

ARM® DS-5™

Version 5.17

ARM DS-5 Debugger User Guide

ARM®

ARM® DS-5™**ARM DS-5 Debugger User Guide**

Copyright © 2010-2013 ARM. All rights reserved.

Release Information**Document History**

Issue	Date	Confidentiality	Change
A	30 June 2010	Non-Confidential	First release
B	30 September 2010	Non-Confidential	Update for DS-5 version 5.2
C	30 November 2010	Non-Confidential	Update for DS-5 version 5.3
D	30 January 2011	Non-Confidential	Update for DS-5 version 5.4
E	30 May 2011	Non-Confidential	Update for DS-5 version 5.5
F	30 July 2011	Non-Confidential	Update for DS-5 version 5.6
G	30 September 2011	Non-Confidential	Update for DS-5 version 5.7
H	30 November 2012	Non-Confidential	Update for DS-5 version 5.8
I	28 February 2012	Non-Confidential	Update for DS-5 version 5.9
J	30 May 2012	Non-Confidential	Update for DS-5 version 5.10
K	30 July 2012	Non-Confidential	Update for DS-5 version 5.11
L	30 October 2012	Non-Confidential	Update for DS-5 version 5.12
M	15 December 2012	Non-Confidential	Update for DS-5 version 5.13
N	15 March 2013	Non-Confidential	Update for DS-5 version 5.14
O	14 June 2013	Non-Confidential	Update for DS-5 version 5.15
P	13 September 2013	Non-Confidential	Update for DS-5 version 5.16
Q	13 December 2013	Non-Confidential	Update for DS-5 version 5.17

Proprietary Notice

Words and logos marked with ® or ™ are registered trademarks or trademarks of ARM® in the EU and other countries, except as otherwise stated below in this proprietary notice. Other brands and names mentioned herein may be the trademarks of their respective owners.

Neither the whole nor any part of the information contained in, or the product described in, this document may be adapted or reproduced in any material form except with the prior written permission of the copyright holder.

The product described in this document is subject to continuous developments and improvements. All particulars of the product and its use contained in this document are given by ARM in good faith. However, all warranties implied or expressed, including but not limited to implied warranties of merchantability, or fitness for purpose, are excluded.

This document is intended only to assist the reader in the use of the product. ARM shall not be liable for any loss or damage arising from the use of any information in this document, or any error or omission in such information, or any incorrect use of the product.

Where the term ARM is used it means “ARM or any of its subsidiaries as appropriate”.

Confidentiality Status

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by ARM and the party that ARM delivered this document to.

Unrestricted Access is an ARM internal classification.

Product Status

The information in this document is Final, that is for a developed product.

Web Address

<http://www.arm.com>

Contents

ARM® DS-5™ ARM DS-5 Debugger User Guide

Preface

<i>About this book</i>	14
------------------------------	----

Chapter 1

Getting started with DS-5™ Debugger

1.1	<i>About the debugger</i>	1-18
1.2	<i>Debugger concepts</i>	1-19
1.3	<i>Launching the debugger from Eclipse</i>	1-21
1.4	<i>About DS-5™ headless command-line debugger</i>	1-22
1.5	<i>Headless command-line debugger options</i>	1-23
1.6	<i>Specifying a custom configuration database using the headless command-line debugger</i>	1-26
1.7	<i>DS-5 Debug perspective keyboard shortcuts</i>	1-28
1.8	<i>DS-5 Debugger command-line console keyboard shortcuts</i>	1-29
1.9	<i>Standards compliance in the DS-5 Debugger</i>	1-30

Chapter 2

Configuring and connecting to a target

2.1	<i>Types of target connections</i>	2-32
2.2	<i>Configuring a connection to a Fixed Virtual Platform (FVP)</i>	2-33
2.3	<i>Configuring a connection to a Linux target using gdbserver</i>	2-35
2.4	<i>Configuring a connection to a Linux Kernel</i>	2-37
2.5	<i>About configuring connections to a Linux target using Application Debug with Rewind Support</i>	2-39
2.6	<i>About configuring connections to an Android target using Native Application/Library Debug with Rewind Support</i>	2-44

2.7	Configuring a connection to a bare-metal target	2-47
2.8	Configuring an Event Viewer connection to a bare-metal target	2-49
2.9	About the target configuration import utility	2-51
2.10	Adding a new platform	2-53
2.11	Adding a new configuration database to DS-5	2-55
2.12	Exporting an existing launch configuration	2-57
2.13	Importing an existing launch configuration	2-60
2.14	Disconnecting from a target	2-62

Chapter 3

Working with the target configuration editor

3.1	About the target configuration editor	3-64
3.2	Target configuration editor - Overview tab	3-65
3.3	Target configuration editor - Memory tab	3-67
3.4	Target configuration editor - Peripherals tab	3-69
3.5	Target configuration editor - Registers tab	3-71
3.6	Target configuration editor - Group View tab	3-73
3.7	Target configuration editor - Enumerations tab	3-76
3.8	Target configuration editor - Configurations tab	3-78
3.9	Scenario demonstrating how to create a new target configuration file	3-80
3.10	Creating a power domain for a target	3-92
3.11	Creating a Group list	3-94
3.12	Importing an existing target configuration file	3-96
3.13	Exporting a target configuration file	3-98

Chapter 4

Controlling execution

4.1	About loading an image on to the target	4-101
4.2	About loading debug information into the debugger	4-103
4.3	About passing arguments to main()	4-105
4.4	Running an image	4-106
4.5	Working with breakpoints and watchpoints	4-107
4.6	Working with conditional breakpoints	4-114
4.7	About pending breakpoints and watchpoints	4-118
4.8	Setting a tracepoint	4-119
4.9	Setting Streamline start and stop points	4-120
4.10	Stepping through an application	4-121
4.11	Handling Unix signals	4-123
4.12	Handling processor exceptions	4-125
4.13	Configuring the debugger path substitution rules	4-127

Chapter 5

Examining the target

5.1	Examining the target execution environment	5-130
5.2	Examining the call stack	5-132
5.3	About trace support	5-133
5.4	About post-mortem debugging of trace data	5-136

Chapter 6

Debugging embedded systems

6.1	About endianness	6-138
6.2	About accessing AHB, APB, and AXI buses	6-139
6.3	About virtual and physical memory	6-140
6.4	About debugging hypervisors	6-141

6.5	About debugging big.LITTLE systems	6-142
6.6	About debugging bare-metal symmetric multiprocessing systems	6-143
6.7	About debugging multi-threaded applications	6-145
6.8	About debugging shared libraries	6-146
6.9	About debugging a Linux kernel	6-148
6.10	About debugging Linux kernel modules	6-150
6.11	About debugging FreeRTOS™	6-152
6.12	About debugging TrustZone enabled targets	6-153
6.13	About debugging a Unified Extensible Firmware Interface (UEFI)	6-155
6.14	About application rewind	6-156
6.15	About debugging ThreadX	6-158
6.16	About DTSL (Debug and Trace Service Layer)	6-159
6.17	About CoreSight™ Target Access Library	6-160
Chapter 7	Controlling runtime messages	
7.1	About semihosting and top of memory	7-162
7.2	Working with semihosting	7-163
7.3	Enabling automatic semihosting support in the debugger	7-164
7.4	Controlling semihosting messages using the command-line console	7-165
7.5	Controlling the output of logging messages	7-166
7.6	About Log4j configuration files	7-167
7.7	Customizing the output of logging messages from the debugger	7-168
Chapter 8	Debugging with scripts	
8.1	Exporting DS-5 Debugger commands generated during a debug session	8-172
8.2	Creating a DS-5 Debugger script	8-173
8.3	Creating a CMM-style script	8-174
8.4	About Jython scripts	8-175
8.5	Jython script concepts and interfaces	8-177
8.6	Creating Jython projects in Eclipse for DS-5™	8-179
8.7	Creating a Jython script	8-182
8.8	Running a script	8-184
Chapter 9	Working with the Snapshot Viewer	
9.1	About the Snapshot Viewer	9-187
9.2	Components of a Snapshot Viewer initialization file	9-189
9.3	Connecting to the Snapshot Viewer	9-192
9.4	Considerations when creating debugger scripts for the Snapshot Viewer	9-193
Chapter 10	DS-5 Debug perspectives and views	
10.1	App Console view	10-196
10.2	ARM Asm Info view	10-198
10.3	ARM assembler editor	10-199
10.4	Breakpoints view	10-202
10.5	C/C++ editor	10-206
10.6	Commands view	10-209
10.7	Debug Control view	10-212
10.8	Disassembly view	10-216
10.9	Events view	10-220
10.10	Expressions view	10-221

10.11	Functions view	10-224
10.12	History view	10-226
10.13	Memory view	10-228
10.14	Modules view	10-232
10.15	Registers view	10-236
10.16	RTOS Data view	10-239
10.17	Screen view	10-241
10.18	Scripts view	10-244
10.19	Target Console view	10-246
10.20	Target view	10-247
10.21	Trace view	10-249
10.22	Trace Control view	10-253
10.23	Variables view	10-255
10.24	Auto Refresh Properties dialog box	10-258
10.25	Memory Exporter dialog box	10-259
10.26	Memory Importer dialog box	10-260
10.27	Fill Memory dialog box	10-261
10.28	Export trace report dialog box	10-262
10.29	Breakpoint properties dialog box	10-264
10.30	Watchpoint properties dialog box	10-269
10.31	Tracepoint properties dialog box	10-270
10.32	Manage Signals dialog box	10-271
10.33	Event Viewer dialog box	10-273
10.34	Functions Filter dialog box	10-274
10.35	Jython Script Parameters dialog box	10-275
10.36	Debug Configurations - Connection tab	10-276
10.37	Debug Configurations - Files tab	10-279
10.38	Debug Configurations - Debugger tab	10-283
10.39	Debug Configurations - OS Awareness tab	10-286
10.40	Debug Configurations - Arguments tab	10-287
10.41	Debug Configurations - Environment tab	10-289
10.42	DTSL Configuration Editor dialog box	10-291
10.43	Configuration database panel	10-293
10.44	About the Remote System Explorer	10-295
10.45	Remote Systems view	10-296
10.46	Remote System Details view	10-297
10.47	Target management terminal for serial and SSH connections	10-298
10.48	Remote Scratchpad view	10-299
10.49	Remote Systems terminal for SSH connections	10-300
10.50	New Terminal Connection dialog box	10-301
10.51	DS-5 Debugger menu and toolbar icons	10-303

Chapter 11

Troubleshooting

11.1	ARM Linux problems and solutions	11-307
11.2	Enabling internal logging from the debugger	11-308
11.3	Target connection problems and solutions	11-309

