` register, then asks to display 10 lines:

```
mdb> isi
    sp 00027a2c
    1018a main+0x02          b1.d  __ac_push_13_to_15
mdb> isi
    r1 00000001  blink 00010190
    14c28 __ac_push_13_to_15      push  %r15
mdb> isi
    sp 00027a28
    14c2c __ac_push_13_to_14      push  %r14
mdb> isi
    sp 00027a24
    14c30 __ac_push_13_to_13      j_s.d [%blink] ; main+0x08
mdb> display smart lines=10
    9 102f0 _mwcall_main+0x74  st_s      %r1,[%r0,8]
    8 102f2 _mwcall_main+0x76  mov_s    %r0,%r15
    7 102f4 _mwcall_main+0x78  b1.d    main
    6 102f8 _mwcall_main+0x7c  mov_s    %r1,%r16
    5 10188 main              push_s   %blink
    4 1018a main+0x02          b1.d  __ac_push_13_to_15
    3 1018e main+0x06          mov_s    %r1,1
    2 14c28 __ac_push_13_to_15  push    %r15
    1 14c2c __ac_push_13_to_14  push    %r14
    0 14c30 __ac_push_13_to_13  j_s.d [%blink] ;
```

After a **run**, this example displays trace history with instructions between flow-control changes omitted:

```
mdb> run
Execution stopped.
No source information at 0x100f0.
    100f0 _exit_halt+0x04  nop
mdb> display smart compressed
    19 14c80 __ac_pop_13_to_17    mov_s  %r12,4
    18 14c82 __ac_pop_13_to_17v   mov_s  %r13,20
```

```

17 14c84 __ac_pop_13_to_17v+0x02 b_s    __ac_pop_17
16 14cca __ac_pop_17          ld     %r17,[%sp,16]
15 .
14 14cde __ac_pop_13+0x08      j_s    [%blink]
13 13854 _hl_message+0x10     pop_s  %blink
12 13856 _hl_message+0x12     j_s.d [%blink]
11 13858 _hl_message+0x14     add_s  %sp,%sp,32
10 13432 _exit+0x1a          cmp_s  %r0,0
9   13434 _exit+0x1c          blnz   _hl_delete
8   135f4 _hl_delete          ld_s   %r0,[%pcl,12]
7 .
6   135fe _hl_delete+0x0a     j_s    [%blink]
5   13438 _exit+0x20          bl.d   _mwmutex_unlock
4   1343c _exit+0x24          ld_s   %r0,[%r13]
3   14f78 _mwmutex_unlock    j_s    [%blink]
2   1343e _exit+0x26          bl    _exit_halt
1   100ec _exit_halt         flag   1
0   100f0 _exit_halt+0x04    nop

```

This example disables, then reenables SmaRT and displays 10 lines.

```

mdb> display smart enabled=0
4
3
2      SmaRT capture disabled
1
0
mdb> display smart enabled=1 lines=10
9   135f4 _hl_delete          ld_s   %r0,[%pcl,12]
8   135f6 _hl_delete+0x02     mov    %r1,-1
7   135fa _hl_delete+0x06     st.di %r1,[%r0,8]
6   135fe _hl_delete+0x0a     j_s    [%blink]
5   13438 _exit+0x20          bl.d   _mwmutex_unlock
4   1343c _exit+0x24          ld_s   %r0,[%r13]
3   14f78 _mwmutex_unlock    j_s    [%blink]
2   1343e _exit+0x26          bl    _exit_halt
1   100ec _exit_halt         flag   1
0   100f0 _exit_halt+0x04    nop

```

Debugging ARC Media Extensions and SIMD Instructions (ARC 700 Only)

You can debug ARC Media Extensions and the SIMD instructions on hardware or the simulator. The debugger automatically detects the presence of ARC Media extensions when you connect to hardware containing them. When you debug using the simulator, the debugger must locate and read the simulator-extension DLL containing the ARC Media Extensions.

After the debugger is aware of the presence of ARC Media Extensions, you can display information on the command line or open the related windows using the **Display** menu.

Note The debugger retrieves and displays SIMD data only when the processor is stopped.

GUI

If the IP Library correctly copied the SIMD simulator-extension DLL to the debugger's /bin directory, you can enable ARC Media Extensions by selecting **Enable SIMD (media) extensions** under **ARC700** in the **Debugger options** dialog.

You can also manually specify the location of the simulator-extension DLL in the **Debugger options** dialog (**Simulator Extensions | User Extensions**). Be sure you also select **ARC 700** in **Debugger options | Target Selection**.

See your ARC Media Extensions documentation for the location of the DLL.

After the ARC Media Extensions components are enabled, you can select **Display > SIMD processor** and display the SIMD-specific windows.

You can see queue and data memory in the **Memory** window (**Display > Memory**).

Command Line

Load your SIMD simulator-extension DLL as follows:

```
mdb myApp.out -simext=/path/simd.dll
```

See your ARC Media Extensions documentation for the *path* to `simd.dll`.

Using `-simext*` is the same as using `-simext*` and `-prop`.

You can display the ARC Media Extensions internal registers with the following **display** commands:

- **display simd** displays the SIMD configuration registers.
- **display vrN** displays the vector registers in an 8-, 16, or 32-bit view, where *N* is **8**, **16**, or **32**. If you have more than one SIMD unit, use **display vrN_UN**, where *UN* is the unit number.

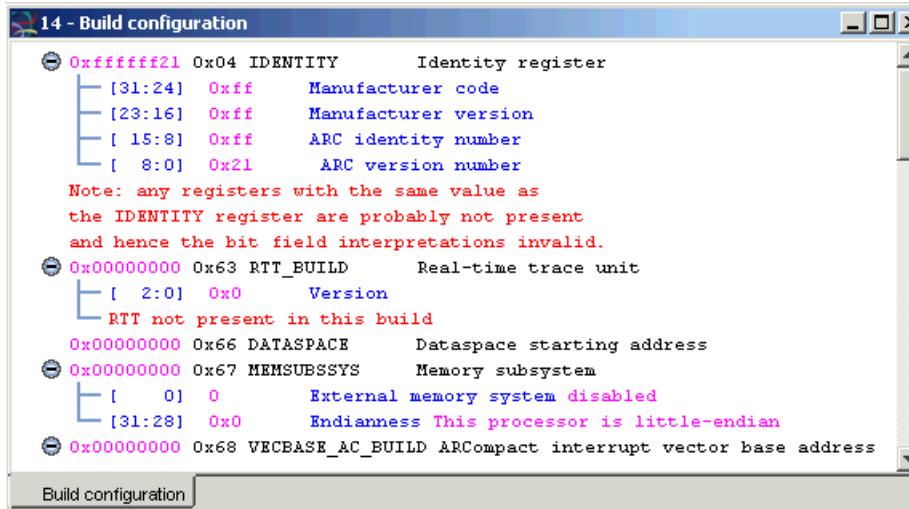
Viewing Hardware Data and Instruction Caches

To view cached data and instructions while you step through a program, specify command-line option **-Xcache_display**; see [Command-Line Option Reference](#) on page 274 for information about this option.

When you are debugging hardware, the debugger determines data and instruction cache size, line length, associativity, and replacement algorithm. See [Simulating Instruction and Data Caches](#) on page 91 for information about displaying cache data using the debugger GUI.

Viewing Build-Configuration Registers

When debugging on ARC hardware, select **Display | Hardware | Build configuration** to open a **Build configuration** window. The window displays the contents of the build-configuration registers and explains the meanings of the bit fields.



Debugging an ARC Processor in an SoC

The MetaWare debugger can use SCIT to debug an ARC processor embedded in a larger environment, for example as a component of a system-on-chip (SoC) simulation. The environment incorporates the SCIT interface handler to give the MetaWare debugger access to the ARC processor.

For more information on SCIT, see the following:

- [Debugging Remotely with SCIT](#) on page 118
- [SCIT Interface Handler Usage](#) on page 124

You must start the environment first, as the interface handler is a virtual server that must be running before you can connect the MetaWare debugger to it. For example:

```
soc ...arguments...
```

At some point the environment activates the handler (the exact point depends on the particular environment). After that point you can start the MetaWare debugger with the ARC SCIT client `arccli`:

```
mdb myprog.out -DLL=arccli,shmem=123
```

Shared memory (`shmem=123` in our example) is the preferred conduit; with it, the MetaWare debugger can arbitrarily connect to and disconnect from the handler. Whether you use shared memory or TCP/IP as the communications channel is determined by the provider of the environment. Whether the environment supports configuring the channel in its command-line arguments is determined by the provider. The SCIT environment variables `SCIT_SHMEM` and `SCIT_PORT` can be used instead of arguments to the environment. See [SCIT Interface Handler Usage](#) on page 124 for a description of the SCIT environment variables and arguments.

The environment may make use of the MetaWare debugger engine and execution vehicle access to instantiate an ARC processor within the environment, if a MetaWare debugger engine is present in the target. The target may then make use of the debugger engine's facilities to provide address-counter

statistics (such as instruction counts) when you connect the MetaWare debugger to the environment. Address counters are sufficiently complex that to access them from the MetaWare debugger requires a special DLL “helper” that can drag the address-counter data from the target to the connected MetaWare debugger. To load the DLL helper, load the “acsi” (address counter semantic inspection) DLL along with the ARC SCIT client arccli:

```
mdb myprog.out -DLL=arccli,shmem=123 -semint=install_dir/bin/acsi
```

The file acsi.dll (.so on UNIX) is in the *install_dir/bin* directory. On Windows, you can omit *install_dir/bin/* if the **bin** directory is in your path.

You can avoid traffic to the target by having the MetaWare debugger read the read-only (code) portions of the program from the executable being debugged, rather than from the target. To do so, turn on toggle **read_ro_from_exe**.

Using CMPD

With CMPD, use *cputnum* to specify the processor for which you wish to extract the address counters:

```
mdb myprog.out -DLL=arccli,shmem=123 -semint=dir/acsi,cputnum=N
```

Viewing the ARC 600 MPU Registers

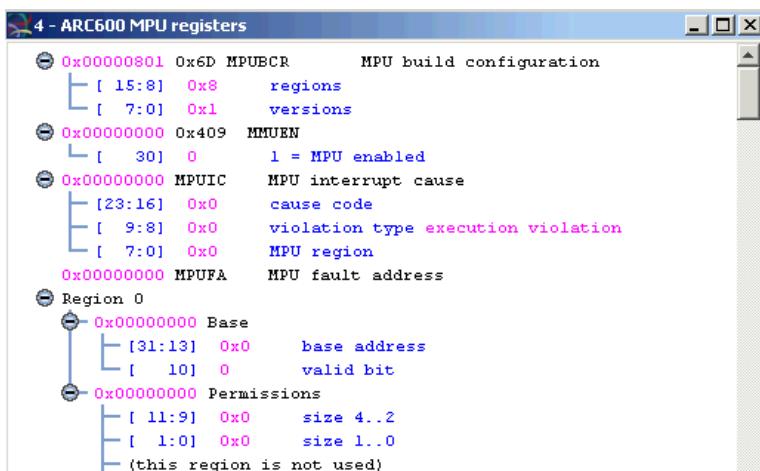
You can simulate the ARC 600 Memory Protection Unit (MPU) in **Debugger Options | Target selection**, or by specifying [mpu](#) and [arc600](#) on the command line. The **Enable MPU** selection box appears only when you select **MetaWare ARC simulator** as the target.

Command Line

Enter [display mpu](#)

GUI

To open the **ARC600 MPU registers** window after you begin debugging, select **Display | Hardware | ARC600 MPU registers**.



Displaying MMU and Exception Registers (ARC 700)

You can simulate the ARC 700 Memory Management Unit (MMU) in the **Debugger options | Target selection** dialog or by specifying [-mmu](#) and [-arc700](#) on the command line. You can view exception registers without specifying the presence of an MMU.

Command Line

Enter [display mmu_contents](#) or [display exmmu](#).

GUI

Select **Display | Hardware** to open one of the following.

- **MMU contents**
- **Exception/MMU registers**

If your processor does not contain an MMU, the **Exception/MMU registers** window displays the exception registers and reports **There is no MMU present**.

The **MMU contents** window includes a **Content:** button for toggling between three types of content:

- all — all MMU contents
- NZ — only sets with one or more non-zero entries
- valid — only sets with one or more valid entries

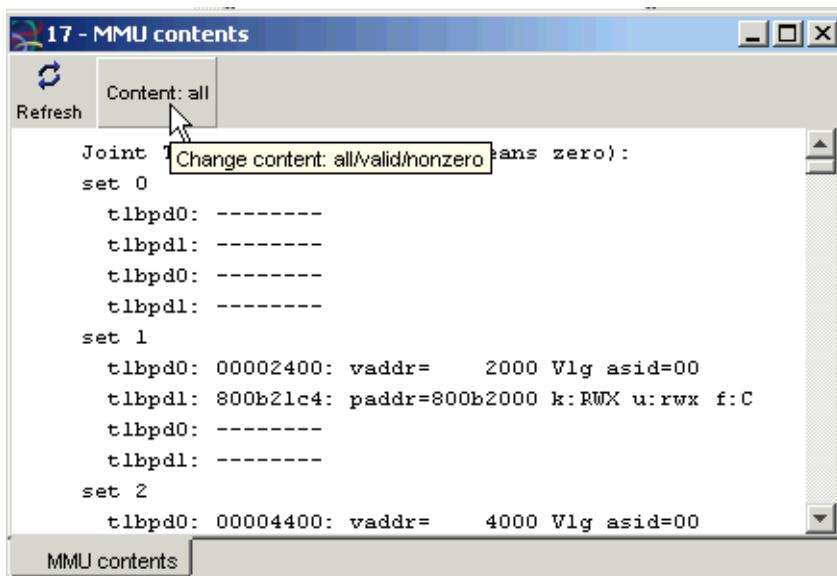
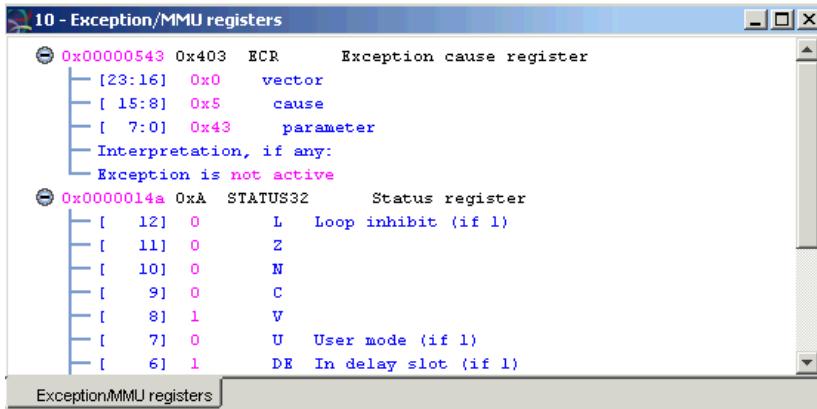


Figure 40 MMU Contents

The **Exception/MMU registers** window shows a tree view of registers related to exceptions and MMU, including the following:

- Exception cause register (ECR)
- Status register (STATUS32)

- MMU build configuration register (MMUBUILD)
- Data uncached configuration register (DUNCBUILD)



If your processor does not contain an MMU, the **Exception/MMU registers** window displays the exception registers and reports There is no MMU present.

Debugging with ARC_DLL or BRC_DLL

The systems ARC_DLL and BRC_DLL are DLL files provided for third-party OEM implementation of functions the debugger needs to debug programs on an ARC processor. This system allows access to an ARC processor through mechanisms the debugger knows nothing about. The DLL files can communicate with a simulator or with hardware.

For example, you can implement a serial-port access to a JTAG-enabled ARC processor with ARC_DLL or BRC_DLL. The only requirement is that you implement the Target Access Interface described in file `install_dir/mdb/inc/arcint.h`.

For more information on writing your own DLL to implement the Target Access Interface, see the *MetaWare Debugger Extensions Guide*.

Specifying a DLL System

To specify the DLL that the debugger should open:

1. Start the debugger and click the **Debugger Options** button. If the debugger is already running, select **Tools | Debugger Options**.
2. Select **Target selection** and select the target **Hardware**.
3. Check **Execute programs using your DLL interface to ARC**.
4. Enter the path to your DLL interface in the **File name** field, or browse to it using the icon.
 - For the default ARC parallel-port DLL, use file `apipar.dll` in the `/bin` directory of your installation.
 - For the default ARC JTAG DLL, use file `apijtag.dll` in the `/bin` directory of your installation.
5. Click **OK**. If the debugger is already running, you must exit and restart it for the changes to take effect.

From the command line, option **-hard** specifies the DLL for parallel communication and **-jtag** specifies the DLL for JTAG connection. You can also use option **-DLL=xxx** to open DLL `xxx`.

Specifying Properties of the DLL System

Properties are an important way to configure your ARC_DLL or BRC_DLL system. To specify system properties, enter a **-c** command-line option like this when you invoke the debugger:

```
-c="sysprop ARC_DLL prop1=val1 prop2=val2 ..."
```

To make these properties permanent, place the **-c** option in the **ARGS=** line in **bin/mdb.cnf**.

Setting up an Ashling Opella Emulator

Ashling supports a connection to ARC JTAG via their Opella-ARC JTAG Emulator box. Opella-ARC is supplied with an ARC Probe Cable (part number TPA-ARC-JTAG-15) with a male DB15 connector that plugs directly into the DB15 female in an ARCanel3.

For online information regarding Ashling's Opella-ARC software, go to www.ashling.com. This software provides the USB driver for Opella-ARC as well as test software that can verify the Opella-ARC connection, download an FPGA build, and test that ARC is responding. You should run the setup from this software before using the Opella-ARC emulator.

Specify the Opella-ARC connection using either of the following methods:

GUI

1. Start the debugger and click the **Debugger Options** button. If the debugger is already running, select **Tools | Debugger Options**.
2. Select **Target selection** and select the target **Hardware**.
3. Under **Hardware connection**, select **Ashling Opella JTAG for ARC**.
4. Verify the DLL and the JTAG frequency.

NOTE Do not choose a frequency greater than the target can handle.

5. Click **OK**. If the debugger is already running, you must exit and restart it for your settings to take effect.

Command Line

1. Connect to the Opella by simply specifying the following:

-DLL=location_of_Opella_DLL

For example: **DLL=D:\ashling\Opella\ARC\OPELLARC**

2. You can then also specify the JTAG frequency using

-prop=jtag_frequency=N

where **N** is a value from [Table 9 Ashling JTAG Frequency Codes](#).

CAUTION The values for **N** are set by Ashling and are subject to change. Consult your Ashling documentation.

Table 9 Ashling JTAG Frequency Codes

Value for N	Frequency
0	8 MHz
1	6 MHz

Table 9 Ashling JTAG Frequency Codes

Value for <i>N</i>	Frequency
2	4 MHz
3	2 MHz
4	1.5 MHz
5	750 kHz
6	375 kHz
7	200 kHz
8	100 kHz

If you do not specify a value for *N*, the default is 2 MHz.

TIP The faster the JTAG speed, the faster the ARC processor must run to be able to properly sample the JTAG clock. If you get erratic behavior, slow down the JTAG clock to see if it improves the situation.

Using ARCanel3

For ARCanel3, switch 2 must be Off (serial) and switch 5 must be On (alternate JTAG connection). For any other configuration, such as your proprietary FPGA board, read the pinout documentation in Ashling's document *AshlingARCDevTools.pdf* for how to wire its connections to your JTAG.

Using the Cycle-Estimating Simulator (CES)

The ARC Cycle-Estimating Simulator (CES) is a component of the ARC ISS that estimates machine cycles in executing programs. The estimation slows down the ISS by 10%, but the speed is better than an order of magnitude faster than the ARC Cycle-Accurate Simulator (CAS). The accuracy of the CES is generally better than 90% of RTL simulation, and is usually 97% or better when there is no cache thrashing. The CES also includes a cache thrash analysis routine to point out cache thrashing.

CES supports ARCTangent-A4, ARCTangent-A5, and ARC 600, and instruction and data caches. Cycle estimation is not available for code fetch where no icache is present. The assumption is that without an icache, you are employing single-cycle access code ram that essentially takes no time.

Basic Operation

You can use cycle estimation by turning on toggle [cycles](#), or by checking the box in **Debugger options | Simulator Extensions | Cycle Counting** to enable cycle estimation.

When CES is enabled, three additional cycle-estimating registers are available.

- `cycles` — contains the accumulation of all cycles for the program.
- `1cycles` — contains the cycles from when the program last began to ran until it stopped
- `dastallcy` — contains the stall cycles caused by data access.

When an icache is simulated, the following two registers are available:

- `icmisscy` — the number of cycles incurred in icache misses
- `icmiss` — the number of icache misses

In addition, the icache miss address counter converts from “# of times the icache missed” to “# of cycles lost due to an icache miss.”

When a dcache is simulated, the following two registers are available:

- `dcmsscyc` — the number of cycles incurred in dcache misses
- `dcmiss` — the number of dcache misses.

In addition, the dcache miss counter converts from “# of times the dcache missed” to “# of cycles lost due to an dcache miss.”

If you specify load/store RAM (with `-prop=ldst_ram_base=` and `-prop=ldst_ram_size=`), the CES takes it into account.

Fine-Tuning Timing

- Data returning from a memory load costs an additional cycle to post the result to a three-port register file. A four-port register file incurs no such penalty. The CES assumes a 3-port register file. To assume a four-port register file, turn on toggle `fast_load_return`, e.g.:

`-on=fast_load_return`
- The data cache has a pipeline depth of two or 3. The CES assumes two. A three-deep pipeline takes longer for data to return. Use

`-prop=dcache_pipeline_depth=N`
 to specify the pipeline size as two or 3 (or longer if you wish, but depths other than two or 3 are not officially supported by ARC).
- The CES assumes that data and code memory throughput is one cycle. That is, after the latency cost is paid, four bytes of memory can be retrieved per cycle. You can change the throughput using

`-prop=data_memory_throughput=N`
 and

`-prop=code_memory_throughput=N`
 where N denotes the number of cycles needed for four bytes of memory, and the default value is 1.
- You can adjust the latency up or down using

`-prop=data_memory_additional_latency=N`
 and

`-prop=code_memory_additional_latency=N`
 where N can be a positive or negative number.

Extensions

The following extensions include cycle estimation:

- **`mul64`** and **`mulu64`**
- **`crc`**
- **`xmac`** in DSP version 3.1

These extensions are estimated based on documentation but not verified against RTL models.

Adding Cycle Estimation to Your Extensions

If you implement a simulator for the ISS, you can add code to include cycle estimates for it. You can see an example of how to do this in the following files, shipped with the debugger. In the files, **`mul64`**, **`mulu64`**, **`crc`**, and the **`mac`** instructions include cycle estimates.

- `mwextlib.c`
- `mwextxmac3.c`
- `mwextcrc.cpp`

Extensions cost cycles in one of several ways:

- An extension instruction can produce a register result that might not be available until a certain number of cycles in the future (register scoreboardding).
- The extension itself may not be available for use until that future time. (By contrast, it may be pipelined, permitting multiple instances of the extension to be in progress.) For example, **mac** instructions are pipelined, whereas **crc** and **mul64** are not.
- The extension might consume all or some of its cycles at the outset, not allowing any other instructions to be issued until these cycles are charged. For example, a load from or store to an auxiliary register (which is not scoreboxed in the processor) may delay until complete.

Functions are available in the `ARC_simext_simulator_access` object passed to your `set_simulator_access()` function that allow you to inform the ISS of the cycle requirements of your extension. They are documented in file `simext.h` shipped with the debugger, but we summarize them here and provide examples.

- You tell the ISS that the extension result is not ready until the future by calling `corereg_cycle_delay()` to specify the delay in producing the result.
- If your extension is not pipelined, you check that your extension is available for use by keeping an internal variable showing when it is next available and comparing that with the current cycle count. You can read the current cycle count by reading internal register 0 with function `rw_internal_register()`.
- You consume cycles by calling `consume_cycles()`.

[Example 8 Informing the ISS of the Cycle Requirements of an Extension](#) is an extension that takes 10 cycles until its result is available. In addition, it initially consumes 2 cycles to get started, stalling the pipeline.

Example 8 Informing the ISS of the Cycle Requirements of an Extension

```
// If not available, stall until available.  
unsigned now;  
simulator_access->rw_internal_register(0,&now,0);  
if (now < i_am_available) {  
    consume_cycles(i_am_available-now);  
    now = i_am_available;  
}  
// We stall the pipeline for 2 cycles setting up.  
simulator_access->consume_cycles(2);  
// .. compute result in core register R ...  
unsigned cycles_left = 10-2;  
corereg_cycle_delay(R,cycles_left,0);  
// Specify next time when my resource is available.  
i_am_available = now + cycles_left;
```

Initialize `i_am_available` to 0 in the `prepare_for_new_program()` and `reset()` functions of your extension. This is also a good place to ask whether the ISS is counting cycles; you can avoid your cycle computation if no cycle counting happens. You can determine this because of the fact that if you read the cycle register, the read fails if the ISS is not counting cycles.

[Example 9 Code to Execute on prepare_for_new_program or reset](#) is code you might execute on prepare_for_new_program or reset.

Example 9 Code to Execute on prepare_for_new_program or reset

```
unsigned dummy;
counting_cycles =
    simulator_access->version() >=
ARC_SIMEXT_SIMULATOR_ACCESS_CYCLES_VERSION &&
    simulator_access->rw_internal_register(0,&dummy,0) & 1;
i_am_available = 0;
```

Execute all of your cycle code only if `counting_cycles`.

You can get a contemporaneous trace of all cycle stalls by turning on toggle [trace_cycles](#).

Cache Thrashing

Cache thrashing significantly reduces the accuracy of the estimation, partly due to extra cycles incurred when the cache runs ahead of the instruction decoder in an ARCTangent-A5. These extra cycles are not counted by the CES; you should avoid counting on the numbers when the caches are thrashing.

Cache Analysis

You can obtain a performance analysis of your ARC processor's icache or dcache by clicking **Analyze** in the **Instruction cache** or **Data cache** window.

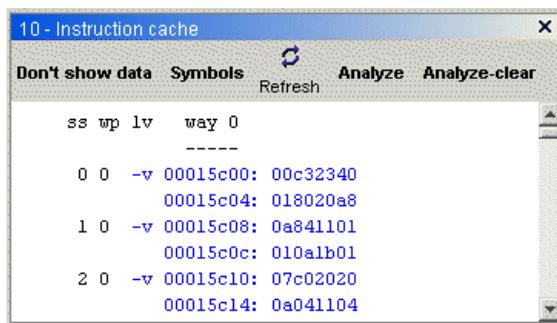


Figure 41 Click Analyze to Analyze Cache Performance

Working with the Cache Analysis Window

- Click **Don't show data** to hide the data column in the display.
- Click **Symbols** to add a column showing the symbols for each address.
- Click **Refresh** to refresh the contents of the window.
- Click **Analyze** to open an **Instruction cache analysis** window (or **Data cache analysis** window)
- Click **Analyze-clear** to clear the cache statistics regenerate the contents of the **Instruction cache analysis** window (or **Data cache analysis** window).

Command Line

```
prop icache_stats=1
display icache analyze

prop dcache_stats=1
display dcache analyze
```

Cache Performance

Performance consists of evaluating estimated memory access time for memory given the presence of the cache, and thrash analysis to see if competing lines in memory are thrashing each other.

Note The analysis includes identification of lines that thrashed during program execution and as such may impact program performance.

Memory in your program is divided up into lines; each line has a size determined by the cache. For example, a line might be 32 bytes (eight 4-byte words). Each line goes into the cache in a particular “set”, based upon the address of the line. The cache is composed of a fixed number of sets. Each set contains one or more lines. In a direct-mapped cache, each set contains just one line. In an N -way cache, each set contains N lines. A line in your program mapped to set S may be placed in any of the N lines in an N -way cache, depending on the cache algorithms.

The cost of accessing memory in a line in memory is the cost of loading it into the cache if the line is not already present in the cache (approximately one cycle per word in the line) and the cost of accessing the cache if the line is already present (often just one cycle).

Explanation of the Cache-Analysis Listing

For purposes of computing cache efficiency, we assume one cycle per word in the line, and present for each line in memory the access cost of memory references in that line. The access cost is $access_cycles/(miss+hits)$, where $access_cycles = (miss*words_in_line + hits)$. You want the access cost to be as close to one as possible, which means that most accesses hit. An access cost that exceeds 1.2 is flagged with $<- !$.

Each line is presented in the form

$(addr\ co\ C\ mi\ M\ cy\ Y)$

where $addr$ is the line address, C is the access cost, M is the total number of misses, and Y the total number of access cycles.

First, each set of the cache is shown, and all lines that at one time or the other occupied that set. In this way you can see which lines compete with others for residence in the cache.

Example 10 List of Historical Cache-Analysis Set Contents

97: Set 041: (10348 co 1.50 mi 1 cy 3) (13b48 co 1.50 mi 1 cy 3) (15148 co 1.50 mi 1 cy 3)

Next, a thrash analysis is performed, identifying for each set lines that appear to have *thrashed* (were loaded into the cache, accessed infrequently, and dumped from the cache by a contending line or lines from memory). The analysis attempts to pair up lines that thrashed each other, signified by similar miss counts. The solution to thrashing lines might be adding ways to your cache or reorganizing your program in memory via linker directives to locate contending lines in different sets.

Finally, all the lines in your program are printed, with the total number of misses, hits, access cycles, access cost, and the program label corresponding to the line. The lines are printed first in address order, then sorted according to access cost.

TIP Click **Analyze-clear** in the **Instruction cache** (or **Data cache**) window to clear the cache statistics and regenerate the **Instruction cache analysis** or **Data cache analysis** window contents.

Using the Cycle Accurate Simulator (CAS)

The Cycle Accurate Simulator (CAS) is a processor simulation tool for assisting in the accurate analysis and optimization of ARC 600 and later processor configuration options and extensions. For more information, see the *CAS User's Guide* for your ARC processor.

A similar simulator exists for ARCTangent-A5, and can be specified using option [-DLL](#). See your ARCTangent-A5 processor documentation for details.

Using ARC xCAM (ARC 700 and ARC FPX)

ARC xCAM is a cycle-accurate model for ARCompact-based processors. The MetaWare debugger treats the xCAM model as a target-access DLL. You can connect the MetaWare debugger to ARC xCAM using RASCAL and option [-DLL](#) as follows:

```
mdb -DLL=rascalint,rascal_env=rascal.env app.out
```

Where

- *rascal.env* is the path and filename of your RASCAL environment file
- *app.out* is the path and filename of the executable to debug

For more information, see your ARC xCAM and RASCAL documentation.

Using the ARC xISS Fast ISS (ARC 600 and ARC 700)

The ARC xISS tool is a fast instruction-set simulator for ARC 600 and ARC 700 processors. It uses its own set of options and configuration file documented in the ARC xISS documentation. The MetaWare debugger treats the ARC xISS tool as a target-access DLL. You can connect the MetaWare debugger to ARC xISS using option [-DLL](#) as follows.

```
mdb -on=prefer_soft_bp -DLL=%XISSL_HOME%\libxiss.dll app.out (Windows)
```

```
mdb -on=prefer_soft_bp -DLL=$XISSL_HOME/libxiss.so app.out (Linux)
```

Where

- [prefer_soft_bp](#) is the toggle to disable hardware breakpoints (not supported by ARC xISS)
- `%XISSL_HOME%` is the ARC xISS environment variable set by the installer for Windows
- `$XISSL_HOME` is the ARC xISS environment variable set by the installer for Linux
- *app.out* is the path and filename of the executable to debug

For more information, see your ARC xISS documentation.

Flash Programming

Vendors of hardware debug adapters generally provide support for flashing their target from various JTAG hardware connections; the MetaWare debugger is not involved beyond sending commands as documented in this chapter.

The MetaWare debugger's hostlink facility (host OS access by MetaWare C Run-Time Library code) allows you to program using an uploaded ELF image on a target that does a file open and binary read while programming. You might have to build flash programming into the boot (or BIOS) code and so must get the driver source from the flash maker. Then change the code to do file I/O with the MetaWare debugger using `fopen()`, `fread()`, and `fclose()`. For more information on the hostlink facility, see [Hostlink Services](#) on page 50 and the *MetaWare C/C++ Programmer's Guide*.

Chapter 5 — CMPD Extensions

In This Chapter

- [Overview](#)
- [About Multiprocess Debugging](#)
- [Setting up a CMPD Session](#)
- [Running a CMPD Session](#)
- [Viewing and Controlling Processes](#)
- [Using Breakpoints and Watchpoints](#)
- [Using Barriers to Synchronize Processes](#)
- [Using Commands with CMPD](#)
- [Configuring a CMPD Session on the Command Line](#)

Overview

This chapter shows how to use the Coordinated Multiprocess Debugger (CMPD) extensions to the MetaWare debugger. You must license the CMPD extensions separately from the debugger.

About Multiprocess Debugging

Debugging multiple processes at the same time differs in several ways from debugging those same processes one at a time. This section covers features of debugging using CMPD extensions.

Debugging Multiple Processes As a Group

Using the CMPD extensions, you can do the following with multiple processes:

- Apply the same debug operation at the same time to multiple processes: stepping, running, stopping, animating, setting breakpoints, and so on.
- When one process stops, cause others to stop automatically.
- Synchronize processes being debugged.
- Use one **Source** window to view the source for different processes, one at a time, or bring up several **Source** windows to see multiple sources simultaneously.
- In the **Source** window, single-step the process being displayed, or step all processes.
- Specify a subset of processes, and issue commands to operate on all processes in that subset.
- If multiple processes are doing similar tasks, use a multi-way comparison feature to compare the results of these tasks.

Debugging Multiple Processes on Multiple Processors

Using CMPD extensions, you can debug on multiple processors, such as multiple ARC processors with different architectural extensions. Or you can debug a simulator and hardware for the processor at the same time.

NOTE The ability to debug a simulator and hardware for a processor at the same time is known as PARC (parallel ARC comparator) when used to compare performance on the ARC instruction set simulator and ARCanalog hardware. See the readme file `parc.txt` for the latest information.

With CMPD extensions, the debugger can debug processes running on different processors. For example, you can run the same program on a MIPS processor and on an ARM processor, and use CMPD features to compare their behavior.

Process Sets

Processes must each have a unique “handle.” Each CMPD process is identified by an integer in brackets. If there are four processes, typically you refer to them as [1], [2], [3], and [4]. You can refer to a collection of processes as a process set by combining numbers; for example, [1,3] for processes 1 and 3; or [2:4] for processes 2 through 4.

Single-Process Debugging

In the single-process (non CMPD session) debugger, each command applies to the single process; the command is executed by a command processor. The command processor does the following:

- accepts command input from the **Command:** field
- accepts input from buttons or other GUI elements
- reads the command line in the command-line version of the debugger

Commands generally include every action the debugger takes, whether through the **Command:** field, or by clicking buttons, such as any of the stepping buttons.

Multiprocess Debugging

In a CMPD session, a command applies to all processes or only to a subset, or process set, that you designate. Each process has its own command processor, which executes commands for that process.

It would be more precise to say that there are command processors numbered 1, 2, 3, 4 rather than processes numbered 1, 2, 3, 4. Think of 1, 2, 3, and 4 as command processor “slots” in which processes can run, rather than the processes themselves. The concept of process “slots” is important, as processes come and go. A process can die and you can start debugging another process; in CMPD you can debug the new process in the same “slot” as the process that died. If you restart a process, the debugger kills the process and reloads it afresh as a distinct process, but again in the same “slot.” Rather than increment a process number each time a new process is created, or use the operating system’s “process ID,” the slot remains the same, and so you deal only with fixed slot numbers. In most cases, you specify a number of slots before your debugging session, and that number stays constant throughout the session.

Each command processor has a number which is a positive integer. You use the number to refer to the command processor and the process running within it. The number is different from the process’s process ID, which might be an arbitrary integer, assigned by a real time operating system (if you are using one).

Because command processors are numbered and processes are not, the processes you are debugging can change over time, yet you can refer to the set of processes with the same (command-processor) numbers. This strategy preserves defined process sets; the sets remain valid because the numbers remain. If you replace a process in one command processor with a completely different process, you use the same slot number to refer to the new process.

Despite the fact that it is the command processor that has the number, the documentation uses $[n]$ to refer to process n . This number n refers to the process that is currently running in command processor slot n .

Setting up a CMPD Session

This section contains steps that demonstrate how to configure a CMPD session using demonstration programs provided in the distribution. You can use this CMPD configuration to complete the tasks in [Running a CMPD Session](#) on page 163.

Sample Demonstration Programs

The debugger uses the following programs to demonstrate multi-process debugging:

- two copies of the Eight Queens benchmark, **queens**, running as processes [1] and [2]
- **greycode** as process [3]
- Sieve of Eratosthenes, **sieve**, as process [4]
- an infinite loop, **loop**, as process [5]

The programs are located in the following directory specific for your MIPS, PowerPC, ARC, or ARM target processor.

If you licensed the MIPS debugger, go to directory:

install_dir/cmpd/benches.mips

If you licensed the PowerPC debugger, go to directory:

install_dir/cmpd/benches.ppc

If you licensed the ARCTangent-A4 debugger, go to directory:

install_dir/cmpd/benches.arc

If you licensed the debugger for ARCTangent-A5 or later, go to directory:

install_dir/cmpd/benches.ac

If you licensed the ARM debugger, go to directory:

install_dir/cmpd/benches.arm

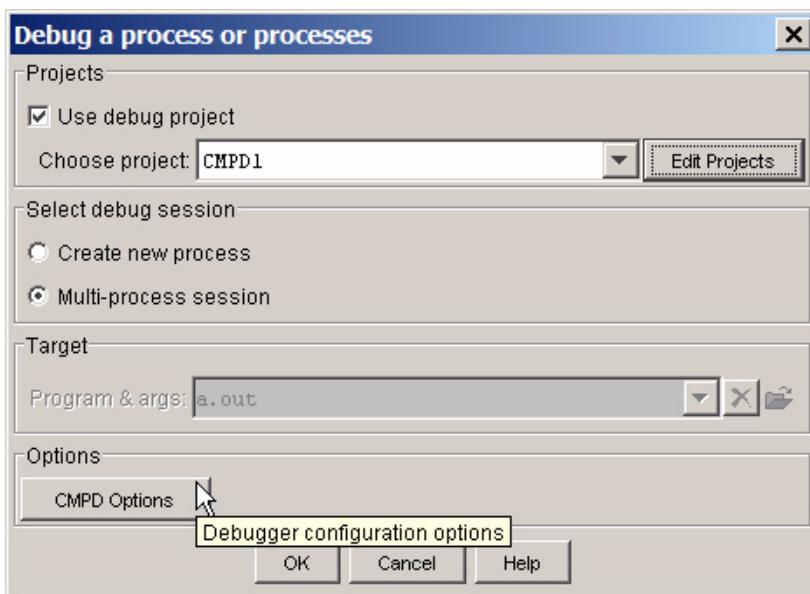
Creating the Multi-Process Debug Session

1. Compile the demo programs. Run **compile.bat** (Windows) or the **compile** (UNIX or Linux). This file is located in the sample programs directory (such as the **cmpd/multi.sieve** directory). If you have trouble compiling the programs, precompiled programs are included in the directory.
2. Launch the debugger.
3. In the **Debug a process or processes** dialog, select **Use debug project**.

Using a project is helpful if you are concurrently debugging both single process and multi-process environments. This allows you to switch the application you are debugging on without needing to set options each time you start a debug session.

4. Click **Edit projects**.
5. In the **Create/Edit Debug Projects** dialog, click **Create**.
6. Enter **CMPD1** in the **Project name:** field.
7. Click **OK** to return to the **Create/Edit Debug Projects** dialog.
8. Click **OK** to return to the **Debug a Process or processes** dialog.
9. In the **Debug a Process or processes** dialog, select **Multi-process session**.

- Click the **CMPD Options** button.

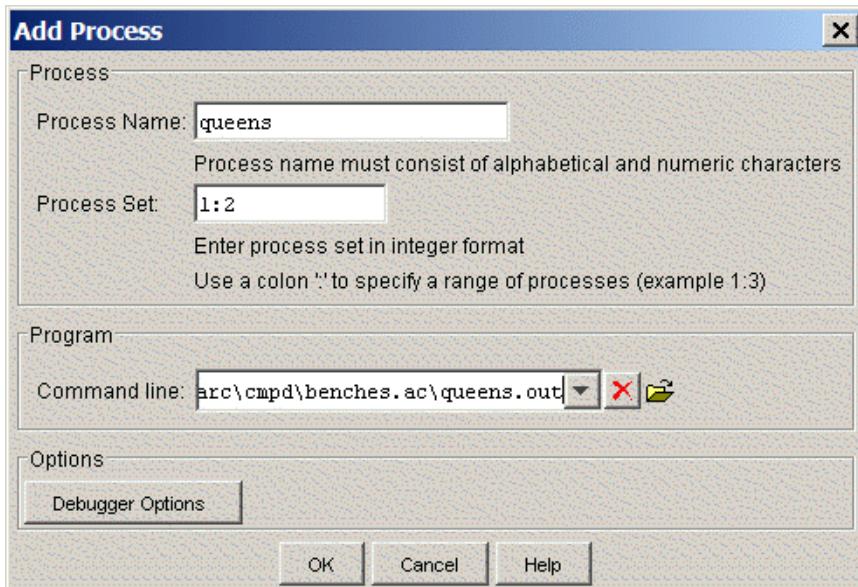


- In the **Set CMPD Options** dialog, click **Add Process**.
- In the **Add Processes** dialog, enter **queens** into the **Process Name:** field. The process name should start with an alpha character and may not contain any spaces.

NOTE After the process is created you can not edit the name.

- In the **Process Set** field enter **1:2**. This creates two processes that share the same target options and debugger command-line options. You can debug each of these processes at once or separately. To learn more about process sets, see [Creating Session Configuration Files](#) on page 173.
- In the **Command line:** field, enter or browse to the location of **queens.out** located in *install_dir/cmpd/benchmarks.target/queens.out*

NOTE When you are debugging your own multi-process project, you can specify options by clicking **Debugger Options** and/or **Target-specific options**. For the demo project, proceed to the next step.



15. Click **OK** to add the process.
16. Repeat the steps to add the remaining three processes using the following process information:

Process Name: sieve

Process Set: 3

Command line: *install_dir/cmpd/benches.target/sieve.out*

Process Name: greycode

Process Set: 4

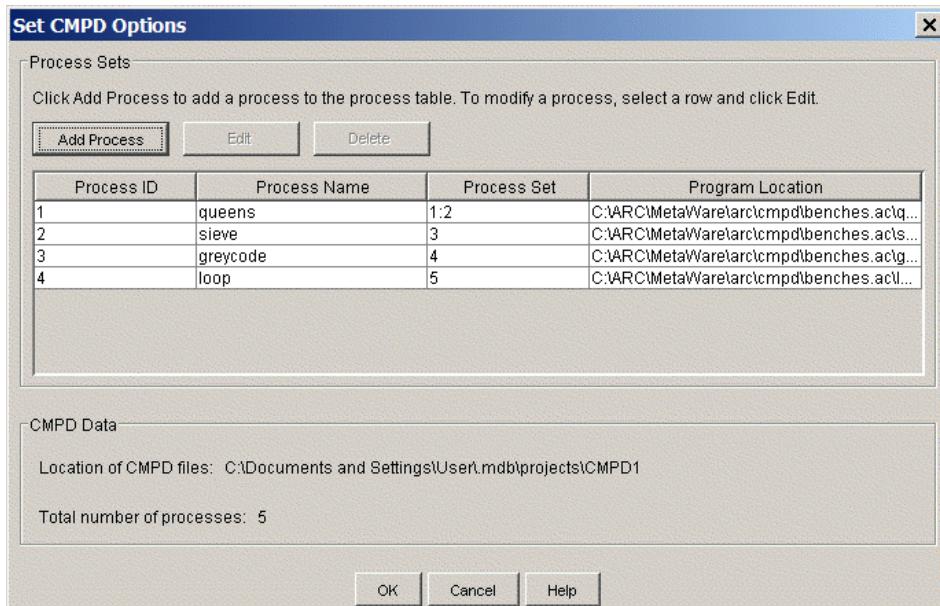
Command line: *install_dir/cmpd/benches.target/greycode.out*

Process Name: loop

Process Set: 5

Command line: *install_dir/cmpd/benches.target/loop.out*

17. After creating each process, click **OK**.



18. Click **OK** twice to begin the debug session. See [Running a CMPD Session](#) on page 163 for examples of how to debug using CMPD.

Editing a Multi-Process Configuration

To edit your CMPD configuration, such as target settings, or to add or eliminate a process while you are currently running a CMPD session:

1. Select **Tools | Multi-Process Debug Options**.
2. Select the process you want to modify and click **Edit**.
3. Change options using the **Edit Process** dialog, click **OK** when done.
4. When finished modifying CMPD settings, click **OK**.
5. Exit and restart the debugger.
6. Select the project containing your CMPD project.
7. Click **OK** to restart the debug session with the new settings.

NOTE You must exit and restart the debugger to view any changes in the options you set.

Running a CMPD Session

Try a demonstration of CMPD extensions. Two demos are included with the MetaWare debugger: one running multiple single-threaded processes, and one running multiple multi-threaded processes.

Running a Single-Threaded Demo

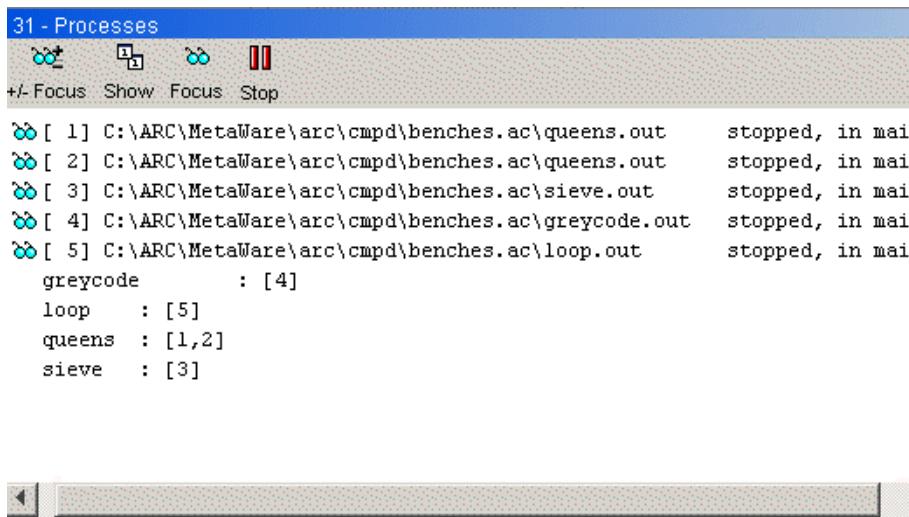
In this demo, the debugger runs four programs found in the directory listed in [Sample Demonstration Programs](#) on page 159. You can use a CMPD configuration by following the steps in [Creating the](#)

[Multi-Process Debug Session](#), or you can navigate to the directory in [Sample Demonstration Programs](#) and run `demo.bat` (Windows) or `demo` (UNIX or Linux).

Running, Stepping, Stopping, and Restarting the Processes

1. Launch the debugger and select the project CMPD1 that you created in [Creating the Multi-Process Debug Session](#).
OR
Navigate to the directory in [Sample Demonstration Programs](#) and type **demo** to start the debugger and bring up a **Source** window and the **Processes** window.
2. Notice the [1] at the top right of the **Source** window; the [1] shows that this window points to the first instance of the **queens** benchmark. You can click the [1] to cycle through the **Source** windows for the other instances.
3. Click [**Run**] on the main toolbar. All processes except [5] run to completion and produce output (if you are running the debugger on Windows, see the **Command Prompt** window in which you started the demo).
4. Process [5] keeps running, and the debugger's run indicator persists. Click [**Stop**] toolbar to stop the process.
5. Click [**Restart**] and select **Display | Processes**.

The **Processes** window displays blue eyeglasses for each process, showing they are all in the focus: any command you issue applies to all. The **Processes** window lists not only processes, but all defined focus sets.



```
31 - Processes
+/- Focus Show Focus Stop
OO [ 1] C:\ARC\MetaWare\arc\cmpd\benches.ac\queens.out      stopped, in mair
OO [ 2] C:\ARC\MetaWare\arc\cmpd\benches.ac\queens.out      stopped, in mair
OO [ 3] C:\ARC\MetaWare\arc\cmpd\benches.ac\sieve.out       stopped, in mair
OO [ 4] C:\ARC\MetaWare\arc\cmpd\benches.ac\greycode.out    stopped, in mair
OO [ 5] C:\ARC\MetaWare\arc\cmpd\benches.ac\loop.out        stopped, in mair
    greycode      : [4]
    loop         : [5]
    queens       : [1,2]
    sieve        : [3]
```

6. Click **Src Into** to source-step all processes. The single **Source** window shows you where process [1] is. Click [1] to rotate through the other processes.
7. Change the focus to just the two **queens** programs: go to the **Processes** window and double-click the line: **queens : [1,2]**
queens is the name of a process set whose contents are [1,2]. After double-clicking, verify that the eyeglasses include just the two instances of **queens**.
8. Click [**Run**]. Only the **queens** programs run to completion.

9. Click [**Restart**]. Because only processes [1,2] are in the focus, only these processes restart. Like other commands, the restart command applies only to those processes that are in the focus.
10. Even though only [1,2] are in the focus, you can resume execution of any process by right-clicking its process line, then clicking [**Run**].

Try this with the loop process, [5]. Stop this process by right-clicking [5] and then clicking [**Stop**].

Using Breakpoints and Synchronizing Processes

Now run the two **queens** processes slightly out of sync.

1. Check the eyeglasses in the **Processes** window to confirm that the two **queens** processes are the only two processes in the focus.
2. Break on the **Try()** function:
 - a. Choose **Tools | Set Breakpoint**.
 - b. Select **Function**.
 - c. Enter **Try** (without parentheses).
 - d. Click [**OK**].

This sets a breakpoint at function **Try()** for all processes in the focus. Now you have the same breakpoint at **Try()** for both **queens** programs.

3. Run just one of the two **queens** processes:
 - a. Double-click [**2**] in the **Processes** window to remove [2] from the focus, and now just [1] is in the focus.
 - b. Click [**Run**] four or five times.
 - c. Bring up a single **Examine** window. The window is labeled [1].
 - d. Type **A** in the **Examine:** field and press **ENTER**, to show array **A** in the benchmark.
 - e. Click [**1**] to rotate to process [2].
 - f. Press **ENTER** again in the **Examine:** field to input **A** again. You see a slightly different array **A**, because [1] changed the values in **A** as the benchmark ran.
 - g. Rotate through processes [1] and [2] repeatedly to see the differences.
 - h. In the **Command:** field, type the following:
[1,2] eval A

The debugger displays a single line where the array values are the same in both processes, and shows individual lines where the values differ.

4. Close the **Examine** window.

Using One Process to Control Another

Now run all processes, but stop the infinite-loop process when the first **queens** finishes.

1. In the **Command:** field, enter the following:
[1] b queens.c!69, stop [5]
2. Make sure every process is in the focus, by double-clicking in the **Processes** window until eyeglasses appear by each.
3. Click [**Run**]. This time you should notice that the infinite-loop program has stopped at the breakpoint.

Animating Multiple Processes

Now animate all the processes at once:

1. Click [**Restart**] to restart the processes.
2. Choose **Display | Disassembly** to bring up disassembly windows for all processes; reposition these windows so you can see all of them.
3. Click [**Animate**] then [**Instr Into**] to start instruction stepping.
4. Increase the animation speed using the [+] button.

NOTE The **Processes** window is updated for each execution event. Animation is faster if you close this window.

Running a Multi-Threaded Demo

The following demo uses precompiled programs and you will not be able to edit settings for the individual processes. Try running three threaded RTOSs on two different architectures:

- ThreadX on an ARCTangent-A4 processor (process [1])
 - Nucleus PLUS on an ARCTangent-A4 processor (process [2])
 - MQX on the ARM processor (process [3])
1. Go to directory *install_dir/cmpd/rtoses.cpus* and type **demo** to start the debugger. The **Processes** window, three RTOS-specific windows, and a **Disassembly** window appear.
 2. Click [**Run**]. All processes run until you click [**Stop**]. Now all the OS-specific windows show the state of the respective RTOSs.
 3. Click [**1**] in the **Disassembly** window to rotate through the processes.
 4. Click the various resources in the RTOS-specific windows to see more information about those resources.
 5. Click any of the eyeglasses in the **Processes** window. Each process expands to list its respective threads.

Locking Windows to a Thread

1. Double-click any thread in the expanded list to lock windows for that process to that thread; a lock icon appears to the left of the locked thread.
2. Place a lock on threads in one or more of the RTOSs and rotate through the processes in the **Disassembly** window. The number of the locked thread appears in the title bar.

For example, to lock to a non-current thread in the ThreadX process:

1. Set the **Disassembly** window to [**1**].
2. Type *compute* in the **Goto:** field.
3. Press ENTER, and you should see `compute_square` displayed.
4. Right-click **TS Break** to set a thread-specific breakpoint. Now the red box shows a 'T', indicating thread-specific.
5. Now run the processes. [1] stops when it reaches the breakpoint; the others resume.

- Click [Stop] to stop all processes.

Setting Thread-Specific Breakpoints for a Process

This method differs from the non-CMPD way to set a thread-specific breakpoint; see [Setting Task- or Thread-Specific Breakpoints](#) on page 63.

- Lock the windows for a process to the thread for which you wish to set the breakpoint.
- Use the **Disassembly** or **Source** window to locate the place where you wish to set the breakpoint.
- Right-click the line you selected, and click **[TS Break]** to set the thread-specific breakpoint.

Viewing and Controlling Processes

The debugger has features specialized for debugging multiple processes. Now that you have configured your multi-process session, you will need to become familiar with how to view and manage debugger windows.

Viewing Different Processes Using One Window

Data windows (like the **Disassembly** window) for CMPD processes have a number, such as [1], in the title bar. You can view the window for different processes by clicking the number.

You can also enter a process number in the field next to the number or select a process from the drop-down list of numbers you previously entered.

```

17 - [1] Disassembly

Goto: [1] Break Mix Ihex Lhex

main      st      *blink,[%sp,4]
main+0x04 st      *fp,[%sp]
main+0x08 mov    *fp,%sp
main+0x0c sub    *sp,%sp,44
main+0x10 st      *r13,[%sp,16]
main+0x14 st      *r14,[%sp,20]
main+0x18 st      *r15,[%sp,24]
main+0x1c st      *r16,[%sp,28]
main+0x20 st      *r17,[%sp,32]
main+0x24 st      *r18,[%sp,36]
main+0x28 mov    *r15,%r0
main+0x2c mov    *r14,%r1
queens.c!49
main+0x30 mov    *r18,%8
main+0x34 mov    *r17,0x1_b210 ; 0x1_b210 = _bufendt;
main+0x3c st      *r18,[%r17,-256]

```

Click the **[n]** number again to cycle to the window for the next process.

Controlling Processes Using the Processes Window

Use the **Processes** window to select and control which processes are currently affected by run and step actions.

Select **Display | Processes** to open the **Processes** window. This window lists:

- every command processor
- each processor's contained process

- the status of each process
- whether a command processor is the focus

A pair of blue eyeglasses appears to the left of a process if that process is in the focus. This window lists all defined (named) process sets, and shows the state of each process: running; stopped; or stopped/held (by a barrier).

Changing the Focus

The debugger operates only on processes that are in the focus. You can use the **Processes** window to change the focus, in a number of ways.

First, click a line that shows a process or a process set. You can use any of the following methods to set the focus to the selected line:

- Click **[+/- Focus]** to toggle the current focus.
- Double-click the selected line.
- Click the eyeglasses button. You can also right-click the line to get the same eyeglasses button.

Showing the Same Process in all Windows

You can direct all windows to show the data for a process:

1. Click a line showing a process.
2. Click **[Show]** to redirect all windows to show data for the process.

Running and Stopping Processes

You can run or stop execution of a process or processes:

1. Right-click a line showing a process or process set in the **Processes** window.
2. Select **Run** to start the process, or all processes in the process set.
3. Select **Stop** to stop the process, or all processes in the process set. (This is the same as left-clicking this line and clicking the **[Stop]** button.)

Adding a Stepping Toolbar to the Processes Window

If you frequently switch between processes, you can add a stepping toolbar and command line to the **Processes** window. The buttons on this stepping toolbar are identical in function to the toolbar at the top of the debugger frame.

To add a stepping toolbar, right-click in the **Processes** window and select **Toolbars | Stepping toolbar**. To add a command line, select **Toolbars | Command line**.

Working with Threads in the Processes Window

When a process is multi-threaded, the **Processes** window can show all the threads in the process, and cause any windows for that process to show a particular thread. Click the eyeglasses for a process to list its threads. If no eyeglasses are visible, click the **[+]** symbol.

Double-click a line listing a thread to lock the process's windows to that thread; a lock icon appears on the left. Double-click again to unlock.

You can set a thread-specific breakpoint as follows:

1. Lock the windows for a process to the thread for which you wish to set the breakpoint.
2. Use the **Disassembly** or **Source** window to locate the line where you want to set the breakpoint.
3. Right-click the located line and click **TS break** to set the thread-specific breakpoint.

Using Breakpoints and Watchpoints

Breakpoints and watchpoints are assigned integers as they are defined. The integer is process-specific, so if you set the same breakpoint in three processes, each breakpoint might be assigned the same integer.

If you are debugging distinct programs, you usually set distinct action points in each. You can do that by prefixing the breakpoint or watchpoint command with a process set (such as [1] b Try or [2] watch X), or by using the **Processes** window to restrict the focus before you set a breakpoint or watchpoint using the **New Breakpoint** or **New Watchpoint** dialogs.

If you are debugging the same program in multiple processes — for example, running the same program on a simulator and on hardware at the same time — you might want the action points to apply to all processes. If the focus is all processes, an action-point command applies to all of them.

Compound Action Points

Whenever a command sets more than one breakpoint or watchpoint, it creates a compound action point. A *compound action point* is a set of one or more of action points for a set of different processes. A compound action point allows you to disable or enable all of the action points in the set by referring to a single integer (that of the compound action point). The numbering of compound action points is artificially higher than those of regular action points; compound action points start at 101.

The action points within a compound action point can be manipulated with the integer assigned the compound action point using the syntax *N, where N is the point number for the compound action, to denote the members of the compound action point. If you use N alone, you are referring solely to the compound point and not its members.

Action-Point Stop Lists

In CMPD, an action point can have an associated a *stop list*: a list of processes that stop when the action point is encountered. You use the syntax

, stop [pset]

in an action-point specification, where *pset* is a process set. For example:

[4] b Try, stop [1:3]

When [4] breaks at Try(), the debugger stops processes 1:3 as well.

Using Barriers to Synchronize Processes

A *barrier* is a set of action points, compound action point or single breakpoint or watchpoint. You can use a barrier to synchronize processes during a debug session.

You might want to step and run several active processes, but ensure that until all reach a certain point (the barrier), none will proceed past that point. For example, processes 1, 2, and 3 might all call the same **enqueue()** function; but until they all do so, you do not want any of them to run past that point. You can achieve this using a barrier.

NOTE You can only set barriers using debugger commands. Enter all commands into the **Command:** field.

Creating a Barrier

You specify a barrier with the command:

```
barrier APnum1 APnum2 ...
```

where *APnum1 APnum2 ...* are action-point numbers. This command creates a new barrier, which is also assigned an action-point integer so it can be manipulated. The “affected process set” consists of those processes addressed by the action points in the barrier.

You can create a barrier from a single compound action point:

```
barrier CPnum
```

where *CPnum* is the compound action point. The barrier semantics apply to each regular action point in the compound action point.

To obtain a list of barriers, use the **barrier** command with no arguments. Currently there is no graphical display of barriers, so you must type **barrier** in the **Command:** field.

Understanding How Barriers Work

A newly created barrier is unsatisfied. A barrier is said to be *satisfied* when all processes in the affected set have encountered their respective action points. When a process encounters its action point, if the barrier is unsatisfied, the process is “held”; the debugger does not resume execution of that process until the barrier is satisfied. When the barrier is satisfied, all processes are free to resume execution.

Each non-compound action point in the barrier must be encountered before any process in the affected process set may continue past the point at which it encountered its action point in the barrier. That is, each process can be run, stepped, and so on, until it hits the action point in the barrier. Then, the debugger does not resume execution of that process until all the other processes in the barrier have similarly hit their action points. When this occurs the barrier is said to be satisfied.

The debugger analyzes a barrier specification to be sure that it does not contain two action points for the same process, which would almost always make it impossible to satisfy a barrier.

In other words:

1. Set a barrier. It is initially unsatisfied.
2. Debug processes. When a process encounters an action point in the barrier, it is held. The debugger ignores attempts to resume execution of the process, even if you enter a command (for example, [3] run).
3. Keep debugging processes until the barrier is satisfied.
4. All processes in the barrier’s affected set are now free to resume execution upon command.

Setting Different Action Points for Different Processes

When you define a barrier, you can set different action points for different processes. For example:

```
[1] b foo // set point 1, process 1
[3] b goo // set point 1, process 3
[5] watch X // set point 1, process 5
barrier 1 // set barrier 101.
```

Processes 1, 3, and 5 are not allowed to proceed past their respective action points until all have reached their respective action points: namely, process 1 has broken at **foo()**, 3 at **goo()**, and process 5 has modified memory at location X.

This example sets barrier number 1, where 1 refers to action-point 1 in all processes. If the action-point number after a **barrier** command is greater than 100, it is taken to be a compound action point number. If the number is 100 or less, it is taken to be a regular action point, and all action points with that number for processes in the focus are included in the barrier.

You can change the focus to restrict the set of processes whose action points are specified in the barrier command.

As another example, suppose this breakpoint command:

b Try

defines:

- action point 1 in each of several processes
- compound point 101 that refers to these action points

Then you can specify:

barrier 101

to effectively place the action points in a barrier. This command creates action point 102, which is the barrier.

Disabling and Enabling Barriers

The **disable** command disengages a barrier, and all affected processes are now free to continue.

The **enable** command re-enables the barrier and re-establishes it from the beginning for each affected process: that is, each must now encounter its respective action point again.

Deleting a Barrier

To delete a barrier, use the **delete** command, specifying the action-point number of the barrier.

Using Commands with CMPD

Several aspects of command usage are different when you use the CMPD extensions. This section describes the differences.

NOTE You can enter all commands in the debugger **Command:** field.

Combining Command Output

For most commands, CMPD has a unique multi-way comparison feature that shows the results for each process, combining identical output for multiple processes on a single line, interleaving combined results with non-combined results. This feature allows you to see more quickly where distinct processes differ and where they are the same. This is most useful when you are running the same program in different processes.

For example, this register listing shows that the register set is largely the same for all processes:

```
> reg  
[all] r0 00000001 r12 00000001 r24 00000000 ZNCV 0000
```

```
[a11] r1 0002cc77 r13 00000008 r25 00000000 21HR 0010
[a11] r2 00000001 r14 0002cc90 gp 0001af5c stat 02004053
[a11] r3 0001c88e r15 0002cc88 fp 0002cc78 sem 00000000
[a11] r4 0000000a r16 0001b06c sp 0002cc60 LPS 00004e61
[a11] r5 0001bac4 r17 00000004 ILNK1 00000000 LPE 00004e66
[1] r6 00000001 r18 00000001 ILNK2 00000000 ident ffff0107
[2] r6 00000001 r18 00000001 ILNK2 00000000 ident ffff0207
[3] r6 00000001 r18 00000001 ILNK2 00000000 ident ffff0307
[4] r6 00000001 r18 00000001 ILNK2 00000000 ident ffff0407
[a11] r7 0001b9dc r19 0002cc94 blink 000040d8 debug 20000000
[a11] r8 00000000 r20 00000001 LPC 00000000 timer 00000000
[a11] r9 00000000 r21 0001bb2c pc 0001014c tctrl 00000000
[a11] r10 00000000 r22 00000000 retpc 00010360
[a11] r11 00000000 r23 00000000 icnt 00000ff8
```

Four lines are different for the four processes; these lines are prefixed with their respective process numbers. (These lines happen to be different because the processor `ident` register contains the distinct processor number in a four-processor configuration.)

Not all commands are subject to output combination:

Commands Subject to Output Combination

Yes	No
<code>source</code>	<code>isi</code>
<code>eval</code>	<code>iso</code>
<code>locals</code>	<code>ssi</code>
<code>globals</code>	<code>sso</code>
<code>dis</code>	<code>run</code>
<code>mem</code>	<code>stop</code>
<code>break</code>	<code>macro</code>
<code>watch</code>	<code>endm</code>
	<code>read</code>

Generally, the debugger combines a command's output if it makes sense to do so. Execution commands produce output (for example, `program stopped`) at unrelated times, so it does not make sense to combine them.

Completing Commands

When you use the command-line version of the debugger without CMPD, each command finishes before the debugger returns to the command prompt. If a command initiates execution, completion includes the program stopping execution (for example, finishing, or encountering an action point). Pressing **CTRL+C** stops the process and returns the debugger to the command prompt.

When you use the command-line version of the debugger with CMPD, and you issue a command that initiates process activity, the debugger does not return to the command prompt until all processes in the focus have stopped. If you press **CTRL+C**, the debugger returns to the prompt, leaving the processes running.

While processes are running and you have a command prompt, you can issue additional commands:

- | | |
|----------------|--------------------------------|
| process | Look at the state of processes |
| stop | Stop all running processes |

[A] stop	Stop processes in process set [A]
wait	Wait for all processes in the focus to stop

The debugger with CMPD can run processes on simulators and on hardware simultaneously; each simulator gets a time slice, as long as the simulator is still executing.

If you interrupt command completion and obtain a debugger prompt, simulators make no further progress, even though they are technically “running”. You must use the **wait** command to resume progress in the simulators. Because the debugger itself is not multi-threaded, the simulators do not execute in the background; they execute only when you do *not* have a debugger prompt.

Adding Processes During a Debug Session

You can start the debugger with no command processor slots and load individual processes one at a time.

Use **newcmd** to create a new command processor slot, followed by **load** to load a process. Or, use **loadm** to create a slot for a new process and load it at the same time.

Configuring a CMPD Session on the Command Line

When you use the command-line debugger to configure a session, you run the debugger once for each slot; but instead of actually debugging a process, you instruct the debugger to save the configuration for that slot. The debugger runs in a “configure but do not debug” mode; after it configures the command processor slot, the debugger exits.

After you run your configuration commands, the result is a set of session configuration files that can be used whenever you want to run the debugger in command-line or GUI mode.

Creating Session Configuration Files

A configured session consists of a set of files with special names; these files contain debugger macros and commands that implement the configuration. Each process set’s configuration is stored in a file named:

`.sc.args.pset_name.multi`

where *pset_name* is the name you assign to a process set. The location of the `.sc.args...` files is determined by the current option set, which by default is “local,” denoting the debugger’s current working directory.

Note You must use the debugger GUI to create new option sets; however, you can use the command-line debugger to switch between them. See option [set optset](#) in [Command-Line Option Reference](#) on page 274.

You execute the command-line debugger repeatedly in “configuration mode” to configure the session. Configuration mode looks like this:

`mdb -psetname=N -pset=P [more opts] prog_name args`

This command creates a configuration *N* for process set *P* and writes the configuration into a file named `.sc.args.N.multi`. The nature of the configuration is determined by the [*more opts*] command-line options you supply, and by the program *prog_name* and its arguments *args*.

CAUTION If you specify an argument to *prog_name* that is identical to a debugger control, the debugger does not pass that argument to your program. Be particularly careful when using arguments that start with a dash (-). If you precede the argument with two dashes (--) and a space, the debugger discards both dashes and treats all subsequent arguments on that command line as arguments to *prog_name*.

For example:

```
mdb -psetname=abc -pset=1 -Xlib queens
mdb -psetname=def -pset=2:3 -hard greycode
mdb -psetname=xyz -pset=4 someprog 1 2 3
```

These commands define three named process sets:

Name	Process Set
abc	1:1
def	2:3
xyz	4:4

These three sets comprise four processes; process set [2:3] runs the same process twice in command processors 2 and 3. The configuration is stored in three separate files:

```
sc.args.abc.multi
sc.args.def.multi
sc.args.xyz.multi
```

Each file contains the debugger macros and commands necessary to create the command processor slots for the process set, and to initiate execution of the programs.

Specifying a Session Configuration

When you invoke the debugger, you use command-line option **-multifiles** to tell the debugger to read the session configuration files when debugging. For example:

```
mdb -multifiles=abc,def,xyz -OK
```

NOTE Because the filename of the application being debugged is embedded in your session-configuration files, you must include option **-OK** when invoking a CMPD session with **-multifiles** on the command line.

You can specify a subset of configuration files, and tell the debugger to process them in any order (subject to shared memory constraints). For example:

```
mdb -multifiles=abc,xyz -OK
mdb -multifiles=abc -OK
mdb -multifiles=xyz,def -OK
```

When you choose the process sets for a configuration, you must make sure they do not intersect. If the same process appears in more than one set, the debugger refuses to load the process more than once. However, process sets do not have to be contiguous. For example, you can specify [1], [2:7], and [9:11,13]. The sets do not have to be in ascending order.

Specifying When to Load Programs

Programs are initially loaded when the debugger starts up, unless you specify otherwise by turning off toggle `initial_load`. For example:

```
mdb -psetname=abc -pset=1 -Xlib queens
mdb -psetname=def -pset=2:3 -hard greycode -off=initial_load
mdb -psetname=xyz -pset=4 someprog 1 2 3
```

When you invoke the debugger for debugging, the programs in process sets [1] and [4] would be loaded initially, but the programs in process set [2:3] would not. When you are ready, you use the `load` command to load the remaining programs:

```
mdb> [2] load -- loads the program in 2
mdb> [3] load -- loads the program in 3
```

or:

```
mdb> [2,3] load -- loads the programs in 2 and 3
```

Debugging Shared Memory on ARCTangent-A4 Systems

For older ARC systems, debugging a shared-memory configuration takes some care. The debugger consumes memory to store breakpoint information in a breakpoint table, and the debugger has to share that table among all the processes being debugged in the same shared memory. For ARC systems using processor core version 7 or later, with the `brk` instruction, this is not an issue.

To debug a shared memory system that requires breakpoint tables, you must designate one process to be the “owner” of the memory, and specify that all other processes in the same memory space use the memory of the owner. You do this by “exporting” the memory from the owner, and “importing” the memory into the other processes. In addition, you must load the owner before you load the others, so that the owner “comes into existence” before the other processes need to use its memory. For example:

```
mdb -a4 -psetname=abc -pset=1 -exportmem=MAIN -Xlib queens
mdb -a4 -psetname=def -pset=2:3 -importmem=MAIN -hard greycode
mdb -a4 -psetname=xyz -pset=4 -importmem=MAIN someprog 1 2 3
```

Here process [1] is the owner of the memory and the other processes import it. The debugger knows to share the breakpoint table among all the processes.

In a shared memory application you might have just one executable program that you download once, and have each processor run some portion of that executable. In this case it is pointless to have each process download the same program multiple times. You can turn off toggle `download` to ensure that only the memory owner downloads the program:

```
mdb -a4 -psetname=abc -pset=1 -exportmem=MAIN queens
mdb -a4 -psetname=def -pset=2:4 -importmem=MAIN queens \
-off=download
```

The debugger halts the processors for processes 2:4 and downloads nothing. You must set the program counter and allocate a stack appropriately for the processes in 2:4 to begin execution. If your program does not figure this out by itself, we suggest executing a debugger macro at the outset. If you specify:

```
mdb -a4 -psetname=def -pset=2:4 -importmem=MAIN queens \
-loadexec="read share.scm $cid" -off=download
```

the debugger executes the macro `share.scm` (located in directory `install_dir/cmpd/shared_mem`) upon process creation. For convenience `share.scm` is also passed the value of the process number (2, 3, 4). Here is an example of a macro we have used to set up the stack and initialize a register to tell the program which CPU it is:

```
eval r23=$1
if ($1 > 1) { eval pc = main ; eval sp =
(int)((char*)&stacks)+stacksize*($1-2)+stacksize ; }
```

The macro sets up a separate stack for each of the processors and uses `r23` to tell each processor which one it is.

See the example in directory `install_dir/cmpd/shared_mem` for a working program that runs on four processors.

A shared memory system has additional considerations. The C run-time library uses static data; two processors reading and writing that same data can wreak havoc. Even though the library supports locking around such data for multi-threaded applications, the semaphore locking calls need to be implemented and are processor-specific. You can avoid this problem by ensuring that only the first processes makes use of any such aspects of the run-time library: file I/O, memory allocation, and any hostlink communication with the debug host. This protocol is observed in the example in the `shared_mem` directory.

Generating Complete Configurations on the Command Line

You can use option `-multifiles` along with option `-multiout` to generate a configuration that includes every option and argument you would ordinarily specify on the command line. This allows you to run the debugger with no command options. For example:

```
mdb -multifiles=abc,def,xyz -multiout
```

The debugger writes the entire configuration in file `.multi`, which contains commands to read in the individual `.sc.args.N.multi` files. The debugger places the `.multi` file in the currently selected project (option set) or in local. Then you can do the following:

1. Invoke the debugger GUI with `mdb` (and any [--arc*](#) series).
2. Select **Multi-process session**.
3. Click **[OK]**.
4. Start debugging.

NOTE Some configurations generated by this command-line process *do not* show up in the debugger CMPD Options dialog.

Combining all Configurations in a Single File

Rather than using the `.sc.args.N.multi` files, you can direct the configuration output to a single file. Use the option `-multiout=stdout` to direct the output to standard output rather than to the normal configuration files. For example:

```
del total
mdb -psetname=abc -pset=1 -Xlib queens \
-multiout=stdout >> total
mdb -psetname=def -pset=2:3 -hard greycode \
-multiout=stdout >> total
```

You use this configuration with:

```
-multifile=total
```

This prevents creating the `.sc.args...` files.

Using Standard ARC Processor Configurations

Command-line option **-multiarc** provides a shortcut for configuring multiple ARC processors identically. When you use this option, you cannot configure any of the ARC processors differently. Specify:

```
-multiarc=N
```

where N is the number of ARC processors you want to debug. This creates N clones of the appropriate system. The clones are named `arc2`, `arc3`, Each cloned system receives the property assignment `cpunum=N` where N is 1, 2, 3, and so on. This way the implementation of the DLL interface can know which processor it is intended for. If you specify option **-DLL**, the system `ARC_DLL` is cloned. Otherwise, the MetaWare debugger system `ARC_simulator` is cloned.

Because the interface to the DLL was designed at the outset to be object oriented, the debugger merely creates multiple instantiations of the object. The `cpunum` property allows each DLL to know which ARC processor it is.

Specifying:

```
mdb -multiarc=N progname args...
```

is equivalent to specifying:

```
mdb -psetname=noname -pset=1:N progname args...
mdb -multifile=noname
```

to place the same program in each of N processors.

The intended operation is illustrated in the following scenarios:

Scenario 1:

```
mdb -multiarc=N progname args
```

Creates N ARCTangent-A4 command processors [1]..[N], and loads *progname* in [1]. Then:

```
mdb> [2] load anotherprog -- loads anotherprog in [2]
mdb> [3] load ... -- etc.
```

Scenario 2:

```
mdb -multiarc=N -off=initial_program progname args
```

Creates N ARCTangent-A4 command processors [1]..[N], and does *not* load any program. Then:

```
mdb> [1] load progname
mdb> [2] load anotherprog -- loads anotherprog in [2]
mdb> [3] load ... -- etc.
```

or:

```
mdb> load progname -- loads progname in [1:N]
```

Scenario 3:

```
mdb -multiarc=N -off=initial_program \
-on=shared_program progname args
```

Creates N ARCTangent-A4 command processors $[1]..[N]$, and does *not* load any program. Processors $[2:N]$ share memory with $[1]$ and will neither download the specified program upon load nor go to **main()**. Then:

```
mdb> load programe -- loads programe in [1:N]
```

Downloads in $[1]$ but not in $[2:N]$. On first program load in $[1:N]$ the file **share.scm** in the current directory is processed as a macro, with the number of the command processor ($1..N$) as parameter **\$1**.

Scenario 3: allows initialization of a stack and PC for processes $2..N$. The following test case uses the file (macro):

```
eval r23=$1
if ($1 > 1) { eval pc = main ; eval sp =
(int)((char*)&stacks)+stacksize*($1-2)+stacksize ; }
```

Here **r23** is passed in to indicate to the program which processor it is. For $[2:N]$ the PC is initialized to **main()** (because downloading is turned off for $[2:N]$, you get whatever the PC is on the machine; on the simulator it is 0), and so you have to initialize it. The test program allocates storage for stacks for $[2:N]$, and the macro initializes **sp**.

If you have shared-memory ARC processors and you are running the same program on all of them, only one of the processors can execute the standard run-time library initialization and use hostlink. See the example in directory *install_dir/cmpd/shared_mem* where we configure one processor to do the initialization and service **printf()** requests, and three other processors to do actual computation.

Configuring a Session with the Debugger GUI

You can use the debugger GUI **Options** dialogs to configure a session. You use a separate **Options** dialog to configure each command processor slot. The options you specify are saved across invocations of the debugger, in ASCII files you can modify off line or transmit via e-mail.

You do not necessarily have to configure each command processor slot separately. If two or more of the slots are to be configured identically—for example, they run the same program—you can combine them in a process set (for example, $1,2$ or $3:7$) and configure the entire process set at once.

You must identify the process set to which each configuration applies. If a process set contains just one number, you are configuring just one command processor slot; if it contains more than one number, you are configuring multiple command processor slots simultaneously. You must navigate the **Options** dialogs for each process set.

Using the CMPD Options Dialogs

1. Start the debugger.
2. In the **Debug a process or processes** dialog, select **Multi-process session**.

- Click **CMPD Options**. The **Configure multi-process session** dialog appears. Use this dialog to define and configure named process sets.

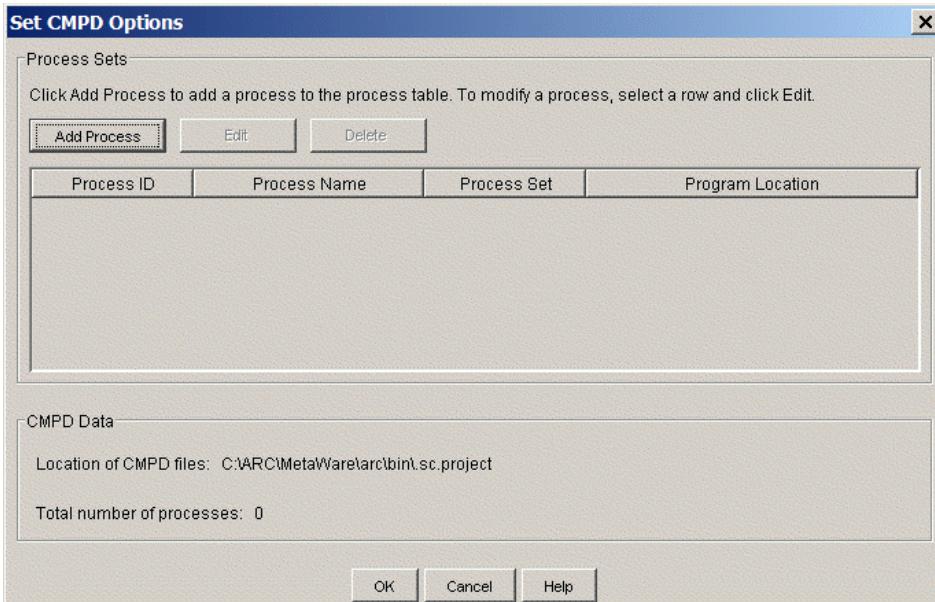


Figure 42 Set CMPD Options

- Click **Add Process** and enter a **Process Name** and **Process Set** definition for each process set. For example, in the **Process Name** box, enter `main` and in the **Process Set** box enter `1:7`. This defines `main` to have seven processes, each with the same configuration.
- In **Program | Command line:** enter or navigate to the location of the program you wish to debug.

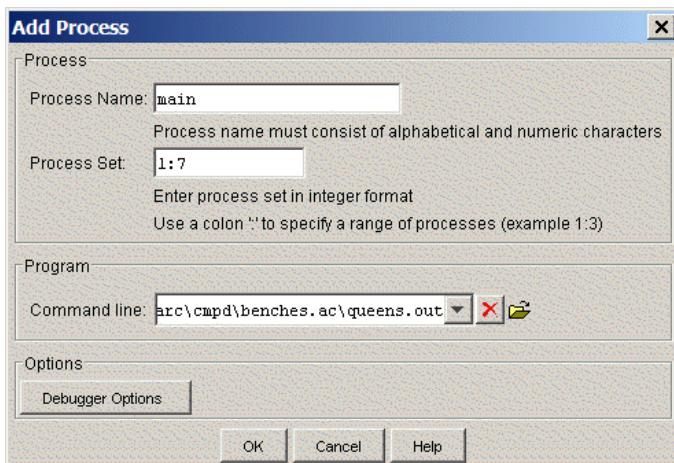


Figure 43 Add Process Dialog

- Click **Debugger Options** to bring up a single **Debugger options** dialog dedicated to configuring only the processes in that process set. Here you can set unique hardware or software simulator characteristics for the process set.
- Click **OK** to exit the **Debugger options** dialog and save the configuration, and click **OK** again to exit the **Add Process** dialog and save the process definition.

Configuring a CMPD Session on the Command Line

Your process is listed in the Set CMPD Options dialog.

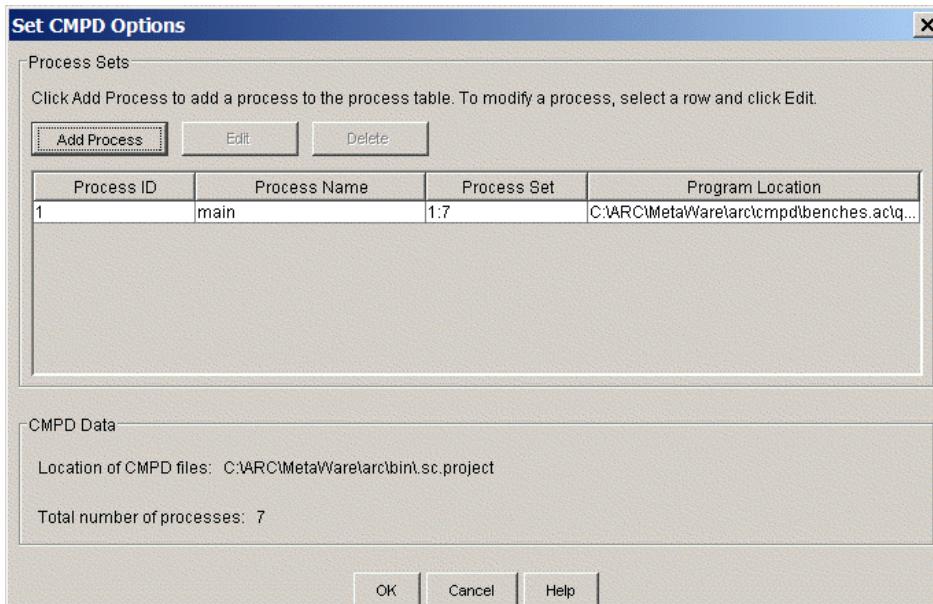


Figure 44 CMPD Options with Saved Processes

8. Click **OK** to save and exit.

You have now created a configuration that can be used for multiprocess debugging. To begin debugging, click **OK** in the **Debug a process or processes** dialog.

To use this multiprocess configuration on the command line, specify one of the following.

```
mdb -multifile=options
```

See [Specifying a Session Configuration on page 174](#) for information about specifying configurations on the command line.

Chapter 6 — Command Features

In This Chapter

- [Specifying Controls](#)
- [Using Debugger Commands](#)
- [Conventions for Debugger Input](#)
- [Creating and Using Debugger Macros](#)
- [Modifying the Driver Configuration File](#)

Specifying Controls

You can control the debugger's operation by specifying options and toggles on the driver command line or in the GUI dialogs.

On the command line, you can specify options, toggles, program name, and program arguments in any order.

CAUTION If you specify an argument to your program that is identical to a debugger control, the debugger does not pass that argument to your program. Be particularly careful when using arguments that start with a dash (-). If you precede the argument with two dashes (--) and a space, the debugger discards both dashes and treats all subsequent arguments on that command line as arguments to your program.

To specify command-line options or turn toggles on or off in the GUI, do the following:

1. Select **Tools | Debugger Options**.
2. Select **Command-Line Options**.
3. Enter the options you want to specify and the toggles you want to turn on or off.
4. Click **OK**.

The option and toggle settings take effect the next time you launch the debugger. To set options at the beginning of a debug session, see [Starting Your Debug Session](#) on page 16.

Controls specified on the command line are in effect only for the current debug session. However, you can save control specifications either of the following ways:

- Select **Tools | Debugger Options** to access settings dialogs where you can set options that will persist throughout your debug session.
- Make the controls defaults by adding them to the driver configuration file; see [Specifying Default Values for Command-Line Options](#) on page 228 for more information.

When you use the debugger GUI, the options you specify are saved to the current debug project (or the default debug project, if you are not using a project of your own creation). For more information, see [Working with Debug Projects](#) on page 41.

NOTE Some debugger options and toggles affect only the debugger GUI.

About Command-Line Options

Driver command-line options (also known as “switches”) configure the MetaWare debugger’s interface and control the debugger’s operation. Command-line options remain in effect for an entire debug session, including restarts. You must exit and restart the debugger to cancel the options.

[Command-Line Option Reference](#) on page 274 lists options in alphabetical order.

Option names are case sensitive. An option name is always prefixed with a dash (-). If an option takes an argument, there can be no spaces between the option, the “=” character, and the argument; for example, `-x=y` is valid; `-x = y` is not.

About Toggles

Toggles turn special debugger features on or off. Unlike command-line options, many toggles can be set and reset during the debugging session. However, you must restart the program being debugged in order for the changes to take effect. You can use a command-line option to specify a toggle when you first invoke the debugger, or you can use a debugger command to set a toggle from within the running debugger.

To enable a toggle when you invoke the debugger, use the syntax **-on=togglename**. To disable a toggle when you invoke the debugger, use the syntax **-off=togglename**.

[Toggle Reference](#) on page 298 lists toggles in alphabetical order.

Toggle names are case sensitive. Syntax depends on the context in which you specify a toggle:

Context	Syntax	
	Turn on	Turn off
On the command line when you start the debugger	-on=toggle=toggle_n ame	-off=toggle=toggle_n ame
In a Command: field or at the debugger prompt	prop toggle_name=1 or -on=toggle=toggle_n ame	prop toggle_name=0 or -off=toggle=toggle_n ame

Using Debugger Commands

You can control the debugger's operation interactively using debugger commands. You can use commands by doing any of the following:

- [Entering Commands at the Debugger Command Prompt](#)
- [Entering Commands in the GUI Command Field](#)
- [Entering Commands in a Script File](#)

Rules and Guidelines for Using Commands

MetaWare debugger commands function much like an interpreted language that follows standard C conventions. Commands have the following characteristics:

- Commands are case sensitive, and cannot contain uppercase letters. For example, entering **HELP** instead of **help** generates an error message.
- You can abbreviate most commands to a minimum character string, instead of specifying the full command name. For example, entering **attach** or **at** (or **atta**) produces the same result. The entries in [Alphabetical Listing of Commands](#) on page 232 document the abbreviations of the commands.
- Spaces are required in multi-word commands; for example:
`delete break`
- Compound commands: To enter more than one command per line, you must separate the commands with semicolons (with or without spaces) and enclose them in braces: `{isi ; register ; ssi}`
- Commands must not exceed one line. However, you can extend a long line by putting a backslash at the end of the line, for example:

```
if ((ap_match_value | ap_match_mask) == \
    (ap_hit_value_reg | ap_match_mask)) print ...
```

- Comments begin with // and extend to the end of the line.

CAUTION But beware of using comments in loops with compound commands, as your comment might nullify subsequent continuation lines:

```
while (condition) { \
    command1; \
    // Check card reader status: \
    command3; \
}
```

Here *command3* does not execute.

- Expressions use C operator precedence:

```
if ((x & y) == 0) print ...
```

- Commands may declare and reference debugger variables and reference built-in debugger variables; for more information, see [Using Debugger Variables](#) on page 188.
- Command lines must begin with a command, a command prefix, a command modifier, or a variable.

Related Topics

- [Commands Reference](#) on page 231
- [Creating and Using Debugger Macros](#) on page 224

Using Command Prefixes and Modifiers

You can prefix many MetaWare debugger commands with prefixes and modifiers that affect their execution.

- [Using Command Prefixes](#)
- [Using Command Modifiers](#)

Using Command Prefixes

Command prefixes can be used before a command to do the following:

- Filter the output of most commands using pattern matching
- Execute selected commands without waiting while hardware is running

Pattern-Matching Prefixes

A pattern-matching prefix causes any line of command output that does not satisfy its filter to be discarded. The pattern-matching prefixes and their filters are shown in [Table 10 Pattern-Matching Prefixes](#).

Table 10 Pattern-Matching Prefixes

Prefix	Filter
-grep=xxx	The output line must contain an occurrence of xxx.
-!grep=xxx	The output line must <i>not</i> contain an occurrence of xxx.

Table 10 Pattern-Matching Prefixes

Prefix	Filter
-regex=xxx	The output line must match xxx.
-!regex=xxx	The output line must <i>not</i> match xxx.
In all cases, xxx may contain the wildcard characters “?” and “*”. If xxx contains a space, enclose xxx in quotes: “xx x”	

Prefix **-grep=xxx** is equivalent to **-regex=“*xxx*”**, but faster.

Examples

For example, if you have a recursive function called **Try()** you can obtain a stack display containing only **Try()**, by entering the following:

```
-grep=Try stack
```

To see only **long long** global variables, enter the following:

```
-grep="long long" globals
```

Prefix **-nowait**

Many debugger commands execute then wait for the processor to stop before returning control to you. These commands are generally commands that change the processor’s run state or have the ability to do so indirectly. Here is a list of such commands as of this writing:

attach	call	go	evaluate	isi	iso	kill	load
quit	restart	return	run	ssi	sso	stop	

You can prevent a command from waiting by prefixing it with **-nowait**, for example:

```
-nowait run
```

This prefix is most useful for hardware-testing scripts, when you wish to start the target running and execute further commands while it is running. For an example of such use, see [Halting the Instruction-Set Simulator](#) on page 207.

When you use **-nowait** before a command, the debugger reminds you that waiting is suppressed and you can wait for execution to stop via the command **wait**:

Wait suppressed following ‘run’; you can use command ‘wait’.

You can silence this message by using **-nowaitq** (quiet).

Using **-nowait** has no effect when you are debugging on the simulator.

Example

This example initiates execution, sleeps one second, and then stimulates the target and waits:

```
-nowait run
mem 0x1000 // Look at memory (on ARC, possible while running).
sleep 1000 // Sleep 1 second.
<macro_call> // Generate an interrupt using a macro; see
// Using Commands to Generate Interrupts on page 207.
wait // Now wait for execution to stop.
```

Using Command Modifiers

Modifiers modify commands by doing such things as making the command repeatable, or controlling the format of information displayed.

A modifier consists of a slash character (/) followed by a string of one or more letters like the following:

/string

The following modifiers work with several MetaWare debugger commands:

- the repeat modifier (/r)
- the limiting modifier (/n)

Other modifiers work primarily with the [memory](#) command to control the format of the dumped memory. For information about these format modifiers, see the [memory](#) command in [Commands Reference](#) on page 231.

Repeat Modifier /r

Modifier /r makes a command repeatable; that is, once you have entered /r followed by a command, you can repeat the command until the next reset by pressing ENTER. For example, the following command allows you to perform three [isi](#) 5 commands by pressing [Enter] twice:

```
/r isi 5
```

Modifier /r is most useful for the extended forms of the stepping commands [isi](#), [iso](#), [ssi](#), [sso](#).

NOTE The stepping commands in simple form are already repeatable.

Including /r in a macro heading makes the macro repeatable:

```
/r macro sievx
```

See [Creating and Using Debugger Macros](#) on page 224 for information about writing your own debugger macros.

Limiting Modifier /n

Modifier /n^{num} controls the number of lines of output a command displays. For example, this command prints six lines of memory dump starting at function [max\(\)](#):

```
/n6 memory max
```

Modifier /n works with commands that display multiple lines of output:

[break](#) [globals](#) [memory](#) [stack](#)
[disassemble](#) [locals](#) [register](#) [watch](#)

Entering Commands at the Debugger Command Prompt

Type a command at the mdb> prompt and press ENTER. The debugger prints the result of the command to standard output.

Entering Commands in the GUI Command Field

In the debugger GUI, you can use MetaWare debugger command alternatives for most stepping and window-option toolbar operations.

Type a command in the **Command:** input field in the debugger GUI and press ENTER. The output of the command, if any, appears in the **Debug console**. To repeat a previously entered command, select it from the drop-down list.



Figure 45 Command: Input Field

You can combine entering commands in the **Command:** input field with using the debugger's windows, menus, and toolbar buttons. For example, if you set a breakpoint by entering a [break address](#) command in a **Command:** input field, that breakpoint is visible in the **Breakpoints** window and can be disabled, enabled, or removed with the controls in the **Breakpoints** window.

Entering Commands in a Script File

This section discusses the following topics:

- [Reading a Script File](#)
- [Converting HMSL to Debugger Commands](#)

Reading a Script File

You can use the [read](#) command to execute a sequence of commands in a script file. Because debugger commands work with any valid C/C++ expression, including function calls into the program being debugged, you can use them like a scripting language.

Tip The [system](#) command allows you to issue commands to the host operating system. This allows you to edit a command script and return to the debugger when you exit the editor. On Windows, you can use the [system](#) command as follows:

system *filename.ext*

to open *filename.ext* with the application associated on your system with files of extension *.ext*, as long as *filename.ext* is in the directory where you launched the debugger. You can use this shortcut to quickly launch your favorite editor to edit a script file.

The file is treated like a macro with no argument names, or with arguments if it contains \$*n* argument references. For example:

```
read cmdfile.txt arg1 arg2 ...
```

Within *cmdfile.txt*, argument references \$1, \$2, and so on expand to *arg1*, *arg2*, and so on when the commands in *cmdfile* execute.

Example

The following example automatically processes the sequence of MetaWare debugger commands in command script file *cmdfile.txt*:

```
read cmdfile.txt
```

Example with Arguments

In this example, we assume a command script file called *cmdfile.txt* containing the following:

```
fillmem $1 $2 $3
memory $2
```

This script uses the [fillmem](#) and [memory](#) commands to fill memory with the argument referenced by \$1 beginning at address \$2 and continuing for \$3 bytes, then displays memory starting at \$2. Executing the file with the following arguments fills the specified memory, displays it, and returns control to the debugger command prompt:

```
mdb> read cmdfile.txt 0 0x0 512
0x200 bytes filled.
0000: 00000000 00000000 00000000 00000000 .....|.....
0010: 00000000 00000000 00000000 00000000 .....|.....
0020: 00000000 00000000 00000000 00000000 .....|.....
0030: 00000000 00000000 00000000 00000000 .....|.....
0040: 00000000 00000000 00000000 00000000 .....|.....
0050: 00000000 00000000 00000000 00000000 .....|.....
0060: 00000000 00000000 00000000 00000000 .....|.....
0070: 00000000 00000000 00000000 00000000 .....|.....
0080: 00000000 00000000 00000000 00000000 .....|.....
0090: 00000000 00000000 00000000 00000000 .....|.....
```

mdb>

Converting HMSL to Debugger Commands

If you are familiar with the ARC Host Module Scripting Language (HMSL), you can use debugger commands to accomplish many of the same tasks. See the *HMSL Programmer's Guide* for information on converting HMSL to debugger commands.

TIP You can use command prefix [nowait](#) in your scripts to start the target running and execute further commands while it is running without waiting for the debugger to stop execution and return control to you.

Using Debugger Variables

Commands may declare and reference debugger variables and reference built-in debugger variables.

Declaring Debugger Variables

You use the [variable](#) command to declare a debugger variable. The declaration may include an optional initialization.

Example 11 Declaring a Debugger Variable with an Initial Value

```
variable A = initial_value;
```

Example 12 Declaring a Debugger Variable and Assigning a Value Later

```
variable X;
...
X = initial_value;
```

NOTE The initial value may not include '=' or ','.

CAUTION Take care that a variable name does not conflict with the name of an entity in the application you are debugging. The expression evaluator gives precedence to the latter.

Any debugger command beginning with a declared variable is assumed to be prefixed with [evalq](#), a “quiet” evaluation that does not print its results.

Example 13 Commands Beginning with Declared Variables Assume evalq

```
variable X;
...
X++          // 1. A quiet evaluation.
evalq X++;   // 2. Same as 1.
evaluate X++; // 3. Evaluate and then print the answer.
```

Using Built-in Debugger Variables

You can also use variables starting with a '\$' without declaring them; they are "built-in." Generally you can use these variables without fear of conflicting with an entity in your program. Variables starting with a '\$' must be prefixed with [evaluate](#) or [evalq](#).

Example 14 Using a Built-in Debugger Variable

```
evalq $x = 4
evaluate ++$x
int (++$x) = 5
```

Arrays

The decimal value of a declared variable V is substituted when you write \$V. This allows simulated arrays. The following example uses the [while](#) command and a macro to fill such an array.

Example 15 Simulating an Array with Debugger Variables**Input**

```
variable index, m;
variable A0 = 1, A1 = 2, A2 = 3, A3 = 4;
```

Output

```
debugger int (A0 = 1) = 1
debugger int (A1 = 2) = 2
debugger int (A2 = 3) = 3
debugger int (A3 = 4) = 4
```

Input

```
macro callee p
    m = $p // set m to the value of the parameter, so m=0,1,2,3
    m = A$m // Now substitute $m with the actual value of m.
    printf("A[%d] has the value %d\n", $p, m);
    endm

index = 0;
while (index < 4) { callee index ; index++; }
```

Output

```
A[0] has the value 1
A[1] has the value 2
A[2] has the value 3
A[3] has the value 4
```

Another Array Example

You can use a macro to declare the variables, as shown in [Example 16 Simulated Array Declared in a Macro](#).

Example 16 Simulated Array Declared in a Macro

```
macro declare_ap num
    variable ap_match_value$num, ap_match_mask$num, ap_ctrl_value$num
    variable ap_hit$num, ap_mask$num = 1 << $num
endm
```

Here, ap_match_valueN is declared and initialized, as are several other variables. You can then call declare_ap with different values for your actionpoint variables, and reset them as shown in Example 17

Example 17 Macro to Reset Values in an Array

```
//This example assumes the variables on the right are defined.  
macro reset_values which  
    evaluate ap_match_value$which = ZERO;  
    evaluate ap_match_mask$which = AC_MASK_FULL;  
    evaluate ap_ctrl_value$which = AC_RESET_VALUE;  
    evaluate ap_hit$which      = 0;  
endm
```

Assuming the variables on the right are defined, calling macro `reset_values` with 0 for `which` produces the following:

```
ap_match_value0 = ZERO;  
ap_match_mask0 = AC_MASK_FULL;  
ap_ctrl_value0 = AC_RESET_VALUE;  
ap_hit0 = 0;
```

Related Topics

- [Creating and Using Debugger Macros](#) on page 224.
- [Example 47 Using a Macro with an Array to Start Extension Timers](#) on page 227

Printing to stdout

Debugger commands provide two ways of printing: unformatted and formatted.

- [Unformatted Printing](#)
- [Formatted Printing](#)

Unformatted Printing

The `print` command prints its argument on a new line. The command-line debugger prints to `stdout`; the GUI debugger prints to the **Command** window.

Having the debugger say “Hello, World!” is as simple as the following:

```
> print Hello, World!  
Hello, World!
```

Formatted Printing

For formatted printing, the command is the same as the C `printf()` function, including a format string with format specifiers. Each argument is a string literal or an expression that can be converted to an integer. For example:

```
printf("Match value: 0x%x\n",ap_match_value | ap_match_mask);
```

NOTE Be sure to include `\n` where appropriate, as it is not supplied.

Creating while Loops

The `while` command executes a debugger command while an expression evaluates to true (non-zero).

TIP If you want to execute many commands in a while loop, consider placing the commands in a macro:

```
while (condition) macro
```

For more information on using macros, see [Creating and Using Debugger Macros](#) on page 224.

Example 18 Using the while command

```
evaluate r14
machine register int r14 = 4
while (r14 > 0) evaluate r14--
int (r14--) = 4
int (r14--) = 3
int (r14--) = 2
int (r14--) = 1
evaluate r14
machine register int r14 = 0
```

Analysis

[Example 18](#) shows the debugger commands and output for a sequence that does the following:

1. Displays the value of r14 using [evaluate](#).
2. Decrement r14 using [evaluate](#) while r14 is greater than zero.
3. Uses [evaluate](#) again to display the result.

Example 19 Script Using the while Command

```
evaluate $h = 0
while ($h < 4) {evalq $h++; ssi}
evaluate $h
```

Analysis

[Example 19](#) does the following:

1. Uses [evaluate](#) to initialize built-in debugger variable \$h to 0.
2. Uses a compound command while \$h is less than 4 to do the following:
 - a. Increment \$h without display using [evalq](#) \$h++
 - b. Execute an [ssi](#) command.
3. Evaluates \$h.

Invoked using [read](#) on the command line, the script prints output such as the following, depending on the application being debugged:

```
debugger int ($h = 0) = 0
queens.c!29:      if (Asub(J) && Bsub(I+J) && Csub(I-J)) {
queens.c!42+0xe:      else *Q = True;
queens.c!45:      while (!(*Q || J==8));
queens.c!28:      J++; *Q = False;
debugger int $h = 4
```

Executing Commands Conditionally

The debugger provides a set of [if](#) and [if ... endif](#) commands for conditional execution of other commands. This section discusses using the conditional commands.

- [Using if](#)
- [Using if, then, else, elseif, and endif](#)

Using if

The debugger provides the [if](#) command for conditional execution of the subsequent command.

Syntax: **if** (*expression*) *command*

command is executed if *expression* evaluates to non-zero. No “else” command is provided, but “else” options are available using the **if...endif** syntax. See [Using if, then, else, elseif, and endif](#) on page 192.

Example 20 Using the if Command

```
register r10
r10 00000000
if (r10 == 0) evaluate r10++
int (r10++) = 0
register r10
r10 00000001
```

Analysis

[Example 20](#) shows the debugger commands and output for a sequence that does the following:

1. Uses **register** to display the value of r10.
2. Uses **evaluate** to increment the value of r10 if the value of r10 is zero.
3. Uses **register** again to display the value of r10.

Example 21 Script Using the if Command

```
register r11
if (r11 == 0) evaluate r11++
register r11
```

Analysis

[Example 21](#) does the following:

1. Uses **register** to display the value of register r11.
2. Uses **evaluate** to increment the value of register r11 if the value is currently 0.
3. Uses **register** again to display the value of register r11.

Invoked using **read** on the command line, the script prints the following if the initial value of register r11 is 0:

```
r11 00000000
int (r11++) = 0
r11 00000001
```

If the initial value of register r11 is not 0, the script displays the value of register r11 twice, for example:

```
r11 00000001
r11 00000001
```

Using if, then, else, elseif, and endif

The **if ... endif** set of commands provides the ability to nest conditional commands using **if**, **then**, **else**, and **elseif**, followed by **endif**. The **if ... endif** set of commands must begin with **if** and end with **endif**; the other commands are optional. See the listing at [if...endif on page 246](#) for detailed syntax.

Example Script

```
register r11
if (r11 == 0) then
    evaluate r11++
    register r11
else isi
endif
```

Analysis

This example script does the following:

1. Displays the value of register `r11`.
2. If the value of register `r11` is currently 0, increments the value, and displays the value.
3. If the value of register `r11` is not 0, executes an `isi` command.

Invoked using `read` on the command line, the script prints the following if the initial value of register `r11` is 0:

```
r11 00000000
int (r11++) = 0
r11 00000001
```

Because value of register `r11` is no longer 0, subsequent invocations of this script might print the following, depending on the application being debugged:

```
r11 00000001
100e8 _exit_halt flag 1
```

Reading and Modifying Processor Data

This section discusses using commands and scripts to read and write data to and from the processor.

- [Reading Registers](#)
- [Writing Registers](#)
- [Reading Memory](#)
- [Writing Memory](#)

Reading Registers

The basic commands for reading processor registers are the `register` and `aux` commands, but you can read registers in several ways:

- `register` displays the contents of all registers, a register entered as an argument, or the registers specified in a regular expression entered as an argument.
- `evaluate` with a register as an argument displays the value of the register.

CAUTION The debugger abbreviates some register names. If the debugger reports that the register you want to evaluate is an undeclared identifier, execute a `register` command with no argument to see the debugger names for all registers.

- `aux` displays the contents of all auxiliary registers or the auxiliary registers starting at an auxiliary register entered as an argument.
- Debugger-intrinsic function `$reg()` reads core registers, and `$bank_reg()` reads auxiliary registers; for function prototypes, see [Debugger-Intrinsic Functions](#) on page 223.

The debugger does not provide the ability to access individual bits of most registers; see [Reading Special Registers in Detail](#) on page 194 for exceptions.

Example

The following example uses the `register` and `aux` commands together (note semicolon and braces) to display the content of core register `r10` followed by a subset of auxiliary registers beginning with auxiliary register 3:

```
mdb> {register r10 ; aux 3}
r10 00000079
aux0003: 000115ec ffffff21 00000000 000101dc
aux0007: 00000000 00000000 00000000 00000001
...
aux0037: 00000000 00000000 00000000 00000000
aux003b: 00000000 00000000 00000000 00000000
```

Reading Special Registers in Detail

The debugger provides detailed display of specific registers using the [display](#) command. This section discusses the special command-line register displays currently available. Note that you can enter [display](#) without an argument for a list of available command-line displays.

display mmu_contents — Display ARC 700 MMU contents

If you have specified option [-mmu](#) (and [-a7](#)), you can use this command to display the contents of the MMU on the command line.

display exmmu — Display ARC 700 Exception and MMU registers

If you have specified option [-mmu](#) (and [-a7](#)), you can use this command to display the contents of the exception and MMU registers on the command line.

display mpu — Display ARC 600 MPU registers

If you have specified option [-mpu](#) (and [-a6](#)), you can use this command to display the contents of the MPU registers on the command line.

display simd|vrN — Display ARC 700 SIMD information

After you have specified the location of your SIMD DLL and started debugging, you can use [display simd](#) to display the SIMD configuration registers, or use [display vrN](#) to display the vector registers in an 8-, 16, or 32-bit view, where *N* is **8**, **16**, or **32**.

If you have more than one SIMD unit, use [display vrN_UN](#), where *UN* is the unit number.

Examples

```
mdb> display simd
0xa3310101 0xfd SIMD_DMA_BUILD SIMD & DMA engine
|-[ 7:0] 0x1      SIMD version
|-[ 15:8] 0x1      DMA version
|-[19:16] 0x1      SDM size -- 2048 locations, 32K
|-[23:20] 0x3      SCM size -- 4096 locations, 20K
|-[27:24] 0x3      SCQ size -- 256 locations, 1280 bytes
|-[30:28] 0x2      VRF size -- 24 registers
|-[ 31] 1          1 = entropy decoder present
0x00000001 0xfc VLC_BUILD VLC extensions
|-[ 7:0] 0x1      Version
|-[11:8] 0x0      FIFO depth -- 4 x 32 bits
|-[ 12] 0          1 = mpeg 1,2,4 Huffman tables are supported
|-[ 13] 0          1 = VC-1 Huffman tables are supported
|-[ 14] 0          1 = CAVLC H.264 baseline supported
|-[ 15] 0          1 = CABAC H.264 main profile supported
|-[30:23] 0x0      VLC DMA version
|-[ 31] 0          1 = DMA acceleration present
0x00000000 0xA1 VE_CTRL           Control
|-[ 31] 0          RST reset vector engine
|-[ 18] 0          PDO pause DMA outward channel
|-[ 17] 0          PDI pause DMA inward channel
|-[ 16] 0          PVE pause vector engine
|-[ 7:5] 0x0        SWAP mode: No address swapping
----->ENTER for more, `x' ENTER to stop...x
```

Vector-Register Example from the H.264 Decoder

```
mdb> display vr16
      7   6   5   4   3   2   1   0
vr00: 0000 0000 0b2c 0a8c 0000 0000 0930 0000
vr01: 0020 0020 0020 0020 0020 0020 0020 0020
vr02: 0002 0002 0002 0002 0002 0002 0002 0002
vr03: 00c4 00b5 00a1 0051 0001 0000 0004 0011
vr04: 0000 0000 0000 0000 c4b5 a151 0100 0411
vr05: 0000 0000 0000 0000 c4b5 a151 0100 0411
vr06: 0000 0000 0000 0000 0000 0000 c4b5 a151
vr07: 0000 0000 0000 0000 0000 0000 c4b5 a151
vr08: ffff ffff ffff ffff ffff ffff ffff ffff
vr09: ffff ffff ffff ffff ffff ffff ffff ffff
vr10: 0000 0000 0000 0000 f1ec e1de fafd ffff
vr11: 0000 0000 0000 0000 f1ec e1de fafd ffff
vr12: 0000 0000 0000 0000 0000 0000 0000 0000
vr13: 0000 0000 0000 0000 0000 0000 0000 0000
vr14: 00f1 00ec 00e1 00de 00fa 00fd 00ff 00ff
vr15: 00f1 00ec 00e1 00de 00fa 00fd 00ff 00ff
vr16: 0000 0000 0000 0000 0000 0000 0000 0000
vr17: 00f1 00ec 00e1 00de 00fa 00fd 00ff 00ff
vr18: 00f1 00ec 00e1 00de 00fa 00fd 00ff 00ff
vr19: 0004 0004 0004 0004 0004 0004 0004 0004
vr20: 0004 0004 0004 0004 0004 0004 0004 0004
vr21: fffc fffc fffc fffc fffc fffc fffc fffc
...

```

display timer0|timer1

After you specify ARC extension timer 0 or 1 using the applicable [-X*](#) option or in the GUI, you can use this command to display the timer registers on the command line.