Chapter 12

File-based flash programming in DS-5™

12.1	About file-based flash programming in DS-5™	12-311
------	---	--------

12.2	Flash programming configuration	12-314
12.3	Creating an extension database for flash programming	12-316
12.4	About using or extending the supplied Keil flash method	12-317
12.5	About creating a new flash method	12-319
12.6	About testing the flash configuration	12-323
12.7	About flash method parameters	12-324
12.8	About getting data to the flash algorithm	12-325
12.9	About interacting with the target	12-326

Chapter 13

Writing OS awareness for DS-5™ Debugger

13.1	About Writing operating system awareness for DS-5™ Debugger	13-334
13.2	Creating an OS awareness extension	13-335
13.3	Implementing the OS awareness API	13-339
13.4	Enabling the OS awareness	13-341
13.5	Implementing thread awareness	13-345
13.6	Implementing data views	13-348
13.7	Programming advice and noteworthy information	13-351

List of Figures

ARM® DS-5™ ARM DS-5 Debugger User Guide

Figure 2-1	Adding a new configuration database	2-56
Figure 2-2	Export launch configuration dialog box	2-57
Figure 2-3	Launch configuration selection panels	2-58
Figure 2-4	Import launch configuration dialog box	2-60
Figure 2-5	Launch configuration file selection panels	2-61
Figure 3-1	Target configuration editor - Overview tab	3-66
Figure 3-2	Target configuration editor - Memory tab	3-68
Figure 3-3	Target configuration editor - Peripherals tab	3-70
Figure 3-4	Target configuration editor - Registers tab	3-72
Figure 3-5	Target configuration editor - Group View tab	3-74
Figure 3-6	Target configuration editor - Enumerations tab	3-76
Figure 3-7	Target configuration editor - Configuration tab	3-79
Figure 3-8	LED register and bitfields	3-80
Figure 3-9	Core module and LCD control register	3-81
Figure 3-10	Creating a Memory map	3-82
Figure 3-11	Creating a peripheral	3-83
Figure 3-12	Creating a standalone register	3-84
Figure 3-13	Creating a peripheral register	3-85
Figure 3-14	Creating enumerations	3-86
Figure 3-15	Assigning enumerations	3-87
Figure 3-16	Creating remapping rules	3-88
Figure 3-17	Creating a memory region for remapping by a control register	3-89
Figure 3-18	Applying the Remap_RAM_block1 map rule	3-90

Figure 3-19	Applying the Remap_ROM map rule	3-91
Figure 3-20	Power Domain Configurations	3-92
Figure 3-21	Creating a group list	3-95
Figure 3-22	Selecting an existing target configuration file	3-96
Figure 3-23	Importing the target configuration file	3-97
Figure 3-24	Exporting to C header file	3-98
Figure 3-25	Selecting the files	3-99
Figure 4-1	Load File dialog box	4-101
Figure 4-2	Load additional debug information dialog box	4-104
Figure 4-3	Setting an execution breakpoint	4-109
Figure 4-4	Setting a data watchpoint	4-110
Figure 4-5	Viewing the properties of a data watchpoint	4-110
Figure 4-6	Breakpoint Properties dialog	4-115
Figure 4-7	Debug Control view	4-121
Figure 4-8	Managing signal handler settings	4-123
Figure 4-9	Manage exception handler settings	4-125
Figure 4-10	Path Substitution dialog box	4-127
Figure 4-11	Edit Substitute Path dialog box	4-128
Figure 5-1	Target execution environment	5-130
Figure 5-2	Debug Control view	5-132
Figure 6-1	Threading call stacks in the Debug Control view	6-145
Figure 6-2	Adding individual shared library files	6-146
Figure 6-3	Modifying the shared library search paths	6-147
Figure 7-1	Typical layout between top of memory, stack, and heap	7-162
Figure 8-1	Commands generated during a debug session	8-172
Figure 8-2	PyDev project wizard	8-179
Figure 8-3	PyDev project settings	8-180
Figure 8-4	Jython auto-completion and help	8-182
Figure 8-5	Scripts view	8-184
Figure 10-1	App Console view	10-196
Figure 10-2	ARM Asm Info view	10-198
Figure 10-3	ARM assembler editor	10-199
Figure 10-4	Breakpoints view	10-202
Figure 10-5	C/C++ editor	10-206
Figure 10-6	Show disassembly for selected source line	10-208
Figure 10-7	Commands view	10-209
Figure 10-8	Set the current working directory	10-212
Figure 10-9	Debug Control view	10-213
Figure 10-10	Disassembly view	10-216
Figure 10-11	Expressions view	10-221
Figure 10-12	Functions view	10-224
Figure 10-13	History view	10-226
Figure 10-14	Memory view	10-228
Figure 10-15	Modules view showing shared libraries	10-233
Figure 10-16	Modules view showing operating system modules	10-233
Figure 10-17	Registers view	10-236
Figure 10-18	Typical RTOS view for a RTX table	10-239
Figure 10-19	Screen buffer parameters for the Fireworks example running on a BeagleBoard	10-241
Figure 10-20	Screen view	10-242

Figure 10-21	Scripts view	10-244
Figure 10-22	Target view	10-247
Figure 10-23	Trace view with a scale of 1:1	10-250
Figure 10-24	Trace Control view	10-253
Figure 10-25	Variables view	10-255
Figure 10-26	Auto Refresh properties dialog box	10-258
Figure 10-27	Memory Exporter dialog box	10-259
Figure 10-28	Memory Importer dialog box	10-260
Figure 10-29	Memory Fill dialog box	10-261
Figure 10-30	Export trace report dialog box	10-263
Figure 10-31	Breakpoint properties dialog box	10-265
Figure 10-32	Watchpoint properties dialog box	10-269
Figure 10-33	Tracepoint properties dialog box	10-270
Figure 10-34	Managing signal handler settings	10-272
Figure 10-35	Manage exception handler settings	10-272
Figure 10-36	Function filter dialog box	10-274
Figure 10-37	Jython Script Parameters dialog box	10-275
Figure 10-38	Connection configuration for a model using VFS	10-278
Figure 10-39	File system configuration for an application on a model	10-280
Figure 10-40	Debugger configuration to set application starting point and search paths	10-285
Figure 10-41	OS Awareness tab	10-286
Figure 10-42	Application arguments configuration	10-288
Figure 10-43	Setting up target environment variables	10-289
Figure 10-44	Environment configuration for a model	10-290
Figure 10-45	DTSL configuration editor	10-292
Figure 10-46	Configuration Database panel	10-294
Figure 10-47	Remote Systems view	10-296
Figure 10-48	Remote System Details view	10-297
Figure 10-49	Terminal view	10-298
Figure 10-50	Remote Scratchpad	10-299
Figure 10-51	Remote Systems Terminals view	10-300
Figure 10-52	Terminal Settings dialog box	10-301
Figure 12-1	DS-5 File Flash Architecture	12-313
Figure 13-1	Eclipse preferences for mydb	13-336
Figure 13-2	Custom OS awareness displayed in Eclipse Debug Configurations dialog	13-338
Figure 13-3	myos No OS Support	13-341
Figure 13-4	myos waiting for target to stop	13-342
Figure 13-5	myos Enabled	13-343
Figure 13-6	myos waiting for OS initialization	13-344
Figure 13-7	myos Debug Control view data	13-346
Figure 13-8	myos Empty Tasks table	13-349
Figure 13-9	myos populated Tasks table	13-350

List of Tables

ARM® DS-5™ ARM DS-5 Debugger User Guide

<i>Table 2-1</i>	<i>ETM/PTM versions for each type of processor</i>	<i>2-52</i>
<i>Table 3-1</i>	<i>DMA map register SYS_DMAPSR0</i>	<i>3-80</i>
<i>Table 3-2</i>	<i>Control bit that remaps an area of memory</i>	<i>3-81</i>
<i>Table 7-1</i>	<i>Log4j Components</i>	<i>7-167</i>
<i>Table 10-1</i>	<i>Files tab options available for each Debug operation</i>	<i>10-279</i>
<i>Table 10-2</i>	<i>DS-5 Debugger icons</i>	<i>10-303</i>
<i>Table 10-3</i>	<i>Perspective icons</i>	<i>10-304</i>
<i>Table 10-4</i>	<i>View icons</i>	<i>10-304</i>
<i>Table 10-5</i>	<i>View markers</i>	<i>10-304</i>
<i>Table 10-6</i>	<i>Miscellaneous icons</i>	<i>10-305</i>

Preface

This preface introduces the *ARM® DS-5™ ARM DS-5 Debugger User Guide*.

It contains the following:

- *About this book on page 14.*

About this book

This book describes how to use the debugger to debug Linux applications, bare-metal, *Real-Time Operating System* (RTOS), Linux, and Android platforms.

Using this book

This book is organized into the following chapters:

Chapter 1 Getting started with DS-5™ Debugger

Gives an introduction to some of the debugger concepts and explains how to launch the debugger.

Chapter 2 Configuring and connecting to a target

Describes how to configure and connect to a debug target using ARM DS-5 Debugger in the Eclipse *Integrated Development Environment* (IDE).

Chapter 3 Working with the target configuration editor

Describes how to use the editor when developing a project for an ARM target.

Chapter 4 Controlling execution

Describes how to stop the target execution when certain events occur, and when certain conditions are met.

Chapter 5 Examining the target

This chapter describes how to examine registers, variables, memory, and the call stack.

Chapter 6 Debugging embedded systems

Gives an introduction to debugging embedded systems.

Chapter 7 Controlling runtime messages

Describes semihosting and how to control runtime messages.

Chapter 8 Debugging with scripts

Describes how to use scripts containing debugger commands to enable you to automate debugging operations.

Chapter 9 Working with the Snapshot Viewer

This chapter describes how to work with the Snapshot Viewer.

Chapter 10 DS-5 Debug perspectives and views

Describes the DS-5 Debug perspective and related views in the Eclipse *Integrated Development Environment* (IDE).

Chapter 11 Troubleshooting

Describes how to diagnose problems when debugging applications using DS-5 Debugger.

Chapter 12 File-based flash programming in DS-5™

This chapter describes the DS-5 file-based flash programming architecture, how to add support for new boards, and how to add support for new types of flash devices.

Chapter 13 Writing OS awareness for DS-5™ Debugger

This chapter contains information for programmers who wish to extend the operating system awareness in DS-5 Debugger to support additional operating systems.

Glossary

The ARM Glossary is a list of terms used in ARM documentation, together with definitions for those terms. The ARM Glossary does not contain terms that are industry standard unless the ARM meaning differs from the generally accepted meaning.

See the [ARM Glossary](#) for more information.