Parsing the Data Obtained from Reading Registers

The following interactive example reads a register, parses the data, and takes action conditionally:

Example 22 Finding the Stack Location at a Conditional Breakpoint

```
> evaluate $low_water_mark=0xFFFFFFFF
debugger int ($low_water_mark=0xFFFFFFFF) = 268435455 (0xffffffff)
> macro stk_chk
Enter contents for macro stk_chk:
> if (sp < $low_water_mark) evalq $low_water_mark=sp
> endm
End of macro stk_chk.
> break Try, evaluate (0), exec stk_chk
Breakpoint 1 set.
1:    0x010118 condition:[expn:(0)]
                  Try          push_s      %blink
> run
go
1 5 8 6 3 7 2 4
Execution stopped.
No source information at 0x10108.
  10108 _exit_halt+0x04  nop
> evaluate _fstack
_fstack() = 0x17000 (_fstack)
> evaluate $low_water_mark
debugger int $low_water_mark = 96980 (0x17ad4)
```

Analysis

[Example 22](#) uses the queens demo application to do the following:

1. Uses [evaluate](#) to set built-in debugger variable \$low_water_mark to 0xFFFFFFFF.
2. Defines a macro `stk_chk` to set \$low_water_mark to the value of the stack-pointer register `sp` if the current value of `sp` is less than the current value of \$low_water_mark.
3. Sets a conditional breakpoint on `Try()` using the `eval` and `exec` options to execute macro `stk_chk` and set the breakpoint only if the result is 0. Note that `b` is an accepted abbreviation for the [break](#) command.
4. Runs `queens`.
5. Evaluates `_fstack` and \$low_water_mark to determine the minimal stack location at the conditional breakpoint.

Parsing the Data Obtained from Reading Auxiliary Registers

Example 23 defines and calls a macro to check the data-cache control register and print a message stating whether or not it is enabled.

Example 23 Checking the Status of the Data-Cache Control Register

```
mdb> prop use_aux=0x48,dcctl
mdb> macro dc_chk
Enter contents for macro dc_chk:
mdb> if (dcctl & 1) then print Data cache is disabled.
mdb> else print Data cache enabled.
mdb> endif
mdb> endm
End of macro dc_chk.
mdb> dc_chk
Data cache is enabled.
```

Analysis

This example does the following:

1. Uses [prop use_aux](#) to enable auxiliary register 0x48 and assign it the name `dcctl`.
2. Creates a macro to test the enable bit of `dcctl` and print a message depending on the result.
3. Invokes the macro and receives the message that `dcctl` is enabled.

Macros can call script files, and script files can define and invoke macros.

Related Topics

- [Entering Commands in a Script File](#) on page 187
- [Creating and Using Debugger Macros](#) on page 224

Writing Registers

This section discusses ways you can write registers using the debugger.

- [Writing Core Registers](#)
- [Writing Auxiliary Registers](#)

Writing Core Registers

NOTE The debugger handles starting and stopping the processor, so you normally do not need to write registers directly to do so.

Command [evaluate](#) evaluates any valid C or C++ expression and displays the result. When you enter a core register as part of the expression, the debugger changes the value of the register as directed and displays the result.

Note Command **evaluate** can also evaluate expressions containing debugger-intrinsic functions, including **\$put_reg()**, which you can use to put a value in a register; see [Debugger-Intrinsic Functions](#) on page 223.

Command [evalq](#) functions like **evaluate** but does not show the results. This form of “quiet” evaluation is useful for assignments in scripts.

TIP You can make any auxiliary register accessible to read and modify using [evaluate](#) by using **prop use_aux=N, name**, where *N* is the auxiliary-register number and *name* is the auxiliary-register name.

Example 24 Using the evaluate and evalq Commands on a Register

```
evaluate r10
machine register int r10 = 123 (0x7b)
evalq r10++
evaluate r10
machine register int r10 = 124 (0x7c)
```

Analysis

[Example 24](#) includes both commands and debugger responses, and does the following:

1. Uses **evaluate r10** to display the current value of register **r10**.
2. Uses **evalq** to increment the value of register **r10**.
3. Uses **evaluate** again to display the new value of register **r10**.

Example 25 Using the evaluate Command to Assign a Value to a Register

```
evaluate r10=25
machine register int (r10=25) = 25 (0x19)
evaluate r10
machine register int r10 = 25 (0x19)
```

Analysis

[Example 25](#) includes both commands and debugger responses, and does the following:

1. Uses **evaluate** to assign the value 25 to **r10**.
2. Uses **evaluate** again to display the new value of **r10**.

Related Topics

- [Halting by Writing to a Register](#) on page 206
- [Hardware Interrupts](#) on page 207

Writing Auxiliary Registers

The debugger offers several means to write auxiliary registers.

Built-in Auxiliary Registers

To change the value of auxiliary registers, use the [evaluate](#) command with the debugger-intrinsic function **\$put_bank_reg()** as follows:

```
evaluate $put_bank_reg(1,aux,value)
```

Where *aux* is the auxiliary-register number and *value* is the new value.

Example

```
mdb> aux 3
aux0003: 000115ec ffffff21 00000000 000101dc
aux0007: 00000000 00000000 00000000 00000001
aux000b: 00000000 00000000 00000000 00000000
...
mdb> evaluate $put_bank_reg(1,3,0)
debugger int $put_bank_reg(1,3,0) = 0
mdb> aux 3
aux0003: 00000000 ffffff21 00000000 000101dc
aux0007: 00000000 00000000 00000000 00000001
aux000b: 00000000 00000000 00000000 00000000
...
```

Analysis

This example shows the debugger commands and output for a sequence that does the following:

1. Uses **aux 3** to display the values of a subset of auxiliary registers starting with aux0003 (truncated).
2. Uses **evaluate \$put_bank_reg(1,3,0)** to change the value of aux0003.
3. Uses **aux 3** to display the new values of a subset of auxiliary registers starting with aux0003 (truncated).

You can make any auxiliary register accessible to read and modify using [evaluate](#) by using the [prop](#) command (or the **-prop=** option when launching the debugger):

```
prop use_aux=N,name
```

Where *N* is the auxiliary register number and *name* is the auxiliary register name.

Example

```
mdb> prop use_aux=0000,aux0000
```

This example makes aux0000 accessible to read and modify using [evaluate](#) as described in [Writing Core Registers](#) on page 196.

Writing to Your Own Auxiliary Registers

When you add an extension auxiliary register using the compiler or assembler mechanism, you can view and modify the auxiliary register in the **Registers** window and on the command line using [evaluate](#) as described in [Writing Core Registers](#) on page 196.

Related Topics

- [Evaluating Expressions](#) on page 78
- [Example 47 Using a Macro with an Array to Start Extension Timers](#) on page 227
- *MetaWare C/C++ Programmer's Guide for ARC*
- *ELF Assembler User's Guide for ARC*
- [Debugger-Intrinsic Functions](#) on page 223

Reading Memory

The debugger provides the **memory** command for dumping memory beginning at a specific address, and debugger-intrinsic functions for reading a single **byte** or **short** from an address. On ARC processors, you can access memory while a process is running.

- The **[memory](#)** command dumps 160 bytes of memory, starting at the address you specify and displayed in the format, data type, radix, and number of lines you specify.

The *address* argument itself can be an address, a function a register, a regular expression, a C/C++ expression, or a source-code line number. See the listing for **[memory](#)** in [Alphabetical Listing of Commands](#) on page 232 for details.

```
> memory main
main:
0101c4: c5e1c0f1 b6c81cfcc 3700230a ce00c1a1 .....|.#.7....
0101d4: 3f812200 00040000 00000882 0d17dd01 ."?.?....|.....
0101e4: d9011253 f00c17cc 18ff6598 e5018042 S.....|.e..B...
0101f4: dd02f1f7 14530d15 17bcd801 6599f00c .....S.|.....e
010204: 800219fe f1f8e501 1e7f258a 12130d13 .....|.%.....
010214: 17a8d801 6599f00c e501a907 2342f1f8 .....e|.....B#
010224: d8013041 13fffffb6 e811b080 0d1fdd01 A0.....|.....
010234: ce021253 793b79bb f00c1784 11fc6199 S....y;y|.....a..
010244: e5018001 00000816 ce03f1f3 0000080e .....|.....
010254: 09ced800 78e00080 0f3ec0f1 c5e10240 .....x|..>.@...
```

- Debugger-intrinsic function **\$read_byte()** reads a byte from a memory address using a one-byte-wide access to memory if the memory interface supports one-byte-wide access.

```
> evaluate $read_byte(0x1000)
unsigned char $read_byte(0x1000) = '\340' (224=0xe0)
```

- Debugger-intrinsic function **\$read_short()** reads a short from a memory address using a two-byte-wide access to memory if the memory interface supports two-byte-wide access.

```
> evaluate $read_short(0x1000)
unsigned short $read_short(0x1000) = 31200 (0x79e0)
```

Related Topics

- [Debugger-Intrinsic Functions](#) on page 223

Writing Memory

The debugger provides commands and intrinsic functions for writing to memory. On ARC processors, you can access memory while a process is running.

- [memset](#)** is like the **memset()** function in C, setting memory beginning at a certain address to a certain value for a certain number of bytes:

```
> memset 0x0 0 160 // zero memory from 0x0 for 160 bytes
Flushing and invalidating data cache.
memset 0xa0 bytes total.
> memory 0x0 // memory command to show the results from 0x0
0000: 00000000 00000000 00000000 00000000 .....|.....
0010: 00000000 00000000 00000000 00000000 .....|.....
0020: 00000000 00000000 00000000 00000000 .....|.....
0030: 00000000 00000000 00000000 00000000 .....|.....
0040: 00000000 00000000 00000000 00000000 .....|.....
0050: 00000000 00000000 00000000 00000000 .....|.....
0060: 00000000 00000000 00000000 00000000 .....|.....
0070: 00000000 00000000 00000000 00000000 .....|.....
0080: 00000000 00000000 00000000 00000000 .....|.....
0090: 00000000 00000000 00000000 00000000 .....|.....
```

- [emem](#)** enters an address with a value, then shows memory from that address for 160 bytes:

```
> emem 0x0 11111111
0000: 00a98ac7 00000000 00000000 00000000 .....|.....
0010: 00000000 00000000 00000000 00000000 .....|.....
```

```
0020: 00000000 00000000 00000000 00000000 .....|.....
0030: 00000000 00000000 00000000 00000000 .....|.....
0040: 00000000 00000000 00000000 00000000 .....|.....
0050: 00000000 00000000 00000000 00000000 .....|.....
0060: 00000000 00000000 00000000 00000000 .....|.....
0070: 00000000 00000000 00000000 00000000 .....|.....
0080: 00000000 00000000 00000000 00000000 .....|.....
0090: 00000000 00000000 00000000 00000000 .....|.....
```

- Debugger-intrinsic function `$write_byte()` writes an address with a `byte` using a one-byte-wide access to memory if the memory interface supports one-byte-wide access.
- Debugger-intrinsic function `$write_short()` writes an address with a `short` using a two-byte-wide access to memory if the memory interface supports two-byte-wide access.

Related Topics

- [Debugger-Intrinsic Functions](#) on page 223

Using Breakpoints and Watchpoints

You can use commands to set breakpoints and watchpoints to control the application and processor you are debugging. For information on using breakpoints and watchpoints in the GUI, see [Debugging with Breakpoints](#) on page 60 and [Debugging with Watchpoints](#) on page 66.

Using Breakpoints

A breakpoint stops execution of a process. If you step over a function that contains a breakpoint, execution stops on the breakpoint, even if this means stepping into the function.

NOTE To set a breakpoint, you must first compile with debugging information (option `-g`).

You can set a breakpoint:

- on a line in a source file
- on a single disassembled machine instruction
- at an address
- on a function
- at the current location of the program counter

Breakpoints can be temporary, counted, conditional, command-execution, or task-specific, or mixture thereof. For syntax information on using the `break` command with different arguments to achieve various results, see the listing in [Alphabetical Listing of Commands](#) on page 232.

You use the `delete break` command to delete a single breakpoint or all current breakpoints. See the listing in [Alphabetical Listing of Commands](#) on page 232 for details.

You can use the `disable break` command to disable some or all currently enabled breakpoints without deleting them, or the `enable break` command to enable some or all of the currently disabled breakpoints. See the listings in [Alphabetical Listing of Commands](#) on page 232 for details.

Examples

This example displays current breakpoints:

```
break
```

This example sets a breakpoint at line 55 in source-code file `my_file.c`:

```
break my_file.c!55
```

This example sets a breakpoint on the machine instruction at hexadecimal address 0804931c:

```
break 0x0804931c
```

This example sets a breakpoint at offset 0xd8 in function **main()** that becomes enabled when **done==1** evaluates to true:

```
break main+0xd8, evaluate done==1
```

This example sets a breakpoint at line 371 in the current source file; control stops on the breakpoint after three passes:

```
break !371, 3
```

This example sets a breakpoint on function **myfunc()** if the thread ID is 50 or is odd:

```
break myfunc, evaluate $tid == 50 || ($tid & 1) == 1
```

Using Watchpoints

Watchpoints monitor a region of memory and pause execution if any changes occur in the contents of that region.

Use watchpoints to find statements that unexpectedly change your program's code or data (for example, an array index that addresses elements beyond the array's allocated space). If the value of a variable, array, structure, or other data item in your program is changing unexpectedly, you can set a watchpoint on the region of memory containing that data item, then step through your program. When a watchpoint's value changes, the program stops.

You set watchpoints (except register watchpoints) using the [watch](#) command. See the listing in [Alphabetical Listing of Commands](#) on page 232 for the syntax to use for setting read and write (or read/write) watchpoints on addresses, and expressions, including the special arguments for ARC targets. See the listing for the [watchreg](#) command for information on setting watchpoints on registers.

You can use the [delete watch](#) command to delete either a single watchpoint or all current watchpoints. See the listing in [Alphabetical Listing of Commands](#) on page 232 for details.

You can use the [disable watch](#) command to disable some or all currently enabled watchpoints, and the [enable watch](#) command to enable some or all currently disabled watchpoints. See the listings in [Alphabetical Listing of Commands](#) on page 232 for details.

Examples

The following example displays current watchpoints:

```
watch
```

The following example sets a watchpoint on local variable X:

```
w X
```

Set a watchpoint on 16 bytes of memory, starting at hexadecimal address 0x0001023c:

```
watch 0x0001023c, 16
```

Set a watchpoint for read operations on local variable X:

```
w X, mode r
```

Set a watchpoint for read operations on local variable X if the current thread is thread 1:

```
w X, mode r, thread 1
```

or

```
w X, $tid == 1, mode r
```

Prepare a list of watchpoints for re-entry:

```
watch X[3]
Watchpoint 1 set (hardware) data-write.
watch, list
watch X[3]
watch, list input
watch X[3]
watch, list addr
watch 0x170d4, len 4
```

Example 26 Using Special Watch Arguments for ARC Targets

The examples in this section assume the following invocation:

```
mdb queens.ac -c1 -a6 -on=a6_ap_sim
```

With a simple **watch**, any write to X[3] stops the program:

```
mdb> watch X[3]
Watchpoint 1 set (hardware) data-write.
mdb> run
go
queens.c!32: Asub(J) = False;

Watchpoints:
C 1: H X[3] [hit:1 time] data-write (last hit value: 0x170d4)
C value: 0 => 2
Execution stopped at watchpoint 1.
```

When you are not using **value v** and **mask k** (ARC 600 and ARC 700 only), **invert** inverts the address range:

```
mdb> watch 0x1000, 0x1000, invert
```

The program stops on any reference to memory outside of 0x1000 for 0x1000.

When you use **mask k** and **value v**, without **invert**, the processor does not stop until the data **value & 4 == 4**.

```
mdb> watch X[3], mask 4, value 4
Watchpoint 1 set (hardware) data-write.
mdb> run
go
queens.c!32: Asub(J) = False;

Watchpoints:
C 1: H X[3] [hit:1 time] data-write value: 0x4 mask: 0x4 (last hit value: 0
x170d4) (data hit value: 0x7)
C value: 0 => 7
```

In this case, the store of 2 is ignored, and the program does not stop until 7 is stored, as $7 \& 4 == 4$.

If you invert the condition, the program stops when the data **value & 4 != 4**:

```
mdb> watch X[3], mask 4, value 4, invert
Watchpoint 1 set (hardware) data-write-inverted.
mdb> run
go
queens.c!32: Asub(J) = False;

Watchpoints:
C 1: H X[3] [hit:1 time] data-write-inverted value: 0x4 mask: 0x4 (last hit
```

```
value: 0x170d4) (data hit value: 0x2)
C value: 0 => 2
```

Execution stopped at watchpoint 1.

Processor Status and Run Control

The debugger handles starting and stopping the processor, so you normally do not need to write registers directly to do so; you simply issue the [run](#) command or the various stepping commands ([isi](#), [iso](#), [ssi](#), [ssq](#)). The debugger does not return control to you until the processor stops (unless you prefix commands with [-nowait](#)).

CAUTION Normally, the debugger controls the target processor. If you modify the run state, the debugger might lose track of the process.

Checking Processor Status

Generally, the debugger polls processor status and informs you through of the status of the application being debugged as you run or step. You can extrapolate processor status from the status of the application being debugged.

You can also use the [status](#) command to show status of the application being debugged at any time.

Example 27 Status Messages During Debugging

Can't display registers at this time (process probably running).

```
...
Process stopped.
Execution stopped.
...
Process is not executing.
...
queens: Process terminated
...
No process; will try to restart.
queens: Process created
...
Execution stopped at breakpoint.
...
mdb> -nowait run // Don't wait for completion to accept commands
Wait suppressed following `run'; you can use command `wait'.
mdb> evaluate stat32 // Trying to read status; processor running
ERROR: stat32 is an undeclared identifier.
Execution stopped.
No source information at 0x100f8.
    100f8 _exit_halt+0x10  b          _exit_halt
mdb> evaluate stat32 // After execution stops, debugger can read
machine register int stat32 = 1
```

Manually Testing for a Running Processor

You can manually test for a running processor by taking an action such as trying to write to that `status32` register and evaluating the result. You can use commands in a script or macro to test, as shown in [Example 28 Script for Testing Processor Running or Stopped](#):

Example 28 Script for Testing Processor Running or Stopped

```
if (stat32 & 1) then {evalq *((int *) 0x60000)=4 ; print Processor is stopped}
else print Processor is running //We know because we can't write
endif
```

Analysis

This example tests the halt bit (bit 0 in the `status32` register) by **ANDing** with 1. If the halt bit is set, it uses **`evalq`** (evaluate without showing results) to cast a constant (0x60000) to a pointer to an **int** and then deaddresses it and gives it a new value of 4.

If the write fails, the script prints `Processor is running` (because the debugger cannot write to a running control register). If the write succeeds, the processor must be stopped, and the script prints `Processor is stopped`.

NOTE The debugger abbreviates `status32` among other register names. If you receive an unexpected error when working with a register name, execute a **`register`** command with no argument to verify the debugger names for all registers.

To test this script, you might need to use a **`-nowait run`** command so that the debugger runs without waiting for the processor to stop before returning control to you. Assuming the commands placed in file `stat.txt` in the directory where the debugger was invoked, the results are as shown in [Example 29](#)

Example 29 Example Runs of Status Script

```
mdb> read stat.txt
Processor is stopped
mdb> -nowait run
Wait suppressed following `run'; you can use command `wait'.
mdb> read stat.txt
Processor is running
```

Macros can call script files, and script files can define and invoke macros.

Finding Processor ID

To find the processor ID, use the following (command and response shown):

```
mdb> register ident
ident ffffff31
```

Resets and Pipe Flushing

The debugger can reset hardware or a simulated processor using the **`restart -reset`** command, but it does not actually detect resets, as it does not have access until reset is complete and the processor is either running or halted.

To flush the processor pipe between tests, you can do one of the following:

- Restart the application being debugged with the **`restart`** command
This method also does a new download of non-read-only portions of the program.
- Ask the debugger to flush the pipe using the following command:
`prop do_flush_pipe=1`

If you use this method, be sure you have removed any conditions (such as action-point settings) that might cause the pipe flush to fail.

Related Topics

- [Entering Commands in a Script File](#) on page 187
- [Creating and Using Debugger Macros](#) on page 224

Terminating a Process

The debugger provides two commands for terminating a process.

- The [stop](#) command terminates (halts) the current process, leaving its debug information in the debugger.
- The [kill](#) command terminates the current process and removes its debug information.

Waiting and Sleeping

This section discusses using debugger commands for waiting and sleeping.

- [Waiting for Commands to Complete](#)
- [Waiting for Processor Events](#)
- [Pausing Using the Sleep Command](#)

Waiting for Commands to Complete

The debugger provides the following command and command prefix:

- [wait](#) (the default)
- [-nowait](#) (prefix commands for run-related commands)

Normally, the debugger waits for commands to complete before returning control to you ([wait](#)). You can change this behavior for run-related commands by preceding the commands with [-nowait](#). Control then returns to the command prompt while the command is executing.

After issuing a [-nowait](#) command, you can use [wait](#) to tell the debugger to wait for execution to stop. See [Example 30 Using -nowait, sleep, and wait](#).

Waiting for Processor Events

An *event* implies a transition, but the debugger does not detect most events on the processor in that sense. You can use actionpoints (breakpoints or watchpoints) to capture some events.

The debugger can detect interrupts or exceptions if you place a breakpoint at the correct vector location and then execute some commands. See the examples in [Using Commands to Generate Interrupts](#) on page 207.

The instruction-set simulator allows watchpoints on most core or auxiliary registers; ARC hardware allows watchpoints to be set on hardware only if actionpoints are included in the processor build.

Related Topics

- [Using Breakpoints and Watchpoints](#) on page 200
- [Using Commands to Generate Interrupts](#) on page 207
- [Example 28 Script for Testing Processor Running or Stopped](#) on page 203.
- [Example 23 Checking the Status of the Data-Cache Control Register](#) on page 196

Pausing Using the Sleep Command

The debugger provides the [sleep](#) command to tell the processor or simulator to pause for the specified number of milliseconds:

`sleep nnn`

Example 30 Using -nowait, sleep, and wait

```
-nowait run
sleep 500      // Allow hardware to run
simulate       // or push simulator.
level_intr 5   // Generate an interrupt from a macro.
wait          // Now wait for execution to stop.
```

Changing the Program Counter

Normally, the debugger controls the target processor. If you modify the run state, the debugger might lose track of the process. To change the program counter for an application being debugged, use

```
evaluate pc=address
```

rather than writing the auxiliary register yourself.

Halting Target Hardware

CAUTION Normally, the debugger controls the target processor. If you modify the run state, the debugger might lose track of the process.

Halting by Writing to a Register

Halting target hardware generally involves writing to a register. Most targets do not allow register or memory access while they are running. On ARC processors, you can access memory, but not registers. If you attempt to access a register while the processor is running, the debugger responds with a message such as the following:

```
Can't display registers at this time (process probably running).
```

You can use a “back-door” method to obtain the value of certain ARC dual-ported auxiliary registers that you know can be read during execution, such as `pc32` and `status32` (the effect of accessing a non-dual-ported register is undefined).

The debugger-intrinsic functions `$bank_reg()` and `$put_bank_reg()` allow access to registers. By writing to `status32` you can halt the ARC after `-nowait run`.

Example 31 Halting Target Hardware

```
> -nowait run
Wait suppressed following `run'; you can use command `wait'.
> evaluate $put_bank_reg(1,10,1)
debugger int $put_bank_reg(1,10,1) = 0
Execution stopped.
No source information at 0xbc.
      bc main          flag          6
```

For the prototype of `$put_bank_reg()`, see [Debugger-Intrinsic Functions](#) on page 223.

Using `evaluate` without the `-nowait` prefix means that after you write `status32`, the debugger waits for the processor to stop before it returns control to you. You can also stop the processor by using the `stop` command.

When you use `-nowait`, hostlink requests are not serviced unless the application being debugged is running on the instruction-set simulator. Use the `status` command to update the debugger's status on the application being debugged; this allows a single hostlink request to be serviced. On hardware, the `simulate` command shows status.

Halting After a Certain Number of Cycles

Using RASCAL, you can force an ARC processor to stop after a certain number of cycles using the `prop` command:

```
prop ras_stop_after=N
```

The next time you start up the processor, RASCAL stops it after N RASCAL cycles. Note that the RASCAL clock `ra_ck` might be slower than the system clock.

The `ras_stop_after` value remains in effect until it is decremented to zero, so it can span multiple commands. If the processor stops before N is counted to 0, RASCAL might stop the processor later when you run it again. To reset the RASCAL countdown timer, use the following command:

```
prop ras_stop_after=0
```

Halting the Instruction-Set Simulator

Unlike hardware, the instruction-set simulator advances only when the debugger is waiting for it to finish. While waiting, the debugger repeatedly queries the ISS state, and each query causes the ISS to advance. If you use `-nowait` on the ISS, hostlink requests can be serviced., but the ISS does not advance. You can advance the ISS using the [simulate](#) command. The [simulate](#) command permits all hostlink requests to the ISS to be serviced.

Using the [simulate](#) command advances execution by the amount specified by option `-instrs_per_pass=NNN` (the default is 512 instructions). Using the [simulate](#) command with hardware has the same effect as [status](#), so you can include it in your test scripts and use a single test script for both hardware and the simulator. For example:

```
-nowait run
sleep 500          // Allow hardware to run.
simulate          // or push simulator.
level_intr 5      // Generate an interrupt from a macro.
wait              // Now wait for execution to stop.
```

Using Commands to Generate Interrupts

This section discusses how to generate hardware and software interrupts using debugger commands.

- [Hardware Interrupts](#)
- [Software Interrupts](#)

Hardware Interrupts

Hardware interrupts are generally triggered by a timer or an external, hardware-related event. To test your timer using the debugger, you can set the timer count to the rollover value and set a breakpoint on the timer-vector address, then run your application until the timer rolls over the value and triggers the interrupt.

TIP To execute a command after hitting the breakpoint, use
`break address, exec command`

Example 32 Triggering an Interrupt Using the Timer (Interactive Session)

```
mdb> evaluate stat= stat | 0xC000000
machine register int (stat= stat | 0xC000000) = 234897478 (0xe004046)
mdb> evaluate tctl0=3
machine register int (tctl0=3) = 3
mdb> evaluate timr0=tlim0
machine register int (timr0=tlim0) = 16777215 (0xffffffff)
mdb> break 0x18
Breakpoint 1 set.
1: 0x000018
00000018 b1_s 0x4
mdb> run
No source information at 0x18.
@ 18 00000018 b1_s 0x4
Execution stopped at breakpoint 1, 0x18.
```

1. Enables interrupts by setting the status register (called `stat` in the debugger) using [evaluate](#).

CAUTION The debugger abbreviates some register names. If the debugger reports that the register you want to evaluate is an undeclared identifier, execute a [register](#) command with no argument to see the debugger names for all registers.

2. Uses [evaluate](#) to set timer 0 control to interrupts enabled and no tick in debug mode (3).
3. Uses [evaluate](#) to set the timer 0 count to the timer 0 limit.
4. Uses [break](#) to set a breakpoint on the timer 0 interrupt-vector address (timer 0 is vector 3 with address $3*8$ or `0x18`).
5. Uses [run](#) to trigger the interrupt and hit the breakpoint.

TIP You can put the commands into a text file called *filename* and execute them anytime using `read filename`.

Related Topics

- [Using Breakpoints](#) on page 200
- [Commands Reference](#) on page 231

Software Interrupts

Software interrupts are generally triggered by a `swi` (or equivalent) instruction or by writing to the software-triggered-interrupt register.

This section presents several examples of software interrupts, which write registers using various means.

- [Software Interrupt by Writing a Register with Commands](#)
- [Software Interrupt by Writing a Register with a Debugger-Intrinsic Function](#)
- [Software Interrupt by Writing a Register with an Intrinsic from a Script](#)

Software Interrupt by Writing a Register with Commands

You can generate a software interrupt on an ARC processor by writing to the software-triggered-interrupt register.

Example 33 Software Interrupt Using Commands

```
mdb> evaluate stat = stat | 0xC000000 // Enable interrupts
machine register int (stat = stat | 0xC000000) = 234897478 (0xe004046)
mdb> prop use_aux=0x201,reg_irq_trig
mdb> evaluate reg_irq_trig=5
machine register int (reg_irq_trig=5) = 5
mdb> break 5*8
Breakpoint 2 set.
2: 0x0000028
00000028 bnz_s 0x6
mdb> run
No source information at 0x28.
@ 28 00000028 bnz_s 0x6
Execution stopped at breakpoint 2, 0x28.
mdb> evaluate reg_irq_trig=0
```

Analysis

Example 33 does the following:

1. Enables interrupts by setting the `stat` register using the [evaluate](#) command.
2. Uses `prop use_aux` to enable the AUX_IRQ_HINT auxiliary register (0x201) and assign it the name `reg_irq_trig`.
3. Uses the [evaluate](#) command to set `reg_irq_trig` to vector 5, latching an interrupt for vector 5.
4. Uses [break](#) to set a breakpoint on vector 5 (address 8*5, or 0x28).

TIP To execute a command after hitting the breakpoint, use
`break address, exec command`

5. Uses [run](#) to trigger the interrupt and hit the breakpoint on vector 5.
6. Uses [evaluate](#) to clear the AUX_IRQ_HINT auxiliary register (otherwise, the interrupt is asserted again when interrupts are reenabled).

TIP You can put the commands into a text file called *filename* and execute them anytime using `read filename`.

Related Topics

- [Using Breakpoints](#) on page 200
- [Commands Reference](#) on page 231

Software Interrupt by Writing a Register with a Debugger-Intrinsic Function

[Example 34](#) shows an interactive debugging session that generates a software interrupt on ARCanel hardware.

Example 34 Software Interrupt Using a Breakpoint and an Intrinsic

```

mdb> evaluate stat32=7 // enable interrupts
machine register int (stat32=7 ) = 7
mdb> break (7*8) // set break on interrupt vector 7
Breakpoint 1 set.
1: 0x000038
        _init+0x24    b          0x38 = _init+0x24
mdb> -nowait run
Wait suppressed following `run'; you can use command `wait'.
// set IRQ_HINT register to vector 7:
mdb> evaluate $put_bank_reg(1,0x201,7)
debugger int $put_bank_reg(1,0x201,7) = 0
No source information at 0x38.
@ 38 _init+0x24    b          0x38 = _init+0x24
Execution stopped at breakpoint 1, 0x38.
mdb>
```

Analysis

[Example 34](#) does the following:

1. Uses [evaluate](#) (short form: **eval**) to set the `stat32` register (called `stat32` in the debugger) to 7, enabling interrupts.

CAUTION The debugger abbreviates some register names. If the debugger reports that the register you want to evaluate is an undeclared identifier, execute a [register](#) command with no argument to see the debugger names for all registers.

2. Uses [break](#) (short form: **b**) to set a breakpoint on interrupt vector 7.

TIP To execute a command after hitting the breakpoint, use
`break address, exec command`

3. Uses [-nowait run](#) to run the target process without returning control to the user.
4. Uses debugger-intrinsic function [\\$put_bank_reg\(\)](#) to set the `IRQ_HINT` register to vector 7.

TIP You can put the commands into a text file called *filename* and execute them anytime using `read filename`.

For prototypes and descriptions of debugger-intrinsic functions, see [Debugger-Intrinsic Functions](#) on page 223.

Related Topics

- [Using Breakpoints](#) on page 200
- [Commands Reference](#) on page 231
- [Debugger-Intrinsic Functions](#) on page 223

Software Interrupt by Writing a Register with an Intrinsic from a Script

This section provides an assembly file and test script that runs on the ISS and hardware (with `rascal12jtag` modified to support test-bench MADI registers). The test script uses the `-nowait` prefix and a back-door method with debugger-intrinsic functions to read and write registers on a running processor. Finally, this section shows a test run on ARCangel hardware using the command-line debugger.

The assembly test code in [Example 35](#) executes a loop until an interrupt has been received, then stops. Upon restart it executes another loop for a short time. The test script injects the interrupt into the target and shuts off interrupts after one has been taken, and verifies that the interrupts are shut off.

To reproduce this example, do the following:

1. Copy the assembly code in [Example 35 Software Interrupt Using a Macro and an Intrinsic \(Assembly File\)](#) and paste it into a file named `int.s` in the directory where you invoke the debugger.
2. Build `int.s` using the following command line:
`mcc int.s -o int.out -Hstack=0 -Hnocrt -Hnolib -Bbase=0`
3. Copy the debugger script in [Example 36 Hardware Interrupt Using a Macro and an Intrinsic \(Script File\)](#) and paste it into a file named `int.txt` in the directory where you invoke the debugger.
4. Debug `int.out` with the following command-line options:

```
mdb int.out -cl -hard -jtag
```

Note You must blast your ARCanel hardware if you have not blasted it since the last time your booted it. For more information, see [Debugging on ARCanel Hardware](#) on page 134.

5. Use the [read](#) command to read in and execute the debugger script:

```
read int.txt
```

6. Press ENTER occasionally.

Example 35 Software Interrupt Using a Macro and an Intrinsic (Assembly File)

```
.text
j    oops
j    oops
j    oops
int3: j    handle_level1
int4: j    handle_level1
int5: j    handle_level1
int6: j    handle_level2
j    oops
j    oops
j    oops
j    oops

oops:
flag 1 ` nop ` nop ` nop
handle_level1:
st   r0,[save]
ld   r0,[ctrl]
add  r0,r0,1
st   r0,[ctrl]
ld   r0,[save]
j    [ilink1]      // Leave interrupts off.
handle_level2:
st   r0,[save]
ld   r0,[ctrl]
add  r0,r0,1
st   r0,[ctrl]
ld   r0,[save]
j    [ilink2]      // Leave interrupts off.

.macro nops
nop`nop`nop`nop
.endm
.macro stop
flag 1
nops
.endm
_start::
.define lcnt,r8          ; loop count
mov   lcnt,0
main::
1:
flag 2|4    // Enable interrupts.
ld   r1,[ctrl]
brle r1,0,1b
first_interrupt_taken:
```

```
stop
mov lcnt,0
1:
add lcnt,lcnt,1
brle lcnt,250,1b
stop

; The test restarts us.
1:
flag 2|4 // Enable interrupts.
ld r1,[ctr2]
brle r1,0,1b
second_interrupt_taken:
stop
mov lcnt,0
1:
add lcnt,lcnt,1
brle lcnt,250,1b
stop
; Now loop forever, except that the test macro will stop the ARC.
mov lcnt,0
1: nop
add lcnt,lcnt,1
b 1b

stop
.data
save: .long 0
; counters for interrupt handlers.
ctr1: .long 0
ctr2: .long 0
.type save,@object
.type ctr1,@object
.type ctr2,@object
```

Example 36 Hardware Interrupt Using a Macro and an Intrinsic (Script File)

```
# First define macros to access the testbench registers:
printf("Version of RASCAL testbench is %x\n", $bank_reg(2,0x100))
evaluate $TESTBENCH_PULSE_INTERRUPTS=0x102
evaluate $TESTBENCH_SET_INTERRUPTS=0x103
evaluate $TESTBENCH_CLEAR_INTERRUPTS=0x104
macro level_intr n
    -nowaitq evaluate $put_bank_reg(2,$TESTBENCH_SET_INTERRUPTS,1<<$n)
    endm
macro pulse_intr n
    -nowaitq evaluate $put_bank_reg(2,$TESTBENCH_PULSE_INTERRUPTS,1<<$n)
    endm
macro clear_interrupts N
    -nowaitq evaluate $put_bank_reg(2,$TESTBENCH_CLEAR_INTERRUPTS,$N)
    endm
macro clear_all_interrupts
    # This clears all interrupts.
    # Do we want something that clears only the ones we specify,
    # while leaving others posted?
    clear_interrupts 0xffff_ffff
    endm
macro stop_after N
    # stop arc after N cycles
    prop ras_stop_after=$N
```

```

        endm

# Now the test code:
macro test_interrupt num ctr level
    -nowait run
    # Sleep and then wait for execution to stop.
    sleep 300
    # Should be able to read memory while running.
    /n2 mem 0x100
    # Shouldn't be able to read a register.
    reg r0
    # But can read an aux reg through back-door
    # Here, read status 32 and the PC while running.
    -nowaitq evaluate $bank_reg(1,10)
    -nowaitq evaluate $bank_reg(1,6)
    # Twice...
    -nowaitq evaluate $bank_reg(1,10)
    -nowaitq evaluate $bank_reg(1,6)
    simulate                                // Push the simulator if present.
    level_intr $num
    # Wait for execution to stop.
    wait
    # Should have stopped after detecting the interrupt.
    print Clearing all interrupts...
    clear_all_interrupts
    # $ctr must be non-zero by now; that tells the interrupt fired.
    if ($ctr > 0) then print SUCCESS: interrupt $num fired at level $level
    else print FAIL: interrupt $num did not fire at $level
    endif
    evaluate $save=$ctr
    run
    # ctr should be the same (no more interrupts)
    evaluate $ctr
    if ($ctr == $save) then print SUCCESS: interrupts were cleared
    else print FAIL: more interrupts occurred!
    endif
    endm
# Test interrupt level 1
test_interrupt 5 ctr1 1
# Test interrupt level 2
test_interrupt 6 ctr2 2
# Now run the ARC, wait a bit, and stop it myself.
-nowait run
sleep 300
simulate
evaluate $put_bank_reg(1,10,1)
dis pc

```

Example 37 Hardware Interrupt Using a Macro and an Intrinsic (Output with Occasional Keyboard Hits)

```

C:\ARC\MetaWare\arc\demos\mdb>mdb int.out -cl -hard -jtag
...
Welcome to the command line ... MetaWare Debugger, v7.4ENG/3f-1244.
...
No source information at 0xbc.
    bc main      flag      6
Execution stopped at breakpoint.
mdb> read int.txt
Version of RASCAL testbench is da9566
debugger int  ($TESTBENCH_PULSE_INTERRUPTS=0x102) = 258 (0x102)

```

Using Debugger Commands

```
debugger int ($TESTBENCH_SET_INTERRUPTS=0x103) = 259 (0x103)
debugger int ($TESTBENCH_CLEAR_INTERRUPTS=0x104) = 260 (0x104)
Wait suppressed following `run'; you can use command `wait'.
 0100: 7000264a 01802069 70011600 01780000 J&.pi ...|...p..x.
 0110: 805209f5 00402069 7000264a 7000264a ..R.i @.|J&.pJ&.p
Can't display registers at this time (process probably running).
int $bank_reg(1,10) = 6
int $bank_reg(1,6) = 188 (0xbc)
int $bank_reg(1,10) = 6
int $bank_reg(1,6) = 192 (0xc0)
debugger int ($put_bank_reg(2,$TESTBENCH_SET_INTERRUPTS,1<<5)) = 0
Keyboard hit!
Process stopped.
Execution stopped.
No source information at 0xbc.
  bc main      flag      6
Elapsed time 18.06 seconds.
Clearing all interrupts...
debugger int $put_bank_reg(2,$TESTBENCH_CLEAR_INTERRUPTS,0xffff_ffff) = 0
Flushing data cache.
FAIL: interrupt 5 did not fire at 1
debugger int ($save=ctr1) = 0
Keyboard hit!
Process stopped.
Execution stopped.
No source information at 0xc0.
  c0 main+0x04    1d      %r1,[0x174]      ; ctr1
Elapsed time 19.06 seconds.
unsigned int ctr1 = Flushing data cache.
0
SUCCESS: interrupts were cleared
Wait suppressed following `run'; you can use command `wait'.
  0100: 7000264a 01802069 70011600 01780000 J&.pi ...|...p..x.
  0110: 805209f5 00402069 7000264a 7000264a ..R.i @.|J&.pJ&.p
Can't display registers at this time (process probably running).
int $bank_reg(1,10) = 6
int $bank_reg(1,6) = 200 (0xc8)
int $bank_reg(1,10) = 6
int $bank_reg(1,6) = 200 (0xc8)
debugger int ($put_bank_reg(2,$TESTBENCH_SET_INTERRUPTS,1<<6)) = 0
Keyboard hit!
Process stopped.
Execution stopped.
No source information at 0xc4.
  c4 main+0x08    b???.d      0x28c4
Elapsed time 11.06 seconds.
Clearing all interrupts...
debugger int $put_bank_reg(2,$TESTBENCH_CLEAR_INTERRUPTS,0xffff_ffff) = 0
Flushing data cache.
FAIL: interrupt 6 did not fire at 2
debugger int ($save=ctr2) = 0
Keyboard hit!
Process stopped.
Execution stopped.
No source information at 0xbc.
  bc main      flag      6
Elapsed time 14.06 seconds.
unsigned int ctr2 = Flushing data cache.
0
```

```

SUCCESS: interrupts were cleared
Wait suppressed following `run'; you can use command `wait'.
debugger int $put_bank_reg(1,10,1) = 0
Execution stopped.
No source information at 0xcc.
    cc first_interrupt_taken flag      1
> cc first_interrupt_taken flag      1
    d0 .0+0x04    nop
    d4 .0+0x08    nop
    d8 .0+0x0c    nop
    dc .0+0x10    nop
    e0 .0+0x14    mov      %r8,0
    e4 .0+0x18    add      %r8,%r8,1
    e8 .0+0x1c    brge    250,%r8,0xe4 = .0+0x18 ; 0xfa = .0+0x2e
    f0 .0+0x24    flag     1
    f4 .0+0x28    nop
mdb>
    f8 first_interrupt_taken+0x2c  nop
    fc .0+0x30    nop
    100 .0+0x34   nop
    104 .0+0x38   flag     6
    108 .0+0x3c   ld       %r1,[0x178] ; ctr2
    110 .0+0x44   brlt    %r1,1,0x104 = .0+0x38
    114 second_interrupt_taken flag     1
    118 .1+0x04   nop
    11c .1+0x08   nop
    120 .1+0x0c   nop
mdb>
    124 second_interrupt_taken+0x10  nop
    128 .0+0x14   mov      %r8,0
    12c .0+0x18   add      %r8,%r8,1
    130 .0+0x1c   brge    250,%r8,0x12c = .0+0x18 ; 0xfa =
first_interrupt_taken+0x2e
    138 .0+0x24   flag     1
    13c .0+0x28   nop
    140 .0+0x2c   nop
    144 .0+0x30   nop
    148 .0+0x34   nop
    14c .0+0x38   mov      %r8,0
mdb>
    150 second_interrupt_taken+0x3c  nop
    154 .0+0x40   add      %r8,%r8,1
    158 .0+0x44   b       0x150 = .0+0x3c
    15c .0+0x48   flag     1
    160 .0+0x4c   nop
    164 .0+0x50   nop
    168 .0+0x54   nop
    16c .0+0x58   nop
    170 .0+0x5c   b       save
    174 .0+0x60   b       ctr1
mdb>

```

Related Topics

- [Using Breakpoints](#) on page 200
- [Creating and Using Debugger Macros](#) on page 224
- [Commands Reference](#) on page 231
- [Debugger-Intrinsic Functions](#) on page 223

Running a Program up to a Specific Address

For information on running a program up to a specific address, see [Using Breakpoints](#) on page 200.

Running a Program Until Data Changes

For information on running a program until data changes, see [Using Watchpoints](#) on page 201.

Executing Commands When a Program Halts

You can execute any debugger command when the application you are debugging halts at a breakpoint by using the following syntax:

```
break address, exec command
```

TIP *command* can be a macro name or a [read](#) command to invoke a script.

Example

This example sets a breakpoint on function **maximum()**, and executes five **ssi** commands as soon as the application stops at the breakpoint:

```
break maximum, exec ssi 5
```

For further details and additional options, see the listing for the [break](#) command in [Alphabetical Listing of Commands](#) on page 232.

Using Commands to Read and Write Files

The debugger provides the [file2mem](#) and [mem2file](#) commands to read files into memory and write data from memory into files, respectively.

- [Loading Data into Memory and Reading Back](#)
- [Saving Data from Memory](#)

Loading Data into Memory and Reading Back

The debugger provides the [file2mem](#) command for reading a binary, ELF, or Motorola S3 file into memory. For the syntax for loading a specific type of file to a specific address, see the listing in [Alphabetical Listing of Commands](#) on page 232.

Related Commands

- [memset](#)
- [emem](#)
- [fillmem](#)

Loading Data from a File One Word at a Time

The debugger does *not* provide a buffer-type facility for loading a small set of data, waiting, then loading more. Assuming you have enough memory space, you can load a file to memory and then move one word at a time to a variable using a macro. If you are debugging on the simulator, you can add memory without limit using [-memext](#).

Example 38 Macro for Loading Data from a File to Memory One Word at a Time

```
//Read data to memory address 0x10001000 from my_file.bin:  
file2mem 0x10001000 my_file.bin  
  
//Set up variables:  
variable ptr = 0x10001000 // Start with the first word.  
variable my_var
```

```
// Define macro getword to get each data word:
macro getword addr dataw
    evalq $dataw = *(long*)$addr // "Word" is a long on ARC 600
//Now move the pointer to the next word:
    evalq $addr += 4 //Note 4 not 1 required here (unlike in C)
endm

//Call with "ptr my_var" to get the next data word into my_var:
getword ptr my_var

//Then you can do something interesting with the data word,
//like put it in a register:
evaluate r0 = my_var
```

Note Calling the macro beyond the end of the data you read in continues to return the next word in memory to my_var, even if the next word in memory is garbage. At the end of memory, the debugger reports
ERROR: Can't change left side of += operator.

Loading Data from an Array one Word at a Time

The debugger does not support debugger arrays, but debugger variables can be used to simulate arrays, as described in [Using Debugger Variables](#) on page 188.

Example 39 Example of Loading Data Using an Array

```
// Place the following in a debugger command file my_file.cmd
// to set up data variables:
variable A0 = 0xDE
variable A1 = 0xAD
variable A2 = 0xBE
variable A3 = 0xEF

//Declare and initialize data variables:
read my_file.cmd

// Define variables and a macro to get each data word:
variable m, index
macro getword ptr dataw
    m = $ptr
    $dataw = A$m
// Advance the pointer to the next word:
    index++
endm

//Declare variable my_var to work with;
variable my_var

//Put a value from the simulated array into my_var:
getword index my_var;

//Then you can do something interesting with the data word,
//like put it in a register:
evaluate r0 = my_var
```

Note If index in this example becomes greater than 3, the debugger reports ERROR: A4 is an undeclared identifier.

Saving Data from Memory

The debugger provides the [mem2file](#) command for writing a binary file with the contents of memory. For the syntax for specifying the start address and length of data to be saved, see the listing in [Alphabetical Listing of Commands](#) on page 232.

For a text-only save, you can redirect the output of one or more [memory](#) commands to a file using [log](#) and [log off](#), for example:

```
> {log mem.txt ; memory ; memory ; log off}
```

Using Commands to View Profiling Information

To view profiling information on the command line, use the [disassemble](#) command with the [-ctr=counter_name](#) option. To display the list of all available counters, use [disassemble -ctr=](#) without a *counter_name*:

```
disassemble -ctr=
```

Note that this command also clears the counters, so you must use [-ctr=counter_name](#) again to reenable each counter you wish to reenable.

Examples

The following example shows a disassembly listing of the current process, starting at a four-byte offset in function **main()**:

```
dis main+0x04
```

The following example shows a disassembly listing preceded by a listing of the counters available:

```
dis main -ctr=
```

Available counters:

```
    icnt
    "LIMM cnt"
    "call count"
    killed
    ...
> 101c4 main      push_s      %blink
  101c6 main+0x02  push_s      %r13
  ...
  
```

The following example shows a disassembly listing with two counters:

```
mdb> dis memcpy -ctr=icnt -ctr=killed
```

```
28  0 11760 memcpy      mov.f   %lp_count,%r2
28  0 11764 memcpy+0x04  or      %r4,%r0,%r1
28  28 11768 memcpy+0x08 jz      [%blink]
28  0 1176c memcpy+0x0c and.f   0,%r4,3
28  0 11770 memcpy+0x10 lsr     %r4,%r2
28  3 11774 memcpy+0x14 bnez    0x1_1804 = memcpy+0xa4
3   0 11778 memcpy+0x18 lsr     %r4,%r4
3   0 1177c memcpy+0x1c lsr     %r4,%r4
3   0 11780 memcpy+0x20 lsr.f   %lp_count,%r4
3   0 11784 memcpy+0x24 sub     %r1,%r1,4
  ...
  
```

Clearing and Sorting Counters

Use [display](#) without an argument to see a list of available command-line displays.

Type **display prof** to view the profiling display. The profiling display includes the `icnt` counter on the instruction-set simulator. If you turn on toggle [cycles](#) to have the ISS estimate CPU cycles, the profiling display includes the `cycles` counter.

- To display the count sorted by the current counter use the following command:
`display prof prof`
- To clear the counter in use (`icnt` or `cycles`), use the following command:
`display prof clear`
- If you turn on toggle [cycles](#) to have the ISS estimate CPU cycles, you can launch the ISS by using `disassemble _start`

Examples

[Example 40 Command-Line Profiling Run](#) shows some commands and sample responses:

Example 40 Command-Line Profiling Run

```

mdb> dis _start -ctr=cycles
1 10070 _start mov %r1,0
1 10074 _start+0x04 mov %r2,0
1 10078 _start+0x08 mov %r3,0
1 1007c _start+0x0c mov %r4,0
1 10080 _start+0x10 mov %r5,0

mdb> display prof prof
Profile of Instruction cycles:
% cumulative self num avg.
total count count calls cnt/call address name
14.19 266 266 00014dfa __ac_pop_13_to_13v
11.31 478 212 2 106 00013730 _hl_blockedPeek
10.94 683 205 6 34 000150d4 priority_enq
7.58 825 142 2 71 000118b0 memchr
7.58 967 142 2 71 000138cc _hl_message_ap
2.99 1023 56 2 28 000136dc _hl_send

mdb> display prof clear
Program has no statistical profiling data
mdb> dis _start -ctr=cycles
0 10070 _start mov %r1,0
0 10074 _start+0x04 mov %r2,0
0 10078 _start+0x08 mov %r3,0

mdb> display prof prof
Address counter `Instruction cycles' contains no nonzero values.
Program has no statistical profiling data
mdb>
```

Using Commands to Dump Counters and Cache Analysis

You can use debugger commands to dump counters (with disassembly) and cache analysis to the command line or to a file to save or analyze using other tools. This example assumes you launched the compiler and debugger with the options for profiling, timers, and instruction cache.

Example 41 Script File for Dumping Profiling Counters and Cache Analysis

```

off=cr_for_more // Disable carriage return for more display.
log log_123.txt // adding an "a" would be open and append.
display icache analyze // dcache also exists, if enabled
-n400 dis 0x10000 -ctr=cycles -ctr=icnt // dump 400 lines
```

```
// disassembly and enable cycles and instruction-count columns
log off
on=cr_for_more
```

Placed in a file in the directory where you launch the debugger and invoked using [read filename](#), this script writes file log_123.txt with output such as the following (excerpt), depending on the application being debugged:

Example 42 Output File from Script to Dump Counters and Cache Analysis (Excerpt)

```
***** Statistics and analysis of Instruction cache:
Lines that appeared in the cache sets:
    (address, access cost, misses, access cycles)
    address = base address of line.
    access cost = average cost of accessing a word in the line.
    (access cost close to 1.00 is ideal -- 1 cycle per word.)
    misses = number of times this line missed.
    access cycles = cost of all line loads (2 cycles per line) and line hits (1
cycle).
    Assumption: cache takes 1 cycle/word to fill,
    and a cache hit takes only 1 cycle.

...
***** List of historical set contents:
Set 000: ( 10100 co 1.50 mi  1 cy   3) ( 10200 co 2.00 mi  1 cy   2) ( 10300 co
1.50 mi  1 cy   3) ( 10400 co 1.50 mi  3 cy   9)
        ( 10600 co 1.50 mi  1 cy   3) ( 10900 co 1.50 mi  1 cy   3) ( 10f00 co
1.67 mi  2 cy   5) ( 12400 co 1.07 mi  1 cy   15)
        ( 14100 co 1.50 mi  1 cy   3) ( 14200 co 1.50 mi  2 cy   6) ( 14400 co
1.25 mi  2 cy  10) ( 15900 co 1.50 mi  2 cy   6)
        ( 15f00 co 1.50 mi  1 cy   3)

***** Thrash analysis:
***** Each line in memory, in address order:
Set 003: 000018:  3 misses    3 hits    9 access cycles; avg access cost: 1.50
Set 004: 000020:  3 misses    0 hits    6 access cycles; avg access cost: 2.00

...
Total memory loaded in cache 4504 bytes.
391 distinct memory lines incurred a total of 563 misses.
Total cycle cost = 4503 (563 misses * 2 + 3377 hits * 1).
Note that cache hit cost is often free as it may concur with instruction execution.
(You can print these results at the command line
    by typing 'display icache analyze'.)
Available counters:
icnt
“LIMM cnt”
“call count”
killed
delay_killed
“Icache miss”
profint
“Stmt cnt from icnt”
    0      10000 _fini      push      %blink
    0      10004 _fini+0x04  jl       __mw_cpp_exit ; __mw_cpp_exit
    0      1000c _fini+0x0c  pop      %blink
    0      10010 _fini+0x10  j       [%blink]
    1      10014 _init      push      %blink

...
```

Using Commands to Set the Debugger Return Code

The pseudo-variable \$retcode contains the return code that the command-line debugger returns on exit. Normally, the return value from **main()** sets \$retcode. You can set \$retcode using the [evaluate](#) command.

Example 43 Setting the Debugger Return-Code Variable

```
evaluate $retcode = r26*2
```

Setting the return code might be useful for automated testing. When you are running a C application using hostlink, the return value of **main()** is normally the return code of the debugger. [Example 44 Windows DOS Script for Passing an Assembly Return Value](#) shows a Windows DOS script for regression tests on an assembly application without hostlink. The script passes the return value of the assembly application (posted by the application in r13) to the debugger return code. DOS errorlevel then tests the debugger return code.

Example 44 Windows DOS Script for Passing an Assembly Return Value

```
mdb prog.out -cmd=run -cmd="evaluate $retcode=r13" -cmd=quit -cl -off=gverbose
if errorlevel 1 goto report_error
```

Conventions for Debugger Input

You can pass data to the MetaWare debugger in the following forms:

- as arguments to debugger commands
- as values in debugger GUI input fields

Data consists of values such as file or path names, integers and character strings, expressions, variable or register names, and so on. Certain rules and conventions apply to how you can input these values.

Using Regular Expressions

Some debugger GUI input fields and MetaWare debugger commands can accept regular expressions as part of their input. For example, **Name:** and **Goto:** input fields accept regular-expression input; so does the **break** command.

To use a regular expression in a debugger GUI input field or as part of a MetaWare debugger command, follow the conventions listed in [Table 11 Regular Expression Conventions](#).

Table 11 Regular Expression Conventions

Convention	Matches:
*	Zero or more of any character.
?	Zero or one of any character.
[]	Any of one or more characters enclosed in brackets. For example, [xyz] matches x, y, or z.
[^]	Any characters except the one(s) enclosed in brackets. For example, [^xyz] matches any character except x, y, or z.
[-]	Any of a range of characters. For example, [a-z] matches any lowercase character from a to z.
[^ -]	Any character not in the specified range. For example, [^a-z] matches any character not in the range a to z.

Using Environment Variables in Path Names

The MetaWare debugger recognizes environment variables of the form `$var_name` in path names (for example, `$HOME`). The tilde (~) is recognized on all hosts as a synonym for `$HOME`.

The debugger hosted on Windows also recognizes environment variables with the syntax `%var_name%` (for example, `%PATH%`).

You can use path-name environment variables such as `$HOME` and `~` in these debugger GUI locations:

- **Source window Source:** input field
- **Debug a process or processes dialog Command line:** input field

You can also use these path-name environment variables in debugger commands that take a path or file name, such as [load](#), [log](#), [read](#), and [source](#). For example:

```
source $HOME/testfile.c  
read ~/myscript.txt
```

Using Expressions

Any expression you enter in a debugger GUI dialog (or pass to a MetaWare debugger command) must be a valid C or C++ expression.

Character Constants and String Expressions

Enclose character constants in single quotes, as in 'k'. In ANSI C, single-character constants are data type **int**; in C++, they are type **char**.

NOTE The MetaWare debugger does not support evaluation of wide characters (data type **wchar_t**) or Kanji characters.

Enclose character strings "like this" To represent the double-quote character ("") in a string, use the backslash/double-quote escape sequence (\"). For example, "This -->\"<-- is an embedded double-quote character".

Numeric Constants in Expressions

By default, the expression evaluator interprets numeric constants in expressions as decimal values.

To enter an octal integer, precede the constant with a 0 (zero). You can also use an optional suffix of 'u' or 'U' (for **unsigned**), or l or L (for **long**).

To enter a hexadecimal integer, precede the constant with 0x or 0X (zero and lowercase 'x' or uppercase 'X'). You can also use an optional suffix of 'u' or 'U' (for **unsigned**), or 'l' or 'L' (for **long**).

To enter a floating-point constant, use this standard format:

mantissa [*exponent*] [*suffix*]

where *exponent* is one of the following:

- 'e,' 'e+,' or 'E+'
- 'e-' or 'E-'

and *suffix* is one of the following:

- 'f' or 'F' for **float**
- 'l' or 'L' for **long double**

For example: 1.234, or 5.6e78f, or 9E-10L, or 6.02e+23F.

MetaWare Extensions to the C Language

The MetaWare debugger C-language expression evaluator supports evaluation of the following MetaWare extensions to the C language:

- underscores used between the digits of an integer constant (for example, 1_234_567 to represent the value 1,234,567)
- a second ‘x’ character used to specify a number in any base (radix) from 1 to 16 (for example, 0x2x1101 to represent the binary value of decimal 13).

Operators

The MetaWare debugger C-language expression evaluator accepts all valid C expression operators.

Debugger-Intrinsic Functions

Expressions can contain instances of the debugger-intrinsic functions listed in this section. You can use these functions with the [evaluate](#) command to examine and change aspects of your target processor and the application being debugged.

\$bank_reg(*bank,reg*)

Return the value of register *reg* in bank *bank* where the register is a four-byte quantity numbered according to `inc/regnames.h`.

For ARC processors, the values for argument *bank* are as follows:

0 = core
1 = auxiliary
2 = MADI

\$put_bank_reg(*bank,reg,val*)

Write value *val* to register *reg* in bank *bank* where the register is a four-byte quantity numbered according to `inc/regnames.h`.

For ARC processors, the values for argument *bank* are as follows:

0 = core
1 = auxiliary
2 = MADI

You can halt an ARC processor by writing to `status32` after a **-nowait** run; see [Software Interrupt by Writing a Register with an Intrinsic from a Script](#) on page 210.

\$reg(*n*)

Return the value of register *n* where the register is a four-byte quantity numbered according to `inc/regnames.h`.

TIP Always check `regnames.h` to ensure the external numbering has not changed.

\$put_reg(*n,value*)

Put *value* in register *n*.

\$swap_long(*value*)

Return four-byte value *value*, swapped. For example, `$swap_long(0xaabbccdd)` returns 0xddccbbaa.

\$swap_short(*value*)

Return two-byte value *value*, swapped. For example, `$swap_short(0xaabb)` returns 0xbbaa.

\$endian_long(*value*)

Return four-byte value *value* swapped when the endianness of the target is different from the endianness of the host. If the host and target have the same endianness, return *value*.

\$endian_short(*value*)

Return two-byte value *value*, swapped when the endianness of the target is different from the endianness of the host. If the host and target have the same endianness, return *value*.

\$read_byte(*addr*)

Read a byte from *addr*, using a one-byte-wide access to memory if the memory interface supports one-byte-wide access.

\$read_short(*addr*)

Read a **short** from *addr*, using a two-byte-wide access to memory if the memory interface supports two-byte-wide access.

\$write_byte(*addr,v*)

Write byte *v* to *addr*, using a one-byte-wide access to memory if the memory interface supports one-byte-wide access.

\$write_short(*addr,v*)

Write **short** *v* to *addr*, using a two-byte-wide access to memory if the memory interface supports two-byte-wide access.

Using Identifiers That Match Register Names

If you give an identifier the same name as a register (for example, **r14**), and you specify that name to examine, the debugger defaults to examining the variable.

To explicitly specify the register rather than the variable, enter **r_** in front of its name:

r_r14

The **r_** identifier is recursive. For example, if you have an identifier **r_r14** in your program, and you specify **r_r14** to examine, the **Examine** window shows variable **r_r14** rather than register **r14**. To examine register **r14** in this situation, you must enter **r_r_r14**.

You can add any number of **r_** prefixes to a register name, until the resulting name does not conflict with a program identifier.

Creating and Using Debugger Macros

The debugger supports user-definable commands in the form of macros. A macro consists of a named sequence of debugger commands (which can include other macros). You invoke a macro by entering its name as a command, optionally followed by a sequence of arguments; the debugger executes the macro body.

NOTE You can place macro invocations and other commands in a file and read them in as a script using the **read** command; see [Entering Commands in a Script File](#) on page 187.

Declaring and Defining a Macro

You declare a macro with the command:

macro *macro_name* [*arg1* [*arg2* ...]]

This is the macro heading.

Each macro argument is a sequence of characters separated by spaces, or a quoted string. In the latter case, the quotes are stripped and do not form part of the argument value.

NOTE If a macro argument contains whitespace, it must be enclosed in quotes.

Next you supply a sequence of debugger commands that comprise the macro body. The macro body can include calls to other macros.

TIP You can use the [print](#) command to print a message when a macro executes.

The combination of macro heading and macro body constitutes the macro definition. You terminate the definition with the command **endm** (or **mend**).

Common Errors in Macro Arguments

Use whitespace with caution, as spaces separate macro arguments. Consider the following invocation:

Example 45 Erroneous Use of Whitespace in Macro Arguments

```
write_aux base[2 * i++] AC_MASK_FULL
```

This macro is invoked with four arguments, which is probably not the intention:

- base[2
- *
- i++]
- AC_MASK_FULL

A more likely correct invocation is as follows:

Example 46 Correct Use of Whitespace to Separate Macro Arguments

```
write_aux base[2*i++] AC_MASK_FULL
```

Also, be aware that every time the first argument is mentioned in the body of the macro, **i++** is evaluated again.

Macro Example

Suppose you want to print a variable each time you statement-single-step. You could define a macro to do this; call it **my_ssi**:

```
macro my_ssi
  ssi
  evaluate some_var
endm
```

After you have defined this macro, whenever you enter the command **my_ssi**, the debugger executes the **ssi** command followed by the **evaluate some_var** command.

Conditional Macros

You can conditionally declare a macro with command:

```
cond_macro macro_name . . .
```

The macro definition is silently ignored if macro *macro_name* is already defined. This allows the debugger to re-read script files that contain macro definitions.

Invoking a Macro

You can invoke a macro using a keyboard shortcut, or by entering its name as a command, at the debugger command prompt or in a **Command:** input field.

- To use a keyboard shortcut, see [Adding a Keyboard Shortcut for a Macro](#) on page 32.
- To enter a macro name at the debugger command prompt or in a **Command:** input field, the syntax is as follows:

macro_name arg1 arg2 ...

where *arg1 arg2 ...* are *macro_name*'s (optional) arguments.

Referencing Macro Arguments

Within a macro you can refer to an argument (make an argument reference) in either of the following ways:

- Use the argument name as it appears in the macro heading, preceded by a dollar sign (\$): \$*arg*.
- Use numerical argument references of the form \$*n*, where *n* is a digit from 1 to 9 that refers to the *n*th argument in the macro heading.
- Use \$* to refer to all the arguments.

You can refer to an argument that belongs to any macro whose body is currently being expanded; to do so you must use the first form: \$*arg*. The debugger looks up macro argument names first in the innermost macro being expanded, and works outward from there.

In the following example, macro B is invoked from within macro A. Within macro B, \$*a1* (which is an argument to macro A) evaluates to x:

```
macro A a1 a2
  B
endm
...
macro B
  evaluate $a1
endm
...
A x y
```

However, if B is invoked by itself, \$*a1* remains unsubstituted, because there is no macro argument name *a1*. The command executed is then literally **evaluate \$a1**.

A command in a macro body first undergoes substitution of argument references; then the command is executed. Therefore, you can use argument substitution to define the command being executed:

```
macro doit A B
  $A$B
endm
...
doit s si # equivalent to ssi
doit i si # equivalent to isi
```

Within a macro body, if you access an argument that was not supplied, the result is an empty string.

Although the debugger does not support debugger arrays, you can simulate arrays to some extent by declaring the variables in macros; for examples, see [Using Debugger Variables](#) on page 188.

Example of a Macro Using an Argument to Reference a Simulated Array

[Example 47 on page 227](#) is a macro that starts the extension timer specified in its argument. The debugger does not support debugger arrays, but debugger variables can be used to simulate arrays, as described in [Using Debugger Variables](#) on page 188. This example uses an array to handle multiple timers.

Example 47 Using a Macro with an Array to Start Extension Timers

```
macro start_timer which
    evaluate timr$which = 1;
    evaluate tlim$which = -1;
    evaluate tctl$which = 2;
    endm
```

Analysis

This macro writes timer registers to start the extension timer (0 or 1) specified in argument *which*, assuming that the compiler and debugger were invoked with the **-Xtimer0** (and possibly **-Xtimer1**, if desired). Called with 0, it produces the following output (commands and debugger response shown):

```
mdb> start_timer 0
machine register int (timr0 = 1;) = 1
machine register int (tlim0 = -1;) = -1 (0xffffffff)
machine register int (tctl0 = 2;) = 2
mdb> display timer0
      0x00000001 COUNT      Timer count
      0x00000002 CONTROL    Timer control
      |-[ 3] 0 IP          Interrupt pending
      |-[ 2] 0 W           Watchdog reset generation disabled
      |-[ 1] 1 NH          Count clocks only when processor is not halted
      |-[ 0] 0 IE          Interrupts disabled
      0xffffffff LIMIT     Timer limit
mdb>
```

Undefining a Macro

To remove macro *macro_name*, enter the command:

```
unmacro macro_name
```

The macro is deleted, and its contents are no longer accessible.

Listing Currently Defined Macros

To see what macros are defined, enter the **set** command. This command shows various debugger settings; macros are listed at the end.

Modifying the Driver Configuration File

MetaWare debugger driver program **mdb** reads a driver configuration file, then executes the debugger according to values specified in that file. The driver configuration file is an editable, plain-text file containing default values for debugger start-up options. This file is named **mdb.cnf** and is located in the same directory as the driver program.

Directions for modifying the configuration file appear as comments in the file itself; any line that begins with the pound (#) character, followed by a space, is a comment.

Before you modify the .cnf file, make copies of the following files:

<code>mdb.exe</code>	driver executable
<code>mdb.cnf</code>	configuration file

Rename the copies, and leave the original files unmodified. The new driver and configuration file must have the same file name, but different extensions. For example, if you name the copied driver file `driver0_c1.exe`, you must rename the corresponding copied configuration file `driver0_c1.cnf`.

Specifying the Driver Location

The driver and configuration file must reside in the same directory, which by default is the MetaWare debugger bin directory. If you move these files to another directory, you must change the SCDIR= line in the configuration file to point to the new location; for example:

`SCDIR=/ROOT`

Specifying Default Values for Command-Line Options

To specify a default value for a debugger start-up option, add the option (and any relevant arguments) to the ARGS= line in the driver configuration file. For example:

`ARGS=$*ARC_ARGS -OK -startup=myfile`

The debugger treats these options as if you had typed them on the command line when you invoked the debugger. If the ARGS= line gets long enough to run over onto another line, you must add a continuation character (\) to the end of the first line.

CAUTION Do not move the ARGS= line, and do not remove the entry ARC_ARGS. Moving this line to the wrong location in the driver configuration file can render the debugger inoperable.

See [Command-Line Option Reference](#) on page 274 for a complete list of options whose values you can specify on the ARGS= line.

Changing the Default Start-up to Command-Line Only

The debugger driver program `mdb` launches the debugger with a GUI by default. If you want to use the command-line-only debugger by default, add option `-cl` to the ARGS= line in the driver configuration file, `install_dir/bin/mdb.cnf`.

Reducing Debugger GUI Start-up Time

If you always include the program name on the command line when you start the debugger, you can specify option `-OK` to skip the **Debug a process or processes** dialog and go directly to the debugger desktop. Specify `-OKN` to also skip the MetaWare debugger splash screen.

Specifying the Location of Java

If the location of your Java Virtual Machine is different from the JVM installed by the MetaWare debugger installation program, modify .cnf configuration file in one of these ways:

- Add `-javadir=dir_where_my_java_resides` to the ARGS= line.
- Modify the JAVADIR= line: `JAVADIR=dir_where_my_java_resides`.

NOTE Even if you choose to use your own copy of the JVM, you must use the same version as the JVM that ships with the debugger. Other versions of the JVM can cause the debugger GUI to exhibit performance quirks.

See MetaWare debugger Release Note or readme file for JVM version information.

Chapter 7 — Commands Reference

- [Overview](#)
- [Alphabetical Listing of Commands](#)

Overview

This section lists complete information about MetaWare debugger commands alphabetically.

Related Topics

- For a quick-access listing of commands, see [Quick Commands List](#) on page 345.
- For information about using commands, see [Using Debugger Commands](#) on page 183.

Alphabetical Listing of Commands

addmap — Add information to a memory map

Syntax: **addmap** *map_name* *lo*,*hi*[,*width N*][, *readonly*][, *verify*] [,reverse_endian] [*delta D*]

Adds a range (*lo*, *hi*) of memory addresses (*hi* is the first address *not* in the range) to memory map *map_name*. Underscores can be used in address values for readability. *width* is the size in bytes of each address in the range. The *width* may be 4, 8, or 16 (only 4 is permitted for ARC targets). If no *width* is specified, 4 is assumed.

<i>lo</i> , <i>hi</i>	Decimal or hex values representing the first address in the address range and the first address <i>not</i> in the range. The range is closed on the left and open on the right. This means you can specify addresses without FFFF, for example: addmap 0x1000, 0x2000 addmap 0x2000, 0x3000 addmap 0x2000, 0x4000 ...
-----------------------	---

NOTE Because *hi* is the first address *not* in range, if you want your address range to extend to the end of memory, you must specify 0 (zero) as *hi*.

<i>width N</i>	When <i>N</i> =1, memory accesses, addresses, and sizes are 0 mod 1. When <i>N</i> =2, memory accesses, addresses, and sizes are 0 mod 2. When <i>N</i> =4, addresses and sizes are 0 mod 4.
<i>readonly</i>	Causes the debugger to use hardware comparator breakpoints within the declared memory region.
<i>verify</i>	Enables verification of every memory write to the address range. This slows access speeds but can be useful when initially configuring board initialization.
<i>reverse_endian</i>	Indicates that memory in the given address range is endian-reversed with respect to the normal endianness of the target. The debugger reverses data written to or read from the area in chunks that are multiples of the memory width.
<i>delta</i>	The offset <i>D</i> used with is added to the address when the target memory is accessed.

Related Topics

- [Using a Memory Map to Indicate an Offset](#) on page 129
- [memmap — Define or print memory map](#)

args — Display name and arguments of the current executableSyntax: **args**

Shows the file name and arguments of the current executable.

Identical to the **file** command.See also [file — Display name and arguments of the current executable](#).**attach — Attach an already running process**Syntax: **at[tach] pid**The **attach** command loads the running process identified by *pid* into the debugger for debugging. *pid* is the process ID.

Use your system utilities to determine the process IDs of running processes.

NOTE The **attach** command works only on systems that allow the attaching and detaching of already running processes.

The following example attaches a process with process ID 10475:

attach 10475See also [detach — Detach current process from the debugger](#).**aux — Display ARC auxiliary registers**Syntax: **aux [reg_number]**The **aux** command shows the contents of the ARC auxiliary registers. Enter **aux reg_number** to display a subset of auxiliary registers starting at register *reg_number*.See also [register — Display machine registers](#).**break — Set a breakpoint or display breakpoints**Syntax: **b[reak] [address] [, breakpoint_option [, ...]]**or: **b[reak], list [input|addr]**Entered without an argument, the **break** command shows a listing of the index number, hexadecimal address, and source-code line for each current breakpoint. Hardware breakpoints are indicated with an 'H.'**Breakpoint Address**Entered with the *address* argument, **break** sets a breakpoint on *address*, where *address* is one of the following:

- A number (hexadecimal, decimal, or octal) indicating the address of a machine instruction.
- The name of a register that contains *address* (as in **break pc**).
- *function_name*, a function definition in the current source-code file, with or without an optional offset. The offset can be positive or negative.
- A regular expression, as in p??p* (see [Using Regular Expressions](#) on page 221).
- !*line_number*, where *line_number* is a line in the current source-code file.

- *file_name!line_number*, a line number in the specified file *file_name*, where *file_name* is an available source-code file for your program. In this case, *file_name* can be a regular expression.
- Any valid C/C++ expression indicating a location in program code.

Breakpoint Options

Use one or more *breakpoint_option* arguments preceded by a comma to further define the breakpoint:

- *global* sets a global, non-thread-specific breakpoint when you are debugging a specific thread.
- *hard* indicates your preference is for a hardware breakpoint. Hardware breakpoints are supported for some architectures and target interfaces. They are indicated with an ‘H’ in lists and windows.
- *soft* indicates your preference is for a software breakpoint, not a hardware breakpoint. See also toggle [prefer soft bp](#).
- *temp* specifies a temporary breakpoint (hit once, then deleted); same as [tbreak](#).
- *[count] NNN* sets a conditional breakpoint on *address* such that the application runs past the address *NNN* times before execution stops. *NNN* must be a positive integer (you can specify the integer following *count* or by itself).
- *eval expression* sets a conditional breakpoint on *address* that becomes enabled only when *expression* evaluates to true. *expression* can be any expression in the supported languages. (Beware of side effects; for example, if *expression* is *i++*, variable *i* is incremented when the expression is evaluated.) See [Using Expressions](#) on page 222.
- *exec debugger_command* sets a breakpoint on *address* such that *debugger_command* is executed every time control stops on the breakpoint. *debugger_command* is any MetaWare debugger command.

See [Setting Task- or Thread-Specific Breakpoints](#) on page 63 for more information about special kinds of breakpoints.

Thread-specific breakpoints are examples of conditional breakpoints. You can write a **break** command that checks for a complex thread-related condition. Such a condition uses \$tid, a debugger pseudo-variable that is always set to the current thread.

The thread ID is a unique number that an operating system assigns to a thread to identify the thread. Thread numbers can be small, such as 1, 2, 3, and so on. For some operating systems (such as ThreadX), the thread ID is the address of a control block that stores information about the thread.

The form *Tlist [input|addr]* is designed to be used by an OEM to prepare **break** commands for re-entering to the debugger when the debugger resumes. *addr* lists the address of each item input; *input* is a list of the breakpoints as input by the user. Compare toggle [restore_ap](#).

NOTE The breakpoints might or might not work when re-input, depending on the program context when they are re-entered.

For examples, see [Using Breakpoints](#) on page 200.

See also the following:

[delete break — Delete a breakpoint](#)

[disable break — Disable a breakpoint](#)

[enable break — Enable a breakpoint](#)

[**evaluate — Evaluate or change the value of an expression**](#)[**tbreak — Set a temporary breakpoint**](#)**call — Call a function**

Syntax: **call** *function*

The **call** command sends *function* to the expression evaluator, which parses and evaluates *function*; that is, **call** executes *function*.

The **call** command differs from the **evaluate** command because **call** enables you to debug a function call in your program.

To do this, you insert breakpoints in the debugged program before using **call**. When the debugger encounters a breakpoint in your program, it abandons the expression evaluation so you can debug the function call.

When the function call completes, or when you remove the debugger-created temporary breakpoint (created to catch the return from the call), the program is restored to the state it was in before the call evaluation.

See [Evaluating Expressions](#) on page 78 for information about the debugger's expression evaluator.

The following example calls function **strlen()**:

```
call strlen(0x1000)
```

See also the following:

[**break — Set a breakpoint or display breakpoints**](#)[**evaluate — Evaluate or change the value of an expression**](#)[**watch — Set a watchpoint or display watchpoints**](#)**continue — Start or continue execution of current process**

Syntax: **cont[inue]**

The **continue** command starts or continues execution of the current process at full speed from the current location of the program counter. The debugger automatically halts execution from a **continue** command if any of the following occurs:

- Execution encounters a breakpoint.
- Your program causes a signal or exception to occur.
- The program terminates normally.

See also the following:

[**restart — Restart the current running process**](#)[**run — Start or continue execution of the current process**](#)**delete break — Delete a breakpoint**

Syntax: **del[ete] b[reak] {index | all}**

Entered with an *index* argument, **delete break** deletes a single breakpoint indicated by *index*, the index number (1, 2, 3...) of the breakpoint.

NOTE To find the index of a breakpoint, enter the **break** command without any arguments; this lists the current breakpoints.

Entered with the **a11** argument, **delete break** deletes all current breakpoints.

This example deletes a breakpoint with index 3:

```
delete break 3
```

This example deletes all current breakpoints:

```
del b all
```

See also [break — Set a breakpoint or display breakpoints](#).

delete mod — Delete an executable module

Syntax: **delete mod n**

Use the **delete mod** command to delete an executable module and its symbols from the list of modules currently loaded. For a numbered list of loaded executable modules, use the **modules** command.

This example deletes executable module 2:

```
delete mod 2
```

delete watch — Delete a watchpoint

Syntax: **del[ete] w[atch] {index | a11}**

Entered with an *index* argument, **delete watch** deletes a single watchpoint indicated by *index*, the index number (1, 2, 3...) of the watchpoint.

NOTE To find the index of a watchpoint, enter the **watch** command without any arguments; this lists the current watchpoints.

Entered with the **a11** argument, **delete watch** deletes all current watchpoints.

This example deletes a watchpoint with index 9:

```
delete watch 9
```

This example deletes all current watchpoints:

```
del w all
```

See also [watch — Set a watchpoint or display watchpoints](#).

delmap — Delete an entry from a memory map

Syntax: **delmap map_name low_address**

Delete the entry with *low_address* from memory map *map_name*.

detach — Detach current process from the debugger

Syntax: **det[ach]**

The **detach** command detaches the current process from the debugger. The process can be either one that the debugger loaded and initialized, or one that the debugger attached.

NOTE The **detach** command works only on systems that allow the attaching and detaching of already running processes.

See also [attach — Attach an already running process](#).

disable break — Disable a breakpoint

Syntax: **disable b[reak]** {*index* | **a**ll}

Entered with an *index* argument, **disable break** disables a single breakpoint indicated by *index*, the index number (1, 2, 3...) of the breakpoint.

NOTE To find the index of a breakpoint, enter the **break** command without any arguments; this lists the current breakpoints.

Entered with the **a**ll argument, **disable break** disables all current breakpoints.

This example disables breakpoint number 8:

```
disable break 8
```

This example disables all current breakpoints:

```
disable b all
```

See also the following:

[break — Set a breakpoint or display breakpoints](#)

[delete break — Delete a breakpoint](#)

[enable break — Enable a breakpoint](#)

disable watch — Disable a watchpoint

Syntax: **disable w[atch]** {*index* | **a**ll}

Entered with an *index* argument, **disable watch** disables a single watchpoint indicated by *index*, the index number (1, 2, 3...) of the watchpoint.

NOTE To find the index of a watchpoint, enter the **watch** command without any arguments; this lists the current watchpoints.

Entered with the **a**ll argument, **disable watch** disables all current watchpoints.

This example disables watchpoint number 3:

```
disable w 3
```

This example disables all current watchpoints:

```
disable watch all
```

See also the following:

[watch — Set a watchpoint or display watchpoints](#)

[delete watch — Delete a watchpoint](#)

[**enable watch** — Enable a breakpoint](#)

disassemble — Show a disassembly listing of the current process

Syntax: [-n*N*] **dis**[**assemble**] *address* [-ctr=*cname* ...]

The **disassemble** command causes the debugger to show a disassembly listing of the current process, starting at *address*, where *address* is one of the following:

- A number (hexadecimal, decimal, or octal) indicating the address of a machine instruction.
- The name of a register that contains *address* (as in **dis pc**).
- A regular expression, as in p??p* (see [Using Regular Expressions](#) on page 221).
- *function_name*, a function definition in the current source-code file, with or without an optional offset. The offset can be positive or negative.
- !*line_number*, where *line_number* is a line in the current source-code file.
- *file_name!**line_number*, a line number in the specified file *file_name*, where *file_name* is an available source-code file for your program.
- Any valid C/C++ expression indicating a location in program code.

Optional argument -ctr= shows profiling counter *cname* in the disassembly. It may be specified as many times as desired; specifying -ctr= without an argument displays the counters available. The counters are listed left-to-right in the order you requested. The setting persists until you specify -ctr= without an argument.

Preceding the command with the optional modifier -n*N* (like /n*N*) prints *N* lines. This can be used to limit the number of lines displayed, but also to increase the number of lines displayed, which is useful for obtaining a mixed trace, for example:

```
log dis_main.txt  
-n1000 dis main  
log off
```

Pressing ENTER continues the listing most recently requested. However, **dis** is reset to the current PC location when you step or otherwise come to a halt.

display [*display_name*] — Show named command-line display

The **display** command prints a display on the command line. This command is provided primarily for showing your own command-line displays created as described in *install_dir/chipinit/readxml.txt*, but also provides access to a set of command-line displays available by default. In the GUI debugger, most of the displays are available from the **Display** menu.

Entered without an argument, this command lists the available displays, which vary by target processor.

Special Command-Line Displays for ARC Targets

Of particular interest for ARC targets are the following.

- **display exmmu** — Display ARC 700 Exception and MMU registers

If you have specified option **-mmu** (and **-a7**), you can use this command to display the contents of the exception and MMU registers.

- **display mmu_contents** — Display ARC 700 MMU contents

If you have specified option **-mmu** (and **-a7**) you can use this command to display the contents of the MMU. See [Displaying MMU and Exception Registers \(ARC 700\)](#) on page 146 for more information.

- **display mpu** — Display ARC 600 MPU registers

If you have specified option **-mpu** (and **-a6**) you can use this command to display the contents of the MPU registers on the command line. See [Viewing the ARC 600 MPU Registers](#) on page 146 for more information.

- **display overlays** — Display overlays for applications using overlays

Use this command to display overlays in applications that use the ARC Automated Overlay Manager. For more information on overlays, see the *Automated Overlay Manager User's Guide* and [Debugging with Overlays](#) on page 130.

- **display smart** — Display ARC 600 SmaRT Trace data

Syntax: **display smart [[lines=]count] [c[ompressed][=0|1]] [e[nabled][=0|1]]**

Entered without an argument, **display smart** displays SmaRT Trace data after the debugger has detected the presence of SmaRT Trace.

- Use **lines=count** to display *count* lines, or enter the integer for *count* without **lines=**.
- Use **compressed** or **compressed=1** to omit instructions between flow-control changes, **compressed=0** to include them.
- Use **enabled** or **enabled=1** to enable trace capture, **enabled=0** to disable.

See [Displaying SmaRT Data Using the Command Line](#) on page 142 for examples.

- **display simd | vrN[_UN]** — Display ARC 700 SIMD information

After you have specified the location of your SIMD DLL and started debugging, you can use **display simd** to display the SIMD configuration registers, or use **display vrN** to display the vector registers in an 8-, 16, or 32-bit view, where *N* is 8, 16, or 32.

If you have more than one SIMD unit, use **display vrN_UN**, where *UN* is the unit number.

See [Reading Special Registers in Detail](#) on page 194 for examples.

- **display timer0 | timer1**

After you specify ARC extension timer 0 or 1 using the applicable **-X*** option, you can use this command to display the timer register on the command line.

download — Load an executable and read its symbol table

Syntax: **download exepath**

Use this command to tell the debugger to load an additional executable when debugging an application that consists of multiple executables.

The debugger assumes the new executable was linked at locations that do not overlap the locations of the startup executable. The debugger does not check whether this condition is violated.

If you are debugging on hardware, and additional executables are downloaded by other means, use the **symbols** command to instruct the debugger to read the symbol table of the additional executables.

emem — Enter values into memory

Syntax: **[modifier] emem address values**

Enter values starting at a memory address. For integers, the values can be expressions that resolve to integers. For floating-point numbers, the values must be constants. If address is “.”, the memory enter continues from where you left off in the prior **emem** command. The *modifier* supports 1, 2, and 4-byte integer values and 4 and 8-byte floating-point values. For example, use modifiers **/c** and **/s** for 1 and 2-byte integers, and use **/f** and **/d** for floating-point values.

NOTE Modifiers are not remembered when you use **emem** “.”, so you must always retype them.

The modifiers and addresses are the same as in the **memory** command. See the description of modifiers for the **memory** command for more information.

See also [**memory — Dump memory**](#).

enable break — Enable a breakpoint

Syntax: **enable b[reak] {index | a77}**

Entered with an *index* argument, **enable break** enables a single breakpoint indicated by *index*, the index number (1, 2, 3...) of the breakpoint.

NOTE To find the index of a breakpoint, enter the **break** command without any arguments; this lists the current breakpoints.

Entered with the **a77** argument, **enable break** enables all previously disabled breakpoints.

The following example enables previously disabled breakpoint number 5:

```
enable b 5
```

The following example enables all previously disabled breakpoints:

```
enable break all
```

See also the following:

[**break — Set a breakpoint or display breakpoints**](#)

[**delete break — Delete a breakpoint**](#)

[**disable break — Disable a breakpoint**](#)

enable watch — Enable a watchpoint

Syntax: **enable w[atch] {index | a77}**

Entered with an *index* argument, **enable watch** enables a single watchpoint indicated by *index*, the index number (1, 2, 3...) of the watchpoint.

NOTE To find the index of a watchpoint, enter the **watch** command without any arguments; this lists the current watchpoints.

Entered with the **a** 11 argument, **enable watch** enables all previously disabled watchpoints.

The following example enables previously disabled watchpoint 1:

```
enable watch 1
```

The following example enables all previously disabled watchpoints:

```
enable w all
```

See also the following:

[**watch — Set a watchpoint or display watchpoints**](#)

[**delete watch — Delete a watchpoint**](#)

[**disable watch — Disable a watchpoint**](#)

endm — Terminate a macro definition

Syntax: **endm**

Terminates a macro body. See [Creating and Using Debugger Macros](#) on page 224 for more information. Synonym for **mend**.

See also the following:

[**macro — Declare a macro**](#)

[**unmacro — Remove a macro definition**](#)

evalq — Evaluate or change the value of an expression quietly

Syntax: **evalq** *expression*

Command **evalq** evaluates or changes the value of an expression but does not show the results. This form of “quiet” evaluation is useful for assignments in scripts.

The *expression* can contain registers. If your program already has a variable or function with the same name as a register, the evaluator finds the program entity before it finds the register. To allow the specification of a register, you can prefix the register name with *r*_; for example, *r_r3* denotes register r3. You can prefix a register name with any number of *r*_ prefixes until the resultant name no longer conflicts with any program entity. For example, *r_r_r_r_r_r3* also denotes r3.

The *expression* can contain function calls into the program being debugged. In this case, the debugger saves the current program state, executes the call, and then restores the program state.

Note You cannot use **evalq** to debug code invoked by way of a function call; see the [**call**](#) command.

The following example disables data and instruction caches and initializes CPU registers:

```
evalq DC_CST = 0x04000000
evalq IC_CST = 0x04000000
evalq msr = 0x1002
evalq SRR1 = 0x1002
evalq DER = 0x31C67C0F
```

The following example initializes hardware devices:

```
evalq *(long*)0xff00_0000 = 0x01632640
```

```
evalq *(long*)0xff00_0004 = 0xFFFFFFF88
evalq *(long*)0xff00_0320 = 0x55ccaa33
```

evaluate — Evaluate or change the value of an expression

Syntax: **eval[uate]** *expression*

The **evaluate** command sends *expression* to the MetaWare debugger expression evaluator, which parses and evaluates *expression* and shows the result. *expression* is any valid C/C++ expression.

If *expression* is an assignment statement, you can use the **evaluate** command to change the value of a variable or element of an aggregate data structure.

Use the **evaluate** command with the **break** or **watch** command to set a conditional breakpoint or watchpoint.

The *expression* can contain function calls into the debugged program. In this case, the debugger saves the current program state, executes the call, and then restores the program state.

CAUTION The debugger abbreviates some register names. If you use **evaluate** with a register name and the debugger reports that the register you want to evaluate is an undeclared identifier, execute a [register](#) command with no argument to see the debugger names for all registers.

Examples

The following example evaluates a function that includes a string argument:

```
mdb> wait eval strlen("A test\n")
...
int strlen("A test\n") = 7
Elapsed time 0.25 seconds.
```

```
mdb> eval I
register r13 int I = 178760 (0x2ba48)
```

See [Evaluating Expressions](#) on page 78 for information about the debugger's expression evaluator.

NOTE You cannot use **evaluate** to debug code invoked by way of a function call; see the [call](#) command.

The following example evaluates the expression **max % i** and displays the result:

```
evaluate max % i
```

The following example tests the value of identifier **max**:

```
eval max==100
```

The following example assigns the value 20 to **limit**:

```
evaluate limit=20
```

See also the following:

[break — Set a breakpoint or display breakpoints](#)

[watch — Set a watchpoint or display watchpoints](#)

exec — Execute a command on encountering a breakpoint

Syntax: **b[reak]** *location*, **exec** *debugger_command*

The **exec** command executes the specified command *debugger_command* each time execution encounters the breakpoint at *location*.

The following example shows the contents of register r3 (command **reg r3**) each time execution encounters the breakpoint at function **my_func()**.

```
b my_func, exec reg r3
```

TIP To execute more than one command, you can place the commands in braces and separate them with semicolons:

```
b Try, exec { print Hello ; print World }
```

See also [break — Set a breakpoint or display breakpoints](#).

exit — Exit the MetaWare debugger

Syntax: **ex[it]**

The **exit** command terminates all processes, exits the debugger, and returns to the command line.

Entering **exit** is the same as choosing **File | Exit Debugger** in the GUI or closing the last debugger GUI that is open.

To detach your program before exiting the debugger (so your program continues to run), enter the **detach** command before entering the **exit** command.

Identical to the **quit** command.

See also the following:

[kill — Terminate the current process](#)

[quit — Exit the MetaWare debugger](#)

file — Display name and arguments of the current executable

Syntax: **file**

The **file** command shows the file name and arguments of the current executable.

Identical to the **args** command.

See also [args — Display name and arguments of the current executable](#).

file2mem — Write a file to memory

Syntax: **file2[mem]** {-e | -s | -q | *address*} *file_name*

The **file2mem** command copies a binary, ELF, or Motorola S3 file to memory.

- Use **file2mem** with an *address* specification to copy the contents of binary file *file_name* to memory at the *address* specified.
- Use **file2mem** with option **-e** instead of an address to load the contents of ELF file *file_name* to memory.

- Use **file2mem** with option **-s** instead of an address to load the contents of S-record file *file_name* to memory.
- Use **file2mem** with option **-q** instead of an address to load the contents of Mentor Quicksim hex file *file_name* to memory.

file_name can include environment variables of the form %XXX%, \$XXX, and ~, which is short for \$HOME.

This command is useful, for example, for loading a dataset into memory so your embedded program can access it.

See also [**mem2file — Dump memory to a binary file**](#).

fillmem — Fill memory

Syntax: **[/s|/c] fillmem value start length**

The **fillmem** command fills memory at *start* for *length* bytes with the value contained in *value*. Prefix the command with **/s** when *value* is a **short**, or **/c** when *value* is a **char**.

value can be any of the following:

- a string (enclosed in quotes)
- a hex digit
- a 4-byte value
- a 2-byte value (prefix the command with **/s**)
- a 1-byte value (prefix the command with **/c**)

Examples

```
fillmem "mystring" 0x0 512  
0x200 bytes filled
```

```
fillmem 0x7000 0x100 256  
0x100 bytes filled
```

See also [**file2mem — Write a file to memory**](#)

focus — Set CMPD focus

Syntax: **focus [N]**

The CMPD debugger operates only on processes that are currently in the focus. Use **focus** to set the focus to process(es) or process set(s) *N*, where *N* is a numeric value for processes and an alphabetic value for process sets (see [Examples](#)). Note that the brackets surrounding the argument *N* are part of the command and must be typed.

Examples

```
focus [1,2,3] //Set the focus to processes 1, 2, and 3.  
focus [1:4] //Set the focus to processes 1 through 4.  
focus [1,3:5] //Set the focus to processes 1, 3, 4, and 5.  
focus [A] //Set the focus to process set A previously defined.  
focus [A, B, 5] //Set the focus to all processes in sets A and B, and process 5.  
focus [A*B, 5] //Set the focus to processes that appear in both sets A and B, and process 5.
```

For more information on using CMPD and defining process sets, see [Chapter 5 — CMPD Extensions](#).

globals — Display global variables

Syntax: **glob[als] [reg_expr]**

Entered without an argument, the **globals** command displays a listing of all global variables for your program.

Entered with the *reg_expr* argument, **globals** displays a listing of all global variables that match regular expression *reg_expr*.

See [Using Regular Expressions](#) on page 221 for information about regular expressions.

The following example lists all variables beginning with ‘L’ followed by a zero, one, or two character string, or a period and another string:

```
glob L??.*
```

See also [locals — Display local variables](#).

go — Stop execution at a specified address

Syntax: **go address**

The **go** command runs the current process and stops it at the specified *address*, where *address* is one of the following:

- A number (hexadecimal, decimal, or octal) indicating the address of a machine instruction.
- The name of a register that contains *address* (as in **go pc+4**).
- A regular expression, as in **p??p*** (see [Using Regular Expressions](#) on page 221).
- *function_name*, a function definition in the current source-code file, with or without an optional offset. The offset can be positive or negative.
- **!line_number**, where *line_number* is a line in the current source-code file.
- *file_name!***line_number**, a line number in the specified file *file_name*, where *file_name* is an available source-code file for your program.
- Any valid C/C++ expression indicating a location in program code.

The command **go address** is equivalent to **tbreak address** followed by **run**.

The following example runs the program, then stop execution at function **main()**:

```
go main
```

Use **go main** to display source code in the **Source** window when you restart a process.

The following example stops execution at a 12-byte offset beyond function **average()** in the current process:

```
go average+0x0c
```

See also [tbreak — Set a temporary breakpoint](#).

help — Get help on a specific topic

Syntax: **help [topic | summary | a77]**

Entered without an argument, **help** shows information about the help commands, **help topic**, **help summary**, and **help a77**.

Entered with a *topic* argument, **help** displays information about *topic*, where *topic* can be any MetaWare debugger command or one of the following:

- misc** Display help for the **log** and **log off** commands.
- show** Display help for all commands that display information.
- step** Display help for all stepping commands.
- tip** Display tips for using the MetaWare debugger.

Entered with the *summary* argument, **help** prints the usage for all MetaWare debugger commands.

Entered with the *all* argument, **help** prints information about all the MetaWare debugger commands.

history — Display command history

Syntax: **h[istory]** [*number*]

The **history** command displays the command history, a list of all the commands executed since the debugging session began.

NOTE A **restart** does not cancel the history list, but is added to it.

number is a positive integer limiting the command history to *number* commands.

To execute a command in the command history, just enter its number.

Repeated commands and history-related commands are not saved in the debugger command history. This is different from the behavior of a shell history. The debugger command history is thus revisionist, not a literal history of events as they occurred.

if — Execute a command if condition is true

Syntax: **if** (*expression*) *command*

Use the **if** command to conditionally execute *command* if *expression* evaluates to non-zero. The **if** command does not have a corresponding “else” command, but see also **if (expn) then** under [if...endif](#). *command* is any valid MetaWare debugger command.

expression is any valid C/C++ expression and can contain function calls into the program being debugged (for more information, see [evaluate](#)).

Related Topics

- [evaluate — Evaluate or change the value of an expression](#)
- [if ... endif — Nest conditional commands](#)
- [while — Execute a command while condition is true](#)
- [Executing Commands Conditionally](#) on page 191

if ... endif — Nest conditional commands

Syntax:

if (*expn*) **then** *command_if*

command_if

```
...
[els[e]if (expn_elsif) command_elsif
command_elsif
... ]
[else command_else
command_else
... ]
endif
```

If *expn* evaluates to non-zero, perform zero or more *command_if* commands. Otherwise, for the first *expn_elsif* that evaluates to non-zero, perform zero or more *command_elsif* commands. Otherwise, perform zero or more *command_else* commands. The **elsif** and **else** portions are optional.

expn is anything that can be evaluated with the [evaluate](#) command. The difference between **if...endif** and the simple **if** command is the presence of **then**, **else**, **elseif**, and **endif**. (However, if you use **if** and **endif** without a **then**, the debugger warns of a mismatched **endif**.)

NOTE **if** (*expn*) **then** must be on a single line, and **elsif**, **else**, and **endif** must begin a command.

For example, the following is *not* acceptable:

```
if (a==b) then eval r3=1 else isi endif
```

but the following is:

```
if (a==b) then eval r3=1
else isi
endif
```

See also the following:

[evaluate — Evaluate or change the value of an expression](#)

[while — Execute a command while condition is true](#)

[if — Execute a command if condition is true](#)

isi — Step through machine instructions, into functions

Syntax: **isi[q] [n]**

The **isi** command causes execution to step from one machine instruction to the next. When execution encounters a function call, it steps into the function.

Entered without an argument, **isi** steps one machine instruction at a time.

You can also specify a number *n* of machine instructions to step, starting at the current position of the program counter. If one of the instructions is a function call, execution steps into the function the remaining number of instruction steps.

Use **isiq** to step without showing the results. This “quiet” mode is useful for stepping in scripts.

TIP If you enter the **isi** command without an argument, press ENTER to repeat the command.

If you enter **isi n**, pressing ENTER yields a single step.

See [Using Command Prefixes and Modifiers](#) on page 184 for how to repeat the **isi** *n* command.

Step five machine instructions, starting at the current position of the program counter:

```
isi 5
```

See also the following:

[**iso — Step through machine instructions, over functions**](#)

[**ssi — Step through source-code statements, into functions**](#)

[**sso — Step through source-code statements, over functions**](#)

iso — Step through machine instructions, over functions

Syntax: **iso[q] [n]**

The **iso** command causes execution to step from one machine instruction to the next. When execution encounters a function call, it steps over the function; that is, it causes the function to execute and moves to the command following the function call.

Entered without an argument, **iso** steps one machine instruction at a time.

You can also specify a number *n* of machine instructions to step, starting at the current position of the program counter. If one of the instructions is a function call, execution steps over it, executing the function, and continues the remaining number of instruction steps.

Use **isoq** to step without showing the results. This “quiet” mode is useful for stepping in scripts.

TIP If you enter the **iso** command without an argument, press ENTER to repeat the command.

If you enter **iso n**, pressing ENTER yields a single step.

See [Using Command Prefixes and Modifiers](#) on page 184 for how to repeat the **iso n** command.

The following example steps eight machine instructions, starting at the current position of the program counter:

```
iso 8
```

See also the following:

[**isi — Step through machine instructions, into functions**](#)

[**ssi — Step through source-code statements, into functions**](#)

[**sso — Step through source-code statements, over functions**](#)

kill — Terminate the current process

Syntax: **kill**

The **kill** command terminates the current process running in the debugger and removes its debug information. To stop the current process without removing its debug information, see [**stop**](#).

lbreak — Set a breakpoint on a symbol that has not been loaded yet

Syntax: **lb[reak]** *address* [, *NNN*]

The **lbreak** command sets a breakpoint on a symbol in a shared library that has not been loaded yet.

address is a number (hex, decimal, octal), function +- an optional offset, *filename!function*, *filename!line_num*, or any valid C/C++ expression.

NNN is a positive integer that indicates the number of times the breakpoint is hit before stopping.

Note *NNN* can be specified after the word `count' or by itself.

See [Setting Task- or Thread-Specific Breakpoints](#) on page 63 for more information about special kinds of breakpoints.

See also [**break — Set a breakpoint or display breakpoints**](#), and [**tbreak — Set a temporary breakpoint**](#).

load — Load a program into the debugger

Syntax: **load** *program_name* [*args* ...]

The **load** command loads a program *program_name* into the debugger and initializes it. This command is the same as choosing **File | New process** in the GUI. *program_name* is the path and name of the program to load. You can also specify arguments *args* to the program.

Note If a process already exists (that is, you are debugging some process A and you want to load and initialize some other process B), use the [**kill**](#) command to terminate process A before loading B.

See also the following:

[**kill — Terminate the current process**](#)

[**restart — Restart the current running process**](#)

locals — Display local variables

Syntax: **loc[als]**

The **locals** command displays a listing of all local variables currently in scope.

See also the following:

[**globals — Display global variables**](#)

[**stack — Display the run-time function-call stack**](#)

log — Log commands and results to a file

Syntax: **log** *file_name* [*a*][*i*][*o*] | **off**

The **log** command opens a log file *file_name* and writes MetaWare debugger commands and their results to it. The optional arguments have the following meanings:

- a** Append to log information in *file_name*.
- i** Log input commands.
- o** Log output results.

If you do not specify **a**, the log file gets overwritten.

If you specify **a** with a file that does not exist, the debugger creates the file and starts appending.

If you specify neither **i** nor **o**, the default is both.

NOTE The log file cannot be named off, because that conflicts with the **log off** command.

log off terminates the current log session and closes the log file.

The **log** command can be nested, which means that each **log off** command closes the “inner-most” (most recent) log file.

The following example opens existing log file `log.txt`, and appends both input commands and output results to the information already in it:

```
log log.txt a i o
```

The following example opens three log files to log both input and output, performs some debugging tasks, then close the log files in FILO order:

```
log one.log a
log twos.log a
log three.log a
...
// Do some debugging
...
log off (closes three.log)
log off (closes twos.log)
log off (closes one.log)
```

macro — Declare a macro

Syntax: **macro** *macro_name* [*arg1*[*arg2*[...]]]

Declares a macro (a user-defined command) named *macro_name*. A macro is a named sequence of debugger commands. You invoke a macro by entering its name as a command, optionally followed by a sequence of arguments. See [Creating and Using Debugger Macros](#) on page 224 for more information.

See also the following:

[**endm — Terminate a macro definition**](#)

[**unmacro — Remove a macro definition**](#)

mem2file — Dump memory to a binary file

Syntax: **mem2**[*file*] *address length file_name*

The **mem2file** command dumps *length* bytes of memory to binary file *file_name*, starting at *address*.

See also [file2mem — Write a file to memory](#).

mem2mem — Copy memory from one location to another

Syntax: **mem2mem** *source dest length*

The **mem2mem** command copies memory from *source* address to *dest* address for *length* bytes.

You might see a message such as the following:

```
Not supported directly on target; will use read-to-host/write-from-host to
implement
```

This message means that the target processor or simulator does not support **memcpy()** directly, and host resources are being used (a slower method).

See Also

[mem2file — Dump memory to a binary file](#)

memmap — Define or print memory map

Syntax: **memmap** [*map_name*]

Defines a memory map named *map_name*. The map is a data structure that records which memory regions are of what kind, so that regions of non-standard size or type can be addressed. The map is initialized to hold no addresses.

Entering **memmap** without an argument prints the memory map currently in use.

You can use the **prop** command to enable and disable memory maps while debugging. To enable a memory map, use the following command:

```
prop memmap=map_name
```

To disable the current memory map, use the following command:

```
prop memmap=0
```

The debugger reports an error only the first time you force an access to an area outside the memory map.

Example 48 Temporarily Disabling Memory Maps

```
> memmap M1
> addmap M1 0,0x80000
> prop memmap=M1
> prop memmap=0
> eval *0xA0000
int (*0xA0000) = -1075839105 (0xbfdffff7f)
> prop memmap=M1
> eval *0x90000
int (*0x90000) = Can't find address 0x90000 in memory map M1.
Map is:
Start:0x00000000 end:0x00080000 width:0 readonly:0 verify:0 delta:0x0
0x0bad0bad (read failed)
> eval *0x90000
int (*0x90000) = 0
```

Related Topics

See the following for examples of memory maps:

- [Using a Memory Map to Indicate an Offset](#) on page 129

See also [addmap — Add information to a memory map](#).

memory — Dump memory

Syntax: `[radix][format] mem[ory] address`

The **memory** command dumps 160 bytes of memory, starting at *address*, where *address* is one of the following:

- A number (hexadecimal, decimal, or octal) indicating the address of a machine instruction.
- The name of a register that contains *address* (as in **mem pc**).
- A regular expression, as in `p??p*` (see [Using Regular Expressions](#) on page 221).
- *function_name*, a function definition in the current source-code file, with or without an optional offset. The offset can be positive or negative.
- `!line_number`, where *line_number* is a line in the current source-code file.
- `file_name!line_number`, a line number in the specified file *file_name*, where *file_name* is an available source-code file for your program.
- Any valid C/C++ expression indicating a location in program code.

Once you have entered a **memory** command, pressing ENTER dumps another 160 bytes of memory. If you have used a *format* modifier, that modifier continues in effect.

The *format* modifier determines the format of the display. It can be any of the following:

<code>/nlines</code>	Limit number of lines displayed to <i>lines</i>	
<code>/b</code>	byte view	(single-byte units)
<code>/c</code>	byte view	(single-byte units)
<code>/s</code>	short view	(two-byte units)
<code>/w</code>	long view	(four-byte units)
<code>/i</code>	int view	(four-byte units)
<code>/l1</code>	long long view	(eight-byte units)
<code>/f</code>	float view	(floating-point representation of four-byte units)
<code>/d</code>	double view	(floating-point representation of eight-byte units)
<code>/ld</code>	long double view	(floating-point representation of eight-byte units)
<code>/ub</code>	unsigned byte view	(single-byte units)

/us	unsigned short view	(two-byte units)
/ui	unsigned int view	(four-byte units)
/ull	unsigned long long view	(eight-byte units)

The default format is **int** view (four-byte units).

Unsigned variants of the *format* modifiers are meaningful only if the radix of the displayed data is 10. You can specify a radix for the displayed data by preceding the memory command with **/radix**, either by itself or with any of the *format* modifiers appended. Valid values for *radix* are 2, 8, 10, or 16.

This command dumps memory for function **main()** in four-byte units (default **int** view) of octal values:

```
/8 memory main
```

This command dumps memory for function **maximum()** in eight-byte units (**unsigned long long** view) of decimal values:

```
/10ull mem maximum
```

The following example dumps 160 bytes of memory, starting at function **min_max()**:

```
memory min_max
```

The following example dumps 160 bytes of memory, starting at memory address 0x01013c:

```
mem 0x01013c
```

See also [emem — Enter values into memory](#).

memsb — Search backward through memory

Syntax: **memsb** *pattern* [*start_addr*] [*length*]

The **memsb** command searches memory for *pattern*, starting at address *start_addr* and searching backward for *length* bytes. *start_addr* and *length* are optional expressions.

- If you omit *start_addr*, the debugger starts at address 0.
- If you omit *length*, the debugger searches all of memory.

See the [memsearch](#) command for a complete description.

memsearch — Search forward through memory

Syntax: **[/s|/c] mems[earch]** *pattern* [*start_addr*] [*length*]

The **mems** command searches memory for *pattern*, starting at address *start_addr* and searching forward for *length* bytes. *start_addr* and *length* are optional expressions.

- If you omit *start_addr*, the debugger starts at address 0.
- If you omit *length*, the debugger searches all of memory.

The *pattern* argument can be a character string, a hexadecimal string, or an integer.

- A character string is a sequence of characters enclosed in double-quote ("") characters. A character string can also contain a wildcard character (?) and certain two-character escape sequences. For

more information about these special characters, see [Searching for Patterns in Memory](#) on page 88.

- A hexadecimal string is a sequence of hexadecimal digits or the wildcard character (?). The hexadecimal string begins with the characters “0h”. The wildcard character denotes “zero or one digit”.
- An integer is an arbitrary expression that evaluates to a one, two, or four-byte value. Use the characters “0x” to prefix a hexadecimal integer. Use the character “0” to prefix an octal integer. Use a /**s** (two-byte) or /**c** (single-byte) modifier when searching for an integer less than four bytes. (For more information about /**s** and /**c**, see the description of the [memory](#) command.)

The following example searches for hexadecimal strings:

```
mems 0h7ff
mems 0hdeadbeef
mems 0hdea?beef
```

The wildcard character ‘?’ allows the hexadecimal string dea?beef to match both of these hexadecimal strings:

```
deadbeef
deafbeef
```

Search for character strings:

```
mems "abc"
mems "Hello, world.\n"
mems "Hello, ?????.\n"
```

The last example finds either of these strings: "Hello, world.\n" or "Hello, earth.\n" due to the use of the wildcard character ‘?’.

Search for integers:

```
mems 0x1234
mems p->x
/s mems b15 (where b15 evaluates to a two-byte value)
/c mems a[i+1] (where a[i+1] evaluates to a one-byte value)
```

See also [memsb — Search backward through memory](#).

memset — Set memory to value

Syntax: **mems[et]** *dest val length*

Sets memory at *dest* to *val* for *length* bytes. *val* is assumed to be a byte value; for a **char**, use quotes. This command matches the **memset()** function in C.

Examples

```
memset 0x10bc 0xfe 1000
memset 0x100 'a' 50
```

mend — Terminate a macro definition

Syntax: **mend**

Terminates a macro body. See [Creating and Using Debugger Macros](#) on page 224 for more information. Synonym for **endm**.

See also the following:

[**macro — Declare a macro**](#)

[**unmacro — Remove a macro definition**](#)

modules — List current executable and shared libraries

Syntax: **mod[ules]**

The **modules** command shows a list of the current executable and any currently loaded shared libraries (DLLs).

onexit *command* — Execute command on exiting the debugger

Adds *command* to a list of commands to be executed when the debugger terminates normally.

See also the following:

[**exit — Exit the MetaWare debugger**](#)

[**quit — Exit the MetaWare debugger**](#)

print — Print a message

Syntax: **print | print1** *message*

Use **print** *message* to print a string to the debugger's output; for the command-line debugger the debugger's output is **stdout**; for the GUI debugger the debugger's output is the **Command** window.

Use **print1** *message* to print a message string to **stdout** (handle 1).

prop — Set a target system property

Syntax: **prop** *prop1=val1 [prop2=val2 ...]*

Sets target system or simulator property *prop1* to value *val1*, and so on for all specified properties and values. This is generally of interest to those who are implementing interfaces that extend debugger functionality.

Examples

The following example stops the simulator after 432 instructions.

```
prop stop_after=432
```

The **stop_after** value is decremented until it reaches zero; the simulator halts.