Typographic conventions

italic

Introduces special terminology, denotes cross-references, and citations.

bold

Highlights interface elements, such as menu names. Denotes signal names. Also used for terms in descriptive lists, where appropriate.

`monospace`

Denotes text that you can enter at the keyboard, such as commands, file and program names, and source code.

`monospace`

Denotes a permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name.

`monospace italic`

Denotes arguments to monospace text where the argument is to be replaced by a specific value.

`monospace bold`

Denotes language keywords when used outside example code.

<and>

Encloses replaceable terms for assembler syntax where they appear in code or code fragments. For example:

```
MRC p15, 0 <Rd>, <CRn>, <CRm>, <Opcode_2>
```

SMALL CAPITALS

Used in body text for a few terms that have specific technical meanings, that are defined in the *ARM glossary*. For example, IMPLEMENTATION DEFINED, IMPLEMENTATION SPECIFIC, UNKNOWN, and UNPREDICTABLE.

Feedback

Feedback on this product

If you have any comments or suggestions about this product, contact your supplier and give:

- The product name.
- The product revision or version.
- An explanation with as much information as you can provide. Include symptoms and diagnostic procedures if appropriate.

Feedback on content

If you have comments on content then send an e-mail to errata@arm.com. Give:

- The title.
- The number ARM DUI0446Q.
- The page number(s) to which your comments refer.
- A concise explanation of your comments.

ARM also welcomes general suggestions for additions and improvements.

Other information

- [*ARM Information Center.*](#)
- [*ARM Technical Support Knowledge Articles.*](#)
- [*Support and Maintenance.*](#)
- [*ARM Glossary.*](#)

Chapter 1

Getting started with DS-5™ Debugger

Gives an introduction to some of the debugger concepts and explains how to launch the debugger. It contains the following:

- *1.1 About the debugger on page 1-18.*
- *1.2 Debugger concepts on page 1-19.*
- *1.3 Launching the debugger from Eclipse on page 1-21.*
- *1.4 About DS-5™ headless command-line debugger on page 1-22.*
- *1.5 Headless command-line debugger options on page 1-23.*
- *1.6 Specifying a custom configuration database using the headless command-line debugger on page 1-26.*
- *1.7 DS-5 Debug perspective keyboard shortcuts on page 1-28.*
- *1.8 DS-5 Debugger command-line console keyboard shortcuts on page 1-29.*
- *1.9 Standards compliance in the DS-5 Debugger on page 1-30.*

1.1 About the debugger

DS-5™ Debugger provides a powerful tool for debugging applications on both hardware targets and models using ARM® architecture-based processors.

Using DS-5 Debugger, you can have complete control over the flow of the execution so that you can quickly isolate and correct errors.

The following features are provided:

- Loading images and symbols.
- Running images.
- Breakpoints and watchpoints.
- Source and instruction level stepping.
- Accessing variables and register values.
- Navigating the call stack.
- Support for handling exceptions and Linux signals.
- Debugging multi-threaded Linux applications.
- Debugging Linux kernel modules, boot code, and kernel porting.
- Debugging bare-metal *Symmetric MultiProcessing* (SMP) systems.

The debugger supports a comprehensive set of DS-5 Debugger commands that can be executed in the Eclipse *Integrated Development Environment* (IDE), script files, or a command-line console. In addition, there is a small subset of CMM-style commands sufficient for running target initialization scripts. CMM is a scripting language supported by some third-party debuggers. To execute CMM-style commands, you must create a debugger script file containing the CMM-style commands and then use the DS-5 Debugger source command to run the script.

To help you get started, there are some tutorials that you can follow showing you how to run and debug applications using DS-5 tools.

Related references

[2.1 Types of target connections on page 2-32.](#)

[1.2 Debugger concepts on page 1-19.](#)

Related information

[ARM DS-5 tutorials.](#)

[DS-5 Debugger commands.](#)

[CMM-style commands supported by the debugger.](#)

1.2 Debugger concepts

Lists the main concepts involved when debugging applications.

Debugger

A debugger is software running on a host computer that enables you to make use of a debug adapter to examine and control the execution of software running on a debug target.

Debug session

A debug session begins when you connect the debugger to a target or a model for debugging software running on the target and ends when you disconnect the host software from the target.

Debug target

At an early stage of product development there might be no hardware so the expected behavior of the hardware is simulated by software. This is referred to in the debugger documentation as a model. Even though you might run a model on the same computer as the debugger, it is useful to think of the target as a separate piece of hardware.

Alternatively, you can build a prototype product on a printed circuit board, including one or more processors on which you run and debug the application. This is referred to in the debugger documentation as a hardware target.

Debug adapter

A debug adapter performs the actions requested by the debugger on the target.

For example:

- Setting breakpoints.
- Reading from memory.
- Writing to memory.

The debug adapter is not the application being debugged, nor the debugger itself.

Examples include:

- Debug hardware adapter:
 - ARM DSTREAM™ unit.
 - ARM RVI™ unit.
 - ARM VSTREAM™ unit.
- Debug software adapter:
 - **gdbserver**.

Configuration database

The configuration database is where DS-5 Debugger stores information about the processors, devices, and boards it can connect to.

The database exists as a series of XML files, python scripts, and other miscellaneous files within the <DS-5 installation directory>/sw/debugger/configdb/directory.

DS-5 comes pre-configured with support for a wide variety of devices out-of-the-box, and you can view these within the Debug Configuration dialog within Eclipse IDE.

You can also add support for your own devices using the **cdbimport** tool.

Contexts

Each processor in the target can have a process currently in execution. Each process uses values stored in variables, registers, and other memory locations. These values can change during the execution of the process.

The context of a process describes its current state, as defined principally by the call stack that lists all the currently active calls.

The context changes when:

- A function is called.
- A function returns.
- An interrupt or an exception occurs.

Because variables can have class, local, or global scope, the context determines which variables are currently accessible. Every process has its own context. When execution of a process stops, you can examine and change values in its current context.

Scope

The scope of a variable is determined by the point within an application at which it is defined.

Variables can have values that are relevant within:

- A specific class only (class).
- A specific function only (local).
- A specific file only (static global).
- The entire application (global).

Related concepts

[1.4 About DS-5™ headless command-line debugger on page 1-22.](#)

Related tasks

[2.5.1 Connecting to an existing application and application rewind session on page 2-39.](#)

[2.5.2 Downloading your application and application rewind server on the target system on page 2-41.](#)

[2.5.3 Starting the application rewind server and debugging the target-resident application on page 2-42.](#)

[2.2 Configuring a connection to a Fixed Virtual Platform \(FVP\) on page 2-33.](#)

[2.3 Configuring a connection to a Linux target using gdbserver on page 2-35.](#)

[2.4 Configuring a connection to a Linux Kernel on page 2-37.](#)

[2.7 Configuring a connection to a bare-metal target on page 2-47.](#)

[2.8 Configuring an Event Viewer connection to a bare-metal target on page 2-49.](#)

[1.6 Specifying a custom configuration database using the headless command-line debugger on page 1-26.](#)

Related information

[Setting up the ARM DSTREAM Hardware.](#)

[Setting up the ARM RVI Hardware.](#)

1.3 Launching the debugger from Eclipse

Describes how to launch Eclipse and select the DS-5 Debug perspective.

Procedure

1. Launch Eclipse:
 - On Windows, select **Start > All Programs > ARM DS-5 > Eclipse for DS-5**.
 - On Linux:
 - If you installed the shortcut during installation, you can select **Eclipse for DS-5** in the **Applications** menu.
 - If you did not install the shortcut during installation:
 1. Add the `install_directory/bin` directory to your PATH environment variable. If it is already configured then you can skip this step.
 2. Open Unix bash shell.
 3. Enter `eclipse` at the prompt.
2. Select **Window > Open Perspective > DS-5 Debug** from the main menu.
3. To connect to the target:
 - If you have not run a debug session before then you must configure a connection between the debugger and the target before you can start any debugging tasks.
 - If you have run a debug session before then you can select a target connection in the **Debug Control** view and click on the **Connect to Target** toolbar icon.

Related tasks

- [1.5 Headless command-line debugger options on page 1-23.](#)
- [2.5.1 Connecting to an existing application and application rewind session on page 2-39.](#)
- [2.5.2 Downloading your application and application rewind server on the target system on page 2-41.](#)
- [2.5.3 Starting the application rewind server and debugging the target-resident application on page 2-42.](#)
- [2.2 Configuring a connection to a Fixed Virtual Platform \(FVP\) on page 2-33.](#)
- [2.3 Configuring a connection to a Linux target using gdbserver on page 2-35.](#)
- [2.4 Configuring a connection to a Linux Kernel on page 2-37.](#)
- [2.7 Configuring a connection to a bare-metal target on page 2-47.](#)
- [2.8 Configuring an Event Viewer connection to a bare-metal target on page 2-49.](#)

Related references

- [2.1 Types of target connections on page 2-32.](#)
- [10.36 Debug Configurations - Connection tab on page 10-276.](#)
- [10.37 Debug Configurations - Files tab on page 10-279.](#)
- [10.38 Debug Configurations - Debugger tab on page 10-283.](#)
- [10.40 Debug Configurations - Arguments tab on page 10-287.](#)
- [10.41 Debug Configurations - Environment tab on page 10-289.](#)
- [10 DS-5 Debug perspectives and views on page 10-194.](#)
- [10.7 Debug Control view on page 10-212.](#)

1.4 About DS-5™ headless command-line debugger

DS-5 Debugger can operate in a headless command-line mode.

You launch DS-5 headless command-line debugger using the DS-5 Command Prompt or Unix shell with the required environment variables set. You must provide command-line arguments to the debugger to select which target to connect to from those available in the debugger's configuration database.

1.5 Headless command-line debugger options

You can use command-line options listed below to configure the DS-5 headless command-line debugger.

Launch the headless command-line debugger using the following syntax:

```
debugger [--option <arg>] ...
```

Where:

debugger

Invokes the DS-5 headless command-line debugger.

[--option <arg>]

Option and its arguments to configure the command-line debugger.

...

Any further options.

———— **Note** ————

When connected, use DS-5 Debugger commands to access the target and start debugging.

For example, **info registers** displays all application level registers.

Options

-h or --help

Displays a summary of the main command-line options.

--cdb-list [filter]

Lists entries in the configuration database. This option is typically used on its own and does not connect to any target.

Where *[filter]* is a string with this syntax

[<manufacturer>[:<board>[:<project type>[:<activity>::<connection type>]]]]For example:

```
debugger --cdb-list
```

This lists all first level entries in the configuration database.

```
debugger --cdb-list="Altera"
```

This lists all the configuration database entries related to the Altera platform.

--cdb-entry <arg>

Specifies the option to define an entry in the configuration database.