The following example defines special-purpose registers:

```
prop use_spr=26,SRR0
prop use_spr=27,SRR1
prop use_spr=144,CMPA
prop use_spr=145,CMPB
```

The following example specifies that memory map A is to be handed to system **ARC_simulator**:

```
prop ARC_simulator memmap=A
```

For additional properties for selected targets, see the following:

- [Debugging ARC Systems](#) on page 126

quit — Exit the MetaWare debugger

Syntax: **q[uit]**

The **quit** command terminates all processes, exits the debugger, and returns to the command line.

Entering **quit** is the same as choosing **File | Exit Debugger** in the GUI, or closing the last open debugger window.

To detach your program before exiting the debugger (so your program continues to run), enter the **detach** command before you enter the **quit** command.

Identical to the **exit** command.

See also the following.

[**exit — Exit the MetaWare debugger**](#)

[**kill — Terminate the current process**](#)

read — Automatically process commands in a file

Syntax: **read** *file_name*

The **read** command causes the debugger to automatically process the sequence of MetaWare debugger commands in command file *file_name*.

You can use the **read** command almost like a macro if you invoke it with arguments and reference the arguments from within the command file.

register — Display machine registers

Syntax: **reg[ister]** [*bank* | ? | *reg_expr*]

The **register** command shows the contents of the machine registers.

Enter **reg bank** to make the debugger display a subset of the target-system machine registers.

Enter **reg ?** to list possible values for *bank* for your system.

Enter **reg reg_expr** (where *reg_expr* is any regular expression) to display all the registers in the current bank that match the regular expression. For example, the following sequence of commands displays all the registers in the **core** bank that begin with the letter r:

```
register core  
...  
register r*
```

NOTE The first time you enter the **register** command, the default for *bank* is **a11**. If you have already specified some value other than **a11**, the default for *bank* is the last bank or banks displayed.

See also [**aux — Display ARC auxiliary registers**](#).

restart — Restart the current running process

Syntax: **restart [-reset][args]**

The **restart** command terminates the current running process and by default reloads it with the same arguments it took on the previous load. To reload with new arguments, specify them as *args*. Entering **restart** is the same as choosing **File | Restart** in the GUI.

Optional **-reset** resets the target before restarting. This also implies a full download on restart: The debugger no longer assumes that code previously downloaded is intact, due to reset. In addition, the debugger re-runs any chipinit file you supplied.

You can obtain **-reset** behavior by default by turning on toggle [reset upon restart](#).

See also the following commands:

[**kill — Terminate the current process**](#)

[**load — Load a program into the debugger**](#)

return — Return from the current function

Syntax: **ret[urn]**

The **return** command causes the process to execute until it returns from the current function. Execution stops on the machine instruction or source-code statement following the call to the function.

run — Start or continue execution of the current process

Syntax: **run**

The **run** command starts or continues execution of the current process at full speed from the current location of the program counter. The debugger automatically halts execution from a **run** command if any of the following occurs:

- Execution encounters a breakpoint or hardware watchpoint.
- Your program causes a signal or exception to occur.
- You press CTRL+C.
- The program terminates normally.

Identical to the **continue** command.

Note If you are debugging for a native system, when your program has run to completion, **run** behaves like the **restart** command. This is not the case for embedded targets.

See also the following:

[**continue — Start or continue execution of current process**](#)

[**restart — Restart the current running process**](#)

set — Set or display debugger settings

Syntax: **set [setting=value]**

Entering the **set** command without an argument causes the debugger to list all debugger settings and their current values. The list appears in the command window from which you invoked the debugger (command-line debugger) or in the **Debug console** (GUI debugger).

Entering the **set** command with debugger setting *setting* as an argument assigns a new value to the debugger setting.

Following are the debugger settings that you can reset with the **set** command:

- [addr_width](#)
- [args](#)
- [dir_xlation](#)
- [double_format](#)
- [float_format](#)
- [printlines](#)
- [source_path](#)

addr_width

The minimum size of hex addresses in certain windows, such as **Disassembly**, **Memory**, **Modules**, and **Instruction history -- Simulator trace**.

args

An array of strings representing the arguments with which you invoked the debugger; `args[0]` is always the name of the program you are debugging.

The following example changes the command-line arguments with which you invoke your program:

```
set args=-Hldopt "-D cmdfile" myprog.c -o myobj
```

NOTE Use double-quote characters to group elements into one argument.

In the preceding example, the quotes identify “-D cmdfile” as a single argument.

dir_xlation=old,new [...] (Windows)

dir_xlation=old,new [...] (UNIX and Linux)

If the location of your source has changed since compilation, set `dir_xlation=old,new` where `new` replaces `old` if `old` is a prefix of the directory of the source when compiled.

You can enter multiple string pairs, separated by semicolons (Windows) or colons (UNIX and Linux).

Example 49 Using dir_xlation (Windows Example)

This example references source for a debug build of the MetaWare C Run-Time Library. The command tells the Windows debugger to look for source in `d:\ARC\MetaWare\arc` instead of `c:\ARC\MetaWare\hcarc`:

```
set dir_xlation=c:\ARC\MetaWare\hcarc,d:\ARC\MetaWare\arc
```

Example 50 Using dir_xlation (UNIX Example)

To use ‘;’ as a separator on UNIX or Linux systems, place a ‘;’ as the first character of `dir_xlation`. The following example tells the UNIX debugger that source built on Windows is now located in a directory tree with the same structure on UNIX:

```
set dir_xlation=;c:\,/home/
```

double_format

float_format

By default, the debugger uses the C format specifier `%20e` to display **doubles** and `10%e` to display **floats**.

Use `set double_format=F` or `set float_format=F` to specify a new format *F* for use in all contexts. For example, `set double_format=%20.15e` gives you more digits after the “.” than `%e` typically does. The format *F* must be a standard C format string.

CAUTION Specifying a nonstandard format might cause the debugger to crash.

You can also use `-prop=double_format|float_format` or `-prop=double_format|float_format`.

printlines

The number of lines printed by the [memory](#) command.

source_path

A sequence of directory paths separated by the system-dependent path separator — a colon (:) on UNIX or Linux systems or a semicolon (;) on Windows systems.

The debugger searches these directories for source code if it cannot find the source based on the program’s debugging information.

See also [unset — Restore debugger setting to its default](#).

show log — Show all commands and outputs thus far

Show a log of all commands and outputs since the debugger was started. To send the output to a file, use the [log](#) command first.

Example

```
mdb> log mylog.txt
mdb> show log
1: Debugging queens on ARCompact Instruction Set Simulator (ARC_simulator).
2: queens: Process created
3: Simulator extensions loaded: Extension Library.
4: Downloading to ARCompact Instruction Set Simulator.
5: ELF segment #0 addr 0x010000 size 0x005b27 ...
6: ELF segment #1 addr 0x016000 size 0x0011d8 .
7: ELF segment #1 addr 0x0171d8 zero 0x0108c8 .....
8: Semantic inspection loaded: Profiling Plugin.
9: mdb> run
10:   queens.c!48: void main () {
11: Execution stopped at breakpoint.
12: mdb> readq ".scrc"
13: mdb> readq "C:\ARC\MetaWare\arc/bin/scrc"
14: mdb>
15: mdb> log mylog.txt
16: mdb> show log
mdb> log off
```

File mylog.txt now contains the same information displayed in response to **show log**.

NOTE In a log from a GUI session, you might see `wait` before your commands. This `wait` tells the debugger “Do not execute the command until all the processes in focus have stopped.”

simulate — Update status quietly

Like the [status](#) command, the **simulate** command updates the status of the application being debugged, but does not print output (useful for scripts).

On the instruction-set simulator, this command advances execution by the amount specified by option **-instrs_per_pass**=*NNN* (the default is 512 instructions). This permits all hostlink requests to the ISS to be serviced.

This command is most useful for hardware-testing scripts, when you wish to run the target and execute further commands while it is running using command prefix **-nowait**. For an example of such use, see [Halting the Instruction-Set Simulator](#) on page 207.

sleep — Sleep specified number of milliseconds

Syntax: **sleep** *nnn*

Enter **sleep** *nnn* to force the debugger to wait *nnn* milliseconds. This command is useful for waiting, for example after a board reset.

source — Display source code

Syntax: **sour[ce]** [*file_name* | *address* | {+|-} *num*]

Entered without an argument, the **source** command shows the contents of the source-code file associated with the current location of the program counter. The lines of source code are numbered, starting with line 1.

Entered with any argument, the **source** command displays, by default, 14 lines of source code, starting at the indicated address. Once you have entered the **source** command, you can press ENTER to display the next 14 lines of code.

Entered with a *file_name* argument, **source** displays the source-code file *file_name*, with line numbers, starting with line 1. *file_name* is the path and file name of the source file you want to display.

NOTE You must include the source-file extension.

Entered with an *address* argument, **source** attempts to display source at a specified address, where *address* is one of the following:

- A number (hexadecimal, decimal, or octal) indicating the address of a machine instruction.
- The name of a register that contains *address* (as in **source pc**).
- A regular expression, as in p??p* (see [Using Regular Expressions](#) on page 221).
- *function_name*, a function definition in the current source-code file, with or without an optional offset. The offset can be positive or negative.
- !*line_number*, where *line_number* is a line in the current source-code file.
- *file_name!**line_number*, a line number in the specified file *file_name*, where *file_name* is an available source-code file for your program. In this case, *file_name* can be a regular expression.
- Any valid C/C++ expression indicating a location in program code.

The debugger tries to evaluate the *address* argument as a file name. If that fails, the debugger then evaluates the argument as an address expression, which can be a numeric address or an identifier. For example, if *address* is the name of a function, you can use the **source** command to display the source code for that function.

Entered with a +*num* argument, **source** displays *num* lines of source code following the position of the location counter. Entering **source** -*num* displays the *num* lines of code preceding the location counter

The following example displays source code for `my_prog.c`:

```
source my_prog.c
```

The following example displays the source code for function `main()`:

```
sour main
```

ssi — Step through source-code statements, into functions

Syntax: `ssi[q] [n]`

The `ssi` command causes execution to step from one source-code statement to the next. When execution encounters a function call, it steps into the function.

Entered without an argument, `ssi` steps one source-code statement at a time.

You can also specify a number *n* of statements to step, starting at the current position of the program counter. If one of the statements is a function call, execution steps into the function the remaining number of statement steps.

Use `ssiq` to step without showing the results. This “quiet” mode is useful for stepping in scripts.

TIP If you enter the `ssi` command without an argument, press ENTER to repeat the command.

If you enter `ssi n`, pressing ENTER yields a single step.

See [Using Command Prefixes and Modifiers](#) on page 184 for how to repeat the `ssi n` command.

Same as the `step` command.

The following example steps seven source-code statements, starting at the current position of the program counter:

```
ssi 7
```

See also the following:

[**isi — Step through machine instructions, into functions**](#)

[**iso — Step through machine instructions, over functions**](#)

[**sso — Step through source-code statements, over functions**](#)

sso — Step through source-code statements, over functions

Syntax: `sso[q] [n]`

The `sso` command causes execution to step from one source-code statement to the next. When execution encounters a function call, it steps over the function; that is, it causes the function to execute and moves to the statement following the function call.

Entered without an argument, `sso` steps one source-code statement at a time.

You can also specify a number *n* of statements to step, starting at the current position of the program counter. If one of the statements is a function call, execution steps over the statement, executing the function, and continues the remaining number of statement steps.

Use `ssoq` to step without showing the results. This “quiet” mode is useful for stepping in scripts.

TIP If you enter the **sso** command without an argument, press ENTER to repeat the command.

If you enter **sso n**, pressing ENTER yields a single step.

See [Using Command Prefixes and Modifiers](#) on page 184 for how to repeat the **sso n** command.

The following example steps four source-code statements, starting at the current position of the program counter:

```
sso 4
```

See also the following:

[**isi — Step through machine instructions, into functions**](#)

[**iso — Step through machine instructions, over functions**](#)

[**ssi — Step through source-code statements, into functions**](#)

stack — Display the run-time function-call stack

Syntax: **stack**

The **stack** command shows the current contents of the function-call stack. If the program counter is stopped in code associated with an available source-code file (that is, “stopped in known source”), each line lists a function name and the value of each argument to that function.

If the program counter is not stopped in known source, each line lists the hexadecimal address of a called function.

See also [**locals — Display local variables**](#).

status — Update status of application being debugged

The **status** command updates and displays the status of the application being debugged.

On the instruction-set simulator, this command advances execution by the amount specified by option **_instrs_per_pass=NNN** (the default is 512 instructions). This permits all hostlink requests to the ISS to be serviced.

For a “quiet” version of this command that does not print the results, use [**simulate**](#).

step — Step through source-code statements, into functions

Synonym for [**ssi**](#).

stepi — Step through machine instructions, into functions

Synonym for [**isi**](#).

stop — Stop the current process

The **stop** command stops (halts) the current process. Compare the [**kill**](#) command, which stops the current process and removes its debug information from the debugger.

symbols — Load symbols from executable

Syntax: **symbols filename[-load=+address -quiet]**

The **symbols** command causes the debugger to add all the symbolic information from the ELF executable *filename*. This command is useful when an application (such as an RTOS or an application using overlays) has loaded additional executables without the knowledge of the debugger.

You can include **-quiet** with this command to suppress debugger confirmation of successful loading.

If you attempt to load an executable file that has the same memory address as one already loaded, the debugger reports that it is unable to load the module. You might see an error message such as `Unable to comply for filename: Physical address 0 already mapped.`

The following example loads the symbols from `myfile.o` at an offset of `0x30000` from the location where it was linked:

```
symbols myfile.o -load=+0x30000
```

To have the debugger load an additional executable *and* read its symbol table, use the [download](#) command.

See also the following:

[**download — Load an executable and read its symbol table**](#)

[**modules — List current executable and shared libraries**](#)

[**delete mod — Delete an executable module**](#)

sysclone — Clone the current system

Syntax: `sysclone existing_system new_sys1 [new_sys2 ...]`

The **sysclone** command makes a clone of a system *existing_system* and names the new, cloned system *new_sys1* (and *new_sys2*, and so on, as more clones are created).

By cloning an existing system, you can run more than one system in a single debugging session, which enables you to debug multiple programs at one time.

sysprop — Set properties of specified target system

Syntax: `sysprop system_name property=value [prop2=val2 ...]`

The **sysprop** command sets properties of the target-system *system_name* as specified: it sets property *property* to the specified value *value* (and so on for all itemized properties and values).

For more information, see [Determining the Target System](#) on page 116.

system — Execute a command on the host system

Syntax: `system command`

The **system** command executes a command on the host system. The output is directed to the command window in which the debugger was invoked.

tbreak — Set a temporary breakpoint

Syntax: `tb[reak] address [, condition[, ...]]`

The **tbreak** command sets a temporary breakpoint on *address*. A temporary breakpoint is one that is removed as soon as execution stops on it. In all other respects, a temporary breakpoint is like a regular breakpoint.

Entered with the *condition* argument, **break** sets a conditional breakpoint on *address*. The *condition* can be any of the following, singly or in combination:

eval[uate] *expression*
count
exec *command*

For more information about the *condition* argument to **tbreak**, see the description of [**break**](#).

See [Setting Task- or Thread-Specific Breakpoints](#) on page 63 for more information about special kinds of breakpoints.

See also [**break — Set a breakpoint or display breakpoints**](#), or [**lbreak — Set a breakpoint on a symbol that has not been loaded yet**](#).

thread — Switch to an alternate thread

Syntax: **thread** *thread_ID*

Whenever you display information, such as registers, that information pertains to the current thread. You can switch to an alternate thread by entering a **thread new_TID** command, where *new_TID* is the thread ID of that alternate thread. All subsequent displays show information for that alternate thread (where relevant).

NOTE To get a listing of your program's thread IDs, enter the **threads** command.

When you resume running the program, the debugger switches back to the current thread, undoing the **thread new_TID** command.

Here is a shortcut for reentering the **thread new_TID** command without retyping it:

1. Enter **h** (for [history](#)) to see the debugger command history.
2. Find the number of the **thread new_TID** command.
3. Enter that number to reissue the command.

See also [**threads — Display a list of thread IDs**](#).

threads — Display a list of thread IDs

Syntax: **threads**

The **threads** command displays a list of thread IDs for all threads in the debugged program, and also identifies the current thread.

The following example displays the current threads:

threads

Resultant output, where thread 2 is the current thread:

```
current thread
37944: "System Timer Thread"
35860: "thread 0"
36004: "thread 1"
36148: "thread 2" (current thread)
36292: "thread 3"
36436: "thread 4"
36580: "thread 5"
```

NOTE This example is very target specific. Your results will vary.

See also [thread — Switch to an alternate thread](#).

trace — Show instruction trace history

Syntax: **trace NNN**

Syntax: **trace modifier**

Show instruction trace history, if available.

trace NNN

Using only the numeric argument *NNN* requests a trace of *NNN* lines (starting from the most recent instruction). Display, however, begins with the oldest instruction within the specified range.

trace modifier

To display traces selectively, use one or more of the following *modifiers*:

- **trace=tracename**

Modifier **trace=tracename** specifies the trace component for which you want the trace history displayed. Replace *tracename* with your component name. Note that this modifier repeats the command name:

```
trace trace=mytrace
```

To see a trace history for all the installed trace components, use a nonexistent name:

```
trace trace=notexist
```

- **fields=1**

Modifier **fields=1** displays fields. Not all traces have fields; the fields are not defined by the debugger but by the component doing the capture. You can see which fields are available by clicking the **Fields** button in the **Instruction history -- Simulator trace** window, which toggles field display on and off. The **1** here is part of the modifier, which is essentially an on/off switch.

- **bool=boolexp** or **regex=regexp**

You can restrict the lines displayed using a regular expression or a Boolean expression (but not both). You can use a Boolean expression only if the trace has fields.

Use a Boolean query **bool=boolexp** to restrict output to those lines satisfying the Boolean expression *boolexp*. Use a regular-expression query **regex=regexp** to restrict output to those lines satisfying the regular expression *regexp*.

A regular expression *regexp* is any valid regular expression as described in [Table 11 Regular Expression Conventions](#) on page 221.

A Boolean expression *boolexp* is a C expression whose constituents include the fields in the trace. Each field has a 32-bit value you can use in the expression. You name a field in the Boolean expression by prefixing it with a '\$'; for example \$addr names the field addr. A trace item matches if the Boolean expression evaluates to non-zero for that item *and* the expression contains at least one field present in that trace item. For example, if you look for items satisfying \$addr >= 0x20000, an item must contain an addr field to match; register writes, for example, would be excluded.

Here are some examples of Boolean expressions:

```
$addr >= 0x2000
$reg = 13 && $value < 10
$reg == 13 ? $value == 10 : $value = 1
• lines=NNN
```

Modifier *l*ines=NNN specifies how many lines of trace to show (starting from the most recent entry). Replace NNN with the number of lines you want to show. Display, however, begins with the oldest instruction within the specified range. When you specify *l*ines=NNN together with a Boolean expression or a regular expression, NNN specifies the number of lines to search, not the number of lines to show.

Examples

```
trace 001
    0      800041dc -> write blink

trace 01
    0      800041dc -> write blink

trace trace=notexist
Couldn't find trace notexist. Possible traces are:
Simulator instruction history
    19  10744  _mwcall_main+0xa8  sub.f      0,%r16,%r19
    18  10748  _mwcall_main+0xac  add        %r14,%r14,4
    ...
    1      0002ba50 -> write r1
    0      800041dc -> write blink

trace fields=1 llines=1
    0      800041dc -> write blink [reg=0x1f value=0x800041dc]

trace bool=$reg>10
    17     00000004 -> write r14
    12     0002ba54 -> write r18
    0      800041dc -> write blink
```

unmacro — Remove a macro definition

Syntax: **unmacro** *macro_name*

Removes a macro definition. See [Creating and Using Debugger Macros](#) on page 224 for more information.

See also the following:

[**macro** — Declare a macro](#)

[**endm** — Terminate a macro definition](#)

unset — Restore debugger setting to its default

Syntax: **uns[et]** *setting*

The **unset** command restores a debugger setting to its default value.

Argument *setting* can be any of the following (see **set** for complete descriptions):

- [addr_width](#)
- [args](#)

- [*dir_xlation*](#)
- [*printlines*](#)
- [*source_path*](#)

See also [set — Set or display debugger settings](#).

variable — Declare a debugger variable

Syntax: **variable** *N* [[:*type*] [=init] [, *N*[[:*type*]=*init*]]]

Declares a debugger variable *N*. Variable *N* may then be used in expressions.

type is the optional data type; acceptable types are **int** or **double**. Default is **int**.

init is the optional initial value.

NOTE The initial value may not include '=' or ','.

Any debugger command beginning with a declared variable is assumed to be prefixed with [**evalq**](#).

CAUTION Take care that a variable name does not conflict with the name of an entity in your program. The expression evaluator gives precedence to the latter.

Example

```
variable A=3, B
debugger int (A = 3) = 3
B=2*A
// evalq assumed. Quiet evaluation, so no debugger response
evaluate A+B
int (A+B) = 9
```

Variables starting with a '\$' are also debugger variables, and are implicitly declared. Generally you can use these variables without fear of conflicting with an entity in your program. Variables starting with a '\$' must be prefixed with [**evaluate**](#) or [**evalq**](#).

Example

```
mdb> evaluate $i = 0
debugger int ($i = 0) = 0
mdb> while ($i < 0x1f8) { evalq *(short*)$i = 0xffff; evalq $i += 2; }
Flushing data cache.
Flushing and invalidating data cache.
mdb> eval $i
debugger int $i = 504 (0x1f8)
```

vec*N* — Display SIMD vector registers

For ARC 700 Only

Synonym for [**display vr*N***](#).

wait — Wait for execution to stop before returning control to command line

The **wait** command tells the debugger to wait for execution to complete before returning control (and feedback) to the command line (this is the default debugger behavior). You can use **wait** after issuing a command with the [**-nowait**](#) modifier, to tell the debugger to wait for execution to stop. For more information and an example, see [Waiting and Sleeping](#) on page 205.

watch — Set a watchpoint or display watchpoints

Syntax: **w[atch]** *idexpression* [, **eval[uate]** *evalexpression*][**exec debugger_command**] [, **mode m**] [, **thread threadno**][, **invert**][, **value v** [, **mask k**]]

or: **w[atch]** *hex_address num_bytes* [**exec debugger_command**][, **mode m**] [, **thread threadno**][, **invert**][, **value v** [, **mask k**]]

or: **w[atch]**, **list** [*input|addr*]

watch with No Arguments

Entered without an argument, the **watch** command shows a listing of the index number, expression (identifier), and value for each current watchpoint.

Argument *idexpression*

Entered with the *idexpression* argument, **watch** sets a watchpoint on a specified expression currently in scope. *idexpression* is any expression that can appear on the left side of an assignment statement; usually it is a local or global variable.

NOTE You can only use the *idexpression* argument if the program was compiled with option **-g**.

For a program compiled without **-g**, you can determine the memory or register location of that expression and then set the **watch** on the memory's *hex_address*, or set a **watchreg** on the register.

watch evaluate evalexpression

Entered with the **evaluate evalexpression** command, **watch** sets a conditional watchpoint on *idexpression* that becomes enabled only when *evalexpression* evaluates to True. *evalexpression* must be a conditional expression. (Beware of side effects; for example, if *evalexpression* is *i*++, variable *i* is incremented when the expression is evaluated.) See [Using Expressions](#) on page 222.

Argument exec debugger_command

Optional argument **exec debugger_command** sets a watchpoint on *address* such that *debugger_command* is executed every time control stops on the watchpoint. *debugger_command* is any MetaWare debugger command.

Arguments hex_address and num_bytes

Entered with the *hex_address* and *num_bytes* arguments, **watch** sets a watchpoint on the specified number of bytes *num_bytes*, starting at hexadecimal address *hex_address*.

TIP On ARC hardware, if two actionpoints are available (not used as hardware breakpoints) you can trap a write from the core to a memory range if the length of the range is a power of 2 and the start address of the range is aligned to the length value. For example:
`watch 0xA000, 0x1000, mode w`

Argument mode m to Specify Read or Write Operation

By default, watchpoints stop execution on data write operations only. Use optional argument **mode m** to limit the watchpoint by the type of operation. Replace *m* with one of the following.

- **read** (or **r**) for read (simulator)
- **write** (or **w**) for write
- **rw** (or **wr**) for read/write (simulator)

Argument *thread threadno*

Use optional argument *thread threadno* to stop execution only if the current thread is *threadno* (replace *threadno* with an integer thread ID). This argument is equivalent to the expression `$tid == threadno`. (`$tid` is a debugger pseudo-variable that is always set to the current thread.)

Argument *invert*

Optional argument *invert* applies only to hardware watchpoints and stops execution when access is made outside of the specified range, as shown in [Example 26 Using Special Watch Arguments for ARC Targets](#) on page 202.

Pair arguments *mask k* and *value v* (ARC 600 and ARC 700 Only)

Specify optional arguments *mask k* and *value v* together. *value* makes the watchpoint conditional so that it stops execution only if the content of *idexpression* (or of *hex_address*) matches *v* after being **anded** with *k*. If you do not specify *mask k*, the default is `0xFFFFFFFF`.

In effect, the watchpoint stops execution when the following condition holds (assuming a watchpoint length of 4):

`(idexpression_or_hex_address & k) == (v & k)`

Using *mask k* and *value v* with *invert*

Normally when you use *mask k* and *value v*, and *m* is the value in memory, the processor stops when $(m \& k) == (v \& k)$. If you also specify *invert*, the processor stops when $(m \& k) != (v \& k)$. Specify these arguments together to restrict the match to certain conditions, as shown in [Example 26 Using Special Watch Arguments for ARC Targets](#) on page 202.

Restrictions on Using *mask k* and *value v*

- ARC 600 and ARC 700 hardware only
- Processor configuration that supports pair action points
- Processor's maximum pair action points not yet exhausted
- Lengths of 1, 2, or 4 bytes only

watch *list*

The form *list [input|addr]* is designed to be used by an OEM to prepare **watch** commands for re-entering to the debugger when the debugger resumes. *addr* lists the address of each item input; *input* is a list of the watchpoints as input by the user.

Note The watchpoints might or might not work when re-input, depending on the program context when they are re-entered.

Related Topics

[delete watch — Delete a watchpoint](#)

[disable watch — Disable a watchpoint](#)

[enable watch — Enable a watchpoint](#)

[evaluate — Evaluate or change the value of an expression](#)

[watchreg — Set a watchpoint on a register](#)

For examples, see [Using Watchpoints](#) on page 201.

watchreg — Set a watchpoint on a register

Syntax: **watchr[eg] register_name [, eval[uate] evalexpression][exec debugger_command][, mode m][, thread threadno]**

The **watchreg** command sets a watchpoint on a register.

Argument evaluate evalexpression

Entered with the **evaluate evalexpression** command, **watchreg** sets a conditional watchpoint that causes execution to stop only when *evalexpression* evaluates to true for the register. *evalexpression* must be a conditional expression. (Beware of side effects; for example, if *evalexpression* is *i*++, variable *i* is incremented when the expression is evaluated.) See [Using Expressions](#) on page 222.

Argument exec debugger_command

Optional argument **exec debugger_command** sets a watchpoint on *register_name* such that *debugger_command* is executed every time control stops on the watchpoint. *debugger_command* is any MetaWare debugger command.

Argument mode m to Specify Read or Write Operation

By default, watchpoints stop execution on data write operations only. Use optional argument **mode m** to limit the watchpoint by the type of operation. Replace *m* with one of the following.

- **read** (or **r**) for read (simulator)
- **write** (or **w**) for write
- **rw** (or **wr**) for read/write (simulator)

Argument thread threadno

Use optional argument **thread threadno** to stop execution only if the current thread is *threadno* (replace *threadno* with an integer thread ID). This argument is equivalent to the expression `$tid == threadno`. (*\$tid* is a debugger pseudo-variable that is always set to the current thread.)

After the watchpoint is set, use the regular **disable watch**, **enable watch**, and **delete watch** commands, respectively, to disable, enable, and delete it.

NOTE The simulator does not allow watchpoints to be set on the *icnt* register. When using a simulator, set a property to tell the simulator how many instructions to execute before stopping, as described in [Stopping Execution by Instruction Count](#) on page 117.

The following example sets a watchpoint on register *r7*:

```
watchr r7
```

See also the following:

[**delete watch — Delete a watchpoint**](#)

[**disable watch — Disable a watchpoint**](#)

[**enable watch — Enable a watchpoint**](#)

[**register — Display machine registers**](#)

[**watch — Set a watchpoint or display watchpoints**](#)

while — Execute a command while condition is true

Syntax: **while (expression) command**

Use the **while** command to execute *command* while *expression* evaluates to non-zero.

command is any valid MetaWare debugger command.

expression is any valid C/C++ expression and can contain function calls into the program being debugged (for more information, see [evaluate](#)).

Related Topics

- [if — Execute a command if condition is true](#)
- [evaluate — Evaluate or change the value of an expression](#)
- [Creating while Loops](#) on page 190

Chapter 8 — Options and Toggles

In This Chapter

- [Command-Line Option Reference](#)
- [Toggle Reference](#)

Command-Line Option Reference

This section lists all command-line options alphabetically.

Related Topics

- [Specifying Controls](#) on page 182
- [Quick Options List](#) on page 349

-a4 — Select the ARCTangent-A4 processor series

Option **-a4** selects the ARCTangent-A4 processor series. [-core8](#) is the default core option.

-a5 — Select the ARCTangent-A5 processor series (default)

Option **-a5** selects the ARCTangent-A5 processor series. [-core3](#) is the default core option.

-a6 — Select the ARC 600 processor series

Option **-a6** selects the ARC 600 processor series. [-core1](#) is the only valid core option with this option.

-a7 — Select the ARC 700 processor series

Synonym for [-arc700](#).

-anim_scale=top_speed — Set maximum animation speed

Targets: All

Option **-anim_scale** specifies the maximum animation speed where *top_speed* is any positive number; the default is 500 units. Your CPU determines the highest actual speed at which animation can run.

Increasing *top_speed* increases the speed at which your animation can potentially run, but also ties up more of the resources of your CPU.

You can adjust the animation speed inside the debugger GUI by clicking the **[+]** and **[-]** buttons.

-arc* — Specify the processor series

Use **-arc600** to select the ARC 600 processor series and **-arc700** to select the ARC 700 processor series. To further specify the processor core version, use a [-core*](#) option.

-arc600 | -a6 — Select the ARC 600 processor series

Option **-arc600** selects the ARC 600 processor series. [-core1](#) is the only valid core option with this option.

-arc700 | -a7 — Select the ARC 700 processor series

Option **-arc700** (synonym: **-a7**) selects the ARC 700 processor series. [-core1](#) is the default core option with this option.

See also [enable_exceptions](#).

-auxlo=addr — Specify the low address of an auxiliary-register range

-auxsz=size — Specify the size of the auxiliary-register range

Options **-auxlo** and **-auxsz** set aside a range of memory *size* bytes in length, starting at address *addr*, that denotes the ARC auxiliary registers. These options are deprecated in favor of option **-memaux**.

Debugger operations that display, read from, and write to addresses in the designated range actually operate on the auxiliary registers.

NOTE When you use options **-auxlo** and **-auxsz** to set aside a range of memory denoting the auxiliary registers, the “real” memory associated with that address range is no longer visible during the debugging session. Be sure you set aside a range of memory whose contents you have no interest in.

See also [-memaux](#).

-blast[=file_name] — Reinitialize the FPGA CPU from a file

Option **-blast** tells the debugger to “blast” (reinitialize) the FPGA CPU from a file.

- If you specify **-blast=file_name**, the debugger blasts from the file you specify. If the debugger cannot find *file_name* in the current directory, it searches for the file in the *mdb/cpu* directory.
- If you specify **-blast** with no *file_name*, the debugger blasts from the file specified by configuration variable **BLASTFILE** in the debugger configuration file. For information about modifying the configuration file, see [Modifying the Driver Configuration File](#) on page 227.

For more information about blasting from a file, see [Viewing the ARC 600 MPU Registers](#) on page 146.

-break=addr, size — Specify address and size of breakpoint table

Option **-break** specifies the address and size of the breakpoint table. The following example specifies a breakpoint table of size (hex) 100 bytes that starts at address ff0000:

`-break=0xff_0000,0x100`

C and C++ programs typically do not need to specify a breakpoint table explicitly, because they are already linked with an internal breakpoint table. If a program is not linked to a breakpoint table, the debugger builds a table for it in high memory, unless you override this behavior with option **-break**.

You can use option **-break** to set a breakpoint table for both the embedded processor and the simulator.

NOTE The compiler’s default link automatically links in object **breakp.o**, which provides the breakpoint table.

If your executable is larger than 2 megabytes, you must link in multiple copies of **breakp.o**, because a breakpoint table must exist within 2 megabytes of any given breakpoint.

The **breakp.o** object resides in the MetaWare C/C++ library directory.

-c=command — Pass commands to the command processor

Targets: All

Option **-c** passes *command* to a command processor dedicated to executing debugger system commands.

You can specify multiple **-c** options. If you add the *command* of each **-c** option to the **ARGS=** line in your debugger configuration file, they will take effect every time you run the debugger. You can also put the commands in a file *filename*, then specify **-c="read filename"** when invoking the debugger.

-chipinit=filename — Configure the target board when the debugger initializes communication

Option **-chipinit** configures your target board when the debugger initializes communications with the board. The debugger evaluates the commands found in file *filename*, which represents the chip initialization (**chipinit**) file. Generally, commands in a chip initialization file are intended for booting the processor, configuring memory, and initializing the board.

Commands in the chipinit file are performed prior to downloading code and data. If you want a set of commands to be executed after reset/download and prior to execution of target code, use the command-line option:

```
-cmd="read cmdfile"
```

where *cmdfile* contains any list of debugger commands.

We recommend that you structure the **chipinit** file in the following order:

1. Specify properties about the target that the debugger will need to know, such as the special-purpose registers that you want the debugger to display.
2. Define a memory map for your target hardware. Describe all known address ranges and their properties.
3. Define parameters for the target connection hardware that the debugger uses during a debug session.
4. Initialize the internal CPU state, such as registers.
5. Initialize any critical board device state, such as memory controllers and interrupt controllers.
6. Define macros that make these definitions and initializations convenient.

In the **chipinit** file, you can use the [**prop**](#) command to enable an auxiliary register *nnn* and assign it a name *Name*, as follows:

```
prop use_aux=nnn,Name
```

If you omit the *Name* argument, the debugger provides *auxnnn* as the default register name.

You can use *Name* in evaluation expressions in the chipinit file; for example:

```
prop use_aux=0x20,memctrl  
evalq memctrl=0x6a
```

See also the *install_dir*/mdb/**chipinit** directory.

-cl — Invoke the command-line debugger

Targets: All

Option **-cl** invokes the command-line debugger rather than the debugger GUI. Compare to option [**-gui**](#), which has the opposite effect.

If you specify both option **-cl** and option [**-gui**](#), the last one specified applies.

If you do not specify option **-cl** or option [**-gui**](#), the GUI is the default.

-cmd=debugger_command — Execute the specified command immediately after loading the application program

Targets: All

Option **-cmd** specifies a debugger command to be executed immediately after your application program has been loaded. This happens only once.

You can enclose *debugger_command* in quotes if the command contains spaces. For example, this command directs the debugger to read a long sequence of commands stored in file *myfile*.

-cmd="read myfile"

See also option [loadexec](#).

-core*— Specify processor core version

Targets: ARC

The **-core*** options direct the simulator to assume a program's executable was built for a particular version of the ARC processor core:

- core1** ARCtangent-A5 and later core version 1
- core2** ARCtangent-A5 or ARC 700 core version 2
- core3** ARCtangent-A5 core version 3
- core5** processor core version 5 (pre-ARCtangent-A4)
- core6** processor core version 6 (pre-ARCtangent-A4)
- core7** processor core version 7 (pre-ARCtangent-A4)
- core8** processor core version 8 (ARCtangent-A4)

- By default, the debugger for ARCtangent-A4 assumes core version 8.
- By default, the debugger for ARCompact assumes **-a5** and core version 3.

You can also change the defaults by modifying configuration file *mdb.cnf*. For example, to make version 5 the default, specify option **-core5** on the *ARGS=* line in *mdb.cnf*. See [Specifying Default Values for Command-Line Options](#) on page 228.

-dcache=csz,lsz,w,attrib — Simulate a data cache

Specifies the size and characteristics of a simulated data cache. Data cache misses are listed in the **Memory** window.

The **Registers** window also includes *dchit* and *dcmis* registers when you simulate a data cache.

Parameters are as follows (refer to your processor documentation for legal values):

csz Total cache size in bytes: *512, 1024, 2048, 4096, 8192, 16384*, or *32768*

lsz Line size in bytes: *8, 16, 32, 64, 128*, or *256*

w Cache associativity (number of “ways” to simulate the cache):
1 = direct-mapped cache
2 = two-way set-associative cache
4 = four-way set-associative cache

attrib You can use one or more of the following optional attribute characters.
a = Use “random” replacement algorithm
o = Use “round robin” replacement algorithm

We recommend you compile with toggle `emit_line_records` on to allow source correlation with code without disturbing optimization. Although the optimizer can move statements around, much of the correlation can be useful.

See also option [-icache](#).

-dir_xlation=old,new — Set new source-directory prefix

Syntax

-dir_xlation=old,new [;...] (Windows)

-dir_xlation=old,new [;...] (UNIX/Linux)

Use option `-dir_xlation` to inform the debugger when your source tree is not the same as when the program was compiled. The arguments *old*, *new* are directory strings where *new* replaces *old* if *old* is a prefix of the directory of the file when compiled. You can enter multiple string pairs, separated by semicolons.

UNIX and Linux

To use Windows drive designators (e.g., C:) in the `dir_xlation` string, place a ; as the first character of `dir_xlation`:

`-dir_xlation=;c:/projects/src,/home/user`

-DLL — Specify a DLL named in the configuration file

-DLL=DLLname[, prop1=val1[, ...]] — Specify a third-party DLL and its properties for target communication

Option `-DLL` specifies the use of a third-party DLL file for target communication through the debugger’s Target Access Interface. Use syntax `-DLL=xxx` to specify a DLL other than the default DLL located in the `mdb.cnf` file.

NOTE For ARC targets, the debugger automatically loads the ARC parallel DLL when you specify option [-hard](#) or the ARC JTAG DLL when you specify option [-itag](#). Use option `-DLL` only when you wish to load another DLL by name.

The DLL receives properties *prop1=val1*, and so on; these properties are not sent to any other client. As soon as the DLL is loaded, its **process_property** method is called with each such property.

One use for this option is to set an IP address for a TCP transport DLL so that the DLL knows where to find a server with the “real” DLL. The transport DLL needs a property that specifies the IP address before any of its other methods are called, because the other methods cause TCP traffic to the real DLL to get the result. Example:

`-DLL=tcpDLL,ipaddr=123.45.67.890`

Note Use option **-DLL=DLLname** only if you are using a DLL that you designed or that someone else supplied.

Load DLL *DLLname* as the implementation of *arcint.h*; programs are then controlled by calling methods in *DLLname*. See the *Debugger Extensions Guide* for information on using *arcint.h*.

See also [**-DLLprop**](#).

For UNIX and Linux

On UNIX and Linux hosts, you may also use the following syntax to specify whether or not to export the global symbols of the shared object to other shared objects:

-dll=libname.so[,symbol1s=globa1 | loca1] [,prop1=val1[, ...]]

The following two values are recognized for *symbol1s*:

- *globa1* makes all global symbols within the shared object (DLL) available to other shared objects.
- *loca1* makes no global symbols within the shared object (DLL) available to other shared objects. *loca1* is the default.

-DLLprop=propname=value — Specify a property for a DLL

Specify a property of the form *propname=value* for the DLL. Use this option only if the property *value* contains a comma; otherwise use:

-DLL=DLLname ,propname=value

See [**-DLL=DLLname**](#).

-engine_wait=NNN — Wait before terminating debugger thread

The debugger waits *NNN* milliseconds after a “stop” or “exit” request is made; if the engine doesn’t respond, the debugger kills the thread in which the engine runs and starts a new one. This option is helpful if an attached DLL consistently causes problems such as not returning from a function call.

The default wait period is a small number of seconds. Using option **-engine_wait**, you can specify a smaller time interval before the debugger terminates. This option applies only to the Java GUI version of the debugger.

-fontsize=size — Set initial font size for text in debugger windows

Option **-fontsize** specifies the initial font size for text in debugger windows. The default size is 12 points.

This option is intended for use with the GUI version of the debugger only.

-generate_config=config — Generate configuration for extracting a debugger-engine DLL

Use this option to generate a configuration file for function **engine->specify_argument_file("config")** if you are using the interface described in *scengine.h* to extract a debugger engine and C-callable ISS as described in the *Debugger Extensions Guide*.

Command Line

On the command line, include other options you require:

mdb -generate_config=config -c1 -other_options

The debugger runs, exits, and leaves the configuration file in the directory where you invoked the debugger.

GUI

```
mdb -generate_config=config
```

When using the GUI, you must choose a program for your target in the **Debug a process or processes** window, but the program itself is irrelevant. Configure your options, save, and then exit the debugger.

The debugger leaves the configuration file in the directory where you invoked the debugger.

Related Topics

- *Debugger Extensions Guide*

-goifmain — Execute start-up code and stop at function main() if main() exists

Option **-goifmain** tells the debugger to test for function **main()**, and if it exists, to initialize your program, execute the start-up code, and stop at function **main()**. If the debugger does not find function **main()**, execution stops in start-up code.

If you use option **-goifmain** and function **main()** is present, your process will step over the start-up code at every restart during the debugging session. If you want access to the start-up code, invoke the debugger with option **-nogoifmain**, then enter the command **go main** to step over the start-up code.

See [Commands Reference](#) on page 231 for information about using MetaWare debugger commands.

-gui — Invoke the debugger GUI

Option **-gui** invokes the debugger GUI rather than the command-line debugger. Compare to option [**-cl**](#), which has the opposite effect. If you specify both option **-cl** and option **-gui**, the last one specified applies.

If you do not specify option **-cl** or option **-gui**, the debugger GUI is the default.

-h — List debugger command-line options

Option **-h** prints a list of the MetaWare debugger command-line options to the screen.

-hard — Specify that the application runs on an embedded processor

Option **-hard** specifies that the application runs on the embedded processor. This is the opposite of option [**-sim**](#). Option **-hard** may be used with option [**-DLL**](#) (the order of the two options is unimportant).

The following example starts the command-line debugger, specifies that *my_prog* runs on the embedded board, sets up communication with the board through serial port 1, uses a DLL *xxx*, and downloads and initializes *my_prog*:

```
mdb -dll=xxx -hard -io=ser,1 my_prog
```

Option **-hard** uses file *apipar.dll* in the debugger's *install_dir/bin* directory to communicate with the parallel port. For information on debugging ARC hardware, see [Debugging ARC Systems](#) on page 126. For a JTAG connection, use option [**-jtag**](#).

-hmem=mem_size — Specify memory size of embedded processor

Option **-hmem** specifies the size of memory for the embedded processor. This option tells the debugger that the hardware memory is *mem_size* bytes. The debugger, in turn, tells the debugged program how large the memory is, so the program knows how much heap space is available.

For information about specifying the memory size of the simulator see options [-mem](#) and [-smem](#).

-icache=csz,lsz,w[,attrib] — Simulate an instruction cache

Specifies the size and characteristics of a simulated instruction cache. Parameters are as follows (refer to your processor documentation for legal values):

<i>csz</i>	Total cache size in bytes: 512, 1024, 2048, 4096, 8192, 16384, or 32768
<i>lsz</i>	Line size in bytes: 8, 16, 32, 64, 128, or 256
<i>w</i>	Cache associativity (number of “ways” to simulate the cache): 1 = direct-mapped cache 2 = two-way set-associative cache 4 = four-way set-associative cache
<i>attrib</i>	You can assign one or more of the following optional attributes: <i>a</i> = Use “random” replacement algorithm <i>c</i> = Instruction cache is outfitted with code RAM. To use this attribute, you must also set the “ways” parameter (<i>w</i>), to 1. <i>o</i> = Use “round robin” replacement algorithm

You can use the **Profiling** menus in the **Disassembly**, **Source**, and **Functions** windows to view the number of cache misses for your instruction cache.

The **Registers** window also includes **ichit** and **icmiss** registers when you simulate an instruction cache.

For example, a simulated instruction cache with *csz* = 2048, *lsz* = 8, and *w* = 2 has these characteristics:

Way 0	Way 1
line 0 (8 bytes)	line 0 (8 bytes)
line 1 (8 bytes)	line 1 (8 bytes)
...	...
line 127 (8 bytes)	line 127 (8 bytes)
Total: $128 \times 8 = 1,024$ bytes	Total: $128 \times 8 = 1,024$ bytes

Total cache size is $1,024 + 1,024 = 2,048$ bytes.

Related Topics

- [-dcache](#)
- [-X*](#)
- [Simulating Instruction and Data Caches](#) on page 91

-instrs_per_pass=NNN — Specify the number of instructions executed before simulator attends to a different operation

If this option is not specified, the simulator defaults to address 0. This option applies only to simulators, and is most useful with CMPD. This option specifies the number of instructions the debugger will execute in a simulator before attending to a different operation, such as running a different the simulator for another process.

The default value of *NNN* is 512. If you specify a small value for *NNN* the debugger's overhead will increase significantly for simulation because of the time taken to rapidly switch operations.

You can also specify a value for this option using the command-line using the following syntax:

```
prop instrs_per_pass=NNN
```

-interrupt_base=*address* — Specify the base address of the interrupt vector table for the simulator

If this option is not specified, the simulator defaults to address 0.

-javadir=*dir* — Specify the Java directory

Option **-javadir** specifies the directory where Java is installed.

If you are using Java from a directory different from the one where Java was placed during debugger installation, you must specify option **-javadir** to tell the debugger where to find Java.

NOTE To successfully run the debugger GUI, you must use the same version of Java as supplied with the debugger.

-jtag — Connect to hardware using a JTAG connection

Connect to the hardware using a JTAG connection.

Option **-jtag** is equivalent to using option **-DLL=api jtag** to access **api jtag.dll** in the debugger's **bin** directory.

-jvmheap=*minheapsizem* — Specify minimum heap for host JVM

Use option **-jvmheap=*minheapsizem*** to specify the minimum size in megabytes of the heap available to the Java Virtual Machine on your host system. Note that you must specify the trailing **m**. The default is 60, and rarely needs to be increased.

This option might be useful in situations such as viewing very large images in the **Image** window.

Example:

```
mdb -jvmheap=200m other_options my_exe.out
```

-load={*A1,A3,...An* |+*offset*}— Specify the load addresses or offsets for program segments

When used with *A1,A3,...An* addresses as arguments, option **-load** Specifies the load addresses for the *n* program segments in your executable. The debugger loads the program at those addresses rather than at the virtual addresses specified in the elf header. If you specify more than one load address you must specify exactly as many as there are program segments. If you specify just one load address, it applies to the first program segment in your executable; subsequent program segments are loaded at an address which is the same offset from the first program segment as is specified in the executable. Typically an executable has two program segments: code, then data. You can view them using the **elfdump** utility (the exact name varies by target—see the *ELF Linker and Utilities User's Guide* for details). The following example indicates two program headers; the first is marked XR, which means it is executable, the second is data:

```
Program header #0
type=LOAD, vaddr=0x10000, off=0x74
paddr=0x0, filesz=0x9403, memsz=0x9403, align=4096, flags=<XR>
```

```
Program header #1
type=LOAD, vaddr=0x1a000, off=0x9477
paddr=0x0, filesz=0xb68, memsz=0x11948, align=4096, flags=<WR>
```

When used with *+offset* as its argument, option **-load** specifies an offset address from which to load program segments. Using **-load**, each program segment that was linked at some address *A* is loaded at address *A+offset*. Option **-load** is useful for testing self-relocatable programs.

-loaddll=XXX — Load additional shared object XXX

Option **-loaddll** loads a shared object in addition to one already supplied. This facility is provided for the GUI debugger on UNIX or Linux where a shared object you supply to the debugger may depend on yet another shared object *XXX* you wish loaded.

-loadexec=*debugger_command* — Execute the specified command immediately after loading the application program

Option **-loadexec** specifies a debugger command to be executed immediately after your application program has been loaded. The command is executed each time your application program loads (i.e., even on restarts).

You can enclose *debugger_command* in quotes if the command contains spaces. For example, the following command directs the debugger to read a long sequence of commands stored in file *myfile*.

```
-loadexec="read myfile"
```

See also the following:

Option [cmd](#)

Option [preloadexec](#)

-max_aggregate_display=*max_size* — Specify maximum size of displayed aggregates

The debugger displays the contents of aggregates (arrays and **struct** types) up to a certain size. The *max_size* parameter specifies this size; the default is 1024 bytes for the simulator and 64 bytes for hardware. If you have a slow communications link to the target, you might want to reduce *max_size* to a very small number.

In the debugger GUI, you can see the full contents of an aggregate by clicking the [+] (“nested data” indicator).

In the command-line debugger, evaluating an individual aggregate automatically expands it. As you repeatedly press ENTER, the debugger reads the aggregate elements successively from memory.

-max_poll_delay=*Y* — Maximum interval in milliseconds between polls

When a program is running, the debugger periodically polls the target board to determine if the program has stopped, or if there are any services the program is requesting of the debugger (via hostlink). The poll interval begins at *X*, where *X* is the value specified for [-min_poll_delay](#). If *X* of [-min_poll_delay](#) is not specified, the default is one millisecond. As the program continues running the interval increases quickly until it reaches *Y* (default 1,000). If the program requests a service via hostlink, or if the program stops, the poll interval is reset back to *X* again. You can use these parameters to slow down debugger polling if the target is incapable of responding to poll requests as fast as the debugger is polling.

See also [-min_poll_delay](#)

-max_stack_depth=*N* — Specify maximum stack frames for Call Stack window

Use option **-max_stack_depth** to specify the maximum number of stack frames (*N*) for the **Call Stack** window to display. The default is 500.

Specifying the maximum stack frames can prevent a debugger crash in cases such as the following:

- Call-stack computation confused
- RTOS has not terminated the task stack area

-mem=mem_size — Specify memory size for hardware or simulator

Option **-mem** specifies the size of memory for an embedded processor or the simulator.

Parameter *mem_size* is the upper bound of memory.

If you do not specify any **-mem=mem_size**, the simulator allows any memory address to be used without generating an error. Use **-mem=mem_size** if you want to restrict the valid memory of the simulator to **0-<size-1>**, in which case the simulator will stop with an error, if your program accesses memory outside that specified region.

You can specify the size of hardware and simulator memory individually; see options [**-hmem**](#) and [**-smem**](#).

-mem=M	Specifies the size <i>M</i> of simulator or hardware memory. If you want to specify the size of each individually, use -smem and -hmem .
-hmem=M	Specifies the size of hardware memory only. This tells the debugger how big the hardware memory is. In turn, the debugger can tell the debugged program how big the memory is, so the program knows how much heap space is available.
-smem=M	Specifies the size of simulator memory only

On ARCTangent-A4 core versions less than 7, the debugger places the breakpoint table at the end of the specified memory block.

See also the following:

- Command [**memmap**](#) — Define a memory map
- [**-break**](#) — Specify address and size of breakpoint table

-memalloc=N — Specify maximum memory allocated by the simulator

Use option **-memalloc** to Specify the maximum memory allocated by the simulator; beyond this amount the simulator stops your program. The default is 32 MB.

Compare option [**-mem**](#), which specifies an upper bound above which the simulator will not allow memory accesses. The default upper bound is -1 (no bound), and your program is free to write anywhere in memory, subject to the maximum memory.

You can also use memory maps to prevent allocation outside defined memory areas. See [**memmap**](#) and related commands in [**Chapter 7 — Commands Reference**](#).

-memaux — Map a memory range into auxiliary registers

Syntax: **-memaux=addr, size, auxlo | addr, size, aux_addr_reg, aux_data_reg**

Used with three arguments, this option maps *size* bytes of memory, starting at memory address *addr*, into the ARC auxiliary registers, starting at register *auxlo*. *addr* and *size* must be multiples of 4. Reads from or writes to that region of memory access instead *size/4* auxiliary registers, starting at register *auxlo*.

If you are using registers in pairs, as in early Harvard builds, use the variant with four arguments. The first two arguments are the same; the third argument, *aux_addr_reg*, is the auxiliary register to be used to set the target memory address. *aux_data_reg* is the auxiliary register to be used to read or write data.

Note If you read from or write to this memory, the debugger accesses the auxiliary registers instead; you can no longer “see” the real memory.

This option is useful if you have a large bank of auxiliary registers that exceeds the debugger’s limit of 128 additional user auxiliary registers that can be displayed in the auxiliary register bank (and 64 that can be read), or if you have not declared your auxiliary registers in your assembly program.

For example, **-memaux=0x1000,16,0** sets aside the range of memory 0x1000 to 0x100f to denote auxiliary registers 0, 1, 2, and 3.

-memaux=0x100_000,0x10000,0x1000

maps memory range 0x100_000..0x100_ffff to 64K bytes of auxiliary registers, starting at auxiliary register 0x1000.

You can specify more than one **-memaux** mapping, to use different regions of memory to access different sets of auxiliary registers.

If *size* is 0, the debugger removes any previous **-memaux** specification that has the same *addr*.

Whenever the memory-to-auxiliary-register mapping changes, the debugger prints a mapping table to standard out.

See also [-auxlo](#), [-auxsz](#). These two options are deprecated; instead, we recommend using **-memaux**. Specifying **-auxlo=addr** and **-auxsz=size** is equivalent to specifying **-memaux=addr,size,0**.

-mem_bus_width=width — Specify the width of a simulated memory bus

Option **-mem_bus_width** specifies the width in bytes of a simulated system memory bus.

Note As of this writing, option **-mem_bus_width** specifies only the number of bits in the BURSTSYS register to ensure correct simulation of the burst operations of the XY memory extension.

Argument *width* can be 8, 16, 24, or 32 bytes. The default is 32.

-memext=adrlo,adrhi [,initval] — Add memory to the simulator

Add memory to the MetaWare debugger simulator, starting at address *adrlo* and ending at address *adrhi* (*adrlo* is the first byte of memory and *adrhi* is the last byte of memory). This memory is not subject to the **-mem=** option.

To add non-contiguous memory to the simulator, specify **-memext** multiple times with different values.

Option **-memext** is not currently required, but is provided for backward compatibility, and for use with [-memextinit](#). Optional argument *initval* is the value to which the memory should be initialized, or the string none. Argument *initval* locally overrides any value specified using **-memextinit** for that memory block. If you enter the string none for *initval*, the memory is not initialized to any value.

Example

The following example adds 0x1_0000 bytes of memory to the simulator, beginning at 0x3_0000 and ending at 0x3_ffff.

```
-memext=0x3_0000,0x3_ffff
```

The following example adds a block of memory from 0xfa0000 to 0xfa8000, and leaves it uninitialized:

```
-memext=0xfa0000,0xfa8000,none
```

See also [-memextinit](#).

-memextinit=*value* — Initial value of memory added to the simulator

Specifies the initial value of each word of any simulator memory defined with option **-memext**. The default for *value* is 0xdead_beef. If the number of bytes of memory is not a multiple of 4, the last few bytes are initialized with the low byte of *value*.