<arg> is a string with this syntax <manufacturer>::<board>::<project type>::<activity>:: <connection type>

--cdb-entry-param <arg>

Specifies the option to define connection parameters.

Use *<arg>* to specify connection parameters in the syntax: key=value key2=value2
--cdb-entry-param "key3=some_key=some comma separated value" ...key=value key2=value2 key3=some_key=some comma separated value specifies the configuration database connection options.

--cdb-root <arg>

Specifies configuration database locations in addition to the debugger's default configuration database.

———— Note ————

If you do not need any data from the default configuration database, use the additional command line option **--cdb-root-ignore-default** to tell the debugger not to use the default configuration database.

———— Note ————

The order in which configuration database roots are specified is important when the same information is available in different databases. That is, the data in the location typed last (nearest to the end of full command line) overrides data in locations before it.

--cdb-root-ignore-default

Ignores the default configuration database.

--continue_on_error=true | false

Specifies whether the debugger stops the target and exits the current script when an error occurs.

The default is **--continue_on_error=false**.

--stop_on_connect=true | false

Specifies whether the debugger stops the target when it connects to the target device. To leave the target unmodified on connection, you must specify **false**. The default is **--stop_on_connect=true**.

--disable-semihosting

Disables semihosting operations.

--disable_semihosting_console

Disables all semihosting operations to the debugger console.

--enable-semihosting

Enables semihosting operations.

-i <arg> or **--semihosting-input <arg>**

Specifies a file to read semihosting stdin.

-o <arg> or **--semihosting-output <arg>**

Specifies a file to write semihosting stdout.

-e <arg> or **--semihosting-error <arg>**

Specifies a file to write semihosting stderr.

--top_mem=address

Specifies the stack base, also known as the top of memory. Top of memory is only used for semihosting operations.

-b=filename or **--image=filename**

Specifies the image file for the debugger to load when it connects to the target.

--interactive

Specifies interactive mode that redirects standard input and output to the debugger from the current command-line console, for example, Windows Command Prompt or Unix shell.

———— Note ————

This is the default if no script file is specified.

--log_config=<arg>

Specifies the type of logging configuration to output runtime messages from the debugger.

The <arg> can be:

<info> - Output messages using the predefined INFO level configuration. This level does not output debug messages. This is the default.

<debug> - Output messages using the predefined DEBUG level configuration. This option outputs both INFO level and DEBUG level messages.

<filename> - Specifies a user-defined logging configuration file to customize the output of messages. The debugger supports log4j configuration files.

--log_file=filename

Specifies an output file to receive runtime messages from the debugger. If this option is not used then output messages are redirected to the console.

--script=filename

Specifies a script file containing debugger commands to control and debug your target. You can repeat this option if you have several script files. The scripts are run in the order specified and the debugger quits after the last script finishes. Add the **--interactive** option to the command-line if you want the debugger to remain in interactive mode after the last script finishes.

Related concepts

[7.6 About Log4j configuration files on page 7-167.](#)

Related tasks

[1.3 Launching the debugger from Eclipse on page 1-21.](#)

Related references

[1.8 DS-5 Debugger command-line console keyboard shortcuts on page 1-29.](#)

[8.1 Exporting DS-5 Debugger commands generated during a debug session on page 8-172.](#)

[7.1 About semihosting and top of memory on page 7-162.](#)

Related information

[Log4j in Apache Logging Services.](#)

[DS-5 Debugger commands.](#)

1.6 Specifying a custom configuration database using the headless command-line debugger

You can use the headless command-line debugger to specify a custom configuration database.

To specify a custom configuration database using the headless command-line debugger:

Procedure

1. Launch a DS-5 command-line console.
 - On Windows, select **Start > All Programs > ARM DS-5 > DS-5 Command Prompt**.
 - On Linux:
 - Add the *install_directory/bin* directory to your PATH environment variable. If it is already configured, then you can skip this step.
 - Open a Unix bash shell.

2. Specify a configuration database location using the following command-line syntax:
`debugger ... --cdb-root /path/to/cdb[:/;/path/to/another/cdb]`

Where:

`--cdb-root`

Is the option to define the path to the configuration database.

`/path/to/cdb[:/;/path/to/another/cdb]`

Specifies the path to a configuration database or, if needed, additional configuration databases.

———— Note ————

DS-5 processes the databases from top to bottom with the information in the lower databases replacing information in the higher databases. For example, if you want to produce a modified Cortex™-A15 processor definition with different registers, then those changes can be added to a new database that resides lower down in the list.

———— Note ————

If you do not need any data from the default configuration database, use the additional command line option `--cdb-root-ignore-default` to tell the debugger not to use the default configuration database.

3. Specify a configuration database entry using the following command-line syntax:
`debugger --cdb-entry "<manufacturer>::<board>::<project type>::<execution_environment>::<activity>::<connection type>"`

Where:

`--cdb-entry`

Specifies the option to define a configuration database entry.

`<manufacturer> <board> <project type> <activity> <connection type>`

4. If connection parameters are required, specify them using the following command-line syntax after the `"..."` which denotes the configuration database entry.

`debugger --cdb-entry "..." --cdb-entry-param key=value key2=value2 --
cdb-entry-param "key3=some_key=some comma separated value" ...`

Where:

`--cdb-entry "..."`

Specifies the configuration database entry.

`--cdb-entry-param`

Specifies the option to define connection parameters.

`key=value key2=value2 key3=some_key=some comma separated value`

Specifies the configuration database connection options.

1.7 DS-5 Debug perspective keyboard shortcuts

Lists the keyboard shortcuts that you can use in the DS-5 Debug perspective.

In any view or dialog box you can access the dynamic help by using the following:

- On Windows, **F1** key
- On Linux for example, **Shift+F1** key combination.

The following keyboard shortcuts are available when you connect to a target:

Commands view

You can use:

Ctrl+Space

Access the content assist for autocompletion of commands.

Enter

Execute the command that is entered in the adjacent field.

DOWN arrow

Navigate down through the command history.

UP arrow

Navigate up through the command history.

Debug Control view

You can use:

F5

Step at source or instruction level including stepping into all function calls where there is debug information. You can also use **ALT+F5** to step in the opposite mode. For example, if you are in source level stepping mode then using **ALT+F5** performs an instruction level step.

F6

Step at source or instruction level but stepping over all function calls.

F7

Continue running to the next instruction after the selected stack frame finishes.

F8

Continue running the target.

Note

A **Connect only** connection might require setting the PC register to the start of the image before running it.

F9

Interrupt the target and stop the current application if it is running.

Related tasks

[1.3 Launching the debugger from Eclipse on page 1-21.](#)

Related references

[10.6 Commands view on page 10-209.](#)

[10.7 Debug Control view on page 10-212.](#)

1.8 DS-5 Debugger command-line console keyboard shortcuts

Lists the line editing features provided, including a command history and some common keyboard shortcuts

Each command you enter is stored in the command history. Use the UP and DOWN arrow keys to navigate through the command history, to find and reissue a previous command.

To make editing commands and navigating the command history easier, the following special keyboard shortcuts are available:

Ctrl+A

Move the cursor to the start of the line.

Ctrl+D

Quit the debugger console.

Ctrl+E

Move the cursor to the end of the line.

Ctrl+N

Search forward through the command history for the currently entered text.

Ctrl+P

Search back through the command history for the currently entered text.

Ctrl+W

Delete the last word.

DOWN arrow

Navigate down through the command history.

UP arrow

Navigate up through the command history.

Related tasks

[1.5 Headless command-line debugger options on page 1-23.](#)

1.9 Standards compliance in the DS-5 Debugger

Lists the level of compliance that DS-5 Debugger conforms to.

ELF

The debugger can read executable images in ELF format.

DWARF

The debugger can read debug information from ELF images in the DWARF 2, DWARF 3, and DWARF 4 formats.

Trace Protocols

The debugger can interpret trace that complies with the ETMv3.4, ETMv3.5, ETMv4, ITM and STM protocols.

———— Note —————

The DWARF 2 and DWARF 3 standard is ambiguous in some areas such as debug frame data. This means that there is no guarantee that the debugger can consume the DWARF produced by all third-party tools.

Related information

[ELF for the ARM Architecture.](#)

[DWARF for the ARM Architecture.](#)

[The DWARF Debugging Standard.](#)

[International Organization for Standardization.](#)

Chapter 2

Configuring and connecting to a target

Describes how to configure and connect to a debug target using ARM DS-5 Debugger in the Eclipse *Integrated Development Environment* (IDE).
It contains the following:

- [2.1 Types of target connections on page 2-32.](#)
- [2.2 Configuring a connection to a Fixed Virtual Platform \(FVP\) on page 2-33.](#)
- [2.3 Configuring a connection to a Linux target using gdbserver on page 2-35.](#)
- [2.4 Configuring a connection to a Linux Kernel on page 2-37.](#)
- [2.5 About configuring connections to a Linux target using Application Debug with Rewind Support on page 2-39.](#)
- [2.6 About configuring connections to an Android target using Native Application/Library Debug with Rewind Support on page 2-44.](#)
- [2.7 Configuring a connection to a bare-metal target on page 2-47.](#)
- [2.8 Configuring an Event Viewer connection to a bare-metal target on page 2-49.](#)
- [2.9 About the target configuration import utility on page 2-51.](#)
- [2.10 Adding a new platform on page 2-53.](#)
- [2.11 Adding a new configuration database to DS-5 on page 2-55.](#)
- [2.12 Exporting an existing launch configuration on page 2-57.](#)
- [2.13 Importing an existing launch configuration on page 2-60.](#)
- [2.14 Disconnecting from a target on page 2-62.](#)

2.1 Types of target connections

To debug an application using DS-5, you must set up a connection between the host workstation running the debugger and the target.

There are several types of connections supported by the debugger:

Linux applications

To debug a Linux application you can use a TCP or serial connection to:

- **gdbserver** running on a model that is pre-configured to boot ARM Embedded Linux.
- **gdbserver** running on a hardware target.
- Application rewind server running on a hardware target.

This type of development requires **gdbserver** or the application rewind server to be installed and running on the target.

Note

- If **gdbserver** is not installed on the target, either see the documentation for your Linux distribution or check with your provider. Alternatively, you might be able to use the **gdbserver** from the DS-5 installation at `install_directory/arm`.
- The application rewind server file `undodb-server` can be found in the `install_directory\DS-5\arm\undodb\linux` folder.

Bare-metal and Linux kernel

To debug an application running on a bare-metal target, a Linux kernel, or a kernel device driver, you can use a debug hardware adapter connected to the host workstation and the target.

Snapshot Viewer

The Snapshot Viewer enables you to debug a read-only representation of your application using previously captured state.

Note

Currently DS-5 only supports DS-5 Debugger connections to the Snapshot Viewer using the command-line console.

Related concepts

[9.1 About the Snapshot Viewer on page 9-187.](#)

[6.14 About application rewind on page 6-156.](#)

Related tasks

[2.5.1 Connecting to an existing application and application rewind session on page 2-39.](#)

[2.5.2 Downloading your application and application rewind server on the target system on page 2-41.](#)

[2.5.3 Starting the application rewind server and debugging the target-resident application on page 2-42.](#)

[2.6.1 Attaching to a running Android application on page 2-44.](#)

[2.6.2 Downloading and debugging an Android application on page 2-45.](#)

2.2 Configuring a connection to a *Fixed Virtual Platform* (FVP)

Describes how to configure a connection to a FVP and set up a *Virtual File System* (VFS).

DS-5 supports serial connections between a FVP and the host machine on both Windows and Linux platforms.

Procedure

1. Select **Window > Open Perspective > DS-5 Debug** from the main menu.
2. Select **Debug Configurations...** from the **Run** menu.
3. Select **DS-5 Debugger** from the configuration tree and then click on **New** to create a new configuration.
4. In the Name field, enter a suitable name for the new configuration.
5. Click on the **Connection** tab to configure a DS-5 Debugger target connection:
 - a) Select the required FVP platform, **Linux Application Debug** project type and the required debug operation. For example, if you are using a VFS then select **Debug target resident application**.
 - b) In the Connections panel, a serial connection is automatically configured.
 - c) If you are using VFS, select **Enable virtual file system support**. The default VFS mounting point maps the Eclipse workspace root directory to the `/writeable` directory on the model. Leave the default or change as required.

Note

VFS is only set-up on initialization of the model. Changes to the VFS directory structure might require restarting the model.

6. Click on the **Files** tab to define the target environment and select debug versions of the application file and libraries on the host that you want the debugger to use.
 - a) In the Target Configuration panel, specify the location of the application on the target. You can also specify the target working directory if required.
 - b) In the Files panel, select the files on the host that you want the debugger to use to load the debug information.

Note

Options in the **Files** tab are dependent on the type of debug operation that you select.

7. Click on the **Debugger** tab to configure the debugger settings.
 - a) Specify the actions that you want the debugger to do after connection to the target.
 - b) Configure the host working directory or use the default.
 - c) Configure the search paths on the host used by the debugger when it displays source code.
8. If required, click on the **Arguments** tab to enter arguments that are passed to the application when the debug session starts.
9. If required, click on the **Environment** tab to create and configure the target environment variables that are passed to the application when the debug session starts.
10. Click on **Apply** to save the configuration settings.
11. Click on **Debug** if you want to connect to the target and begin debugging immediately.

Alternatively, click on **Close** to close the Debug Configurations dialog box. Use the Debug Control view to connect to the target associated with this debug configuration.
12. Debugging requires the DS-5 Debug perspective. If the Confirm Perspective Switch dialog box opens, click on **Yes** to switch perspective.

When connected and the DS-5 Debug perspective opens you are presented with all the relevant views and editors.

For more information on these options, use the dynamic help.

Related tasks

- 2.12 Exporting an existing launch configuration on page 2-57.*
- 2.13 Importing an existing launch configuration on page 2-60.*
- 2.3 Configuring a connection to a Linux target using gdbserver on page 2-35.*
- 2.4 Configuring a connection to a Linux Kernel on page 2-37.*
- 2.7 Configuring a connection to a bare-metal target on page 2-47.*
- 2.8 Configuring an Event Viewer connection to a bare-metal target on page 2-49.*

Related references

- 10.36 Debug Configurations - Connection tab on page 10-276.*
- 10.37 Debug Configurations - Files tab on page 10-279.*
- 10.38 Debug Configurations - Debugger tab on page 10-283.*
- 10.39 Debug Configurations - OS Awareness tab on page 10-286.*
- 10.40 Debug Configurations - Arguments tab on page 10-287.*
- 10.41 Debug Configurations - Environment tab on page 10-289.*

Related information

Model Shell options for Fast Models.

2.3 Configuring a connection to a Linux target using gdbserver

Describes how to connect to a Linux target using **gdbserver**.

Prerequisites

Before connecting you must:

1. Set up the target with an *Operating System* (OS) installed and booted. See the documentation supplied with the target for more information.
2. Obtain the target IP address or name.
3. If required, set up a *Remote Systems Explorer* (RSE) connection to the target.

If you are connecting to an already running **gdbserver** you must ensure that you have:

1. **gdbserver** installed and running on the target.

To run **gdbserver** and the application on the target you can use:

```
gdbserver port path/myApplication
```

Where:

- *port* is the connection port between **gdbserver** and the application.
 - *path/myApplication* is the application that you want to debug.
2. An application image loaded and running on the target.

Procedure

1. Select **Window > Open Perspective > DS-5 Debug** from the main menu.
2. Select **Debug Configurations...** from the **Run** menu.
3. Select **DS-5 Debugger** from the configuration tree and then click on **New** to create a new configuration.
4. In the Name field, enter a suitable name for the new configuration.
5. Click on the **Connection** tab to configure a DS-5 Debugger target connection:
 - a) Select the required platform, **Linux Application Debug** project type and the required debug operation.
 - b) Configure the connection between the debugger and **gdbserver**.
6. Click on the **Files** tab to define the target environment and select debug versions of the application file and libraries on the host that you want the debugger to use.
 - a) In the Target Configuration panel, select the application on the host that you want to download to the target and specify the location on the target where you want to download the selected file.
 - b) In the Files panel, select the files on the host that you want the debugger to use to load the debug information. If required, you can also specify other files on the host that you want to download to the target.

Note

Options in the **Files** tab are dependent on the type of debug operation that you select.

7. Click on the **Debugger** tab to configure the debugger settings.
 - a) In the Run control panel, specify the actions that you want the debugger to do after connection to the target.
 - b) Configure the host working directory or use the default.

- c) In the Paths panel, specify any source or library search directories on the host that the debugger uses when it displays source code.
- 8. If required, click on the **Arguments** tab to enter arguments that are passed to the application when the debug session starts.
- 9. If required, click on the **Environment** tab to create and configure the target environment variables that are passed to the application when the debug session starts.
- 10. Click on **Apply** to save the configuration settings.
- 11. Click on **Debug** to connect to the target.
- 12. Debugging requires the DS-5 Debug perspective. If the Confirm Perspective Switch dialog box opens, click **Yes** to switch perspective.

When connected and the DS-5 Debug perspective opens you are presented with all the relevant views and editors.

For more information on these options, use the dynamic help.

Related tasks

- [2.12 Exporting an existing launch configuration on page 2-57.](#)
- [2.13 Importing an existing launch configuration on page 2-60.](#)
- [2.2 Configuring a connection to a Fixed Virtual Platform \(FVP\) on page 2-33.](#)
- [2.4 Configuring a connection to a Linux Kernel on page 2-37.](#)
- [2.7 Configuring a connection to a bare-metal target on page 2-47.](#)
- [2.8 Configuring an Event Viewer connection to a bare-metal target on page 2-49.](#)

Related references

- [10.36 Debug Configurations - Connection tab on page 10-276.](#)
- [10.37 Debug Configurations - Files tab on page 10-279.](#)
- [10.38 Debug Configurations - Debugger tab on page 10-283.](#)
- [10.39 Debug Configurations - OS Awareness tab on page 10-286.](#)
- [10.40 Debug Configurations - Arguments tab on page 10-287.](#)
- [10.41 Debug Configurations - Environment tab on page 10-289.](#)
- [10.47 Target management terminal for serial and SSH connections on page 10-298.](#)
- [11.1 ARM Linux problems and solutions on page 11-307.](#)
- [11.3 Target connection problems and solutions on page 11-309.](#)

2.4 Configuring a connection to a Linux Kernel

Describes how to configure a connection to a Linux target, load the Linux Kernel into secure memory, and also how to add a pre-built loadable module to the target.

You can connect to running target using a debug hardware adapter.

Note

By default for this type of connection, all processor exceptions are handled by Linux on the target. You can use the Manage Signals dialog box in the Breakpoints view menu to modify the default handler settings.

Prerequisites

Before connecting you must ensure that you have the target IP address or name for the connection between the debugger and the debug hardware adapter.

Procedure

1. Select **Window > Open Perspective > DS-5 Debug** from the main menu.
2. Select **Debug Configurations...** from the **Run** menu.
3. Select **DS-5 Debugger** from the configuration tree and then click on **New** to create a new configuration.
4. In the Name field, enter a suitable name for the new configuration.
5. Click on the **Connection** tab to configure a DS-5 Debugger target connection:
 - a) Select the required platform, **Linux Kernel and/or Devices Driver Debug** project type and the required debug operation.
 - b) Configure the connection between the debugger and the debug hardware adapter.
6. Click on the **Debugger** tab to configure the debugger settings.
 - a) In the Run control panel, select **Connect only** and set up initialization scripts as required.

Note

Operating System (OS) support is automatically enabled when a Linux kernel image is loaded into the debugger from the DS-5 Debugger launch configuration. However, you can manually control this by using the **set os** command.

For example, if you want to delay the activation of OS support until after the kernel has booted and the *Memory Management Unit* (MMU) is initialized then you can configure a connection that uses a target initialization script to disable OS support. To debug the kernel, OS support must be enabled in the debugger.

-
- b) Select the **Execute debugger commands** checkbox.
 - c) In the field provided, enter the following commands:

```
add-symbol-file <path>/vmlinux S:0  
add-symbol-file <path>/modex.ko
```

Note

The path to the vmlinux must be the same as your build environment.

- d) Configure the host working directory or use the default.
- e) In the Paths panel, specify any source search directories on the host that the debugger uses when it displays source code.

7. Click on **Apply** to save the configuration settings.
8. Click on **Debug** to connect to the target.
9. Debugging requires the DS-5 Debug perspective. If the Confirm Perspective Switch dialog box opens, click **Yes** to switch perspective.

When connected and the DS-5 Debug perspective opens you are presented with all the relevant views and editors.

For more information on these options, use the dynamic help.