NOTE Parameter *value* applies only to memory defined using option [-memext](#) with two arguments; when you use option **-memext** with three arguments, the third argument overrides any **-memextinit** value.

See also [-memext](#).

-memxfersize=*N* — Set memory transfer size

Use option **-memxfersize=N** (where *N* is between 2049 and 65535 bytes) to set the memory-transfer size for image downloads and other binary file-transfer operations to and from memory. This option is useful for speeding up debugging via Ethernet when the JTAG emulator handles large buffers.

-metasim=*options_to_metasim* — Launch the Debugger/RTL cosimulator

Option **-metasim** launches MetaSim, the Debugger/RTL cosimulator, which is available as a separate product. See the *MetaSim User's Guide* for *options_to_metasim* and how to use MetaSim.

-min_poll_delay=*X* — Minimum interval in milliseconds between polls

When a program is running, the debugger periodically polls the target board to determine if the program has stopped, or if there are any services the program is requesting of the debugger (via hostlink). The poll interval begins at *X* (default is one millisecond). As the program continues running the interval increases quickly until it reaches *Y*, where *Y* is the value specified for **-max_poll_delay**. If *Y* is not specified, the default is 1,000 milliseconds. If the program requests a service via hostlink, or if the program stops, the poll interval is reset back to *X* again. You can use these parameters to slow down debugger polling if the target is incapable of responding to poll requests as fast as the debugger is polling.

See also [-max_poll_delay](#).

-mmu — Enable the Memory Management Unit (MMU)

Targets: ARC 700

Option **-mmu** enables the ARC 700 Memory Management Unit (MMU) for virtual memory addressing. You can view MMU contents on the command line by entering **display mmu_contents** or by opening the **MMU contents** window under **Display | Hardware**.

NOTE You must also specify option [-arc700](#) for this option to work.

-mpu — Enable the Memory Protection Unit (MPU)

Targets: ARC 600

Option **-mpu** enables the ARC 600 Memory Protection Unit (MPU). You can view the MPU registers by selecting **Display | Hardware | ARC 600 MPU registers**.

NOTE You must also specify option [-arc600](#) for this option to work.

-nogoifmain — Stop in start-up code; do not run to main()

Option **-nogoifmain** overrides the default behavior of the debugger by halting execution in the start-up code instead of running to function **main()**.

-nohostlink_while_running -- Turn off hostlink polling while the processor is running

When this option is used, the debugger refrains from polling target memory to detect hostlink requests while the processor is running. Instead, this option sets an internal breakpoint in the hostlink run time to alert the debugger that a hostlink request has been posted.

You might wish to use this option when running on hardware and using the debugger's hostlink facility to allow the application to make system calls to your host operating system.

-nooptions — Ignore pre-configured options

Option **-nooptions** directs the debugger to ignore any options stored in **.sc.args** (the options file generated by changing options in **Tools | Debugger Options or Tools | Target-specific options**).

NOTE If you use option **-nooptions**, you must specify all debugger options on the command-line at debugger start-up.

-noprofile — Disable profiling for faster simulation

The MetaWare debugger is enabled by default to display profiling information for applications being debugged. Specifying **-noprofile** eliminates this feature.

The benefit of using option **-noprofile** is that the simulator can execute your application slightly faster.

See [Execution Profiling](#) on page 94 for more information about profiling.

-noproject — Run the debugger without a debug project

Option **-noproject** specifies that the debugger is to run without an associated debug project. Debugger settings are saved in, and restored from, the current working directory.

Related Topics

- Option [-project=project-name](#)
- [Working with Debug Projects](#) on page 41

-notrace — Disable save of instruction-history information

Use this option to turn off saving of information for the **Instruction history -- Simulator trace** window. This makes simulation faster, which might be useful, for example, for lengthy regression runs where trace is not important.

-off=toggle — Turn off a toggle

Option **-off** turns off a toggle. Compare to option [-on=toggle](#).

-OK — Skip the “Debug a process or processes” dialog

Option **-OK** causes the debugger to launch its GUI directly without bringing up the **Debug a process or processes** dialog. Specifying option **-OK** makes start-up faster. Use option **-OK** only if you specify your program name when you invoke the debugger.

-OKN — Skip the “Debug a process or processes” dialog and splash screen

Option **-OKN** causes the debugger to skip the **Debug a process or processes dialog** like option **-OK**, but **-OKN** also skips the MetaWare debugger splash screen.

-on=toggle — Turn on a toggle

Option **-on** turns on a debugger toggle. Compare to option [-off=toggle](#).

For example, this command turns on toggle `Reuse_displays`:

`-on=Reuse_displays`

For information about debugger toggles, see [Toggle Reference](#) on page 298.

-OS=rtos_name — Specify RTOS on which the debugged program runs

Option **-OS** specifies the real-time operating system. The method by which the debugger obtains information about threads or tasks differs from one operating system to the next. To enable the debugger’s multi-threaded debugging features, you must specify which real-time operating system (RTOS) your program runs on, or what RTOS you are debugging.

These examples specify ThreadX for ARCTangent-A4:

```
mdb -a4 -OS=TX  
mdb -a4 -OS=ThreadX
```

This example specifies Nucleus PLUS for ARCTangent-A4:

```
mdb -a4 -OS=NU
```

To specify the Precise/MQX RTOS, you must use **-semint=tadsc.d11**. For instructions on how to load MQX, see [Debugging Precise/MQX RTOS Applications](#) on page 316.

To simplify debugger invocation, you can add the **-OS=rtos_name** specification to the **ARGS=** line in the debugger configuration file.

For more information about real-time operating system support, see [Multi-Tasked RTOS Applications](#) on page 313.

-pdisp=filename — Parse pdisp output

Use option **-pdisp** to parse pdisp output and provide it as a trace component in the **Instruction history -- Simulator trace** window. The attribute *filename* is assumed to be the output of the HDL or RASCAL-based pipeline display modules.

You can see a demo of the debugger parsing pdisp output and providing a trace component in the **Instruction history -- Simulator trace** window. To view the demo, go to *install_dir/demos/mdb/pdisp* and type **demo**.

-perip=perip,unit — Enable a peripheral window

Enable a window for the peripheral called *perip*, and unit number *unit*. ARCTangent-A4 supports the VMAC and UART peripherals and unit numbers from 1 to 7. (This option is not necessary to detect predefined peripherals at slot 0.) The peripheral window shows registers for the peripheral hardware devices. The mechanism that displays the peripheral registers relies on an XML file that resides in *install_dir/mdb/chipinit/perip.xml*.

If you have built an ARC processor with the BVCI memory controller, the peripheral *mem* is supported: **-perip=mem,0** displays the state of the controller.

After you have used this option, you can open your peripheral window by selecting **Display | Hardware**.

You can add your own peripherals by creating an XML file for each peripheral that you want to display. For more information on how to create your own XML-based peripheral windows, see *install_dir/mdb/chipinit/readxml.txt*.

-port=p_address — Specify PC parallel port address for I/O communications

Targets: ARC (Windows hosts only)

Option **-port** specifies the parallel port address *p_address* for I/O communications on a PC (x86-based computer). For example:

-port=0x3f8

If you are using the parallel port on the PC, use option **-port** to set the I/O address used by the debugger to the address of your PC parallel port.

Note The parallel port should be configured in the BIOS setup as EPP, not ECP or any other setting.

-preloadexec=debugger_command — Execute debugger command before loading application

Use option **-preloadexec** to execute a MetaWare debugger command immediately before loading your application. This can be useful if you need to set up some parts of your target before all components are initialized.

Compare option **-loadexec**, which executes the command immediately after loading your application.

You can enclose *debugger_command* in quotes if the command contains spaces. For example, the following command directs the debugger to read a long sequence of commands stored in file *myfile*.

-preloadexec="read myfile"

-project=project_name — Specify debug project

Normally, the debugger defaults to the project used the last time it was invoked. Use this option to specify a different project for the debugger session. When you invoke the GUI debugger, the project you specify appears as the initial selection in the **Choose project:** list in the **Debug a process or processes** window (unless you specify [OK](#)).

To bypass the project specification altogether, specify [**-noproject**](#).

Related Topics

- [Working with Debug Projects](#) on page 41
- Option [OK](#)
- Option [**-noproject**](#)

-prop, -props — Specify target-system properties

These options allow you to specify properties on the command line.

If you specify:

`-prop=a=b,c,d`

this is equivalent to specifying the property `a=b,c,d` with command [**prop**](#):

`prop a=b,c,d`

If you specify:

`-props=a=b,c=d,e=f`

this is equivalent to specifying properties `a=b, c=d, and e=f` with command [**prop**](#):

`prop a=b c=d e=f`

See command [**prop**](#) for more information.

`-prop=code_memory_additional_latency=N`

-prop=__HOSTLINK__=address — Set the address of symbol __HOSTLINK__

Use this option to set the address of the `__HOSTLINK__` symbol. This is useful when debugging stripped files.

-prop=code_memory_additional_latency=N — Specify additional cycles to obtain a word from memory

This option specifies an additional number of cycles it takes to obtain a word from memory. The default is 0.

This option has no effect unless toggle [cycles](#) is on.

-prop=code_memory_throughput=N — Specify the number of cycles to obtain a word from memory

Use this option to specify the number of cycles (*N*) it takes to obtain a word from memory. The default is 1.

This option has no effect unless toggle [cycles](#) is on.

-prop=dcache_pipeline_depth=N — Specify the data cache pipeline depth

Enter the dcache pipeline depth of your processor as N . The ARChitect supports a depth of 2 or 3, although the ISS supports any number.

The default is 2.

This option has no effect unless toggle [cycles](#) is on.

-prop=data_memory_additional_latency=N — Specify additional cycles to obtain a word from memory

This option specifies an additional number of cycles it takes to obtain a word from memory. The default is 0.

This option has no effect unless toggle [cycles](#) is on.

-prop=data_memory_throughput=N — Specify the number of cycles to obtain a word from memory

Use this option to specify the number of cycles (N) it takes to obtain a word from memory. The default is 1.

This option has no effect unless toggle [cycles](#) is on.

-prop=dcra=xx — Specify the data cache replacement algorithm

Use the dcra property to modify the data cache replacement algorithm when using a hardware data cache. This property is useful if you are encountering problems with the data cache. Valid values for xx are:

rr round robin (default value)

ra random

la leave alone, which means do not change the replacement algorithm

See also [dcache](#).

-prop=double_format=F | float_format=F — Set display format

By default, the debugger uses the C format specifier %20e to display **doubles** and 10%e to display **floats**.

Use **-prop=double_format=F** or **-prop=float_format=F** to specify a new format F for use in all contexts. For example, **-prop=double_format=%20.15e** gives you more digits after the “.” than %e typically does. The format F must be a standard C format string.

CAUTION Specifying a nonstandard format might cause the debugger to crash.

You can also use the [set](#) command.

-prop=ident=value — Specify value for IDENTITY register

Use this property to specify the *value* for the IDENTITY register of the ARC ISS to make the ISS more closely match your hardware.

-prop=rblast=*my_remote.xbf* — Blast using file on SCIT server

Use **-prop=rblast=*my_remote.xbf*** to blast an ARCangel3 development board using a file residing on the SCIT server. For more information, see [Remote Blasting an ARCangel3](#) on page 124.

-prop=uncached_base=*N* — Specify base of uncached region

Targets: ARC 700 with MMU

Specify the base address *N* of the uncached region. The default is 0xc000_0000.

-prop=uncached_size=*N* — Specify size of uncached region

Targets: ARC 700 with MMU

Specify the size *N* of the uncached region. The default is 1 G.

-rascal — Connect the debugger to an RTL simulation using RASCAL

Use option **-rascal** to connect the debugger to an RTL simulation of an ARC processor using RASCAL. For more information, see your RASCAL documentation.

-run — Download and execute a program with no debugger output

This option makes your program operate as though it is running standalone; that is, the debugger downloads and executes your program, but does not produce any output except what your program produces.

This option turns on toggles [binary_stdin](#) and [binary_stdout](#) for applications that read binary data from `stdin` or write binary data to `stdout`, for example:

```
mdb application.out -run < input
```

```
mdb application.out -run > output
```

-semint=*DLLname*[,*prop1=val1*[,...]] — Specify a semantic inspection DLL and properties

Specify a DLL that implements a semantic inspection object, which allows you to view and interact with the state of the program being debugged. The DLL receives the properties you specify with *prop1=val1*, and so on.

You can specify more than one occurrence of the same DLL and specify different properties to be sent to each; for example:

```
-semint=myfile,a=b -semint=myfile,c=d -semint=myfile,e=f
```

You can also specify multiple properties for a DLL. The following example declares two properties and values, `trace_my_sim=1` and `method=2`, for the DLL `my.dll`:

```
-semint=my.dll,trace_my_sim=1,method=2
```

See also the following:

- [**-DLL=DLLname**](#)
- [*MetaWare Debugger Extensions Guide*](#)

-set_optset=*NAME* — Specify a named option set as the current option set

Directs the debugger to make previously created option set *NAME* the current option set. If *NAME* = `local`, options are read from file `.sc.args` in the debugger's current working directory; otherwise, options are read from file `install_dir/mdb/options/NAME`.

After storing *NAME* in file *install_dir*/mdb/options/current, the debugger exits with the message, “Current options set is now *NAME*.”

See also option [-nooptions](#).

-showtime[=NNN] — Display how long a command took to execute

After a command takes considerable time, the debugger displays the time that it took to execute. *NNN* is by default 100 milliseconds. Increase this value to reduce how often the time is shown.

TIP You can set the time interactively using the command
prop showtime=NNN

-side_effect_auxregs=list — Designate aux registers as having side effects

Use option **-side_effect_auxregs** to designate auxiliary registers as having side effects and prevent the debugger from reading them. Argument *list* is a list of one or more auxiliary-register numbers, separated by commas. The debugger does not read these registers and instead displays `sideefct` for them.

Example: `-side_effect_auxregs=400,0x70f,0x600`

You can force the evaluation of a register with side effects by using the command

[evaluate \\$reg\(64+N\)](#)

where *N* is the register number. (Auxiliary-register numbering in the ARC interface begins at 64, so that 64 denotes auxiliary register 0.)

See also [-side_effect_regs](#).

-side_effect_regs=list — Designate core registers as having side effects

Use option **-side_effect_regs** to designate core registers as having side effects and prevent the debugger from reading them. Argument *list* is a list of one or more core register numbers, separated by commas. The debugger does not read these registers and instead displays `sideefct` for them.

Example: `-side_effect_regs=37,38,0x30`

You can force the evaluation of a register with side effects by using the command

[evaluate \\$reg\(N\)](#)

where *N* is the register number.

See also [-side_effect_auxregs](#).

-sim — Specify simulator as default

Option **-sim** specifies that, by default, the application runs on the MetaWare debugger simulator. This is the opposite of option [-hard](#).

If you specify both option **-hard** and option **-sim**, the last option specified applies.

-simext= DLLname1[,DLLname2[,...]] — Specify simulator-extension DLLs

-simextp= DLLname[,property1=value1[,...]] — Specify a simulator extension DLL with properties and values

The **-simext*** options specify DLLs (or shared libraries) for implementing extensions to the simulator. You can specify either option multiple times. The **-simextp** form allows you to pass properties and their values to the **process_property()** method of the DLL interface object during initialization.

The following example loads the simulator extension DLL `myd11` and passes two properties (`trace` and `connect`) and their values (1 and `0x4bf`):

```
-simextp=myd11,trace=1,connect=0x4bf
```

By default, the debugger includes a set of standard DLL extensions that are included in the debugger distribution. You can also provide your own extension files; see the *MetaWare Debugger Extensions Guide*.

See also the [-X*](#) options, which enable the simulation of specific processor extensions; and option [-sim](#), which specifies that your application runs on the simulator.

Related Topics

- [-simext*](#)
- `install_dir/mdb/inc simext.h`
- `install_dir/etc/dll/readme.txt`
- Examples in directory `install_dir/etc/dll/simext`
- *MetaWare Debugger Extensions Guide*

-simext_dlls=path — Specify path to simulator-extension DLLs

Use option **-simext_dlls** to specify the path to simulator-extension DLLs that the debugger is looking for.

For example, if you specify a floating-point extension, use this option to specify the location where the debugger can find the DLLs generated by ARChitect.

-smem=mem_size — Specify simulator memory size

Option **-smem** specifies the size of memory for the simulator. For information about specifying the memory size of the embedded processor, see options [-hmem](#) and [-mem](#).

-startup=file_name — Specify start-up file

Option **-startup** passes the name of a start-up file to the debugger. When the debugger creates a new command processor, it reads start-up file `.scrc` by default. Use option **-startup** to specify a different start-up file.

You can also provide `null` as an argument to option **-startup**, which tells the debugger not to read any start-up file.

The start-up file is a script that you must write yourself and install in your working directory. It can contain any legitimate MetaWare debugger commands.

See [Chapter 7 — Commands Reference](#) for information about MetaWare debugger commands.

-std|in|out|error=*file_name* — Redirect target's standard in, out, or error

Use **-stdin**, **-stdout**, **-stderr** to redirect your target's **stdin**, **stdout**, or **stderr** to *file_name*. This option works only if you are using the MetaWare hostlink libraries. The files are opened in binary mode.

This option lets you use **stdin**, **stdout**, or **stderr** for debugger commands; when you use it, the program takes its input from a file or sends its output to a file.

Example

```
mdb prog.out -stdin=infile -stdout=outfile
```

-timeout=*num_times* — Specify number of times to retry “get status” on the parallel port

Option **-timeout** specifies the number of times the debugger retries a “get status” on the parallel port. The default is 100.

-win=*file_name* — Specify saved window layout to load

Option **-win** passes the name of a saved window layout to the debugger.

By default, the debugger GUI opens the window layout for the most recently used project. Use option **-win** to tell the debugger to open a different saved window layout. See also the options for saving and loading window layouts on the **Windows** menu.

You can also provide *null* as an argument to option **-win**, which tells the debugger not to read any saved-windows file.

-X* — Enable specific ARC simulator extensions

By default, the ARC instruction set simulator simulates a base-case processor with no extensions. The following command-line options enable the simulation of processor extensions. See also [Simulating DSP Extensions](#) on page 132.

Note The Instruction Set Simulator has fully implemented extensions for ARC systems. Some extensions available in the simulator may not be available for ARCTangent-A5 or later hardware systems. Refer to your processor documentation for information about support for extensions for ARCTangent-A5 and later hardware.

Option	Extension to simulate
-Xadds	Saturating add/subtract extension; enables simulation of ADDS and SUBS instructions. Must be used with one of the XMAC extension options.
-Xbarrel_shifter	Barrel-shifter extension.
-Xbs	
-Xcache_display	This option is no longer required, as the debugger automatically detects hardware cache and loads the related windows.
-Xcrc	Enables simulation of the variable polynomial CRC instruction for ARCTangent-A5 and later.

Option	Extension to simulate
NOTE	When you use option -Xdpfp to debug with the instruction-set simulator, you must inform the debugger of the location of the support DLLs generated by ARChitect during the “Build test scripts” phase. Otherwise, the debugger issues an error message such as <code>-Xdpfp_compact: can't locate simulator extension dpfpcompact_arc700_iss</code> because you have not indicated where the extension DLLs are. When generating the DLLs, ARChitect tells you their location with a message such as <code>Writing unprocessed file 'C:\ARC\Projects\project8\build\lib\win32\dpfpcompact_arc700_iss.dll'</code> . Tell the debugger this path using -simext_dlls=path or set the environment variable SIMEXT_DLLS=path .
-Xdsp_packa	DSP Pack A extension package.
-Xdvbf	Dual Viterbi butterfly; enables simulation of the dual-word Viterbi butterfly instruction (ARCtangent-A4 only). To enable all ARCtangent-A4 telephony extensions with a single option, use -Xtelephony . For the Viterbi butterfly instruction on ARCtangent-A5, use option -Xvbfdw .
-Xea	Extended arithmetic extension; enables simulation of the extended arithmetic instructions (DSP Version 3.1) for ARCtangent-A5 and later.
-Xins_tel	Telephony; enable simulation of the ARCtangent-A4 telephony extended arithmetic extensions. To enable all ARCtangent-A4 telephony extensions with a single option, use -Xtelephony .
-Xinterrupts	Interrupt vector extension for the simulator, allowing extension vectors 16 through 31
-Xlib	Equivalent to specifying all of these options: -Xbarrel_shifter -Xmin_max -Xmult32 -Xnorm -Xswap
-Xmin_max	MIN and MAX instructions (ARCtangent-A4 only).
-Xmpy	General-purpose multiply extension (ARC 700 and later)
-Xmul_mac	MUL/MAC extension (XMAC version 1).
-Xmul32x16	32x16 multiply instructions.
-Xmult32	32-bit multiplier extension.
-Xmult32d	Support multiply extension with the XMAC extension block (remapping of the <code>mlo</code> multiplier register).
-Xnorm	NORM instruction.
-Xscratch_ram	Scratch RAM extension (DSP MEM version 1).
-Xsliding_pointers	Sliding pointers extension (DSP MEM version 1). Selecting this extension also automatically selects the scratch RAM extension.

Option	Extension to simulate
NOTE	When you use option -Xspfp to debug with the instruction-set simulator, you must inform the debugger of the location of the support DLLs generated by ARChitect during the “Build test scripts” phase. Otherwise, the debugger issues an error message such as -Xspfp_compact: can't locate simulator extension spfpcompact_arc700_iss because you have not indicated where the extension DLLs are. When generating the DLLs, ARChitect tells you their location with a message such as Writing unprocessed file 'C:\ARC\Projects\project8\build\lib\win32\spfpcompact_arc700_iss.dll' . Tell the debugger this path using <u>-simext_dlls=path</u> or set the environment variable SIMEXT_DLLS=path .
-Xswap	SWAP instruction.
-Xtelephony	Telephony; enables simulation of all ARCTangent-A4 telephony extensions. Equivalent to specifying options -Xxy_tel , -Xxmac_d16_tel , -Xins_tel , and -Xdvbf .
-Xtimer0	Programmable timer extension (required for execution profiling): For timer 0
-Xtimer1	For timer 1
	Displays are available under Display Hardware , or on the command line using the command display timer0 or display timer1 .
-Xvbfwd	Enables Viterbi butterfly dual-word instruction (ARCTangent-A5 only). For the dual Viterbi butterfly instruction on ARCTangent-A4, use option <u>-Xdvbf</u> .
-Xvraptor	Enable support for the VRaptor SIMD architecture (ARC 700 only)
-Xxmac	Use -Xxmac only on hardware; on simulator, use block extension: 16x16 mode
-Xxmac_16	dual 16x16 mode
-Xxmac_d16	24x24 mode
-Xxmac_24	Enables the XMAC window. Use these options only if your hardware supports XMAC. See XMAC Simulation on page 132 for more information.
-Xxmac_d16_tel	Telephony; enables simulation of the ARCTangent-A4 Telephony dual 16x16 XMAC extension. To enable all ARCTangent-A4 telephony extensions with a single option, use <u>-Xtelephony</u> .
-Xxy -Xxybanks=value -Xxysize=value	XY memory extension (DSP MEM version 2). Includes the XY memory window. Use these options only if your hardware supports XY memory. For information on acceptable parameters, see XY Memory Simulation on page 133.
-Xxy_tel	Telephony; enables simulation of the ARCTangent-A4 telephony XY memory extension. To enable all ARCTangent-A4 telephony extensions with a single option, use <u>-Xtelephony</u> .

-Xnodetect — Do not autodetect cache and peripheral hardware

By default, The debugger auto-detects caches and peripheral hardware that is in slot 0 and loads the relevant windows. Specify this option to turn off autodetection if autodetection causes unforeseen difficulties.

-Xxylsbasesx=addr — Set the base address of X memory**For ARCTangent-A5 and Later**

Set the base address of X memory in the load/store address space to *addr*.

-Xxylsbasey=*addr* — Set the base address of Y memory

For ARCtangent-A5 and Later

Set the base address of Y memory in the load/store address space to *addr*.

Toggle Reference

This section lists all toggles alphabetically.

Related Topics

- [Specifying Controls](#) on page 182
- [Quick Toggles List](#) on page 353

all_regs_volatile — Prevent register caching by the debugger

Default: Off

Turn on toggle `all_regs_volatile` to prevent register caching by the debugger. This is useful for reading or writing registers with side effects while the target program is not running.

bpreg — Enable a hardware breakpoint for the ISS

Default: Off

When debugging on the instruction-set simulator, turn toggle `bpreg` on or off to enable or disable hardware breakpoints you have set.

binary_stdin — Set debugger stdin to binary mode (Windows hosts)

Default: Off unless option `-run` is specified

Turn on this toggle to set the debugger's `stdin` to binary mode. This setting is useful to prevent 'CRLF' sequences (0xd0a) from being reduced to 0xa when target applications read binary data from `stdin`. See option [-run](#) for an example.

Option [-run](#) turns on this toggle automatically.

This toggle has no effect on UNIX or Linux hosts.

See also option [`-std*=file_name`](#).

binary_stdout — Set debugger stdout to binary mode (Windows hosts)

Default: Off unless option `-run` is specified

Turn on this toggle to set the debugger's `stdout` to binary mode. This setting is useful to prevent 'CR' characters (0xd) being inserted for every 'LF' character (0xa) written by target applications that write binary data to `stdout`. See option [-run](#) for an example.

Option [-run](#) turns on this toggle automatically.

This toggle has no effect on UNIX or Linux hosts.

See also option [`-std*=file_name`](#).

branch_self — Allow branch instruction to go to the same location

Default: Off

Targets: ARCTangent-A5 and later

Turn on toggle `branch_self` to allow a branch instruction to go to the same location when using the simulator. When this toggle is off, you can catch the mis-execution of “0,” which is a branch to self in the ARCompact instruction set.

`brk_in_user_mode` — Allow BRK instructions to halt processor in user mode

Default: On

Targets: ARC 700

If this toggle is on, the debugger sets the UB bit (bit 28) in the debug register during download, allowing **BRK** instructions to halt the processor in user mode (otherwise the ARC 700 throws an exception).

If you set this toggle's value to 2 (-prop=`brk_in_user_mode=2`), the debugger sets the UB bit each time it writes the debug register.

`bypass_aom` — Hide internals when loading overlays

Default: On

Targets: ARC

When this toggle is on, the debugger does not display actions of the ARC Automated Overlay Manager. AOM functions do not appear in the call-stack trace, and any attempt to instruction step into the AOM code “fast forwards” directly to the overlay function.

Turn this toggle off if you need to instruction step into the AOM code. The overlays themselves remain active and visible.

Related Topics

- [Debugging with Overlays](#) on page 130

`cache_target_memory` — Cache memory contents

Default: On

Toggle `cache_target_memory` causes the debugger to avoid reading the same target memory twice between execution events. This increases debugging speed by reducing traffic with the target. Turn off toggle `cache_target_memory` if you have volatile memory that changes each time it is read.

`can_step` — Use breakpoints to step

Default: On

Toggle `can_step` allows the debugger to use breakpoints to step through a program. Turn it off to prevent stepping (such as prevent the debugger from stepping into an interrupt). This toggle is useful for substituting breakpoints for stepping if the normal ability of hardware stepping is broken for a target connection.

`check_dcache_lock_bug` — Detect unlocking of cache lines due to load collision

Default: On

This toggle detects an unusual data-cache bug where locked lines become unlocked if a full dcache flush occurs followed by a load that forces a collision with the locked line.

To avoid this bug, turn on [workaround_dcache_lock_bug](#).

check_endian — Check endianness of target against application

Default: On

This toggle causes the debugger to check the endianness of the ARC target to be sure it matches the endianness of the application. The top nybble of the MEMSUBSYS BCR must be 0 for little-endian, and non-zero for big-endian. Turn off this toggle to disable this check.

check_hostlink_immediately — Check for hostlink activity after each instruction (ISS only)

Default: Off

Turn on toggle `check_hostlink_immediately` to check for hostlink activity after each instruction.

This toggle slows down simulation, but prevents hostlink loops from consuming entries in the instruction trace and so allows matching of instruction traces exactly with the ARC 600 CAS, which stops execution for hostlink until the debugger services the request.

check_kb — Enable keyboard input during execution (Windows hosts)

Default: On

Pressing a key in the **Command Prompt** window where the debugger was started (typically the command-line debugger) causes the debugger to stop execution and return the host-system prompt. Turn off this toggle to ensure that accidental keyboard input does not interrupt execution.

NOTE .Pressing CTRL C repeatedly still aborts the debugger when `check_kb` is off.

clear_regs — Clear registers after download

Default: Off

Turn on toggle `clear_regs` to clear registers after code is downloaded to the target hardware.

Use this toggle to ensure the registers on your target are clear before starting your debug session.

cr_for_more — Enable carriage return for more display

Default: On

When a command returns more than one screen of output, the debugger pauses display at the end of the screen with the following message:

----->ENTER for more, 'x' ENTER to stop...

Turn off toggle `cr_for_more` to disable this behavior and continue display until the output is complete. This can be useful when logging commands and results to a file.

Related Topics

- command [log — Log commands and results to a file](#)
- [Example 41 Script File for Dumping Profiling Counters and Cache Analysis](#) on page 219

cycle_step — Step cycles instead of completed instructions (hardware only)

Default: Off

By default the debugger instruction-steps version 7 or later of the processor core. Turn on toggle `cycle_step` to step cycles rather than completed instructions. The main effect of stepping cycles is that you do not see registers updated until the update occurs in the pipeline.

Note The simulator does not support toggle `cycle_step`.

cycles — Estimate CPU cycles

Default: Off

Targets: All ARC targets except ARC 700

When this toggle is on, the ARC simulator estimates CPU cycles. Several new registers are created to contain data related to estimating cycles. For details, see [Using the Cycle-Estimating Simulator \(CES\) on page 150](#).

Cache thrashing reduces the accuracy of the estimation, so do not rely on the numbers when the caches are thrashing. (For automated cache analysis including thrashing analysis, see [Analyzing Cache Performance on page 94](#).)

Cycle estimation is not available for code fetch where there is no icache present. The assumption is that without an icache, you are employing single-cycle access code ram that essentially takes no time.

Cycle estimation slows down the simulator by about 10%.

data_refs — Simulator counts number of times instruction is executed

Default: Off

Turn this toggle on to obtain the number of times each load or store instruction is executed when debugging on the simulator. The resultant data from `data_refs` will show you information useful for profiling. After enabling this toggle, you can view the resultant data in a **Profiling**, **Source**, or **Functions** window.

deadbeef — Initialize memory to 0xdeadbeef before program download (ISS only)

Default: On

By default, the MetaWare debugger simulator initializes memory to 0xdeadbeef before program download. To turn off the default behavior, specify this command when you invoke the debugger:

-off=toggle=deadbeef

Turning this toggle off can help when you allocate a large amount of simulator memory.

If you are running ARCompact code (ARCTangent-A5 or later), turn on toggle `deadbeef` to stop the simulator on a load of 0x7ffe7ffe, which is the value initialized to memory. The load of 0x7ffe7ffe is the ARCompact equivalent of 0xdeadbeef in ARCTangent-A4.

Note Toggle `deadbeef` is relevant to the debugger simulator only.

delay_killeds — Count times delay-slot instructions are not executed, for profiling

Default: On

When toggle `delay_killeds` is on, the simulator counts the number of times each instruction in a delay slot fails to execute due to the setting.

See also [icnts](#), [killeds](#)

delay_reg_write — Delay posting a register write value to the CPU

Default: Off

When you write a register with the debugger (with command `evaluate reg = value`) the value is posted to the CPU when you resume execution.

You can delay this posting by turning on toggle `delay_reg_write`.

detect_uninitialized_data — Stop on uninitialized data

Default: Off

When this toggle is on, the simulator stops when the application being debugged references data that the application did not previously write (or that the debugger did not download). Execution stops immediately after the reference, which is executed normally.

You can then resume running or stepping the application.

Use of this toggle slows execution by about 5%, but it can help in finding bugs.

Some functions, such as `realloc()`, justifiably read uninitialized memory. To accommodate such behavior, use toggle [dud_quiet](#).

dud_quiet — Continue on uninitialized data but display message

Default: Off

Use toggle `dud_quiet` with toggle [detect_uninitialized_data](#) to continue execution after referencing uninitialized data (but display a message).

To disable both halting and message display on uninitialized data but record the data-access information, use the following command:

`prop dud_quiet=2`

Turning off toggle `dud_quiet` cancels the `prop` specification.

download — Download application being debugged to target memory

Default: On

Toggle `download` instructs the debugger to download the application being debugged to target memory (hardware or simulator). If the processor is running, the debugger attempts to stop it before downloading.

Turn this toggle off when the program is already present on the target processor, as when debugging a running target in the field or debugging code in flash. Turning this toggle off also prevents the debugger from making adjustments on the target to run a newly loaded program, such as poking state registers.

enable_bypass — Enable bypass mode in cache (hardware only)

Default: On

This toggle enables bypass mode in the cache. To disable bypass mode, turn this toggle off. Disable bypass mode if you are having difficulty with the cache.

Note This toggle is intended for use with hardware only, not the simulator.

enable_dcache — Enable data cache upon board reset (hardware only)

Default: On

This toggle enables data cache upon board reset. To disable data cache, turn this toggle off. Disable bypass mode if you are having difficulty with the cache.

Note This toggle is intended for use with hardware only, not the simulator.

enable_exceptions — Take exception rather than halt processing

Default: Off

Targets: ARC 700

If this toggle is on, take exceptions for conditions in which the ISS would otherwise halt processing, such as invalid instructions or memory accesses. The toggle should be on for testing operating systems that detect and/or recover from such conditions. The toggle should typically be off for standalone programs, where it is preferable to halt execution rather than go to an unprepared exception vector.

error_box — Send selected errors to error box

Default: On

The GUI debugger reports some errors in an error box. Turn off this toggle to send the errors to the **Debug Console** instead.

examine_one — Show only one entry at a time in Examine window

Default: Off

When this toggle is on, the **Examine** window shows one entry, rather than accumulating multiple entries.

fast_load_return — Write data returns from loads during writeback of computed results

Default: Off

If this toggle is on, cycle estimation assumes a 4-port register file model, which allows data returning from loads to be written to the register file at the same time an instruction is writing a computed result back to the file. If this toggle is off, returning data stalls instructions that are trying to write back results.

fill_halts — Fill memory with halt instructions

Default: Off

Turning this toggle on will fill memory with halt instructions. Halt instructions in memory can provide aid when debugging programs that run without stopping. The halt instruction will catch the program from running “wild.” To fill memory with halt instructions, turn on toggle **fill_halts**.

flush_dcache — Flush data cache when debugger performs a memory transaction in the program's data area

Default: On

If the CPU is stopped, toggle `flush_dcache` flushes the data cache when the debugger performs a memory transaction in the program's data area, and the debugger invalidates the cache on a write transaction. To disable this behavior, turn the toggle off.

NOTE Turning this toggle off can cause unpredictable results in variable windows, memory windows, and computation of previous stack results.

flush_pipe — Clear target-processor pipeline before downloading

Default: On

Toggle `flush_pipe` clears the target processor's pipeline before downloading a program, to eliminate any unfinished instructions remaining there.

fujitsu_fast_serial_hostif — Use Fujitsu high-speed serial communication protocol

Default: Off

Turn on toggle `fujitsu_fast_serial_hostif` to use the host communications protocol defined for Fujitsu's five-wire serial interface communications module. Communications from the host side are conducted using the parallel port (to maintain bandwidth), while on the target side a five-wire high-speed clocked serial interface is used (to minimize pinout). An external pod converts between these two physical interfaces.

hostlink — Process host-link system calls

Default: On

When toggle `hostlink` is on, the debugger checks the memory at address `__HOSTLINK__` (if the symbol `__HOSTLINK__` appears in the program), in order to process host-link system call requests from the target application. When this toggle is off, this checking (and associated host-link service) is not performed.

TIP If you are debugging a stripped file, you can set the address of `__HOSTLINK__` using `-prop=__HOSTLINK__=address`.

icnts — Count instruction executions for profiling

Default: On

When toggle `icnts` is on, the simulator counts the number of times each instruction at every address is executed. This allows for useful profiling.

On the Debugger GUI only, to access to the resultant data from `icnts`:

1. Start the debugger and specify `-on=icnts`.
2. Select **Display | Disassembly**.

NOTE You can also select **Display | Source**, or **Display | Global Functions**.

3. Select **Profiling | icnt display** to view an **Instruction counts** window.
4. Select **Profiling | icnt** to see instruction executions in the **Disassembly, Source, or Functions** window.

See also [delay_killeds, killeds](#).

ignore_zero_debug_entities — Ignore functions and variables whose debug address is zero

Default: On

Ignore functions and variables whose address is set to 0 in debug information.

When you optimize a program by removing unreferenced functions or variables, it is not possible to remove those entities' debug information; instead, the debug information states that the removed entity starts at address 0. Turn this toggle off to debug entities with address 0 if you have not optimized your program by removing unreferenced functions or variables.

NOTE Use compiler toggles `Each_function_in_own_section` and `Each_var_in_own_section` with linker option `-zpurge` to remove unreferenced functions and variables.

image_window — Enable image display

Default: Off

Turn on toggle `image_window` to enable display of images from memory. For more information, see [Viewing an Image from Memory](#) on page 84.

include_local_symbols — Direct ELF reader to include local symbols

Default: Off

When toggle `include_local_symbols` is on, the debugger directs the ELF reader to include global symbols and local symbols.

By default, toggle `include_local_symbols` is off, so the ELF reader includes only global symbols.

init_exception_vectors — Pre-initialize exception vectors in low memory

Default: On

Targets: ARC 700

This toggle pre-initializes exception vectors in low memory. Then if you do not provide a handler for an exception, the exception stops the processor and the debugger informs you that you hit an exception unintentionally.

You might wish to turn this toggle off if your hardware has no low memory.

interrupt_extension — Permit interrupts 16–31

Default: Off

By default, ARC targets may use interrupts 0 through 15. Turn on toggle `interrupt_extension` to allow interrupts 16 through 31 as well.

invalid_instruction_interrupt — Interrupt instead of halt on a bad instruction

Default: Off

By default the simulator halts on a bad instruction. If you specify:

`-on=invalid_instruction_interrupt`

the simulator interrupts to vector 2 instead of halting.

invalidate_dcache — Detect and invalidate dcache prior to download

Default: On

This toggle detects and invalidates the data cache prior to downloading code to the target. See also

[`-prop=dcr=xx`](#).

java_trap — Java traps Windows faults (Windows GUI only)

Default: On

When you debug using the GUI, Java traps all Windows faults that occur in the debugger engine, and shuts down the debugger. This is a problem if you wish to debug code that is causing the fault (such as a simulator extension or a semantic inspection interface you have added to the debugger). Turn toggle `java_trap` off to disable Java's trap handling so that you can debug debugger components.

killeds — Count number of times instructions not executed, for profiling

Default: On

When toggle `killeds` is on, the simulator counts the number of times each instruction at every address fails to execute due to flag settings. This allows useful profiling.

See also [`delay_killeds, icnts`](#).

ldst_from_code — Stop on code memory load or store

Default: Off

When toggle `ldst_from_code` is off, the simulator stops execution whenever a load from or store to code memory is encountered. Any program segment in the executable marked 'XR' (executable/read) is assumed to be code memory. You can use this option to check your code if you are targeting a build with closely coupled memory (a Harvard architecture).

load_at_paddr — Load program segments at physical address

Default: On

Turn off toggle `load_at_paddr` to load program segments at their virtual address rather than their physical address.

loadbeef — Stop when a program loads 0xdeadbeef from memory (ISS only)

Default: Off

To direct the debugger to stop when a program loads 0xdeadbeef from memory (using a fullword load), specify this command when you invoke the debugger:

[`-on=toggle=loadbeef`](#)

Turning this toggle on can help track down bugs in your program.

If you are running ARCompact code (ARCtangent-A5 or later), turn on toggle `loadbeef` to stop the simulator on a load of `0x7ffe7ffe`, which is the value initialized to memory. The load of `0x7ffe7ffe` is the ARCompact equivalent of `0xdeadbeef` in ARCtangent-A4.

Note Toggle `loadbeef` is relevant to the simulator only.

memory_exception_interrupt — Interrupt instead of halt on a bad memory reference

Default: Off

By default the simulator halts on a bad memory reference. If you specify `-on=memory_exception_interrupt`, the simulator interrupts to vector 1 instead of halting.

minimal_interface — Debugger implements some target-interface functions

Default: Off

Targets: Single ARC targets (not multi-ARC)

This toggle is provided for building and debugging target-interface extensions. Turn on `minimal_interface` to have the debugger implement the `step()` and `run()` functions in the interface.

note_box — Send messages to text box

Default: On

The debugger GUI sometimes notes conditions worthy of interest (such as an inability to set a hardware watchpoint) in a text box in addition to the **Debug Console**. Turn off this toggle to disable the box display and send the message only to the **Debug Console**.

oldbp — Use pre-BRK breakpoints

Default: Off

Targets: ARCtangent-A4

Turn on toggle to use the breakpoint method used prior to the BRK instruction. This is useful if you wish to omit the BRK instruction and use branch to breakpoint table breakpoints.

prefer_soft_bp — Try software breakpoint first

Default: On

When toggle `prefer_soft_bp` is on, the debugger tries to set a software breakpoint before trying a hardware breakpoint.

pretty_print — Reduce output of C++ functions

Default: On

Sometimes the length of unmangled output from C++ functions can be overwhelming in some windows. Switch this toggle off to reduce the size of unmangled output from C++ functions in windows.

program_zeros_bss — Do not download BSS

Default: Off

This toggle is useful for reducing download time when the run-time boot or simulator handles BSS zeroing. When you use the MetaWare C/C++ run time, the default initialization of BSS is contingent on variable `__INITBSS__`, which is located in the `.data` section by default and is non-zero. Sometimes it is faster to download or otherwise pre-initialize BSS, such as when running on a hardware description language (HDL) simulator. By default, the debugger zeros BSS and pokes `__INITBSS__` with zero to prevent the runtime doing so.

Turning on toggle **program_zeros_bss** causes the debugger to poke `__INITBSS__`, if present, with a non-zero value (regardless of its current value) so that your application, instead of the debugger, zeros the BSS sections.

`-on=program_zeros_bss`

read_ro_from_exe — Request memory from read-only sections of the executable file

Default: Off

When toggle `read_ro_from_exe` is on, a debugger request to read memory from read-only sections of the executable will be done from the debugged file, rather than the memory of the target. Turning this toggle on can reduce traffic to the target.

Use this option if you know that your program will not accidentally write over read-only sections. Read-only sections typically include code, but also include data that is never modified.

recursive_delay_slot — Permit single-step execution of delay slots within instructions

Default: On

If this toggle is on, delay slots are executed whenever the ISS single-steps an instruction with a delay slot. This usually makes for smoother debugging. Turn this toggle off to have the delay slot executed in a separate single-step instead; this allows interrupts to occur in delay slots.

reset_upon_restart — Reset target on restart

Default: Off

When this toggle is on and the debugger processes a restart command, the debugger attempts to reset the target first by sending a `reset_board` message to the target.

reload_READONLY — Reload read-only portion of executable when restarting

Default: Off

When toggle `reload_READONLY` is off, if you reload an executable (with the `restart` command) the debugger does not reload the read-only portion of the executable, to save reload time. When this toggle is on, the debugger reloads all portions of the executable on a restart.

respect_THREADS — Enable thread awareness

Default: On

When this toggle is on, the debugger is aware of the multiple threads in a multi-threaded RTOS, assuming such RTOS awareness has been provided to the debugger.

If this toggle is off, the debugger ignores the threads. This can be advantageous if you have a large number of threads and it takes time to upload the thread information via the RTOS awareness.

You can turn this toggle on dynamically (`on=respect_THREADS`) when you are ready for the debugger to know about threads.

restore_ap — Save and restore breakpoints

Default: Off

This toggle causes the debugger to save breakpoints into a file upon termination of a program and restore them when the program is reloaded.

Without this toggle, when you exit the debugger, or if you restart a program whose binary file has changed, breakpoint information is lost. With this toggle on, the debugger saves breakpoint information for each program and re-reads and restores the breakpoints when the program is loaded again.

Note Breakpoint information may be inaccurate if your program has changed.

You can also enable this feature by checking **Restore breakpoints** in **Debugger Options | Program Options**.

reuse_displays — Use existing windows instead of opening new ones (GUI)

Default: Off

Turning on toggle `reuse_displays` directs the debugger to reuse an existing window instead of opening a new window.

For example, in the **Call Stack** window you can view the source code for a selected function on the stack. When toggle `reuse_displays` is off, the debugger opens a new **Source** window to show the function's source code. When this toggle is on, the debugger displays the function's source code in an existing **Source** window, rather than opening a new one.

show_memory_loads — Display the contents of a pending memory load

Default: On

When toggle `show_memory_loads` is on and the program counter is at an instruction with a load, disassembly listings (at the command line or in the **Disassembly** window) show the contents of the pending memory load.

Turning this toggle off directs the debugger not to display the load information.

show_reg_diffs — Show which registers have changed after an instruction single-step

Default: On

When toggle `show_reg_diffs` is on and you instruction-single-step through your program, the debugger inspects the value of all registers before and after the step, then shows which registers have changed.

This feature increases register traffic to and from the target; turn off toggle `show_reg_diffs` if I/O to and from the target is too slow.

spot_download_READONLY — Check for read-only code already on target

Default: Off

When this toggle is on, the debugger compares approximately 4 bytes in every 2 K to be downloaded with what exists in memory. If the bytes match, that 2 K is not downloaded.

This toggle is useful if you run the same lengthy program repeatedly and exit the debugger between runs.

During download, the debugger marks sections it skips with “-” rather than the standard “.”

Example

```
ELF segment #0 addr 0x002000 size 0x070134      -----
```

rather than

```
ELF segment #0 addr 0x002000 size 0x070134      .....
```

spot_verify_download — Verify memory operation during download

Default: On

If this toggle is on, approximately 4 bytes in every 2K are read back after download to ensure memory is operating correctly. This check is skipped when the target reports it is a simulator.

See also [verify_download](#).

st_to_code — Stop debugger when a program stores code to memory

Default: On

Turn off toggle **st_to_code** to catch stack overruns. When this toggle is off, the debugger stops whenever a program stores to code memory.

store_cache_data — Simulate cache RAM

Default: Off

When this toggles is on, ISS-simulated caches implement cache RAMs, meaning that cache data can be different from memory. (You can observe such differences in the cache window by clicking **Actions/Compare for Coherency**.) With this toggle off, memory traffic always goes straight to memory, as the caches do not actually store data; they only store the tags, recording the addresses of what would be in the caches were the caches to have RAMs.

Use of **_Uncached** pointers in your program that are not aligned to the size of a data-cache line can cause problems when a line flush for adjacent memory overwrites memory you elsewhere treat as **_Uncached**. With this toggle off, the problem is masked, because the cache simulations do not store memory and hence flushes are not simulated.

Turn on this toggle to avoid masking the problem.

swap_endian — Swap read/write chunks for target connection interface

Default: Off

Turn on this toggle to have memory reads and writes swapped in 4-byte chunks on behalf of the target connection interface.

This toggle overcomes the inability of some older ARC target connection interfaces to handle big- and little-endian ARC targets.

trace — Trace instructions as they execute

Default: Off

Trace instructions as they are executed. This toggle is specific to the MetaWare debugger **ARC_simulator** (little-endian) and **BRC_simulator** (big-endian).

trace_board_interface — Trace calls to interface functions

Default: Off

Prints a message to `stdout` whenever the debugger calls a function in the processor interface. You can turn this toggle on and off with command [prop](#).

Due to timing issues, the first few calls to your interface are not traced. You can turn off toggle [show_reg_diffs](#) to reduce the register read traffic.

trace_board_interface2 — Trace returns from interface functions

Default: Off

This toggle is for use in troubleshooting target-interface extensions. Turn on `trace_board_interface2` to print a separate message to `stdout` each time the call to your interface returns to the debugger; this is useful for detecting if your interface is hanging. You can turn this toggle on and off with command [prop](#).

For a less verbose version, see [trace_board_interface](#).

trace_cycles — Trace cycle stalls as they occur

Default: Off

Toggle `trace_cycles` lists all cycle stalls contemporaneously. The output is useful for discovering precisely when stalls occur, especially when single-stepping at the assembly level.

trace_reg — Trace changes to any register

Default: Off

Trace changes to any register. This toggle is specific to the MetaWare debugger `ARC_simulator` (little-endian) and `BRC_simulator` (big-endian).

use_only_mapped_memory — Do not use memory outside memory map

Default: Off

When this toggle is on, it prevents the debugger from performing any memory transaction with the target that lies outside the current memory map. Use this toggle if your target has memory sensitive to I/O, such as memory that causes a processor fault if you access it.

verify_download — Verify code after downloading to target

Default: Off

Turn on toggle `verify_download` when starting the debugger to verify that a code download was successful.

When this toggle is on, the debugger re-reads the target after a download to verify that the download was successful. Use this toggle if you suspect that there may be an error or problem with memory.

workaround_dcache_lock_bug — Flush locked lines individually to avoid bug

Default: Off

This toggle enables a workaround of the bug detected by [check_dcache_lock_bug](#).

This toggle has no effect unless `check_dcache_lock_bug` is on. If you don't use locked lines in your application, there is no need to enable the workaround.

Normally the debugger flushes the entire data cache when the user asks to examine memory. With the workaround, the debugger flushes lines individually.

xcheck — Check ARC extensions on startup

Default: On

By default, the MetaWare Run-Time Library checks for extensions the program uses, and stops if they are not present. For example, if you compile with the ARC 700 multiply but do not have a multiplier in the target when you run your program, the program stops with the following message:

```
Execution stopped.  
No source information at 0x1553c.  
1553c __Xmult32_conflict2+0x04  nop
```

Turn this toggle off to cancel such checking.

Appendix A — Multi-Tasked RTOS Applications

In This Appendix

- [About Threads and Tasks](#)
- [Debugging Multiple Tasks/Threads](#)
- [Debugging Precise/MQX RTOS Applications](#)
- [Debugging Nucleus PLUS Applications](#)
- [Debugging ThreadX Applications](#)

About Threads and Tasks

The words *thread* and *task* are used interchangeably throughout this guide. The debugger itself is RTOS-aware and labels the affected windows “thread” or “task” depending on the specific RTOS. For example, the debugger refers to *task* for Precise/MQX and Nucleus PLUS, and refers to *thread* for ThreadX.

Debugging Multiple Tasks/Threads

The MetaWare debugger provides multi-tasked/threaded debugging facilities such as task- and thread-specific breakpoints and task-and thread-locked windows. This section describes these facilities. See [Debugging Multi-Tasked/Threaded Applications](#) on page 107 for step-by-step instructions for debugging multiple tasks using the GUI version of the MetaWare debugger.

Specifying the Operating System

To enable the debugger’s multi-tasked/threaded features, you must tell the debugger what operating system you are debugging. The debugger obtains information about tasks in a different way for each operating system.

Specifying the Operating System Using the GUI

1. Start the debugger and click the **Debugger Options** button. If the debugger is already running, select **Tools | Debugger Options**.
2. In **Program Options**, select an RTOS in the **RTOS selection** box.
3. Click **OK** to save settings and exit.

Specifying the Operating System Using the Command Line

To specify an RTOS on the command line, use option **-OS=osname**.

Examples:

```
-OS=TX      # Express Logic's ThreadX  
-OS=ThreadX # Express Logic's ThreadX  
-OS=Nucleus # Accelerated Technology's Nucleus PLUS  
-OS=NU      # Accelerated Technology's Nucleus PLUS
```

Precise/MQX and Task Aware Debug

To specify Precise/MQX, use command-line option **-semint=tadsc.d11**. For more information on how to specify MQX, see [Enabling Task-Aware Debugging in the Debugger](#) on page 317.

You can add these options to the **ARGS=** line in **bin/mdb.cnf** to specify the same operating system each time you invoke the debugger.

Debugging Tasks/Threads

Using the Task Lock or Thread Lock Menu

When a multi-threaded or multi-tasked application starts, threads or tasks usually have not yet been created. After the application has run long enough to start up the tasks/threads, the debugger automatically detects their presence, and adds a new menu item **Task Lock** or **Thread Lock**. The **Task Lock** or **Thread Lock** menu shows a list of the tasks/threads that have been created.

A task-based OS typically has the notion of *current task*: at any given time, just one task is executing. When the program stops, the debugger can find out from the OS which task is current. The **Task Lock** or **Thread Lock** menu shows (**current**) next to the current task.

Each task has a *task ID*: a number the OS assigns to uniquely identify the task. Task IDs can be small integers, such as 1, 2, 3, and so forth. On some operating systems, the task ID might be the address of a control block that stores information about the task.

Locking and Cloning Task Windows

A MetaWare debugger desktop has zero or more associated windows, such as the **Disassembly**, **Source**, or **Local Variables** window. When the debugged program stops, the windows reflect the state of the current task. If you select a non-current task from the **Task Lock** menu, the windows switch to the selected task. Thereafter, the windows are locked to that task, and show only that task.

Commands you type, such as **reg** or **stack**, reflect the state of the selected task. You can switch back to the current task by selecting the **Current Task** item from the menu. To see the state of different tasks at the same time, you can clone the debugger desktop. Select **Clone desktop** from the **File** menu, and you get another debugger desktop. Lock the cloned debugger desktop to a different task, and all its windows show the state of that task.

If you select **Clone desktop and its frames**, the debugger desktop's associated windows are cloned as well.

After cloning, the original and cloned desktops include an alphabetic indicator ([A], [B], and so on) in their title bars. On Windows systems, if you minimize a desktop, the task-bar icon for the desktop includes that desktop's alphabetic indicator ([A], [B], and so on).

If you lock a window to a task, the window shows the task ID in its title bar. If the task displayed is the current task, that information also appears in the title bar. When locked to a task, windows are distinguished by color. The color appears as a stripe across the top of the window.

Using the Task window

If you have so many tasks that the **Tasks** menu no longer fits on the screen, the debugger adds a **More Tasks** menu option. Selecting **More Tasks** brings up the **Task** window. You can also click **Tasks** in the **Windows** menu to bring up the **Task** window.

The **Task** window is scrollable, and so can show an arbitrarily long list of tasks. Double-clicking a task in this window has the same effect as selecting the task from the **Task** menu: it locks the window to that task.

Debugging Tasks with the Command-Line Debugger

When you use the command-line debugger, console notification messages tell you when tasks are created or destroyed. The current task is also specified. Whenever you display information, such as registers, the information pertains to the current task.

Use these commands to identify and select tasks:

tasks to display a list of tasks

task NNN to switch to an alternate task

NNN is the task ID listed by the **tasks** command. All subsequent displays show information for that task (where relevant). When you resume running your program, the debugger switches back to the current task, undoing the **task NNN** command. You can avoid retying **task NNN** by typing **h** to see the command history, finding the number of your task command, and then just typing that number to reissue the command.

To set a task-specific breakpoint, add:

```
, task NNN
```

to the end of a breakpoint command. For more information, type **help break** at the command prompt.

Debugging Precise/MQX RTOS Applications

This section describes debugging for applications that use the Precise/MQX RTOS (MQX) and the Precise/RTCS™ embedded TCP/IP stack (RTCS). The MetaWare debugger uses the task-aware debugging capabilities of Task Aware Debug (TAD). TAD is a task-aware DLL (`tadsc.dll`) that provides MQX- and RTCS-aware support. This section also describes collecting data that can be used by the Performance Tool analyzer and the Precise/EDS™ Client remote monitoring tool (EDS Client).