Related concepts

[6.9 About debugging a Linux kernel on page 6-148.](#)

[6.10 About debugging Linux kernel modules on page 6-150.](#)

Related tasks

[2.12 Exporting an existing launch configuration on page 2-57.](#)

[2.13 Importing an existing launch configuration on page 2-60.](#)

[2.2 Configuring a connection to a Fixed Virtual Platform \(FVP\) on page 2-33.](#)

[2.3 Configuring a connection to a Linux target using gdbserver on page 2-35.](#)

[2.7 Configuring a connection to a bare-metal target on page 2-47.](#)

[2.8 Configuring an Event Viewer connection to a bare-metal target on page 2-49.](#)

Related references

[10.36 Debug Configurations - Connection tab on page 10-276.](#)

[10.37 Debug Configurations - Files tab on page 10-279.](#)

[10.38 Debug Configurations - Debugger tab on page 10-283.](#)

[10.39 Debug Configurations - OS Awareness tab on page 10-286.](#)

[10.40 Debug Configurations - Arguments tab on page 10-287.](#)

[10.41 Debug Configurations - Environment tab on page 10-289.](#)

[10.47 Target management terminal for serial and SSH connections on page 10-298.](#)

[11.1 ARM Linux problems and solutions on page 11-307.](#)

[11.3 Target connection problems and solutions on page 11-309.](#)

Related information

[Debugging a loadable kernel module.](#)

2.5 About configuring connections to a Linux target using Application Debug with Rewind Support

Use the options available under **Application Debug with Rewind Support** in the Debug Configurations dialog to connect to Linux targets.

Note

- Application rewind does not follow forked processes.
- When debugging backwards, you can only view the contents of recorded memory, registers, or variables. You cannot edit or change them.
- Application rewind supports architecture ARMv5TE targets and later, except for the 64-bit ARMv8 architecture.

The options are:

- **Connect to already running application.** This option requires you to load your application and the application rewind server on your target and start the application rewind server manually before attempting a connection between DS-5 and your target.
- **Start undodb-server and debug target-resident application.** This option requires you to load your application and the application rewind server on your target manually. When a connection is established, DS-5 starts a new application rewind server session on your target to debug your application.
- **Download and debug application.** When a connection is established using this option, DS-5 downloads your application and the application rewind server on to the target system, and starts a new application rewind server session to debug your application.

Note

The application rewind feature in DS-5 Debugger is license managed. Contact your support representative for details about this feature.

It contains the following:

- [2.5.1 Connecting to an existing application and application rewind session on page 2-39.](#)
- [2.5.2 Downloading your application and application rewind server on the target system on page 2-41.](#)
- [2.5.3 Starting the application rewind server and debugging the target-resident application on page 2-42.](#)

2.5.1 Connecting to an existing application and application rewind session

Use the **Connect to already running application** option to set up a connection to an existing application and application rewind server session on your target.

Prerequisites

Before connecting to an existing application rewind server session, you must ensure that:

- The *undodb-server* file found in the `install_directory\DS-5\arm\undodb\linux` folder is copied to your target.
- The application that you want to debug is copied to the target.
- The application rewind server session is running and connected to your application.

Note

To run the application rewind server and the application on the target, use:

`undodb-server --connect-port port path/myApplication`

Where:

port is a TCP/IP port number of your choice that is used by application rewind server to communicate with DS-5 Debugger.

path/myApplication is the application that you want to debug.

Procedure

1. From the main menu, select **Window > Open Perspective > Other > DS-5 Debug** to switch to the DS-5 debug perspective.
2. From the **Run** menu, select **Debug Configurations....**
3. Select **DS-5 Debugger** from the configuration tree and then click **New launch configuration** to create a new configuration.
4. In the Name field, enter a suitable name for the new configuration.
5. Select the **Connection** tab to configure the target connection:
 - a) In the Select target panel, select **Linux Application Debug > Application Debug with Rewind Support > Connections via undodb-server > Connect to already running application**.
 - b) Enter the Address of the connection you want to connect to.
 - c) Enter the **UndoDB-server (TCP) Port** that you want to connect to.
6. Select the **Files** tab and in the Files panel, select the files on the host that you want the debugger to use to load the debug information from. If required, you can also specify other files on the host that you want to download to the target.
7. Select the **Debugger** tab to configure the debugger settings.
 - a) In the Run control panel, specify the actions that you want the debugger to perform after connecting to the target.
 - b) In the Host working directory panel, configure the host working directory or use the default.
 - c) In the Paths panel, specify any source or library search directories on the host that the debugger uses when it displays source code.
8. Click **Apply** to save the configuration settings.
9. Click **Debug** to connect to the target.

When connected, and the DS-5 Debug perspective opens, you are presented with all the relevant views and editors.

For more information on these options, use the dynamic help.

Related concepts

[6.14 About application rewind on page 6-156.](#)

Related references

[10.36 Debug Configurations - Connection tab on page 10-276.](#)
[10.37 Debug Configurations - Files tab on page 10-279.](#)
[10.38 Debug Configurations - Debugger tab on page 10-283.](#)
[10.39 Debug Configurations - OS Awareness tab on page 10-286.](#)
[10.40 Debug Configurations - Arguments tab on page 10-287.](#)
[10.41 Debug Configurations - Environment tab on page 10-289.](#)
[2.1 Types of target connections on page 2-32.](#)

2.5.2 Downloading your application and application rewind server on the target system

Use the **Download and debug application** option to download your application and application rewind server to the target system and start a new application rewind session.

Prerequisites

Before connecting, you must:

- Set up the target with an *Operating System* (OS) installed and booted. See the documentation supplied with the target for more information.
- Obtain the IP address or name of the target.
- Set up a *Remote Systems Explorer* (RSE) connection to the target.

Procedure

1. From the main menu, select **Window > Open Perspective > Other > DS-5 Debug** to switch to the DS-5 debug perspective.
2. From the **Run** menu, select **Debug Configurations...**
3. Select **DS-5 Debugger** from the configuration tree and then click **New launch configuration** to create a new configuration.
4. In the Name field, enter a suitable name for the new configuration.
5. Select the **Connection** tab to configure the target connection:
 - a) In the Select target panel, select **Linux Application Debug > Application Debug with Rewind Support > Connections via undodb-server > Download and debug application**.
 - b) Select your **RSE connection** from the list.
 - c) Accept the default values for the **UndoDB-server (TCP) Port**.
6. Select the **Files** tab to define the application file and libraries.
 - a) In the Target Configuration panel, select the application on the host that you want to download to the target and specify the location on the target where you want to download the selected file.
 - b) In the Files panel, select the files on the host that you want the debugger to use to load the debug information. If required, you can also specify other files on the host that you want to download to the target.

———— Note ————

Options in the **Files** tab are dependent on the type of debug operation that you select.

7. Select the **Debugger** tab to configure the debugger settings.
 - a) In the Run control panel, specify the actions that you want the debugger to perform after connecting to the target.
 - b) In the Host working directory panel, configure the host working directory or use the default.
 - c) In the Paths panel, specify any source or library search directories on the host that the debugger uses when it displays source code.
8. If required, use the **Arguments** tab to enter arguments that are passed to the application when the debug session starts.
9. If required, use the **Environment** tab to create and configure the target environment variables that are passed to the application when the debug session starts.
10. Click **Apply** to save the configuration settings.
11. Click **Debug** to connect to the target.

When connected, and the DS-5 Debug perspective opens, you are presented with all the relevant views and editors.

For more information on these options, use the dynamic help.

Related concepts

[6.14 About application rewind on page 6-156.](#)

Related references

[10.36 Debug Configurations - Connection tab on page 10-276.](#)

[10.37 Debug Configurations - Files tab on page 10-279.](#)

[10.38 Debug Configurations - Debugger tab on page 10-283.](#)

[10.39 Debug Configurations - OS Awareness tab on page 10-286.](#)

[10.40 Debug Configurations - Arguments tab on page 10-287.](#)

[10.41 Debug Configurations - Environment tab on page 10-289.](#)

[2.1 Types of target connections on page 2-32.](#)

2.5.3 Starting the application rewind server and debugging the target-resident application

Use the **Start undodb-server and debug target-resident application** option to start the application rewind server on the target system and debug an existing application.

Prerequisites

Before connecting, you must:

- Set up the target with an *Operating System* (OS) installed and booted. See the documentation supplied with the target for more information.
- Obtain the IP address or name of the target.
- Set up a *Remote Systems Explorer* (RSE) connection to the target.
- Ensure that the application rewind server is available on your target and is added to your PATH environment variable.
- Ensure that the application you want to debug is available on the target.

Procedure

1. From the main menu, select **Window > Open Perspective > Other > DS-5 Debug** to switch to the DS-5 debug perspective.
2. From the **Run** menu, select **Debug Configurations...**
3. Select **DS-5 Debugger** from the configuration tree and then click **New launch configuration** to create a new configuration.
4. In the Name field, enter a suitable name for the new configuration.
5. Select the **Connection** tab to configure the target connection:
 - a) In the Select target panel, select **Linux Application Debug > Application Debug with Rewind Support > Connections via undodb-server > Start undodb-server and debug target-resident application**.
 - b) Select your **RSE connection** from the list.
 - c) Accept the default values for the **UndoDB-server (TCP) Port**.
6. Select the **Files** tab to define the location of the Application on target, Target working directory, and additional Files.
 - a) In the Target Configuration panel, enter the location of the Application on target and the Target working directory.
 - b) In the Files panel, enter or select the location of the files on the target that you want the debugger to use to load additional debug information. If required, you can also specify other files on the host that you want to download to the target.

Note

Options in the **Files** tab are dependent on the type of debug operation that you select.

7. Select the **Debugger** tab to configure the debugger settings.
 - a) In the Run control panel, specify the actions that you want the debugger to perform after connecting to the target.
 - b) In the Host working directory panel, configure the host working directory or use the default.
 - c) In the Paths panel, specify any source or library search directories on the host that the debugger uses when it displays source code.
8. If required, use the **Arguments** tab to enter arguments that are passed to the application when the debug session starts.
9. If required, use the **Environment** tab to create and configure the target environment variables that are passed to the application when the debug session starts.
10. Click **Apply** to save the configuration settings.
11. Click **Debug** to connect to the target.

Related concepts

[6.14 About application rewind on page 6-156.](#)

Related references

[10.36 Debug Configurations - Connection tab on page 10-276.](#)
[10.37 Debug Configurations - Files tab on page 10-279.](#)
[10.38 Debug Configurations - Debugger tab on page 10-283.](#)
[10.39 Debug Configurations - OS Awareness tab on page 10-286.](#)
[10.40 Debug Configurations - Arguments tab on page 10-287.](#)
[10.41 Debug Configurations - Environment tab on page 10-289.](#)
[2.1 Types of target connections on page 2-32.](#)

2.6 About configuring connections to an Android target using Native Application/Library Debug with Rewind Support

Use the options available under **Native Application/Library Debug with Rewind Support** in the Debug Configurations dialog to connect to Android targets.

Note

- Application rewind does not follow forked processes.
- When debugging backwards, you can only view the contents of recorded memory, registers, or variables. You cannot edit or change them.
- Application rewind supports architecture ARMv5TE targets and later, except for the 64-bit ARMv8 architecture.