TAD, Performance Tool, and EDS Client are described in the *Host Tools User's Guide*. For general information about debugging multi-task applications, see [Debugging Multi-Tasked/Threaded Applications](#) on page 107.

Task-Aware Debug Features

TAD includes features that help you debug concurrent applications. Features of TAD include task-aware support to browse and analyze MQX and RTCS data structures that control the execution of the application.

You can use the MetaWare debugger to debug MQX and RTCS applications on embedded processors. Task-aware debugging gives you the ability to:

- examine data structures
- obtain a summary of all tasks
- view the context of a specific task
- traverse the ready queues
- browse message queues and message pools
- inspect message contents
- browse lightweight semaphores and view waiting tasks
- browse semaphores and view owning and waiting tasks
- browse event groups and examine their state
- browse lightweight event groups and examine their state
- obtain a summary of mutexes and inspect their contents
- audit task and MQX memory resources
- view partition memory
- view lightweight memory
- view memory pools
- examine lightweight timers
- examine interrupt-handling mechanisms display stack usage
- inspect MQX instrumentation logs

- examine processor usage
- display kernel time
- display RTCS statistics
- examine RTCS internal data

Configuring for Task Aware Debugging

TAD assumes that the MQX initialization variable is called `MQX_init_struct`. All required initialization information is retrieved from this variable. If TAD cannot find the symbol `MQX_init_struct`, it displays a warning.

Enabling Task-Aware Debugging in the Debugger

To enable task-aware debugging for MQX and RTCS applications, do the following:

1. Copy the runtime library `tadsc.dll` to the `install_dir/bin` directory in which the debugger executable is located.
2. Specify the option `-semint=tadsc` in the command line when you start the debugger.

Updating TAD Windows

During a debugging session, all information displayed in the TAD windows is automatically updated after execution of any target application code or modifications to target memory.

Viewing the Context of Blocked Tasks

You can use the **Task Lock** menu to view the context of a blocked task. All tasks are listed under the **Task Lock** menu, and the current (active) task is indicated.

To view the context of a blocked task, select the name of the task. The debugger updates the following windows to reflect values for the task that you selected:

- **Call Stack** window
- **Register** window
- **Local Variables** window
- **Source** window

MQX Menu

The following list describes the function of each item in the MQX menu in the MetaWare debugger:

General menu

Kernel Data

Opens the Kernel Data window, which displays general MQX information.

Memory Blocks

Opens the Memory Blocks Summary window, which lists the blocks of memory that MQX allocated.

Lightweight Memory Pools

Lists the memory pools that MQX allocated.

Partitions

Displays a summary of the partitions that the application created.

Log Summary

Displays the status of the logs.

Initialization

Displays initialization information used by MQX.

Interrupts

Displays a summary of the interrupt-handling routines that the application installed.

IO Devices

Displays a summary of the I/O device handlers that have been installed.

Names

Displays the current names in the name table and their associated values.

Lightweight Timers

Displays the timer queues that have been created.

Check for Errors

Opens the Check System for Errors window, which displays a list of detected errors or data corruption in MQX data structures.

Tasks menu

Task Summary

Displays an overview of all the tasks, including each task's name, ID, descriptor, state, priority, and task error code.

Ready Queue

Opens the Ready Queue window, which lists the tasks that are ready to run at all priority levels.

Stack Usage

Displays a summary of the stack usage by each task and the interrupt stack.

Task Queues

Lists the task queues.

Synchronization menu

Message Queues

Displays a summary of all message queues, including each queue's number, type, owner, and number of entries.

Message Pools

Displays a summary of all message pools.

Mutexes

Displays a summary of all current mutexes.

Lightweight Semaphores

Displays a summary of the current lightweight semaphores.

Semaphores

Lists the current semaphores.

Events

Lists the current events.

Lightweight Events

Lists the current lightweight events.

RTCS menu**Socket Summary**

Displays a summary of the current RTCS sockets.

Config Parameters

Displays the current configuration of RTCS.

Socket Config

Displays an overview of the socket configuration.

TCP Config

Displays the current TCP configuration.

UDP Config

Displays the current UDP configuration.

IP Config

Displays the current IP configuration.

NAT Config

Displays the number of sessions being monitored by the NAT IP-address translator.

TCP Statistics

Displays TCP statistics.

UDP Statistics

Displays UDP statistics.

IP Statistics

Displays IP statistics.

ICMP Statistics

Displays ICMP statistics.

IGMP Statistics

Displays IGMP statistics.

ARP Statistics

Displays ARP statistics.

IP-IF Statistics

Displays statistics gathered by the IP interface modules.

NAT Statistics

Displays the RTCS NAT Statistics window, which displays NAT statistics.

Performance Data

Writes data into a file on the host. The file is compatible with Performance Tool, which lets you view state transitions, stack usage, CPU usage, and MQX usage. With Performance Tool, you can also trace the activity of all tasks and detect errors in resource usage.

MQX Data

Writes the MQX data into a file on the host. The file is compatible with EDS Client. In addition to the features offered by TAD, EDS Client can operate with the target in various communication modes. EDS Client can also operate with multiprocessor MQX applications, and automate Microsoft Excel or Performance Tool to obtain and analyze log data from the target application.

Close All

Closes all windows.

Help

Opens TAD help contents.

About

Displays the version number, processor family, and copyright notice for TAD.

Obtaining Help

You can obtain help by performing either of the following:

- select **MQX | Help** from the debugger GUI
- press F1 to obtain context-sensitive help for an active TAD window

Debugging Nucleus PLUS Applications

The information in this section is specific to debugging Nucleus PLUS applications.

For general information about debugging multi-task applications, see [Debugging Multi-Tasked/Threaded Applications](#) on page 107.

Preparing to Debug a Nucleus PLUS Program

Placing Nucleus PLUS Debug Information in the Linked Program

For the debugger to recognize Nucleus PLUS, the debug information for the basic Nucleus PLUS data structures must be present in the linked program. There are several ways to make this information available.

(See also [Target-Specific Linking Information for Nucleus PLUS Applications](#) on page 320.)

- (Simplest method) Always include `nudebug.o`, a module provided as part of the debugger, in your program (see directory `rtos/nucleus`). This module contains the debug information for all Nucleus PLUS data structures.

If you choose this method, it does not matter whether any other program code is compiled with debug information (compiler option `-g`). Module `nudebug.o` contains everything the debugger needs.
- Compile your programs with option `-g`. In this way precisely those operating system resources you use in your programs have debug information. For information about compiler option `-g`, see the *MetaWare C/C++ Programmer's Guide*.
- Compile all the Nucleus PLUS source with toggle `Emit_line_records` turned on (with command-line option `-Hon`). Turning this toggle on provides some debugging information throughout every function in the RTOS. After compiling, link with `nudebug.o` to get debugging information for the RTOS data structures.
- Compile all of Nucleus PLUS with option `-g` specified. Doing this produces a larger and slower Nucleus PLUS operating system, but is useful if you want to step into or debug parts of Nucleus PLUS itself.

Target-Specific Linking Information for Nucleus PLUS Applications

Here is an example of linking a Nucleus PLUS application:

```
MW_DIR=/arc
NU_DIR=/nucleus
mcc -a4 demo.o \
$(MW_DIR)/rtos/nucleus/nudebug.o \
$(NU_DIR)/int.o \
-Heos=nucleus \
-Heoslib=$(NU_DIR)/libnu.a -o demo.out
```

See the *MetaWare C/C++ Programmer's Guide* for general information about compiling and linking, and for information about command-line options **-Heos** and **-Heoslib**.

Informing the Debugger of a Nucleus PLUS Application

To debug a Nucleus PLUS application, you must use command-line option **-OS=NU** when invoking the debugger. This option notifies the debugger that the operating system is Nucleus PLUS. See [Command-Line Option Reference](#) on page 274 for details.

Getting the Program to a State Where Tasks Are Known

Before a program has executed, there is no task information, and the debugger cannot display any task state. Before using the Nucleus PLUS-specific displays and windows, you must get your program to a state where the tasks are known.

You can break at the end of function **Application_Initialize()**, or on the first call to function **TCT_Schedule()**. At this point all operating system resources are defined and can be found by the debugger.

The Nucleus State Window

Multi-task applications use operating system resources to synchronize tasks. The debugger provides a single Nucleus PLUS-specific window that shows all the operating system resources so you can examine them in detail.

To open the Nucleus State window, choose **Display | Nucleus state**.

Text Display in the Nucleus State Window

The Nucleus State window's text area displays the following Nucleus PLUS resources:

- tasks
- semaphores
- queues
- dynamic memory pools
- partition pools
- pipes
- mailboxes
- timers
- event groups
- drivers

The window shows one line of most-often sought information per Nucleus PLUS resource. For example, for each task, the following information appears:

- The task state — READY, SLEEP, and so on.
- The waited-for resource, if the task is suspended waiting for an operating system resource.
- The number of times the task has run since the last time the debugger updated the display. This information appears in the form ran +*N*, where *N* is the number of times run.

Double-clicking on items in the **Nucleus State** window opens another window. For example:

- Double-clicking most memory pointers opens a **Memory** window showing the pointed-to memory.
- Double-clicking a task's stack pointers opens the **Call Stack** window for that task.
- Double-clicking a task's entry point brings up the **Source** or **Disassembly** window for that task's entry point.

The Nucleus State Options Toolbar

The **Nucleus State Options** toolbar contains tools for displaying more information about queues, and for toggling on and off (or reordering) the display of each category of Nucleus PLUS resources. For all of these resources, toggling the resource off, then on again, places those resources at the bottom of the window's text area.

[More]

(Works only with queues) Opens a **Memory: Queue** window for viewing the contents of a selected queue.

[Hist]

This toggles whether the debugger shows a Nucleus PLUS history log. See [The Nucleus PLUS History Log](#) on page 323 for more information.

[Task]

Toggles on and off the display of tasks in the **Nucleus State** window.

[Sema]

Toggles on and off the display of semaphores in the **Nucleus State** window.

[Queue]

Toggles on and off the display of queue elements in the **Nucleus State** window.

[Pool]

Toggles on and off the display of dynamic memory pools and partition pools in the **Nucleus State** window.

[Pipe]

Toggles on and off the display of pipes in the **Nucleus State** window.

[Mbox]

Toggles on and off the display of mailboxes in the **Nucleus State** window.

[Timer]

Toggles on and off the display of timers in the **Nucleus State** window.

[Event]

Toggles on and off the display of event groups in the **Nucleus State** window.

[Driver]

Toggles on and off the display of drivers in the **Nucleus State** window.

The Nucleus PLUS Memory: Queue Window

To open the **Memory: Queue** window, click the **[More]** button in the **Nucleus State** window. The **Memory: Queue** window is a special **Memory** window uniquely tailored to display Nucleus PLUS queues.

Text Display in the Memory: Queue Window

The **Memory: Queue** window's text area shows only what is in the queue, and nothing more. It wraps when you go past the end of the queue.

- A green arrow marks the head of the queue.
- The line preceding the head of the queue is the tail of the queue. (To see this, scroll up one line.)

Each time the program state changes, the **Memory: Queue** window automatically updates to show the current state of the queue. The number of bytes displayed on each line in the **Memory: Queue** window is the same as the size of each element in the queue.

The Memory-Queue Options Toolbar

The options toolbar in the **Memory: Queue** window has the same features as the options toolbar in the **Memory** window.

The Nucleus PLUS History Log

In the **Nucleus State** window you can disable or enable the logging and clear the log. For an example of using this history-logging feature, see 11. in [Running the Nucleus PLUS Demo](#) on page 327.

NOTE When you clear the log, take care that the program is not within the history-allocation routines — any function with “_History_” in its name — otherwise unpredictable results might occur.

It is always safe to enable history logging when it is disabled.

Viewing OS Resources in the Nucleus State Window

NOTE For a demonstration of using the **Nucleus State** window and related windows, see [Running the Nucleus PLUS Demo](#) on page 327.

1. Be sure you have linked Nucleus PLUS debug information into the program (see [Placing Nucleus PLUS Debug Information in the Linked Program](#) on page 320) and you have run the program far enough that task information exists (see [Getting the Program to a State Where Tasks Are Known](#) on page 321).
2. Choose **Display | Nucleus state**.
3. To get full information for a selected resource in the **Nucleus State** window, double-click the line displaying the resource. This action opens an **Examine** window for the selected resource.

At this point you can explore the resource in full, because you are looking at the actual operating system data variable that represents the resource.

NOTE The **Nucleus State** window accesses a lot of data in the program to determine the state. If this access is across a serial line, you might want to temporarily stop the access. Clicking the **[Lock]** button on the text-options (**A/a**) toolbar causes the display to no longer update. Click **[Lock]** again to unlock the display.

Viewing More Information about Nucleus PLUS Queues

1. In the **Nucleus State** window, select a Nucleus PLUS queue by clicking the line displaying the queue.
2. Click **[More]**. This opens a **Memory: Queue** window, a special **Memory** window uniquely tailored to display Nucleus PLUS queues.

Viewing a Typed Display of Queue Elements

If the content of each queue element is some C data type **T** and the program was compiled with option **-g**, you can cast an address to **T*** and dereference the result to obtain a full display of the data object.

To examine multiple queue elements, all cast to the appropriate type, follow these steps:

1. Open a **Memory: Queue** window.
2. In the **Memory: Queue** window, click a queue element to select it, then right-click to open the window's pop-up menu.
3. On the pop-up menu, left-click **Examine**. This opens an **Examine** window with the memory address of the queue element cast and dereferenced.

The **Memory: Queue** window does not know anything about the queue element, so it casts the address to **int***. For example, if **0x2af0** is displayed in the **Memory: Queue** window, and you select **Examine**, the **Examine** window shows the following:

* (**int***) **0x2af0**

4. In the **Examine:** input field, select **int**, then type over it with the data type of the queue element and press [Enter]. The **Examine** window shows that queue element as that data type.

Subsequent **Examine** commands in the **Memory: Queue** window modify only the address shown in the **Examine** window, not the data type.

Debugging Nucleus PLUS in the Command-Line Debugger

Engaging Task Support

Before a program has executed, there is no task information, and the debugger cannot display any task state. Before using the Nucleus PLUS-specific commands, you must get your program to a state where the tasks are known.

You can break at the end of function **Application_Initialize()**, or on the first call to function **TCT_Schedule()**. At this point all operating system resources are defined and can be found by the debugger.

Viewing Nucleus PLUS-specific Information

To see Nucleus PLUS-specific information, enter this command at the **mdb>** prompt:

```
display nustate
```

This command displays the same information as the initial **Nucleus State** window in the debugger GUI:

- tasks
- semaphores
- queues
- dynamic memory pools
- partition pools
- pipes
- mailboxes
- timers
- event groups
- drivers

For a more detailed explanation, see [Text Display in the Nucleus State Window](#) on page 321.

This command (where *xxx* is any non-blank text) shows the same information you would see in the **Nucleus State** window if you had clicked all the [+] (more information) indicators:

```
display nustate xxx
```

NOTE Although most of the Nucleus PLUS-specific information that is available in the debugger GUI can also be displayed in the command-line debugger, it is not possible to display an equivalent of the **Memory: Queue** window at the debugger command prompt.

Examining a Nucleus PLUS OS Object

To display a Nucleus PLUS OS object at the debugger command prompt, you can get a result similar to the display in the **Examine** window, but you must enter the address information yourself.

For example, suppose you entered a **display nustate** command as shown in [Figure 46 Nucleus PLUS State Command in the Command-Line Debugger](#).

```
display nustate
Task 113832 TASK 0, SLEEP
> Task 114016 TASK 1, READY
Task 114200 TASK 2, READY
Task 114384 TASK 3, SLEEP
Task 114568 TASK 4, SEMAPHORE
Task 114752 TASK 5, EVENT
Semaph 0x1bc80 MW_mutx, count 3, no task waiting
Semaph 0x1b0a4 MW_sema, count 1, no task waiting
Semaph 0x1b0d8 MW_sema, count 1, no task waiting
Semaph 0x1c140 SEM 0, count 0, waiting 1 (TASK 4)
Queue 0x1c0f8 QUEUE 0, 45 msgs, 10 of 100 words avail, no
Dynpool 0x1c18c SYSMEM, avail 8868 of 16436, no task waiting
Partpl 0x1c1d0 PRTPOOL0, avail 4 of 320, no task waiting
Pipe 0x1c20c PIPE0, messages 0, no task waiting
Mailbox 0x1c254 MBOX0, no mail, no task waiting
Mailbox 0x1c288 MBOX1, no mail, no task waiting
Timer 0x1c2bc TIMER 0, disabled
Eventg 0x1c168 EVGROUP0, current 0x0, waiting 1 (TASK 5)
Driver 0x1c300 DRIVER 0
Driver 0x1c320 DRIVER 1
wait log off
```

Figure 46 Nucleus PLUS State Command in the Command-Line Debugger

You can then enter this command at the debugger command prompt:

```
eval *(TC_TCB_STRUCT*) 114016
```

This **eval** command casts the current task ID to a pointer to the operating system type **struct TC_TCB_STRUCT***, dereferences the result, and then displays the operating system object corresponding to that task.

You could write a macro to make this operation more convenient:

```
macro tobj tnum
  eval *(TC_TCB_STRUCT*) tnum
endm
```

To run the macro, just enter this command at the debugger command prompt:

```
tnum taskid
```

[Table 12 Nucleus PLUS Data-Type Names](#) lists the Nucleus PLUS data type names you would use to “examine” a Nucleus PLUS OS object at the debugger command prompt.

Table 12 Nucleus PLUS Data-Type Names

OS Resource	Data Type Name
Dynamic pool	DM_PCB_STRUCT
Event flags	EV_GCB_STRUCT
History entry	HI_HISTORY_ENTRY_STRUCT
Mailbox	MB_MCB_STRUCT

Table 12 Nucleus PLUS Data-Type Names

OS Resource	Data Type Name
Partition pool	PM_PCB_STRUCT
Pipe	PI_PCB_STRUCT
Queue	QU_QCB_STRUCT
Semaphore	SM_SCB_STRUCT
Task	TC_TCB_STRUCT
Timer	TM_TCB_STRUCT

Running the Nucleus PLUS Demo

The MetaWare debugger for selected targets includes **nudemo**, a demo program for Nucleus PLUS. You can use this demo program to tour the debugger's facilities for debugging Nucleus PLUS applications.

This demo is divided into the following parts, with step-by-step instructions for each part:

- [Part 1: Starting the Nucleus PLUS Demo Program](#)
- [Part 2: Debugging in a Task-Locked Set of Windows](#)
- [Part 3: Viewing the Nucleus PLUS OS Resources](#)
- [Part 4: Debugging with Task-Specific Breakpoints](#)
- [Part 5: Quitting the Nucleus PLUS Demo Program](#)

Part 1: Starting the Nucleus PLUS Demo Program

In this part of the demo, you start the debugger GUI and the demo program, open a second debugger desktop so you can view multiple tasks, and run the program long enough to establish the tasks and the associated operating-system resources.

1. Change to the *install_dir*/mdb/rtos/nucleus directory.
2. At the system prompt, enter **nudemo** to run the Nucleus PLUS demo.

The **nudemo** demo file invokes the debugger GUI, starts the demo program, and opens a **Disassembly** window.

3. Clone the debugger desktop: Choose **File | Clone desktop**.

This action opens a second debugger desktop with [B] in its title bar and adds [A] to the title bar of the first debugger desktop.

4. Click **[Run]** in the [A] debugger desktop. Wait three seconds, then click **[Stop]**. This gives the demo program enough time to define the application and establish the tasks and other operating-system ("OS") objects.

After the tasks have been established, each debugger adds a **Tasks** menu. To see the current list of tasks, select **Tasks**. Note that the label **(current)** appears in the menu next to the current task.

The **Disassembly** window shows the execution location of the current task — which might be **task_2_entry**, or might be something in the operating system (typically a queue operation).

Part 2: Debugging in a Task-Locked Set of Windows

In this part of the demo, you lock the set of cloned windows to the non-current task and experiment with viewing task-specific information in the task-locked windows.

1. Open a **Disassembly** window in the [B] debugger desktop. The **Disassembly** window shows the chosen task. Typically this window shows TCC_Suspend_Task, which is where the task has stopped in the operating system.
2. To task-lock the [B] windows to a non-current task, pull down the **Tasks** menu in the [B] debugger desktop and choose the desired task from the menu. (Choose a task that is not current.)
The debugger adds a colored bar containing the task name across their top of all task-specific windows.
The **Disassembly** window shifts to display information for the chosen task. Typically this window shows tx_task_suspended, which is where the task stopped in the operating system.
3. In the [A] debugger desktop, click **[Run]**, then click **[Stop]**. Do this a few times, watching the title bar of the [B] debugger desktop. If the task you selected in the [B] debugger becomes current, both **Disassembly** windows show the same data. If the selected task is not current, the two **Disassembly** windows do not show the same data. Experiment with clicking **[Run]** and **[Stop]** for a while.
4. When the [B] debugger's task is a non-current task, choose **Display | Call stack** in the red [B] debugger desktop to open a **Call Stack** window and see the function-call stack for that non-current task.
5. Close the [B] debugger desktop.

Part 3: Viewing the Nucleus PLUS OS Resources

1. Open the **Nucleus State** window (choose **Display | Nucleus state**). Resize the window to show all the operating system objects.
Observe the basic state of all the Nucleus PLUS OS resources, one line per resource. A resource is a task, queue, timer, and so on. Notice the green arrow pointing to the current task.
To show more detail about any resource, click the **[+]** icon to the left of that resource.
 2. Click **[Run]**, then click **[Stop]**. Notice that the tasks that ran now have a **ran +N** label on the line, showing that the task ran *N* times since the debugger last updated the **Nucleus State** window.
 3. Experiment with examining Nucleus PLUS tasks and operating system resources. The Nucleus PLUS header files contain the C declarations for the operating system objects.
 4. In the **Nucleus State** window, double-click any of the Nucleus PLUS tasks or resources. This action opens an **Examine** window symbolically displaying the operating system object corresponding to that resource. From here you can examine the object in detail.
The **Examine** window can help you see the fine details of the operating system object.
 5. In the **Nucleus State** window, double-click two or more additional resources.
 6. Experiment with displaying operating system object pointers as memory.
 - a. In the **Nucleus State** window, click the **[+]** icon to the left of the Queue, and observe the **read:, write:, start:, and end:** pointers. Double-click any of these pointers to open a **Memory** window positioned to the value of these pointers.
 - b. You can also display as memory the pointers in the operating system objects **DynPool**, **Partpl**, **Pipe**, and **Task**. The **Nucleus State** window provides information about which items in the operating system object can be displayed as memory.
- After you double-click a pointer (such as **start:**) in an operating system object to display that pointer in a **Memory** window, subsequent double-clicking on another operating system object pointer displays the value of that other pointer in the same **Memory** window, rather than opening

- another **Memory** window.
- c. Close the **Memory** window.
7. Experiment with the **Memory: Queue** window.

- a. In the **Nucleus State** window, click the **Queue** line (to select it), then click the [**More**] button. This action opens a specialized **Memory: Queue** window showing the queue contents.

In the demo program, each queue element is two words, so the debugger formats the **Memory: Queue** window to show two words on each line.

- The addresses at the left are the addresses of each element in the queue.
- The green arrow shows the head of the queue.
- The line above the arrow is the tail of the queue.

The **Memory: Queue** window shows nothing except what is in the queue.

- b. The contents of the **Memory: Queue** window wrap around. To see this in action, hold down the scroll-down button in the window to continuously scroll. The green arrow scrolls up off the top of the window, then loops around and shows up again at the bottom of the window.
- c. Click any line in the **Memory: Queue** window.
- d. Right-click to open the window's pop-up menu.
- e. Choose **Examine**.

This action opens an **Examine** window with the contents of that address as an integer. You can widen the window so the **Examine:** field shows fully.

- f. Select the text **int** in the **Examine:** field, then type 'QELT' over it (in all upper-case letters) and press [**Enter**]. **QELT** is the C data type of the elements stored in the queue.

Changing the type in the **Examine:** field from **int** to **QELT** causes the **Examine** window to show the queue element symbolically as it really is: a **QELT**, which consists of a number and its square.

The program contains this C **struct** definition for type **QELT**:

```
typedef struct {
    int count;
    int square_of_count;
} QELT;
```

The **Examine** window shows the two elements of the **QELT** structure symbolically.

- g. To see other queue elements displayed the same way, right-click any other line in the **Memory: Queue** window and select **Examine** from the pop-up menu. Notice that only the address in the **Examine** window changes; the 'QELT' that you typed in earlier stays the same.

8. View the stack frame for a task.

In the **Nucleus State** window, click the [+] icon to the left of any of the Task resources to expand the resource display. You will see the **stack_ptr:** and **stack_start:** items. Double-click either of these items to open a **Call Stack** window showing the stack frame for that task.

Now close any windows opened during this step.

9. View the source (or disassembled instructions) for a task's entry point.

In the **Nucleus State** window, double-click the **entry point:** item in the expanded **Task** resource display. The debugger opens a **Source** window (if the source code is available) or a **Disassembly** window (if the source code is not available) showing the entry point for the task. Notice the

information in the title bar of this newly opened window.

Now close any window opened during this step.

10. Experiment with using the **Tasks** window.

As an alternative to using the **Tasks** menu, you can open the **Tasks** window (choose **Display | Tasks**). The **Tasks** window shows the tasks in your program, just like the **Tasks** menu does, one task per line.

To lock the open windows to a specific task, double-click the line displaying that task (in the **Tasks** window). Notice the padlock icon that appears to the left of the task line, indicating that the windows are locked to that task.

When the open windows are locked to a specific task, double-click that task line to unlock the windows. You can also double-click the `current task` line (at the top of the **Tasks** window) to set the windows so they are not locked to a particular task.

11. Experiment with the history log.

- a. In the **Nucleus State** window, click the `[+]` next to the `History Log` entry. You will see that the history is disabled and that there are no entries. Logging is initially disabled in the Nucleus PLUS demo.
- b. Double-click the line that says `double-click here to enable history`. History logging has now been enabled.
- c. Click **[Run]**, then click **[Stop]**. Observe the history that appears, showing the most recent entry first. The ID string at the far right symbolically tells the nature of the `History Log` entry.
- d. Double-click the line `double-click here to erase history`. This clears the history entries.
- e. Click **[Run]**, then click **[Stop]**. More entries appear.

NOTE The demo program `nudemo` was linked with Nucleus PLUS compiled with option `-DNU_ENABLE_HISTORY` specified. This option makes history logging possible. If Nucleus PLUS is not compiled with this option, the **Nucleus State** window does not show any entry for the `History` resource.

Part 4: Debugging with Task-Specific Breakpoints

A breakpoint specific to task *NNN* is a breakpoint that stops only if the task that encounters the breakpoint is task *NNN*. If any other task encounters the breakpoint, the debugger automatically resumes execution. In this demo program, function `compute_square(ULONG x)` is shared by both task 1 and task 2.

In this part of the demo, you set a breakpoint on the non-current task at function `compute_square()`.

1. Observe which task is current. (Look for the `current task` label in the **Tasks** window, or pull down the **Tasks** menu and see which task is so labeled.)
2. Choose **Tools | Break/watch** to open the Set Breakpoint dialog.
3. In the Set Breakpoint dialog, find an input field labeled **Condition:** — below this is a choice box that says **Not task specific**. This is the **Break on Task** choice box.

Click the down-arrow next to the **Break on Task** choice box to see a list of all the tasks in your program. Click a task to choose it.

4. In the upper left corner of the “breakpoints” dialog, find an input field labeled **Set Breakpoint On:** — this is where you specify that the breakpoint is to be set on function **compute_square()**.

Replace 0xFFFF in the input field with `comp*`. This regular expression resolves to the function name **compute_square()**.

5. Click **[OK]**. This action sets the task-specific breakpoint at function **compute_square()** for the non-current task you chose in step 3.
6. To verify that the task-specific breakpoint has been set, open a **Breakpoints** window by choosing **Display | Breakpoints**.
7. In the **Breakpoints** window you should see a single breakpoint listed.

Notice the condition listed as part of the breakpoint. This is the conditional evaluation the debugger uses to determine whether to stop when execution encounters the breakpoint.

Notice also the red icon denoting the breakpoint (on the far left of the breakpoint line). This icon has a ‘T’ in the middle of it, indicating that the breakpoint is task-specific.

8. Click **[Run]**. After a short time, execution should stop at the **compute_square()** function in the specified task.

Even though the program hit the **compute_square()** function many times during the run, the debugger automatically restarted the program if the task that encountered the function was not the one specified in the breakpoint.

9. View the function-call stack for the task associated with your task-specific breakpoint: Choose **Display | Call stack** to open the **Call Stack** window.

Part 5: Quitting the Nucleus PLUS Demo Program

To terminate the process for the **nudemo** program and exit the debugger, choose **File | Exit debugger**. This ends the Nucleus PLUS demo.

Debugging ThreadX Applications

For general information about debugging multi-threaded applications, see [Debugging Multi-Tasked/Threaded Applications](#) on page 107.

Preparing to Debug a ThreadX Program

Placing ThreadX Debug Information in the Linked Program

For the debugger to be able to recognize ThreadX, the debug information for the basic ThreadX data structures (in `tx_api.h`, which is included in your ThreadX distribution) must be present in the linked program. There are several ways to make this information available.

- Simplest method: Always include `txdebug.o`, a module provided as part of the debugger, in your program (see directory `rtos/threadx`). This module contains the debug information for all ThreadX data structures.

If you choose this method, it does not matter whether any other program code is compiled with debug information (compiler option `-g`). Module `txdebug.o` contains everything the debugger needs.

NOTE If you make changes to tx_api.h (for example, by adding data to TX_THREAD_PORT_EXTENSION), you must recompile txdebug.o. The source is included (txdebug.c).

- Compile your programs with option **-g**. In this way precisely those operating system resources you use in your programs have debug information. For information about option **-g**, see the *MetaWare C/C++ Programmer's Guide*.
- Compile all of ThreadX with option **-g**. Doing this produces a larger and slower ThreadX operating system, but is useful if you want to step into or debug parts of ThreadX itself.

Linking ARC ThreadX Applications

Here is an example of linking a ThreadX application:

```
MW_DIR=/arc
TX_DIR=/threadx
mcc -a4 demo.o $(MW_DIR)/rtos/threadx/txdebug.o \
-Heos=threadx \
-Heoslib=$(TX_DIR)/libtx.a -o demo.out
```

See the *MetaWare C/C++ Programmer's Guide* for general information about compiling and linking, and for documentation of command-line options **-Heos** and **-Heoslib**.

Informing the Debugger of a ThreadX Application

To debug a ThreadX application, you must use command-line option **-OS=TX** when you invoke the debugger. This option notifies the debugger that the operating system is ThreadX. See [Command-Line Option Reference](#) on page 274 for details.

Getting the Program to a State Where Threads Are Known

Before a program has executed, there is no thread information, and the debugger cannot display any thread state. Before using the ThreadX-specific windows, you must get your program to a state where the threads are known.

You can break at the end of function **tx_application_define()**, or on the first call to function **_tx_thread_schedule()**. At this point all operating system resources are defined and can be found by the debugger.

The ThreadX State Window

Multi-threaded applications use operating system resources to synchronize threads. The debugger provides a single ThreadX-specific window that shows all the operating system resources so you can examine them in detail.

To open the **ThreadX State** window, choose **Display | ThreadX state**.

Text Display in the ThreadX State Window

The **ThreadX State** window's text area shows all the ThreadX resources, as follows:

- threads
- semaphores
- queues
- event flags
- block pools

- byte pools
- timers

The window shows one line of most-often sought information per ThreadX resource. For example, for each thread, the following information appears:

- The thread state — READY, SUSPENDED, SLEEP, SEMAPHORE_SUSP, (waiting for) EVENT_FLAG, and so on.
- The waited-for resource, if the thread is suspended waiting for an operating system resource — ("semaphore 0"), ("event flags 0"), and so on.
- The number of times the thread has run since the last time the debugger updated the display. This information appears in the form ran +N, where N is the number of times run (the current value of the thread's run_count minus the previous value of run_count).

If a resource has additional information, a [+] appears to the left of the line; clicking the [+] displays that information (and converts the [+] symbol to a [-] symbol). In particular, for byte pools and block pools, clicking the [+] displays a list of all sections in the pools and whether they are allocated. Doing this helps illuminate your program's current memory allocation status.

The ThreadX Options Toolbar

The **ThreadX Options** toolbar contains tools for displaying more information about queues, and for toggling on and off (or reordering) the display of each category of ThreadX resources. For all of these resources, toggling the resource off, then on again, places those resources at the bottom of the window's text area.

[More]

(Works only with queues) Opens a **Memory: Queue** window for viewing the contents of a selected queue.

[Threads]

Toggles on and off the display of threads in the **ThreadX State** window.

[Semaph]

Toggles on and off the display of semaphores in the **ThreadX State** window.

[Queues]

Toggles on and off the display of queue elements in the **ThreadX State** window.

[Event]

Toggles on and off the display of the event-flags group in the **ThreadX State** window.

[Byte]

Toggles on and off the display of the byte pool in the **ThreadX State** window.

[Block]

Toggles on and off the display of the block pool in the **ThreadX State** window.

[Timers]

Toggles on and off the display of timers in the **ThreadX State** window.

The ThreadX Memory: Queue Window

To open the **Memory: Queue** window, select a queue in the **ThreadX State** window and click [**More**]. The **Memory: Queue** window is a special **Memory** window tailored to display ThreadX queues.

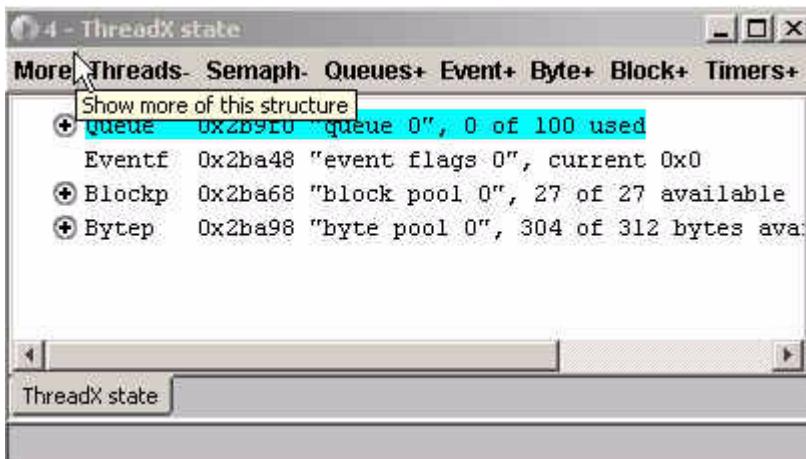


Figure 47 Click **More** to Open the **Memory: Queue** Window

Text Display in the Memory: Queue Window

The **Memory: Queue** window's text-display area shows only what is in the queue, and nothing more. It wraps when you go past the end of the queue.

- A green arrow marks the head of the queue.
- The line preceding the head of the queue is the tail of the queue. (To see this, scroll up one line.)

Each time the program state changes, the **Memory: Queue** window automatically updates to show the current state of the queue. The number of bytes displayed on each line in the **Memory: Queue** window is the same as the size of each element in the queue.

The Memory-Queue Options Toolbar

The options toolbar in the **Memory: Queue** window has the same features as the options toolbar in the **Memory** window.

Viewing OS Resources in the ThreadX State Window

NOTE For a demonstration of using the **ThreadX State** window and related windows, see [Running the ThreadX Demo](#) on page 337.

1. Be sure you have linked ThreadX debug information into the program (see [Placing ThreadX Debug Information in the Linked Program](#) on page 331) and you have run the program far enough that thread information exists (see [Getting the Program to a State Where Threads Are Known](#) on page 332).
2. Choose **Display | ThreadX state**.
3. To get full information for a selected resource in the **ThreadX State** window, double-click the line displaying the resource. This action opens an **Examine** window for the selected resource.

At this point you can explore the resource in full, because you are looking at the actual OS data variable that represents the resource.

NOTE The **ThreadX State** window accesses a lot of data in the program to determine the state. If this access is across a serial line, you might want to temporarily stop the access. Clicking the **[Lock]** button on the text-options (**A/a**) toolbar causes the display to no longer update. Click **[Lock]** again to unlock the window.

Viewing More Information about ThreadX Queues

1. In the **ThreadX State** window, select a ThreadX queue by clicking the line displaying the queue.
2. Click **[More]**. This opens a **Memory: Queue** window, a special **Memory** window tailored to display ThreadX queues. (See [The ThreadX Memory: Queue Window](#) on page 334 for a description.)

Viewing a Typed Display of Queue Elements

If the content of each queue element is some C data type ‘T’ and the program was compiled with option **-g**, you can cast an address to **T*** and dereference the result to obtain a full display of the data object.

To examine multiple queue elements, all cast to the appropriate type, follow these steps:

1. Open a **Memory: Queue** window (see [Viewing More Information about ThreadX Queues](#) on page 335).
2. In the **Memory: Queue** window, click a queue element to select it, then right click to open the window’s pop-up menu.
3. On the pop-up menu, left-click **Examine**. This opens an **Examine** window with the memory address of the queue element cast and dereferenced.

The **Memory: Queue** window does not know anything about the queue element, so it casts the address to **int***. For example, if **0x2af0** is displayed in the **Memory: Queue** window, and you select **Examine**, the **Examine** window opens and examines *** (int*) 0x2af0**.

4. In the **Examine**: input field, select **int**, then type over it with the data type of the queue element and press ENTER. The **Examine** window shows that queue element as that data type.

Subsequent **Examine** commands in the **Memory: Queue** window modify only the address shown in the **Examine** window, not the data type.

Debugging ThreadX in the Command-Line Debugger

Engaging Thread Support

Before a program has executed, there is no thread information, and the debugger cannot display any thread state. Before using the ThreadX-specific commands, you must get your program to a state where the threads are known.

You can break at the end of function **tx_application_define()**, or on the first call to function **_tx_thread_schedule()**. At this point all operating-system resources are defined and can be found by the debugger.

Viewing ThreadX-Specific Information

To see ThreadX-specific information, enter this command at the debugger command prompt:

```
display txstate
```

This command displays the same information as the initial **ThreadX State** window in the debugger GUI:

- threads
- semaphores
- queues
- event flags
- block pools
- byte pools
- timers

For a more detailed explanation, see [Text Display in the ThreadX State Window](#) on page 332.

The following command (where *xxx* is any non-blank text) shows the same information you would see in the **ThreadX State** window if you had clicked all the [+] indicators:

```
display txstate xxx
```

NOTE Although most of the ThreadX-specific information that is available in the debugger GUI can also be displayed in the command-line debugger, it is not possible to display an equivalent of the **Memory: Queue** window at the debugger command prompt.

Examining a ThreadX OS Object

To display a ThreadX OS object at the debugger command prompt, you can get a result similar to the contents of the **Examine** window, but you must enter the address information yourself.

For example, suppose you entered a **display txstate** command as shown in [Figure 48 ThreadX State Command in the Command-Line Debugger](#).

```
display txstate
Thread 178968 "System Timer Thread", SUSPENDED
Thread 177808 "thread 0", SLEEP
Thread 177952 "thread 1", READY
Thread 178096 "thread 2", READY
Thread 178240 "thread 3", SEMAPHORE_SUSP ("semaphore 0")
Thread 178384 "thread 4", SLEEP
Thread 178528 "thread 5", EVENT_FLAG ("event flags 0")
Semaph 0x2ba28 "semaphore 0", count 0; suspended:"thread 3"
Queue 0xb9f0 "queue 0", 39 of 100 used
Eventf 0x2ba48 "event flags 0", current 0x0; suspended:"thread 5"
Blockp 0x2ba68 "block pool 0", 26 of 27 available
Bytep 0x2ba98 "byte pool 0", 244 of 312 bytes available
```

Figure 48 ThreadX State Command in the Command-Line Debugger

You can then enter this command at the debugger command prompt:

```
eval *(TX_THREAD*)177952
```

This **eval** command casts the thread ID to a pointer to the operating system-resource type **TX_THREAD***, dereferences the result, and then displays the operating system object corresponding to that thread.

You could write a macro to make this operation more convenient:

```
macro tobj tnum
```

```
eval *(TX_THREAD*)tnum
endm
```

To run the macro, just enter this command at the debugger command prompt:

```
tnum threadid
```

[Table 13 ThreadX Data-Type Names](#) lists the ThreadX data type names you would use to “examine” a ThreadX OS object at the debugger command prompt.

Table 13 ThreadX Data-Type Names

OS Resource	Data Type Name
Thread	TX_THREAD
Semaphore	TX_SEMAPHORE
Queue	TX_QUEUE
Event flags	TX_EVENT_FLAGS_GROUP
Timer	TX_TIMER
Byte pool	TX_BYTE_POOL
Block pool	TX_BLOCK_POOL

Running the ThreadX Demo

The MetaWare debugger for selected targets includes **txdemo**, a demo program for ThreadX. You can use this demo program to tour the debugger's facilities for debugging ThreadX applications.

This demo is divided into the following parts, with step-by-step instructions for each part:

- [Part 1: Starting the ThreadX Demo Program](#)
- [Part 2: Debugging in a thread-Locked set of Windows](#)
- [Part 3: Viewing the Operating-System Resources](#)
- [Part 4: Debugging with Thread-Specific Breakpoints](#)
- [Part 5: Quitting the ThreadX Demo Program](#)

Part 1: Starting the ThreadX Demo Program

In this part of the demo, you start the debugger and the demo program, open a second debugger so you can view multiple threads, and run the program long enough to establish the threads and the associated operating-system resources.

1. Change to the *install_dir/rtos/threadx* directory.
2. At the system prompt, enter **txdemo** to run the ThreadX demo.

The **txdemo** program (or script) invokes the debugger and starts the demo program.

3. Clone the debugger by choosing **File | Clone desktop**.

This action opens a second debugger desktop with [B] in its title bar and adds [A] to the title bar of the first debugger desktop.

4. Click **[Run]** in the [A] debugger desktop. Wait about three seconds, then click **[Stop]**. This gives the demo program enough time to define the application and establish the threads and other operating-system (“OS”) objects.

After the threads have been established, each debugger adds a **Thread Lock** menu. To see the current list of threads, select the **Thread Lock** menu. Note that the label **(current)** appears in the

menu next to the current thread.

Open a **Disassembly** window in the [A] debugger. The **Disassembly** window is displaying the current thread — which might be `thread_2_entry`, or might be something in the operating system (typically a queue operation).

Part 2: Debugging in a thread-Locked set of Windows

In this part of the demo, you lock the set of cloned windows to the non-current thread and experiment with viewing thread-specific information in the thread-locked windows.

1. Open a **Disassembly** window in the [B] debugger desktop. The **Disassembly** window shows the chosen thread. Typically this window shows `TCC_Suspend_Thread`, which is where the thread has stopped in the operating system.
2. To thread-lock the [B] windows to the non-current thread, pull down the **Thread Lock** menu in the [B] debugger desktop and choose the desired thread from the menu.

The debugger adds a colored bar containing the thread name across their top of all thread-specific windows.

The **Disassembly** window's contents shift to display the chosen thread. Typically this window shows `tx_thread_suspended`, which is where the thread has stopped in the operating system.

3. In the [A] debugger desktop, click **[Run]**, then click **[Stop]**. Do this a few times, watching the title bar of the [B] debugger desktop. If the thread you selected in the [B] debugger becomes current, both **Disassembly** windows are the same. If the selected thread is not current, the two **Disassembly** windows are not the same. Experiment with clicking **[Run]** and **[Stop]** for a while.
4. When the [B] debugger's thread is the non-current thread, choose **Display | Call stack** in the [B] debugger desktop to open the **Call Stack** window and see the function-call stack for that non-current thread.
5. Close the [B] debugger desktop.

Part 3: Viewing the Operating-System Resources

1. Open the **ThreadX State** window (choose **Display | ThreadX state**). Resize the window to show all the operating system objects.

Observe the basic state of all the ThreadX OS resources, one line per resource. A resource is a thread, queue, block pool, timer, and so on. Notice the green arrow pointing to the current thread.

The **Blockp** (block pool) shows which blocks are allocated or free. The **Bytep** (byte pool) shows similar information, except that the regions are of varying size.

To show more detail about any resource, click the **[+]** icon to the left of that resource.

2. Click **[Run]**, then click **[Stop]**. Notice that the threads that ran now have a `ran +N` label on the line, showing that the thread ran *N* times since the debugger last updated the **ThreadX State** window.
3. In the **ThreadX State** window, double-click any of the ThreadX threads or resources. This action opens an **Examine** window symbolically displaying the operating system object corresponding to that resource.

From here you can examine the object in detail. The ThreadX header files contain the C declarations for the contents of the operating system objects.

The **Examine** window can help you view details of the operating system object.

4. In the **ThreadX State** window, double-click two or more additional resources.
5. In the **ThreadX State** window, click the Queue line (to select it), then click the [**More**] button. This action opens a specialized **Memory: Queue** window showing the queue contents.

In the demo program, each queue element is two words, so the debugger formats the **Memory: Queue** window to show two words on each line. The addresses at the left are the addresses of each element in the queue. The green arrow shows the head of the queue, and the **Memory: Queue** window shows nothing except what is in the queue. The line above the arrow is the tail of the queue.

The contents of the **Memory: Queue** window wrap around. To see this in action, hold down the scroll-down button in the window to continuously scroll; the green arrow scrolls up off the top of the window, then loops around and shows up again at the bottom of the window.

6. Now try displaying the queue elements symbolically. Select (click) any line in the **Memory: Queue** window, right-click to open the window's pop-up menu, then choose **Examine** in that menu. This action opens an **Examine** window to examine the contents of that address as an integer. Widen the window so the **Examine:** field shows fully.
7. Select the text **int** in the **Examine:** field, then type 'QELT' over it (in all upper-case letters). **QELT** is the C data type of the elements stored in the queue.

Changing the type in the **Examine:** field from **int** to **QELT** causes the **Examine** window to show the queue element symbolically as it really is: a **QELT**, which consists of a number and its square.

The program contains this C **struct** definition for type **QELT**:

```
typedef struct {
    int count;
    int square_of_count;
} QELT;
```

The **Examine** window shows the two elements of the **QELT** structure symbolically.

8. To see other queue elements displayed the same way, right-click any other line in the **Memory: Queue** window and select **Examine** from the pop-up menu. Notice that only the address in the **Examine** window changes; the **QELT** that you typed in Step 7. stays the same.
9. As an alternative to using the **Thread Lock** menu, you can open the **Threads** window (choose **Display | Threads**). The **Threads** window shows the threads in your program, just like the **Thread Lock** menu does, one thread per line.

To lock the open windows to a specific thread, double-click the line displaying that thread (in the **Threads** window). Notice the padlock icon that appears to the left of the thread line, indicating that the windows are locked to that thread.

When the open windows are locked to a specific thread, double-click that thread line to unlock the windows. You can also double-click the **current thread** line (at the top of the **Threads** window) to set the windows so they are not locked to a particular thread.

Part 4: Debugging with Thread-Specific Breakpoints

A breakpoint specific to thread *NNN* is a breakpoint that stops only if the thread that encounters the breakpoint is thread *NNN*. If any other thread encounters the breakpoint, the debugger automatically resumes execution. In this demo program, function **compute_square(ULONG x)** is shared by both thread 1 and thread 2.

In this part of the demo exercise, you set a breakpoint on the non-current thread at function **compute_square()**.

1. Observe which thread is current — thread 1 or thread 2. (Look for the **current thread** label in the **Threads** window, or pull down the **Thread Lock** menu and see which thread is labeled **(current)**.)
2. Choose **Tools | Break/watch** to open the **Set Breakpoint** dialog.
3. In the Set Breakpoint dialog, find an input field labeled **Condition:** — below this is a choice box that says **Not thread specific**. This is the **Break on Thread** choice box.

Click the down-arrow next to the **Break on Thread** choice box to see a list of all the threads in your program. Choose whichever of thread 1 and thread 2 is the non-current thread. (Click the non-current thread to choose it.)

4. In the upper left corner of the Set Breakpoint dialog, find an input field labeled **Set Breakpoint On:** — this is where you specify that the breakpoint is to be set on function **compute_square()**. Replace 0xFFFF in the input field with **comp***. This regular expression will resolve to the function name **compute_square()**.
5. Click **[OK]**. This action should set the thread-specific breakpoint at function **compute_square()** for the non-current thread you chose in 3.
6. To verify that the thread-specific breakpoint has been set, open a **Breakpoints** window by choosing **Display | Breakpoints**.

The **Breakpoints** window lists a single breakpoint.

Notice the condition listed as part of the breakpoint. This is the conditional evaluation the debugger uses to determine whether to stop when execution encounters the breakpoint.

Notice also the red icon denoting the breakpoint (on the far left of the breakpoint line). This icon has a ‘T’ in the middle of it, indicating that the breakpoint is thread-specific.

7. Click **[Run]**. Execution stops at the **compute_square()** function in the specified thread. Even though the program hit the **compute_square()** function many times during the run, the debugger automatically restarted the program if the thread that encountered the function was not the one specified in the breakpoint.
8. View the function-call stack for the thread associated with your thread-specific breakpoint: Choose **Display | Call stack** to open the **Call Stack** window.

Part 5: Quitting the ThreadX Demo Program

To terminate the process for the demo program and exit the debugger, choose **File | Exit debugger**.

Appendix B — S-Record Files

In This Appendix

- [What Is an S-Record File?](#)
- [Why Use an S-Record File?](#)
- [S-Record Format](#)
- [Example S-Record](#)

What Is an S-Record File?

A Motorola S-record file is an ASCII file consisting of an optional header record and one or more data records, followed by an end-of-file record. Data records may appear in any order. You can use an S-record file to fill memory with values as described in [Copying or Filling Memory to or from a File](#) on page 81.

Why Use an S-Record File?

Using an S-record file allows you to fill separate and distinct regions of memory with values of your choosing. This is advantageous when testing for values that might cause undesired behavior without affecting other regions of memory.

S-Record Format

Each data record, or individual line, adheres to the following format:

SSLAAAADD...DDCC

The component fields are explained in [Table 14 Motorola S-Record Format](#).

Table 14 Motorola S-Record Format

Field	Length	Description
SS	2	This field describes the type of record (S0 through S9). Record S0 is the header, and S9 is the footer. S1..S8 are data records.
LL	2	This field, interpreted as a hexadecimal value, shows the count of character pairs remaining in the record.
AAAA	4, 6, or 8	This field indicates the starting address to load the data into memory. The length of the field depends on the number of bytes necessary to hold the address. A two-byte address uses four characters, a three-byte address uses 6 characters, and a four-byte address uses eight characters.
DD...DD	0 to 64	This string consists of pairs interpreted as hexadecimal values that represent the data or descriptive information to load into memory.
CC	2	This field is a two-character checksum value.

Calculating the Checksum

1. Obtain the one's complement sum of each pair of values from count, address, and data fields (LL, AAAA, and DD...DD).
2. Subtract the value from hexadecimal FF (decimal 255).
3. The checksum is the value of the least-significant byte (the last two characters in hexadecimal notation).

Example S-Record

The following example is an S-record file that can be used to initialize memory in the debugger.

Example 51 Motorola S-Record

```
S00600004844521B  
S1130000285F245F2212226A000424290008237C2A  
S11300100002000800082629001853812341001813  
S113002041E900084E42234300182342000824A952  
S107003000144ED492  
S5030004F8  
S9030000FC
```

The file consists of one S0 record, four S1 records, one S5 record and one S9 record.

The S0 record contains the following components:

- S0: S-record type S0, indicating a header record.
- 06: Hexadecimal 06 (decimal 6), indicating that six character pairs (or ASCII bytes) follow.
- 00 00: Four-character, two-byte address field, zeroes in this example.
- 48 44 52: ASCII H, D, and R: “HDR”.
- 1B: The checksum.

The first S1 record contains the following components:

- S1: S-record type S1, indicating it is a data record to be loaded at a two-byte address.
- 13: Hexadecimal 13 (decimal 19), indicating that nineteen character pairs follow (a two-byte address, 16 bytes of binary data, and a one-byte checksum).
- 00 00: Four-character, two-byte address field; hexadecimal address 0x0000, where the data that follows are to be loaded.
- 28 5F 24 5F 22 12 22 6A 00 04 24 29 00 08 23 7C: Sixteen character pairs representing the actual binary data.
- 2A: The checksum.

The second and third S1 records each contain 0x13 (19) character pairs and end with checksums of 13 and 52, respectively. The fourth S1 record contains 07 character pairs and has a checksum of 92.

The S5 record contains the following components:

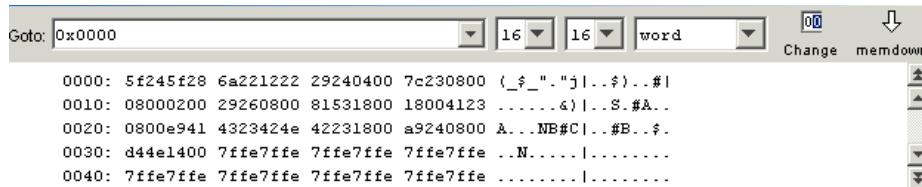
- S5: S-record type S5, a count record indicating the number of S1 records.
- 03: Hexadecimal 03 (decimal 3), indicating that three character pairs follow.
- 00 04: Hexadecimal 0004 (decimal 4), indicating that four data records precede this record.
- F8: The checksum.

The S9 record contains the following components:

- S9: S-record type S9, indicating an end-of-file record.
- 03: Hexadecimal 03 (decimal 3), indicating that three character pairs follow.
- 00 00: The address field, hexadecimal 0 (decimal 0) indicating the starting execution address.
- FC: The checksum.

Example S-Record

[Figure 49 Memory Values from an S-Record](#) shows the values from [Example 51](#) in a **Memory** window.



The screenshot shows a memory dump window with the following interface elements:

- Goto: 0x0000
- 16
- 16
- word
- 00
- Change memdown

The main area displays memory values starting at address 0000:

Address	Value	Description
0000	5f245f28 6a221222 29240400 7c230800	(_\$._"j ..\$)..#!
0010	08000200 29260800 81531800 180041234)1..S.#A..
0020	0800e941 4323424e 42231800 a9240800	A...NB#C ..#B..\$.
0030	d44e1400 7ffe7ffe 7ffe7ffe 7ffe7ffe	..N.....
0040	7ffe7ffe 7ffe7ffe 7ffe7ffe 7ffe7ffe

Figure 49 Memory Values from an S-Record

Appendix C — Quick Commands List

For a complete listing of debugger commands, see [Alphabetical Listing of Commands](#) on page 232.