The options are:

- **Attach to a running Android application.** This option requires you to load your application on your Android target manually before attempting to attach the DS-5 debug session. Once attached, DS-5 starts a new application rewind server session to debug your application.
- **Download and debug an Android application.** When a connection is established using this option, DS-5 downloads your application and the application rewind server on to the Android target, and starts a new application rewind server session to debug your application.

It contains the following:

- [2.6.1 Attaching to a running Android application on page 2-44.](#)
- [2.6.2 Downloading and debugging an Android application on page 2-45.](#)

2.6.1 Attaching to a running Android application

Use the **Attach to a running Android application** option to set up a connection to an existing application and application rewind server session on your target.

Prerequisites

Before connecting, you must:

- Ensure that the ADB application bundle is available under the PATH environment variable on your workstation.
- Obtain root access on the Android device.
- Ensure that the Android target is booted up and running.
- Ensure that your application is installed and running on the Android target.

Procedure

1. From the main menu, select **Window > Open Perspective > Other > DS-5 Debug** to switch to the DS-5 debug perspective.
2. From the **Run** menu, select **Debug Configurations....**
3. Select **DS-5 Debugger** from the configuration tree and then click **New launch configuration** to create a new configuration.
4. In the Name field, enter a suitable name for the new configuration.
5. Select the **Connection** tab to configure the target connection:
 - a) In the Select target panel, select **Android Application Debug > Native Application/Library Debug with Rewind Support > APK Native Library Debug via undodb-server > Attach to a running Android application.**
 - b) Select your device from the Connections panel.

- c) Accept the default values for the **UndoDB-server (TCP)** Port.
6. Select the **Files** tab to define the application file and libraries.
 - a) In the Android panel, select the **Project directory** and **APK file** that you want to use. The **Process** and **Activity** fields are populated by the `AndroidManifest.xml` file.
 - b) In the Files panel, select the files on the host that you want the debugger to use to load the debug information.

Note

Options in the **Files** tab are dependent on the type of debug operation that you select.

7. Select the **Debugger** tab to configure the debugger settings.
 - a) In the Run control panel, select **Connect only**.
 - b) In the Host working directory panel, configure the host working directory or use the default.
 - c) In the Paths panel, specify any source or library search directories on the host that the debugger uses when it displays source code.
8. Click **Apply** to save the configuration settings.
9. Click **Debug** to connect to the target.

Related concepts

[6.14 About application rewind on page 6-156.](#)

Related references

[2.1 Types of target connections on page 2-32.](#)

2.6.2 Downloading and debugging an Android application

Use the **Download and debug an Android application** option to download and install your application on the Android target and load and start an application rewind debug session.

Prerequisites

Before connecting, you must:

- Ensure that the ADB application bundle is available under the PATH environment variable on your workstation.
- Obtain root access on the Android device.
- Ensure that the Android target is booted up and running.

Procedure

1. From the main menu, select **Window > Open Perspective > Other > DS-5 Debug** to switch to the DS-5 debug perspective.
2. From the **Run** menu, select **Debug Configurations...**
3. Select **DS-5 Debugger** from the configuration tree and then click **New launch configuration** to create a new configuration.
4. In the Name field, enter a suitable name for the new configuration.
5. Select the **Connection** tab to configure the target connection:
 - a) In the Select target panel, select **Android Application Debug > Native Application/Library Debug with Rewind Support > APK Native Library Debug via undodb-server > Download and debug an Android application**.
 - b) Select your device from the Connections panel.
 - c) Accept the default values for the **UndoDB-server (TCP)** Port.
6. Select the **Files** tab to define the application file and libraries.

- a) In the Android panel, select the **Project directory** and **APK file** that you want to use. The **Process** and **Activity** fields are populated by the `AndroidManifest.xml` file. If needed, select a different **Activity**.
- b) In the Files panel, select the files on the host that you want the debugger to use to load the debug information.

Note

Options in the **Files** tab are dependent on the type of debug operation that you select.

7. Select the **Debugger** tab to configure the debugger settings.
 - a) In the Run control panel, select **Connect only**.
 - b) In the Host working directory panel, configure the host working directory or use the default.
 - c) In the Paths panel, specify any source or library search directories on the host that the debugger uses when it displays source code.
8. Click **Apply** to save the configuration settings.
9. Click **Debug** to connect to the target.

Related concepts

[6.14 About application rewind on page 6-156.](#)

Related references

[2.1 Types of target connections on page 2-32.](#)

2.7 Configuring a connection to a bare-metal target

Describes how to download and connect to an application running on a target using a debug hardware adapter.

Prerequisites

Before connecting you must ensure that you have the target IP address or name for the connection between the debugger and the debug hardware adapter.

Procedure

1. Select **Window > Open Perspective > DS-5 Debug** from the main menu.
2. Select **Debug Configurations...** from the **Run** menu.
3. Select **DS-5 Debugger** from the configuration tree and then click on **New** to create a new configuration.
4. In the Name field, enter a suitable name for the new configuration.
5. Click on the **Connection** tab to configure a DS-5 Debugger target connection:
 - a) Select the required platform. For example, **ARM-Versatile Express A9x4, Bare Metal Debug, Debug and Trace Cortex-A9x4 SMP via DSTREAM**.
 - b) Configure the connection between the debugger and the debug hardware adapter.
6. Click on the **Files** tab to define the target environment and select debug versions of the application file and libraries on the host that you want the debugger to use.
 - a) In the Target Configuration panel, select the application on the host that you want to download to the target.
7. Click on the **Debugger** tab to configure the debugger settings.
 - a) In the Run control panel, specify the actions that you want the debugger to do after connection to the target.
 - b) Configure the host working directory or use the default.
 - c) In the Paths panel, specify any source search directories on the host that the debugger uses when it displays source code.
8. If required, click on the **Arguments** tab to enter arguments that are passed, using semihosting, to the application when the debug session starts.
9. Click on **Apply** to save the configuration settings.
10. Click on **Debug** to connect to the target.
11. Debugging requires the DS-5 Debug perspective. If the Confirm Perspective Switch dialog box opens, click **Yes** to switch perspective.

When connected and the DS-5 Debug perspective opens you are presented with all the relevant views and editors.

For more information on these options, use the dynamic help.

Related tasks

- [2.12 Exporting an existing launch configuration on page 2-57.](#)
- [2.13 Importing an existing launch configuration on page 2-60.](#)
- [2.2 Configuring a connection to a Fixed Virtual Platform \(FVP\) on page 2-33.](#)
- [2.3 Configuring a connection to a Linux target using gdbserver on page 2-35.](#)
- [2.4 Configuring a connection to a Linux Kernel on page 2-37.](#)
- [2.8 Configuring an Event Viewer connection to a bare-metal target on page 2-49.](#)

Related references

- 10.36 Debug Configurations - Connection tab on page 10-276.*
- 10.37 Debug Configurations - Files tab on page 10-279.*
- 10.38 Debug Configurations - Debugger tab on page 10-283.*
- 10.39 Debug Configurations - OS Awareness tab on page 10-286.*
- 10.40 Debug Configurations - Arguments tab on page 10-287.*
- 10.41 Debug Configurations - Environment tab on page 10-289.*

2.8 Configuring an Event Viewer connection to a bare-metal target

Describes how to connect to a bare-metal target.

The Event Viewer allows you to capture and view textual logging information from bare-metal applications. Logging is captured from your application using annotations that you must add to the source code.

Note

The **Event Viewer** tab in the Debug Configurations dialog box is only enabled for targets where *System Trace Macrocell* (STM) and *Instrumentation Trace Macrocell* (ITM) capture is supported.

Prerequisites

Before connecting you must ensure that you:

- Have the target IP address or name for the connection between the debugger and the debug hardware agent.
- Annotate your application source code with logging points and recompile it. See the ITM and Event Viewer Example for Versatile Express A9x4 provided with DS-5 for more information.

Procedure

1. Select **Window > Open Perspective > DS-5 Debug** from the main menu.
2. Select **Debug Configurations...** from the **Run** menu.
3. Select **DS-5 Debugger** from the configuration tree and then click on **New** to create a new configuration.
4. In the Name field, enter a suitable name for the new configuration.
5. Click on the **Connection** tab to configure a DS-5 Debugger target connection:
 - a) Select the required platform. For example, **ARM-Versatile Express A9x4, Bare Metal Debug, Debug and Trace Cortex-A9x4 SMP via DSTREAM**.
 - b) Configure the connection between the debugger and the debug hardware agent.
6. Click on the **Files** tab to define the target environment and select debug versions of the application file and libraries on the host that you want the debugger to use.
 - a) In the Target Configuration panel, select the application on the host that you want to download to the target.
7. Click on the **Debugger** tab to configure the debugger settings.
 - a) In the Run control panel, specify the actions that you want the debugger to do after connection to the target.
 - b) Configure the host working directory or use the default.
 - c) In the Paths panel, specify any source search directories on the host that the debugger uses when it displays source code.
8. If required, click on the **Arguments** tab to enter arguments that are passed, using semihosting, to the application when the debug session starts.
9. Click on the **Event Viewer** tab to configure the ITM capture settings.
 - a) Select **Enable Event Viewer for ITM events**.
 - b) Enter the maximum size of the trace buffer. For example, you can enter 100MB for a DSTREAM connection. Be aware that larger buffers have a performance impact by taking longer to process but collect more trace data.
10. Click on **Apply** to save the configuration settings.
11. Click on **Debug** to connect to the target.

12. Debugging requires the DS-5 Debug perspective. If the Confirm Perspective Switch dialog box opens, click **Yes** to switch perspective.

When connected and the DS-5 Debug perspective opens you are presented with all the relevant Channel editors for the Event Viewer.

For more information on these options, use the dynamic help.

Related tasks

- 2.12 Exporting an existing launch configuration on page 2-57.*
- 2.13 Importing an existing launch configuration on page 2-60.*
- 2.2 Configuring a connection to a Fixed Virtual Platform (FVP) on page 2-33.*
- 2.3 Configuring a connection to a Linux target using gdbserver on page 2-35.*
- 2.4 Configuring a connection to a Linux Kernel on page 2-37.*
- 2.7 Configuring a connection to a bare-metal target on page 2-47.*

Related references

- 10.36 Debug Configurations - Connection tab on page 10-276.*
- 10.37 Debug Configurations - Files tab on page 10-279.*
- 10.38 Debug Configurations - Debugger tab on page 10-283.*
- 10.39 Debug Configurations - OS Awareness tab on page 10-286.*
- 10.40 Debug Configurations - Arguments tab on page 10-287.*
- 10.41 Debug Configurations - Environment tab on page 10-289.*

2.9 About the target configuration import utility

The import utility, **cdbimporter**, aims to provide an easy method to import platform information into DS-5, and so provide limited debug and trace support for the platform through RVI, DSTREAM, VSTREAM, or model connections.