Command	Description
addmap	Add information to a memory map
args	Display name and arguments of the current executable
attach	Attach an already running process
aux	Display ARC auxiliary registers
break	Set a breakpoint or display breakpoints
call	Call a function
continue	Start or continue execution of current process
delete break	Delete a breakpoint
delete mod	Delete an executable module
delete watch	Delete a watchpoint
delmap	Delete an entry from a memory map
detach	Detach current process from the debugger
disable break	Disable a breakpoint
disable watch	Disable a watchpoint
disassemble	Show a disassembly listing of the current process
display	Show named command-line display
download	Load an executable and read its symbol table
emem	Enter values into memory
enable break	Enable a breakpoint
enable watch	Enable a watchpoint
endm	Terminate a macro definition
evalq	Evaluate or change the value of an expression quietly
evaluate	Evaluate or change the value of an expression
exec	Execute a command when breakpoint encountered
exit	Exit the debugger
file	Display name and arguments of the current executable
file2mem	Write a file to memory
fillmem	Fill memory
focus	Set CMPD focus
globals	Display global variables
go	Stop execution at a specified address
help	Get help on a specific topic
history	Display command history
if	Execute a command if condition is true
if ... endif	Nest conditional commands

Quick Commands List

Command	Description
<u>isi</u>	Step through machine instructions, into functions
<u>iso</u>	Step through machine instructions, over functions
<u>kill</u>	Terminate the current process
<u>lbreak</u>	Set a breakpoint on a symbol that has not been loaded yet
<u>load</u>	Load a program into the debugger
<u>locals</u>	Display local variables
<u>log</u>	Log commands and results to a file
<u>macro</u>	Declare a macro
<u>mem2file</u>	Dump memory to a binary file
<u>mem2mem</u>	Copy memory from one location to another
<u>memmap</u>	Define or print memory map
<u>memory</u>	Dump memory
<u>memsb</u>	Search backward through memory
<u>memsearch</u>	Search forward through memory
<u>memset</u>	Set memory to value
<u>mend</u>	Terminate a macro definition
<u>modules</u>	List current executable and shared libraries
<u>onexit command</u>	Execute <i>command</i> on exiting the debugger
<u>print</u>	Print a message
<u>prop</u>	Set a target system property
<u>quit</u>	Exit the debugger
<u>read</u>	Automatically process commands in a file
<u>register</u>	Display machine registers
<u>restart</u>	Restart the current running process
<u>return</u>	Return from the current function
<u>run</u>	Start or continue execution of the current process
<u>set</u>	Set or display debugger settings
<u>show log</u>	Show all commands and outputs thus far
<u>simulate</u>	Update status quietly
<u>sleep</u>	Sleep specified number of milliseconds
<u>source</u>	Display source code
<u>ssi</u>	Step through source-code statements, into functions
<u>sso</u>	Step through source-code statements, over functions
<u>stack</u>	Display the run-time function-call stack
<u>status</u>	Update status of application being debugged
<u>stop</u>	Stop the current process
<u>symbols</u>	Load symbols from executable
<u>sysclone</u>	Clone the current system
<u>sysprop</u>	Set properties of specified target system

Command	Description
system	Execute a command on the host system
tbreak	Set a temporary breakpoint
thread	Switch to an alternate thread
threads	Display a list of thread IDs
trace	Show instruction trace history
unmacro	Remove a macro definition
unset	Restore debugger setting to its default
variable	Declare a debugger variable
watch	Set a watchpoint or display watchpoints
watchreg	Set a watchpoint on a register
while	Execute a command while condition is true

Appendix D — Quick Options List

In This Appendix

- [Overview](#)
- [Command-Line Options](#)

Overview

This section presents an quick-reference list of command-line options. For full listings, see [Command-Line Option Reference](#) on page 274.

For information on how to specify command-line options, see [Specifying Controls](#) on page 182.

Command-Line Options

-a4	Select the ARCTangent-A4 processor series
-a5	Select the ARCTangent-A5 processor series
-anim_scale	Set maximum animation speed
-arc600	Select the ARC 600 processor series
-arc700	Select the ARC 700 processor series
-auxlo	Specify the low address of an auxiliary-register range
-auxsz	Specify the size of the auxiliary-register range
-blast	Reinitialize the FPGA CPU from a file
-break	Specify address and size of breakpoint table
-c	Pass commands to the command processor
-chipinit	Configure and initialize the target board
-cl	Invoke the command-line debugger
-cmd	Execute the specified command
-core*	Specify processor core version
-dcache	Simulate a data cache
-dir_xlation	Set new source directory prefix
-DLL	Specify a DLL and its properties
-DLLprop	Specify a property for a DLL
-engine_wait	Wait before terminating debugger thread
-fontsize	Set initial font size for text in debugger windows
-generate_config	Generate configuration for extracting a debugger-engine DLL
-goifmain	Execute start-up code and stop at function main()
-gui	Invoke the debugger GUI
-h	List debugger command-line options
-hard	Specify that the application runs on an embedded processor
-hmem	Specify memory size of embedded processor
-icache	Simulate an instruction cache
-interrupt_base	Specify the base address of the interrupt vector table for the simulator
-javadir	Specify the Java directory
-jtag	Connect to hardware using a JTAG connection
-jvmheap	Specify minimum heap for host JVM
-load	Specify the load addresses for specified program segments

-loadexec	Execute the specified command
-max_aggregate_display	Specify maximum size of displayed aggregates
-max_poll_delay	Maximum interval in milliseconds between polls
-max_stack_depth	Specify maximum stack frames for Call Stack window
-mem	Specify memory size for hardware or simulator
-memalloc	Specify maximum memory allocated by the simulator
-memaux	Map a memory range into auxiliary registers
-mem_bus_width	Specify the width of a simulated XY memory bus
-memextinit	Initial value of memory added to the simulator
-memxfersize	Set memory transfer size
-metasim	Launch the Debugger/RTL cosimulator
-min_poll_delay	Minimum interval in milliseconds between polls
-mmu	Enable the ARC 700 Memory Management Unit (MMU)
-nogoifmain	Stop in start-up code; do not run to main()
-nohostlink_while_running	Turn off hostlink polling while the processor is running
-nooptions	Ignore pre-configured options
-noproject	Run the debugger without a debug project
-noprofile	Disable profiling for faster simulation
-notrace	Disable save of instruction-history information
-off=toggle	Turn off a toggle
-OK	Skip the Debug a process or processes dialog
-OKN	Skip the Debug a process or processes dialog and splash screen
-on=toggle	Turn on a toggle
-OS	Specify RTOS on which the debugged program runs
-pdisp	Parse pdisp output
-perip	Enable a peripheral window
-port	Specify PC parallel port address for I/O communications
-preloadexec	Execute MetaWare debugger command before loading application
-project=project-name	Specify debug project
-prop	Specify target-system properties
-prop=_HOSTLINK_=address	Set the address of symbol __HOSTLINK__
-prop=code_memory_additional_latency=N	Specify additional cycles to obtain a word from memory
-prop=code_memory_throughput=N	Specify the number of cycles to obtain a word from memory
-prop=data_memory_additional_latency=N	Specify additional cycles to obtain a word from memory
-prop=data_memory_throughput=N	Specify the number of cycles to obtain a word from memory
-prop=dcache_pipeline_depth=N	Specify the data cache pipeline depth

<u>-prop=dcr=xx</u>	Specify the data cache replacement algorithm
<u>-prop=ident=value</u>	Specify value for IDENTITY register
<u>-prop=uncached_base</u>	Specify base of uncached region
<u>-prop=uncached_size</u>	Specify size of uncached region
<u>-rascal</u>	Connect the debugger to an RTL simulation using RASCAL
<u>-run</u>	Download and execute a program with no debugger output
<u>-semint</u>	Specify a semantic inspection DLL and properties
<u>-set_optset</u>	Specify a named option set as the current option set
<u>-showtime[=NNN]</u>	Display how long a command took to execute
<u>-side_effect_auxregs</u>	Designate aux registers as having side effects
<u>-side_effect_regs</u>	Designate registers as having side effects
<u>-sim</u>	Specify simulator as default
<u>-simext*</u>	Specify a simulator-extension DLL
<u>-simext_dlls</u>	Specify path to simulator-extension DLLs
<u>-smem</u>	Specify simulator memory size
<u>-startup</u>	Specify start-up file
<u>-std*=file_name</u>	Redirect target's standard in, out, or error
<u>-timeout</u>	Specify number of times to retry "get status" on the parallel port
<u>-win</u>	Specify saved window layout to load
<u>-X*</u>	Enable specific ARC simulator extensions
<u>-Xnodetect</u>	Do not autodetect cache and peripheral hardware
<u>-Xxylsbasesx</u>	Set the base address of X memory
<u>-Xxylsbasey</u>	Set the base address of Y memory

Appendix E — Quick Toggles List

In This Chapter

- [Overview](#)
- [Debugger Toggles](#)

Overview

This section presents a quick-reference of debugger toggles. For complete listings, see [Toggle Reference](#) on page 298.

For information on how to specify toggles, see [Specifying Controls](#) on page 182.

Debugger Toggles

all_regs_volatile	Prevent register caching by the debugger
binary_stdin	Set debugger stdin to binary mode
binary_stdout	Set debugger stdout to binary mode
bpreg	Enable a hardware breakpoint for the ISS
brk_in_user_mode	Allow brk instructions to halt processor in user mode
bypass_aom	Hide internals when loading overlays
cache_target_memory	Cache memory contents
can_step	Use breakpoints to step
check_dcache_lock_bug	Detect unlocking of locked cache lines due to load collision
check_endian	Check endianness of target against application
check_hostlink_immediately	Check for hostlink activity after each instruction
check_kb	Enable keyboard input during execution
clear_REGS	Clear registers after download
cr_for_more	Enable carriage return for more display
cycle_step	Step cycles instead of completed instructions
cycles	Estimate CPU cycles
data_refs	Simulator counts number of instruction executions
deadbeef	Initialize memory to 0xdeadbeef
delay_killeds	Count non-execution of delay-slot instructions
delay_reg_write	Delay posting a register write value to the CPU
detect_uninitialized_data	Stop on uninitialized data
download	Download application being debugged to target memory
dud_quiet	Continue on uninitialized data but display message
enable_bypass	Enable bypass mode in cache
enable_dcache	Enable data cache upon board reset
enable_exceptions	Take exception rather than halt processing
error_box	Send selected errors to error box
examine_one	Show only one entry at a time in Examine window
fast_load_return	Write data returns from loads during writeback of computed results
fill_halts	Fill memory with halt instructions
flush_dcache	Flush data cache
flush_pipe	Clear target-processor pipeline before downloading

fujitsu_fast_serial_hostif	Use Fujitsu high-speed serial communication protocol
hostlink	Process host-link system calls
icnts	Count instruction executions for profiling
ignore_zero_debug_entities	Ignore functions and variables whose debug address is zero
image_window	Enable image display
include_local_symbols	Direct ELF reader to include local symbols
init_exception_vectors	Pre-initialize exception vectors in low memory
interrupt_extension	Permit interrupts 16–31
invalid_instruction_interrupt	Interrupt instead of halt on a bad instruction
invalidate_dcache	Detects and invalidates dcache prior to download
java_trap	Java traps Windows faults
killeds	Count number of times instructions not executed
ldst_from_code	Stop on code memory load or store
load_at_paddr	Load program segments at physical address
loadbeef	Stop when a program loads 0xdeadbeef from memory
memory_exception_interrupt	Interrupt instead of halt on a bad memory reference
minimal_interface	Debugger implements some target-interface functions
note_box	Send messages to text box
oldbp	Use pre-BRK breakpoints
prefer_soft_bp	Try software breakpoint first
program_zeros_bss	Do not download BSS
recursive_delay_slot	Permit single-step execution of delay slots within instructions
read_ro_from_exe	Request memory from read-only sections of the executable file
reload_readonly	Reload read-only portion of executable when restarting
reset_upon_restart	Reset target on restart
respect_threads	Enable thread awareness
restore_ap	Save and restore breakpoints
reuse_displays	Use existing windows instead of opening new ones
show_memory_loads	Display the contents of a pending memory load
show_reg_diffs	Show which registers have changed after an instruction single-step
spot_download_READONLY	Check for read-only code already on target
spot_verify_download	Verify memory operation during download
store_cache_data	Simulate cache RAM
swap_endian	Swap read/write chunks for target connection interface
trace	Trace instructions as they execute
trace_board_interface	Trace calls to interface functions
trace_board_interface2	Trace returns from interface functions
trace_cycles	Trace cycle stalls as they occur
trace_reg	Trace changes to any register

Debugger Toggles

[use_only_mapped_memory](#)

Do not use memory outside memory map

[workaround_dcache_lock_bug](#)

Flush locked lines individually to avoid bug

[xcheck](#)

Check ARC extensions on startup

Symbols

comment character in configuration file, 227
\$put_bank_reg(), 206
\$retcode, 221
\$tid, 269, 270
.sc.args.pset_name.multi (CMPD), 173
/n limiting modifier for commands, 186
/r repeat modifier for commands, 186
@ file, 15
\ continuation character in configuration file, 228
_Uncached, 310

Numerics

0xdead_beef
 command-line toggle, 301
 GUI setting, 21
32x16 multiply (command-line option), 296

A

abbreviating commands, 183
aborting (communications error), 127
acsi DLL, 146
action points
 barrier, *defined*, 169
 compound, 169
 for different processes, setting, 170
 stop list, *defined*, 169
actionpoints, 205
actionpoints (commands), 200
adding memory, 21
address
 and size of breakpoint table, option -break, 275
 for I/O communications, option -port, 289
 of auxiliary register range, option -auxlo, 274
 width, 56, 81, 258
ADDS (command-line option), 295
animating
 a process, 49
 multiple processes, 166
animating status, 35
animation
 and watchpoints, 67
 speed, 50
 command-line option, 274
 starting, 49
 stopping, 50
appearance settings, 28
application, embedded (command-line option), 280
ARC 700, specifying, 274

ARC Media, 143, 297
ARC xCAM, 155
-arc* — specify processor core version, 277
ARC_DLL, 126
ARC_hardware, 126
ARC_simulator, 310
ARCangel, 126, 134
 comparing to simulator, 158
ARCangel3
 and SCIT, 124
 remote blasting, 124
ARChitect configuration files, 15
ARGB, viewing, 84
args command, *defined*, 233
argument file, 15
arguments
 displaying with command args, 233
 displaying with command file, 243
 referencing macro, 226
arrays
 debugger (simulated), 189
 loading simulated (commands), 217
Ashling
 setting up an Opella emulator, 149
 GUI option, 20
assembly-level debugging, 44
attaching a running process, 233
autodetection, cache and peripherals, disabling (command-line option), 297
Automated Overlay Manager, 239, 299
automated testing, 15
auxiliary registers, 106, 206
 and run state, 206
 command-line options, 274
 dual-ported, 206
 GUI options, 24
 parsing data (commands), 196
 side effects, 293
 writing (commands), 197

B

barriers
 disabling and enabling, 171
 how they work, 170
 satisfied vs. unsatisfied, 170
base address, XY memory (command-line option), 297
base in Registers window, 106
binary file, loading to memory, 243
binding, deep or shallow, 74
blast
 ARCangel development boards, 138

BLASTFILE configuration variable, 275
 command-line option, 275
 GUI option, 20
 remote (AA3), 124
 board reset, 260
 board, configuring with chipinit file
 command-line option, 276
 GUI option, 23
 branch instruction to same location (toggle), 299
 BRC_DLL, 126
 BRC_hardware, 126
 BRC_simulator, 126, 310
 break command, 62, 187, 233
 breakpoint instruction (ARCtangent-A5), 44
 breakpoint table (command-line option), 275
 breakpoints
 cannot set, 52
 complex, 63
 defined, 60, 200
 delete with command delete break, 235
 disable with command disable break, 237
 display with command break, 233
 enable with command enable break, 240
 enabling and disabling, 65
 hardware, 61
 managing, 65, 68, 110
 mixed, 64
 on symbols not loaded, 249
 preferring software over hardware, 307
 saving and restoring, 309
 setting, 61
 commands, 200
 conditional, 63
 counted, 63
 temporary, 63, 249, 263
 thread-specific, 63, 109, 110
 CMPD, 167, 168
 with command break, 233
 stepping with (command-line toggle), 299
 types of, 63
 user mode (ARC 700)
 GUI option, 23
 toggle, 299
 without BRK, 307
 BRK instruction
 breakpoints without, 307
 in user mode (ARC 700), 299
 BSS (toggle for initializing), 308
 build-configuration registers, 145
 built-in variables, 189
 busy, 203
 BVICI memory controller, 289
 byte value
 set memory to (command), 254
 bytewise loading (commands), 216

C

C and C++ expressions
 entering valid, 222
 evaluation, 78
 C language, MetaWare extensions to, 223
 C or C++
 engine and ISS access, 279
 C++ functions, reducing output, 307
 cache
 algorithms, 93
 analysis, 153, 154
 coherency, 310
 data, simulating (command-line option), 277
 disabling autodetection (command-line option), 297
 instruction, simulating (command-line option), 281
 parameters and algorithms, 93
 thrashing (cycle estimation), 153
 window, enabling with command-line option, 144
 cache RAM (GUI option), 22
 call count, *defined*, 96
 call stack
 displaying, with command stack, 262
 evaluating expressions in scope for a function on the, 79
 function activation records *defined*, 76
 moving up and down in disassembly, 56
 Call Stack window and crashes, 283
 call target counts, 98
 call-graph display programs, 97
 calling functions (expression evaluator), 78, 235
 Can't display registers at this time, 206
 can't locate simulator extension, 296, 297
 capturing a trace, 59
 CAS, 155
 CAS DLL
 specifying on the command line, 278
 case-sensitivity
 command-line options, 11
 debugger commands, 183
 CDE/Motif look-and-feel, 38
 CES, 150
 changing
 expressions, 242
 memory values, 41
 register values, 41
 variable values, 41, 76
 character
 # in configuration file, comment, 227
 \ in configuration file, continuation, 228
 constants and string expressions, 222
 not supported, 222
 set memory to (command), 254
 checking extensions (toggle), 312
 chip initialization file, 130
 command-line option, 276
 C-language expression evaluator, 223
 clear registers after download (toggle), 300
 cloning
 a target system, 177, 263

closely coupled memory, 132, 285, 306
 CMPD, 158, 244
 adding processes, 173
 and SoCs, 146
 command-line options, 174, 176–177
 command-line usage, 172
 commands, 171–173
 completing, 172
 extensions, 158
 demo, 163
 including all command-line options, 176
 list command processors, 172
 macro share.scm, 175
 multi-way comparison of processes, 171
 option
 -multifile, 180
 session configuration file, 173
 simulator and hardware, 158
 state of processes, 172
 stopping processes, 172
 waiting for processes in focus to stop, 173
 code coverage, 102
 accessing, 102
 enabling, 102
 statement counts, 103
 code memory
 additional latency, 290
 stop on load/store (toggle), 306
 throughput, 290
 combining
 CMPD command output, 171
 CMPD configurations in one file, 176
 command
 combining CMPD output, 171
 executing after loading, 277
 executing with breakpoint, 63
 history, 246
 host system, 263
 keys, 29
 processor, 149
 multiple processors, 159
 single-process, 158
 Command: input field, 35, 187
 command-line debugger
 CMPD debugger, 172
 invoke with option -cl, 276
 command-line options, 274–298
 listing on the command line, 280
 commands, 183–221, 232–271
 and debugger variables, 184
 breakpoints, 200
 cache analysis, 219
 case sensitivity, 183
 completing CMPD, 172
 dumping
 command output to file, 250, 300
 entering at the debugger command prompt, 186
 exception registers, 194
 executing conditionally, 191, 246, 271
 executing when application halts, 216
 extension timers, 195
 for loading data, 216
 for loading simulated arrays, 217
 for reading registers, 193
 in a file, processing automatically, 256
 limiting output, 186
 log to file, with command log, 250
 logging to file, 250, 300
 MMU, 194
 modifiers, 185
 MPU registers, 194
 multiple per line, 183
 multi-word, 183
 over one line, 183
 pattern-matching output, 184
 prefixes, 184
 profiling counters, 218, 219
 reading files, 216
 reading memory, 198
 reference, 233–271
 scripts, 187
 SIMD registers, 194
 syntax, 183–184
 terminating a process, 204
 waiting for completion, 205
 watchpoints, 200
 whitespace in multi-word, 183
 writing auxiliary registers, 197
 writing files, 216
 writing memory, 199
 writing registers, 196
 comment character (#) in configuration file, 227
 communication
 with hardware using parallel port, 135
 communications error message, 127
 comparison of processes, CMPD multi-way, 171
 compiling
 with debug information, 43
 with optimizations, 44
 completing CMPD commands, 172
 complex breakpoints, *listed*, 63
 compound action point, *defined*, 169
 compound commands, 183
 conditional
 breakpoint, setting, 63
 debugger commands, 191, 246, 271
 macros, 225
 watchpoints, 68
 configuration file
 CMPD, 173
 comment character #, 227
 continuation character \, 228
 mdb.cnf, 228
 configuration variable BLASTFILE, 275
 configurations
 combining in one file, 176
 generating on the command line (CMPD), 176
 configuring

<p>a session on the command line, 173</p> <p>execution profiling, 97</p> <p>constants and string expressions, character, 222 in expressions, numeric, 222</p> <p>continuation character in configuration file, 228</p> <p>continue command, 235</p> <p>continuing current process with continue command, 235 with run command, 257</p> <p>controls reference, 274</p> <p>conventions debugger input, 221 regular expression, 221 regular expressions, 221 typographic, 11</p> <p>Coordinated Multiprocess Debugger <i>See CMPD</i></p> <p>copying memory (command), 251</p> <p>core register side effects, 293</p> <p>core version (command-line option), 277</p> <p>counted breakpoint, setting a, 63</p> <p>counters, profiling (command line), 218, 219</p> <p>counting instructions not executed (toggle), 306</p> <p>CPU cycles, estimating, 301</p> <p>current process restart with command restart, 257 start or continue with continue command, 235 with run command, 257 terminate with kill command, 205, 248</p> <p>current task, <i>defined</i>, 314</p> <p>currently in scope, evaluating expressions, 79</p> <p>cycle accuracy, 155</p> <p>cycle-accurate simulator, 155 and hostlink, 300</p> <p>cycle-estimating simulator, 150</p> <p>cycles estimating, 301 stepping on hardware GUI option, 23 toggle, 301</p>	<p>additional latency, 291 throughput, 291</p> <p>data returns and result writeback, 303</p> <p>data structures, viewing as image, 84</p> <p>data type in Registers window, 106</p> <p>data type names Nucleus PLUS, 326 ThreadX, 337</p> <p>dcache lock bug, 311</p> <p>dcache locking, 299</p> <p>dcache miss counts, 96, 98</p> <p>dchit register, 277</p> <p>dcmiss register, 277</p> <p>deadbeef, stopping on loads of (toggle), 301</p> <p>Debug a process or processes dialog, skipping (command-line option), 228, 288</p> <p>Debug Console, 27</p> <p>debug information, 43 in the linked program Nucleus PLUS, 320 ThreadX, 331</p> <p>debug projects command-line options, 43</p> <p>debugger entering valid C and C++ expressions in, 222 environment variables recognized by, 222 exit code, 221 exiting, 45 GUI, 24 GUI start-up time, reducing, 228 input, conventions for, 221 main menu, 25 multi-tasked features, enabling, 108 return code, 221 settings, 257 simulated arrays, 189 syntax for starting, 14 variables, 188 built-in, 189 simulated arrays, 189</p> <p>debugger-intrinsic functions, 206, 209, 210, 223, 223–224</p> <p>debugging assembly-level, 44 executable files, 112 files with no debug information, 112 flash, 302 functions with address 0, 305 in the field, 302 memory, 79 MQX applications, 316 multiple processes, 158 multiple tasks/threads, 314 multi-tasked applications, 107 multi-threaded applications, 314 non-symbolic, 112 Nucleus PLUS applications, 320, 324 on hardware, 134 optimized code, 305 preparing programs for, 43</p>
---	---

D

data
 loading (commands), 216
 uncached configuration register, 148

data cache
 analysis, 153, 154
 invalidating (toggle), 306
 pipeline depth, 291
 simulating
 command-line option, 91
 GUI option, 22
 simulating (command-line option), 277

data memory, 144

raw memory, 79
 registers and flags, 104
 shared memory, 175
 source-level, 43
 stripped object files, 112
 tasks
 with the command-line debugger, 315
 threads
 with the GUI debugger, 314
 ThreadX applications, 324, 331, 335
 variables with address 0, 305
 with ARC_DLL or BRC_DLL, 148
 with breakpoints, 60, 200
 with watchpoints, 66, 201
 debugging target (GUI setting), 19
 declaring
 and defining debugger macros, 224
 debugger macros using command, 226, 250
 deep binding, 74
 default
 options, ARG\$= specifies, 228
 target system, overriding, 116
 defined process sets, 159
 defining
 debugger macros, 224
 delay posting register write value (toggle), 302
 delay slot
 killed instruction counts, 96, 98
 delay slots
 executing during instructions, 308
 instructions not executed (toggle), 302
 delay_killed, *defined*, 96
 delete break command, 66, 200, 235
 delete watch command, 70, 201, 236
 deleting
 an entry from a memory map, 236
 delta (memory map), 129
 demo CMPD extensions, 163
 dereferencing a pointer, 75
 desktop settings, 28
 Detach Frame, 37
 detaching the current process, 236
 dir_xlation, 44
 directory
 changing, for source, 258, 278
 disable break command, 65, 237
 disable watch command, 70, 201, 237
 disabling profiling for faster simulation, 287
 disassemble command, 238
 Disassembly window
 customizing, 55
 moving up or down the call stack, 56
 displaying
 a pending memory load (toggle), 309
 call stack (command), 262
 displays
 timers, 101
 DLL
 ARC Media Extensions, 144

specifying
 command-line option, 177, 278
 for CAS
 command line, 278
 properties (command-line option), 279
 system properties, 149
 DLLs
 floating-point extensions, 294
 docking windows, 37
 dots in source display, 18
 double format, setting, 259, 291
 download
 executing commands after, 277
 flush pipeline first (toggle), 304
 multiple executables (command), 239
 skipping, 81
 toggle, 302
 verifying (toggle), 311
 verifying memory during, 310
 download failed error message, 127
 DPPF, 294
 drag-and-drop data tabs, 35
 driver
 configuration file, 16
 executable, 228
 gportio.dll, I/O port, 136
 DSP
 GUI options, 21
 simulating XMAC and XY memory, 132
 Version 3.1, 296
 DSP Pack A (command-line option), 296
 dual 16x16 XMAC, 297
 telephony, 297
 dual Viterbi butterfly instruction, 296
 dumping, 300
 command output to file, 250, 300
 memory 160 bytes at a time, 252
 memory to a file, 81
 memory to a file (command), 251
 DUNCBUILD register, 148

E

ECR register, 147
 EIA, 22
 ELF file, loading to memory, 243
 else command, 247
 elseif command, 247
 embedded application (command-line option), 280
 embedded processor, specifying (command-line option), 280
 enabling
 bypass mode in cache (toggle), 303
 data cache on board reset (toggle), 303
 dual Viterbi butterfly instruction (command-line option), 296
 ISS hardware breakpoint (toggle), 298
 watchpoints, 70, 201

endianness, checking, 300
 ENTER for more, 300
 entering
 C/C++ expressions, 222
 debugger commands
 at the debugger command prompt, 186
 in the GUI, 186
 values into memory, 240
 environment variables
 driver location, 228
 recognized by the debugger, 222
 source directory, 44
 error message
 communications, 127
 download failed, 127
 errors, 28, 303
 evalq, 197
 evaluate command and string arguments, 242
 evaluating
 and side effects, 79
 C/C++ expressions, 78
 expressions, 78, 79, 242
 with command evalq, 197, 241
 with command evaluate, 242
 functions, 78
 events, processor, 205
 EVICTED, 77
 evicted overlays, 130, 131
 Examine window
 showing only one entry, 303
 examining
 data, 73
 deeper levels in an aggregate data structure, 74
 expressions, 70, 74
 memory in various formats, 84
 nested structures, 74
 variables, 70, 73
 exception cause register, 147
 exception registers (ARC 700), 146
 exception registers (commands), 194
 exception vectors (ARC 700)
 GUI option, 23
 toggle, 305
 exceptions
 ARC 700, 303
 exceptions, detecting, 205
 exec command, *defined*, 243
 executable modules
 deleting, with command delete mod, 236
 executables, multiple, 239, 263
 execute command after download, 277
 executing commands when application halts, 216
 execution
 advancing (ISS), 260, 262
 stopping by instruction count, 117
 stopping on loads of a value, 134
 stopping with command go, 245
 execution stopped, 312
 exiting the debugger, 45
 with command exit, 243
 with command quit, 256
 expandable dividers, 39
 expressions
 changing value with command evalq, 197, 241
 changing value with command evaluate, 242
 character constants and string, 222
 conventions for regular, 221
 currently in scope, evaluating, 79
 entering valid C and C++, 222
 evaluation, 78, 79, 223, 235
 examining/modifying variables and, 70
 numeric constants in, 222
 operator precedence, 184
 operators, 223
 extended arithmetic (ARCtangent-A5 and later), 296
 extension
 memory, 127
 timer displays, 101
 extension auxiliary registers
 writing, 198
 extension timers, 207
 extension timers (commands), 195
 extensions
 and cycle estimation, 151
 C language, 223
 CMPD, 158
 CMPD, demo, 163
 disable checking (toggle), 312
 GUI options, 21
 simulator, 118
 GUI option, 21
 XMAC and XY Memory, DSP, 132

F

fast ISS, 155
 faster
 debugger response time, 66
 simulation, 288
 file
 .sc.args.pset_name.multi, CMPD session configuration, 173
 binary, load to memory
 with command file2mem, 243
 binary, write from memory, 81
 binary, write to memory, 81
 combining all configurations in a single, 176
 command, 243
 finding source at run time, 44
 mdb.cnf, configuration, 228
 reading with commands, 216
 writing with commands, 216
 file2mem command, 243
 filling memory, 244
 with halt instructions, 303
 flash
 programming, 155

float format, setting, 259, 291
 floating point extension (ARC), 22
 floating-point-extension DLLs, 294
 flushing
 data cache (toggle), 304
 pipeline (toggle), 304
 pipeline on reset (commands), 204
 following a pointer, 75
 font size, 29, 279
 format, changing in Registers window, 106
 formatted printing, 190
 FPGA development board, 126, 134, 135
 reinitializing (blasting), 275
 frame-relative Local Variables window, 69
 function
 call stack, *see* call stack
 evaluating (command), 235
 return from with command return, 257
 with debug address of zero, 305
 functions
 called most (profiling), 98

G

generating window, *defined*, 23
 global
 symbols in .so files, 279
 variables, displaying, 245
 going
 to a specific address, 245
 to main(), 280
 Goto: input field, 80
 gportio.dll, 136
 graphics, viewing, 84
 grayscale, viewing, 84
 -grep pattern-matching command prefix, 185
 GUI, 24
 invoking from command line, 280
 settings, 16, 19

H

halt, 203
 on bad instruction (toggle), 306
 on bad memory reference (toggle), 307
 using BRK in user mode (toggle), 299
 halting hardware, 206
 after a certain number of cycles, 206
 halting the instruction-set simulator, 207
 hardware
 breakpoints, 61, 307
 communicating with via parallel port, 135
 comparing to simulator, 158
 interrupts (with commands), 207
 watchpoints, 66
 watchpoints vs. software watchpoints, 66
 Harvard architecture *see* closely coupled memory
 heading, debugger macro, 225

heap
 JVM, 282
 Hello, World!, 190
 help
 command, 245
 online, 35, 43, 48
 hex address width, 56, 81, 258
 hex value, filling memory with, 244
 history
 log, Nucleus PLUS, 323
 of commands, 246
 HMSL, 188
 hostlink, 50, 155, 206, 207, 260, 262, 287
 and -nowait, 206, 207
 toggle, 300, 304

|

I/O
 speeding access, 18, 19
 I/O port
 address (command-line option), 289
 driver gportio.dll, 136
 icache miss counts, 96, 98
 ichit register, 281
 icmiss register, 281
 icnt
 defined, 95
 register, 117
 IDENTITY register, 291
 if ... endif commands, 191
 if command, 246
 if then command, 246
 ignoring entities with debug address zero, (toggle), 305
 Image window, 282
 image, viewing, 84
 including local symbols (toggle), 305
 initial value of memory, 286
 input, conventions for debugger, 221
 instruction
 not executed (toggle), 306
 stepping into (command), 247, 263
 stepping over (command), 248
 tracing (toggle), 310
 instruction cache, 91
 analysis, 153, 154
 command-line option, 281
 GUI option, 22
 instruction count, 95
 GUI option, 19
 profiling, 95, 98
 stopping by, 117
 toggle, 304
 Instruction Set Simulator
 extracting, 279
 instruction set simulator *see* simulator
 instructions executed before another operation (command-line option), 281

interface
 settings, 28
interrupt
 16 through 31, 305
 count, 96
 hardware (with commands), 207
 instead of halt (toggle), 306, 307
 software (with commands), 208
 vector extension (command-line option), 296
 vector table base address (command-line option), 282
interrupt on bad instruction
 GUI option, 22
 toggle, 306
interrupt on bad memory access
 GUI option, 22
 toggle, 307
interrupts 16-31
 command-line option, 296
 GUI option, 22
invoking
 a macro, 226
ISS *see* simulator
ISS, fast, 155

J

Java directory
 command-line option, 228, 282
 in configuration file, 228
Java Virtual Machine, heap size, 282
java.lang.OutOfMemoryError
 Java heap space, 282
JTAG, 282
 command-line option, 282
 connection, 134

K

Kanji characters not supported, 222
key bindings, 31
 for stepping, 29
keyboard
 disabling (toggle), 300
 shortcuts for commands and macros, 31
keys
 command, 29
 macro, 29
 stepping shortcuts, 29
kill command, 205, 248
killed instructions
 counts, 96, 98
 GUI option, 20
 toggle, 306

L

labeling lines of instruction history, 59
labels in Disassembly window, 54

libraries, command for listing shared, 255
LIMM cnt, *defined*, 95
LIMM counts, 95, 98
lines of command output, limiting, 186
linked program
 Nucleus PLUS debug information in the, 320
 ThreadX debug information in the, 331
linking information
 for Nucleus PLUS applications, 320
Linux parallel port driver, 137
listing currently defined macros, 227
load
 a dataset into memory, 81
 command, 216, 244
 a program (command), 249
 execute a command immediately after (command-line option), 283
 simulated arrays (commands), 217
 specifying addresses (command-line option), 282
 stop when program loads 0xdeadbeef from memory (toggle), 306
 stopping execution on loads of a value, 134
 symbols from an executable (command), 263
load/store
 RAM, 129
 GUI options, 21
 stop on code memory (toggle), 306
local variables, displaying with command, 249
locking
 and cloning task windows, 315
 cloned windows to a task or thread, 109
 windows to a thread or task, CMPD, 166, 168
log
 command, 250
 Nucleus PLUS history, 323
look-and-feel, 38
LRU value (overlays), 131

M

macro
 arguments, referencing, 226
 body, 224
 CMPD share.scm, 175
 conditional, 225
 declaring and defining a, 224
 declaring with command, 226
 definition, 225
 removing, 266
 terminate, 241, 254
 ending, 225, 254
 example, 225
 heading, 225
 invoking a, 226
 keys, 29
 undefining a, 227
 whitespace in arguments, 225
MADI registers, 210, 223

- main(), stopping before (command-line option), 49, 287
 marking lines of instruction history, 59
 MAX (command-line option), 296
 maximum
 interval between polls (command-line option), 283
 simulator memory (command-line option), 284
 size of displayed aggregates (command-line option), 283
 mdb.cnf configuration file, 227, 228, 277
 mdb.exe driver executable, 228
 mdb> prompt and commands, 186
 MDI desktop model, 28
 memory
 add
 GUI option, 21
 non-contiguous to the simulator, 127, 285
 to the simulator, 128
 command-line option, 285
 closely coupled, 132, 285
 command, 252
 copying (command), 251
 dumping
 command, 252
 to file (command), 251
 entering values, 240
 errors, 311
 extensions, 127
 GUI options, 21
 filling, 244
 initial value, specify with -memextinit, 286
 interrupt on bad reference (toggle), 307
 loading a file to (command), 243
 map, 129, 311
 a range to aux registers, 284
 reversed endianness (command), 232
 problems, 18
 reading (commands), 198
 reading on each execution event, 79
 searching, 88
 backward (command), 253
 forward (command), 253
 setting hardware breakpoints, 234
 setting to a value (command), 199, 254
 simulation, XY, 133
 simulator maximum, 284
 specifying
 bus width, 133, 285
 simulator size (command-line option), 294
 size, 128, 280
 command-line option, 284
 GUI option, 23
 stop on load/store (toggle), 306
 verifying during download, 310
 volatile, 79
 writing (commands), 199
 writing a file to or from, 81
 XY, 132
 Memory Management Unit (ARC 700), 146, 286
 Memory Protection Unit (ARC 600), 146, 287
 Memory-Queue Options toolbar, 323, 334
- Mentor Quicksim file, 244
 menu, 25
 Metal look-and-feel, 38
 MIN (command-line option), 296
 minimum polling interval (command-line option), 286
 mixed breakpoint, 64
 mixed trace, 238
 mlo register, 296
 MMU (ARC 700), 146, 286
 commands, 194
 MMUBUILD register, 148
 modes for gportio port driver, 136
 modifiers for commands, 186
 modifying
 memory values, 41
 register values, 41
 variable values, 41, 76
 modules, listing (command), 255
 Motorola S3 records, 81, 243
 Motorola S-record file, 342
 MPU (ARC 600), 146, 287
 MPU registers
 commands, 194
 MQX RTOS, 316
 MUL/MAC extension (command-line option), 296
 multiple command processors, 159
 multiplier extensions (command-line option), 296
 multiply extension with XMAC, 296
 multiprocess debugger extensions, CMPD, 158
 multi-tasked debugger features, enabling, 108
 multi-way comparison of processes, CMPD, 171
 multi-word debugger commands, 183

N

- name of current executable (command), 233, 243
 non-contiguous memory, adding to the simulator, 127, 285
 non-symbolic debugging, 112
 NORM (command-line option), 296
 Not stopped in known source, 52
 Not supported directly on target, 251
 -nowait, 185, 203, 205, 206, 207, 210
 and hostlink, 206
 -nowaitq, 185
 Nucleus PLUS
 applications, 321
 linking information for, 320
 data type names, 326
 debug information, 320
 history log, 323
 Nucleus State Options toolbar, 322
 numeric base in Registers window, 106
 numeric constants in expressions, 222

O

- offset (memory map), 129
 on-line help, 35, 43, 48

Opella emulators, 149
 GUI option, 20
 operator precedence in expressions, 184
 operators, expression, 223
 optimizing compilation, 44
 options file, ignoring (command-line option), 287
 Options toolbar
 Memory-Queue, 323, 334
 Nucleus State, 322
 ThreadX, 333
 options, command-line, 274–298
 listing on the command line, 280
 specifying in configuration file, 228
 specifying using dialogs, 16
 syntax, 182
 OS, specifying (command-line option), 288
 output
 combining command, 171
 limiting
 from C++ functions, 307
 from commands, 186
 overlay
 mapping table, 131
 overlays, 130, 239
 command-line display, 239
 evicted, 130, 131
 showing internal actions, 299
 Overlays window, 130
 overriding the default system, 116

P

pair action points, 269
 parallel connection (hardware), 134
 parallel port
 address (command-line option), 289
 communicating with hardware via, 135
 Linux driver, 137
 retry attempts (command-line option), 295
 PARC (parallel ARC comparator), 158
 parsing
 auxiliary register data (commands), 196
 register data (commands), 195
 pattern-matching command output, 184
 patterns in memory, searching for, 88
 pausing (sleep command), 205, 260
 pdisp, 289
 perfect call graphing, 98
 peripherals
 disabling autodetection (command-line option), 297
 displaying
 command-line option, 289
 GUI option, 24
 Physical address 0 already mapped, 263
 physical address, loading at (toggle), 306
 pipe flushing, 204
 pipeline display (ARC), 289
 pipeline, flushing (toggle), 304

pipemon, 289
 Plastic look-and-feel, 38
 PlasticXP look-and-feel, 38
 pointer, dereferencing (following), 75
 port
 driver, 136
 automatic mode, 136
 parallel
 address, 135
 command-line option, 289
 communicating with hardware, 135
 retry attempts (command-line option), 295
 Precise/MQX RTOS applications, 316
 preferences, 28
 print from C++ functions, reducing (toggle), 307
 printing to stdout, 190
 process
 CMPD multi-way comparison, 171
 current
 restart with command restart, 257
 start or continue with command run, 257
 terminate with command kill, 205, 248
 setting different action points for different, 170
 terminating (commands), 204
 process commands in a file (command), 256
 process set *defined*, 159
 processor
 core version (command-line option), 277
 halting with commands, 206
 resets, 204
 status and run control, 203
 profiling, 94
 command line, 218
 disabling for faster simulation, 287
 in a Profiling window, 96
 in a window, 94
 interrupts, 98
 count, 96
 statistical, 99
 Profiling memory allocation failure in runtime., 100
 profint, *defined*, 96
 program
 compiling with debug information, 43
 counter, changing, 206
 embedded (command-line option), 280
 restarting a, 50
 startup code
 accessing, 48
 stepping into, 49
 verifying download (toggle), 311
 Program has no statistical profiling data, 96
 Program output window, 49
 projects
 changing, 43
 command-line options, 43
 duplicating, 42
 removing, 42
 prompt, debugger command, 186
 prop command

defined, 255
lastpc, 117
maxlastpc, 117
trace, 117
prop use_aux, 198
properties
 specifying with -prop or -props, 290
 target system
 set with command prop, 255
 set with command sysprop, 263
pseudo-variables, 267
\$retcode, 221
\$tid, 269, 270

Q

Q integer types
 displaying memory as, 252
queue memory, 144
Quicksim hex file, 244
quitting the debugger, 45

R

radix
 changing in Registers window, 106
 modifier for memory command, 253
RAM, load/store, 129
RASCAL, 126, 155, 206, 207, 210, 289, 292
read and read/write watchpoints, 268, 270
read failed, 251
reading
 commands from a file, 256
 files (commands), 216
 from stdin, 292, 298
 memory (commands), 198
 registers (commands), 193
read-only, reloading (toggle), 308
real-time trace, 139
reevaluating a data structure, 74
reference
 commands, 232
 toggles, 298
referencing macro arguments, 226
-regex pattern-matching command prefix, 185
registers, 147, 148, 277, 281, 291
 BCR registers, 145
 caching, 298
 displaying (command), 256
 displaying auxiliary (command), 233
 for cycle estimation, 150
 from trace, 58
 MADI, 210, 223
 parsing data (commands), 195, 196
 reducing read traffic (toggle), 309
 removing a watchpoint, 68
 setting hardware breakpoints, 234
 setting watchpoints (command), 270

showing change (toggle), 309
side effects, 293
SIMD, 194, 239
tracing changes (toggle), 311
volatile, 298
writing (commands), 196
regression testing error reporting, 28
regular expressions, conventions for, 221
reloading read-only (toggle), 308
remote blasting (AA3), 124
removing
 a macro definition (command), 266
 register watchpoints, 68
repeat modifier for commands, 186
reset, 203
response file, 15
restarting a program or process, 50
 and resetting target (GUI option), 24
 and resetting target (toggle), 308
 command, 257
result writeback and data returns, 303
retries for parallel-port status (command-line option), 295
return code, debugger, 221
return command, 257
reusing displays (toggle), 309
reversed endianness and memory maps, 232
RGB, viewing, 84
RTOS, specifying (command-line option), 288
run, 203
run time, finding source code files at, 44
running
 a program, 48
 command-line option, 292
processes, 168
status, 35
stepping, stopping, and restarting (CMPD), 164
to an address *see* breakpoints
until data changes *see* watchpoints, 201

S

sash-panel desktop model, 29
satisfied *vs.* unsatisfied barriers, 170
saving a window to a file, 71, 72, 73, 78, 104
SCDIR environment variable, 228
scengine.h, 279
SCIT
 and ARCan3, 124
 GUI option, 20
 interface handler, 124
 password, 122, 123
 server hold time, 122, 123
 startup
 command line, 121
 GUI, 120
 server, 120
SCIT_PORT, 145
SCIT_SHMEM, 145

scope, evaluating expressions in, 79
 scratch RAM (command-line option), 296
 script file for debugger commands, 187
 scripts, 247
 arguments, 187
 reading in, 187
 SDI desktop model, 29
 search pattern
 escape sequences, 89
 memsearch command, 253
 searching
 backward through memory (command), 253
 forward through memory (command), 253
 in memory using the GUI, 88
 semantic inspection
 DLLs, specify with -semint, 292
 Windows faults, 306
 session configuration file for CMPD, 173
 setting
 breakpoints, 61
 conditional, 63
 counted, 63
 hardware, 61
 in a Source or Disassembly window, 61
 in the Set Breakpoint dialog, 62
 temporary, 63
 command, 263
 thread-specific, 63, 109, 110
 CMPD, 167, 168
 in the Set Breakpoint dialog, 110
 on a line of code, 109
 CMPD action points, 170
 debugger settings with command set, 257
 memory to a value (command), 254
 source path, 44
 watchpoints, 67, 68
 command, 268
 in the Set Watchpoint dialog, 68
 on function arguments, 69
 on registers, 67
 command, 270
 on variables, 67
 task-specific, 110, 111
 in the Set Watchpoint dialog, 111
 on a register, 111
 on a variable, 110
 settings, 28, 227
 change or display (command), 257
 shallow binding, 74
 share.scm (CMPD macro), 175
 shared
 libraries, listing (command), 255
 memory with hostlink, 178
 object, loading additional (command), 283
 SHIMMs, 95
 shortcut keys, 29
 showing a pending memory load (toggle), 309
 side effects
 auxiliary registers, 293
 command-line option, 293
 core registers, 293
 GUI option, 24
 in expression evaluation, 79
 sideefct, 293
 SIMD, 143
 auxiliary registers, 194, 239
 command-line option, 297
 DMA register, 194, 239
 GUI options, 143
 vector registers, 194, 239
 simulate command, 207
 simulating
 ADDS (command-line option), 295
 data cache (command-line option), 277
 instruction cache (command-line option), 281
 MAX (command-line option), 296
 MIN (command-line option), 296
 NORM (command-line option), 296
 scratch RAM (command-line option), 296
 SUBS (command-line option), 295
 SWAP (command-line option), 297
 telephony extended arithmetic (command-line option), 296
 telephony extensions with a single option, 297
 variable polynomial CRC instruction (command-line option), 295
 simulation
 speeding
 by disabling profiling (command-line option), 287
 by disabling tracing (command-line option), 288
 simulator, 126
 adding memory, with option -memext, 285
 adding non-contiguous memory, 127
 comparing to hardware, 158
 cycle-accurate, 155
 and hostlink, 300
 extensions, 118, 295
 command-line options, 295
 DLL, 118
 specifying (command-line option), 294
 GUI options, 21
 making faster, 287
 maximum memory (command-line option), 284
 size (command-line option), 294
 specifying as default (command-line option), 116, 293
 stopping after *x* instructions, 255
 single-process command processor, 158
 single-stepping and delay slots, 308
 skins, 38
 sleep command, 205, 260
 sliding pointers (command-line option), 296
 slow I/O, 18, 19
 SmaRT, 139
 snapshots of a window, 113
 SoC, 145
 and SCIT interface handler, 124
 software
 breakpoints (toggle), 307

interrupts (with commands), 208
 watchpoints, 49, 66
 vs. hardware watchpoints, 66

source code
 changing directory, 258, 278
 coverage, 102
 displaying with command source, 260
 finding files at run time, 44

source command, *defined*, 260

Source window
 troubleshooting, 52

SOURCE_PATH environment variable, 44

source-level debugging, 43

SPFP, 294

splash screen, skipping (command-line option), 228, 288

S-record file, 81, 243, 244, 342

ssi command, *defined*, 261

sso command, *defined*, 261

stack frames, maximum (option), 283

starting
 animation, 49
 current process
 with command continue, 235
 with command run, 257

startup
 code, 49, 287
 accessing, 48
 stopping in, 49

dialog, 228

file of commands (command-line option), 294

speeding (GUI debugger), 228

state registers, 302

statement
 counts, 102
 step into, with command ssi, 261
 step over, with command sso, 261

statement count, 96

statement count from instruction counts, 98

statistical profiling, 100

statistics on
 instruction counts
 GUI option, 19
 toggle, 304
 killed instructions
 GUI option, 20
 toggle, 306

status
 and run control, processor, 203
 bar, 35
 command, 206, 207, 259, 262
 register, 147

stdin, reading from, 292, 298

stdout, writing to, 292, 298

stepping
 breakpoints (command-line toggle), 299
 cycles (toggle), 301
 into program start-up code, 49
 keyboard shortcuts, 29
 machine instructions

into functions, with command isi, 247
 over function calls, with command iso, 248

source code statements
 into functions, with command ssi, 261
 over function calls, with command sso, 261

through instructions and into functions (command), 247
 through instructions, over functions (command), 248

Stmt cnt from icnt, *defined*, 96

stop lists (CMPD action points), 169

stopped status, 35

stopping
 animation, 50
 before main() (command-line option), 49, 287
 execution on loads of a value, 134
 in startup code, 49
 on code memory load/store (toggle), 306
 process execution, with command go, 245

store code to memory, stopping debugger on (toggle), 310

string
 arguments and evaluate command, 242
 expressions and character constants, 222
 filling memory with (command), 244

SUBS (command-line option), 295

support, product, 10

switches *See* options

symbol table, for non-symbolic debugging, 112

symbols, global, in .so files, 279

syntax
 command, 183
 for starting the debugger, 14
 toggle, 183

system
 cloning, 177
 overriding the default, 116
 properties
 set with command prop, 255
 set with command sysprop, 263

T

table, breakpoint (command-line option), 275

TAD (task aware DLL), 316

target
 GUI setting, 19
 reset on restart (GUI option), 24
 reset on restart (toggle), 308

system properties
 set with command prop, 255
 set with command sysprop, 263

Target Access Interface, 278

task *see* thread

Technical Support, 10

telephony
 dual 16x16 XMAC, 297
 extended arithmetic (command-line option), 296
 simulating with a single option, 297
 XY memory, 297

temporary breakpoint

defined, 63
 setting, 63
 with command tbreak, 249, 263
 terminating
 a macro definition with command endm, 241
 a macro definition with command mend, 254
 a process (commands), 204
 current process with command kill, 205, 248
 testing, automated, 15
 text display
 in the Memory: Queue window, 323, 334
 in the Nucleus State window, 321
 in the ThreadX State window, 332
 thread
 awareness, 308
 breakpoints, 63
 CMPD, 167
 setting, 63, 109, 110
 CMPD, 168
 command, 264
 ID, 269, 270, 315
 list (command), 264
 setting watchpoints, 111
 support, engaging, 324, 335
 switch (command), 264
 watchpoints
 setting, 110
 command, 269, 270
 ThreadX
 data type names, 337
 debug information, 331
 debugging applications, 331
 Options toolbar, 333
 timeout for parallel port (command-line option), 295
 timers, 207
 command-line options, 297
 displays, 101
 timers (commands), 195
 title bar, 36
 toggles, 298–312
 syntax, 183
 turning off, 288
 turning on, 288
 toolbar
 Memory-Queue Options, 323, 334
 Nucleus State Options, 322
 ThreadX Options, 333
 trace
 capture, 59
 mixed, 238
 real-time, 139
 tracing
 calls to interface functions (toggle), 311
 changes to a register (toggle), 311
 instructions, 117, 310
 instructions (toggle), 310
 trashing (cache), and cycle estimation, 153
 tree-structured window, defined, 74
 tx_api.h, 331

txdebug.o, 331
 typographic conventions, 11

UART
 command-line option for display, 289
 GUI option for display, 24
 Unable to comply for *filename*, 263
 undeclared identifier, 208, 242
 undefining a macro, 227
 unformatted printing, 190
 unmacro command, 227
 defined, 266
 unsatisfied barriers, vs. satisfied, 170

value, filling memory with, 244
 variable polynomial CRC, simulating (command-line option), 295
 variables, 305
 debugger, 188, 266
 debugger (built-in), 189
 debugger (simulated arrays), 189
 examining/modifying, 70
 global, display with command globals, 245
 local, display with command locals, 249
 viewing as image, 84
 with address zero (toggle), 305
 vcg graphing program, 97
 verifying download (toggle), 311
 viewing
 all tasks in a program, 108
 disassembly code, 56
 at a specific location, 56
 for a line of source code, 56
 function call stack, 76, 81
 functions, 77
 global variables, 70
 graphics or data, 84
 hardware data and instruction caches, 144
 hardware stack, 83
 history of instructions, 57
 last instructions executed, 117
 list of
 current breakpoints, 65
 current watchpoints, 69
 tasks, 108
 local variables, 71
 currently in scope, 72
 machine instructions, 54
 memory at a specific address, 80
 multiple frame-relative Local Variables windows concurrently, 73
 Nucleus PLUS queues, 324
 Nucleus PLUS-specific information, 324
 OS resources

in the Nucleus State window, 323
 in the ThreadX State window, 334
 queue elements, 324, 335
 simulated instruction and data caches, 91
 source code, 51
 at a breakpoint, 66
 for a function on the call stack, 53
 source files, 53
 state of a specific task, 108
 state of one or more threads, 108
 thread-specific breakpoints, 110
 thread-specific watchpoints, 112
 ThreadX queue elements, 324, 335
 ThreadX-specific information, 335
 variables local to a call-stack function, 73
 virtual address, loading at (toggle), 306
 visually inspecting data, 84
 Viterbi butterfly dual-word instruction *see* dual Viterbi butterfly instruction
 VLC auxiliary registers, 194, 239
 VMAC
 command-line option for display, 289
 GUI option for display, 24
 volatile memory, 79
 VRaptor, 297

W

waiting before terminating debugger thread, 279
 watch command, 268
 watchpoints, 205
 and animation, 67
 and icnt register, 117
 defined, 66, 201
 deleting, with command delete watch, 236
 disabling and enabling, 70
 disabling, with command disable watch, 237
 enabling, with command enable watch, 240
 hardware, 66
 hardware vs. software, 66
 managing, 68, 69, 70, 110
 on hardware, 205
 on read, write, or read/write, 268, 270
 pair, 269
 removing from registers, 68
 setting, 67, 68
 command, 268
 in the Set Watchpoint dialog, 68
 on a register (command), 270
 task-specific, 110, 111
 software, 66
 and animation, 49
 watchpoints (commands), 200
 watchreg command, 68

defined, 270
 wchar_t and Kanji characters, 222
 where dcache miss counts, 96, 98
 while command, 271
 while loops (debugger commands), 190
 whitespace in multi-word debugger commands, 183
 wide characters, 222
 width of hex addresses, 56, 81, 258
 windows
 docking, 37
 dragging to another window, 35
 for timers, 101
 layout file (command-line option), 295
 reuse old (toggle), 309
 showing the same process in all, 168
 snapshots, 113
 Windows faults, in debugger extensions, 306
 wordwise loading (commands), 216
 writing
 a binary file to memory, 81
 extension auxiliary registers, 198
 files (commands), 216
 from memory to a file, 81
 memory (commands), 199
 stopping execution on, 268, 270
 to stdout, 292, 298

X

xCAM, 155
 xISS, 155
 XMAC
 command-line options, 297
 DSP extensions, 132
 dual 16x16, 297
 dual 16x16 (telephony), 297
 simulation, 132
 XMAC with multiply extension, 296
 XY memory
 bus width (command-line option), 285
 DSP extensions, 132
 set base address (command-line option), 297
 simulation, 133
 telephony, 297

Y

YUV, viewing, 85

Z

zero debug address (toggle), 305