A database holds the target configuration and connection settings in DS-5. The import utility creates platform entries in a new configuration database using information from:

- A configuration file created and saved using the Debug Hardware Configuration utility, **dbghwconfig** or **rviconfig**.

———— **Note** —————

DS-5 is not yet capable of creating a configuration file from within Eclipse.

- A model that provides a CADI server. The model can be already running or you can specify the path and filename to the executable file in the command-line options.

The import utility creates the following debug operations:

- Single processor and *Symmetric MultiProcessing* (SMP) bare-metal debug for hardware and models.
- Single processor and SMP Linux kernel debug for hardware.
- Linux application debug configurations for hardware.

For hardware targets where a trace subsystem is present, appropriate *Debug and Trace Services Layer* (DTSL) options are produced. These can include:

- Selection of on-chip (*Embedded Trace Buffer* (ETB), *Micro Trace Buffer* (MTB), *Trace Memory Controller* (TMC) or other on-chip buffer) or off-chip (DSTREAM trace buffer) trace capture
- Cycle-accurate trace capture
- Trace capture range
- Configuration and capture of *Instruction Trace Macrocell* (ITM) trace to be handled by the DS-5 Event Viewer.

The import utility does not create:

- debug operations that configure non-instruction trace macrocells other than ITM
- big.LITTLE™ configurations.

For SMP configurations, the *Cross Trigger Interface* (CTI) synchronization is used on targets where a suitable CTI is present. Using a CTI produces a much tighter synchronization with a very low latency in the order of cycles but the CTI must be fully implemented and connected in line with the ARM reference designs, and must not be used for any other purpose. Synchronization without using a CTI has a much higher latency, but makes no assumptions about implementation or usage.

You might have to manually configure off-chip TPIU trace for multiplexed pins and also perform calibrations to cope with signal timing issues.

If you experience any problems or need to produce other configurations, contact your support representative.

Assumptions

The import utility makes the following assumptions when creating debug operations:

- There is a linear mapping between trace macrocells and CoreSight™ trace funnel ports.

- The *Embedded Trace Macrocell* (ETM)/*Program Trace Macrocell* (PTM) versions are fixed for each type of processor.

Table 2-1 ETM/PTM versions for each type of processor

Processor Type	ETM/PTM
Cortex-A15	PTM
Cortex-A7	ETM v3.5
Cortex-A5	ETM v3.5
Cortex-A8	ETM v3.3
Cortex-A9	PTM
Cortex-R4	ETM v3.3
Cortex-R5	ETM v3.3
Cortex-R7	ETM v4
Cortex-M3	ETM v3.4
Cortex-M4	ETM v3.4
ARM9 series	ETM v1.x is not supported.
ARM11 series	ETM v3.1

- The CTI devices are not used for other operations.
- In a target containing multiple CoreSight ETBs, TPIUs or trace funnels, the import utility produces configuration for the first example of each trace funnel, ETB, and TPIU with the lowest base address.

Limitations

It is only possible to import platforms that can be auto-configured using the Debug Hardware Configuration utility or from a model.

To see a list of the processors supported by DS-5 you can run the import utility with the `--list-cores` option (-l).

The import utility produces a basic configuration with appropriate processor and CP15 register sets.

Related tasks

[2.10 Adding a new platform on page 2-53.](#)

[2.11 Adding a new configuration database to DS-5 on page 2-55.](#)

Related references

[10.43 Configuration database panel on page 10-293.](#)

2.10 Adding a new platform

Describes how to create a new configuration database containing a new platform for use with DS-5.

Procedure

1. Launch a command-line console:
 - On Windows, select **Start > All Programs > ARM DS-5 > DS-5 Command Prompt**.
 - On Linux:
 1. Add the *install_directory/bin* directory to your PATH environment variable. If it is already configured then you can skip this step.
 2. Open a Unix bash shell.
2. Launch the import utility using the following command-line syntax:

```
cdbimporter --help
cdbimporter [--cdb=cdbpath] --list-cores
cdbimporter [--cdb=cdbpath] [--target-cdb=targetpath] {file.rvc | --
model[=modelpath]} [option]...
```

where:

--help

Displays a summary of the main command-line options.

--list-cores

lists all the processors defined by the database supplied in the **--cdb** option.

--cdb=cdbpath

Specifies a path to the source configuration database (as shipped in DS-5) containing processor and register definitions to identify the target.

--target-cdb=targetpath

Directory where the destination database is to reside. ARM recommends that you build separate configuration databases in your own workspace to avoid accidental loss of data when updating DS-5. You can specify multiple configuration databases in DS-5 using the **Preferences** dialog. This enables platforms in the new database to use existing processor and register definitions.

file.rvc

Imports from a configuration file (.rv). You can use the Debug Hardware Configuration utility, **dbghwconfig** or **rviconfig** to connect to the target and save the information in a file. The resultant file contains limited debug and trace support for the platform that can be used to populate the DS-5 configuration database.

--model=modelpath

Imports from a model that provides a CADI server.

- If you supply the *modelpath* to the model executable, the utility launches the model for interrogation so that it can determine the connection settings that DS-5 uses to automatically launch the model on connection.
- If you do not supply the *modelpath* to the model executable, you can force the utility to search for a running model to interrogate. You can then manually enter the data for the connection to the model. For example, processors names, IDs, and processor definitions. If you use this option then you must launch the model manually before connecting DS-5 to it.

option

Where *option* can be any of the following:

--no-ctis

Disables the use of *Cross Trigger Interface* (CTI) synchronization in the resulting platform.

--no-trace

Disables the use of trace components in the resulting platform.

--use-defaults

Displays default input when the database requires a user input. This does not apply to the output database path.

--toolkit=key

Specifies a comma separated list of toolkits.

3. During the import process, the import utility enables you to modify details about the processors of the new platform. Follow the instructions in the command-line prompts.

On successful completion a new configuration database is created containing the new platform that can be added to DS-5.

Related concepts

[2.9 About the target configuration import utility on page 2-51.](#)

Related tasks

[2.11 Adding a new configuration database to DS-5 on page 2-55.](#)

Related references

[10.43 Configuration database panel on page 10-293.](#)

2.11 Adding a new configuration database to DS-5

Describes how to add a new configuration database to DS-5.

Procedure

1. Launch Eclipse.
2. Select **Preferences** from **Windows** menu.
3. Expand the **DS-5** configuration group.
4. Select **Configuration Database**.
5. Click **Add...** to locate the new database:
 - a) Select the entire directory.
 - b) Click **OK** to close the dialog box.
6. Position the new database:
 - a) Select the new database.
 - b) Click **Up** or **Down** as required.

Note

DS-5 provides built-in databases containing a default set of target configurations. You can enable or disable these but not delete them.

7. Click **Rebuild database...**
8. Click **OK** to close the dialog box and save the settings.

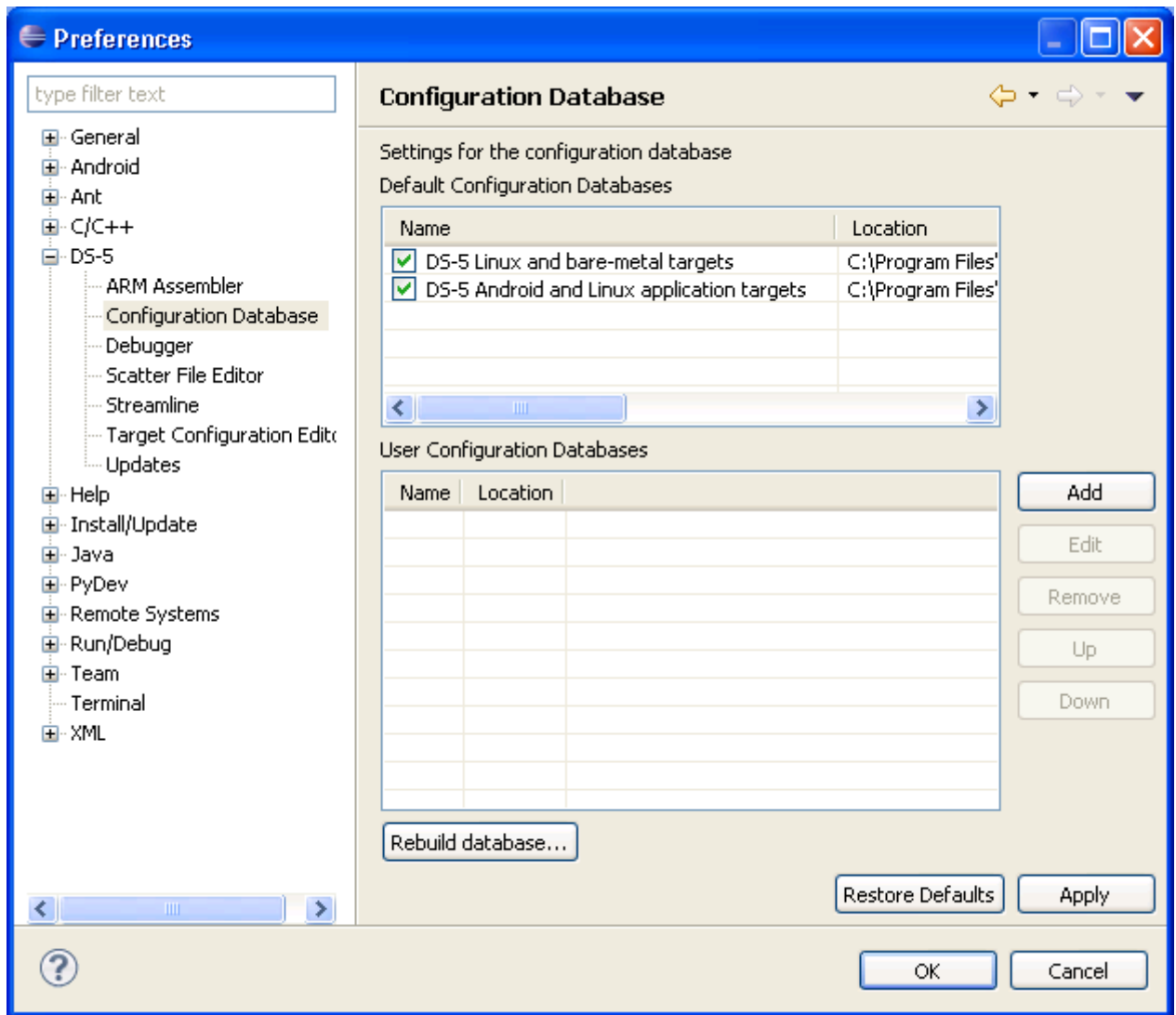


Figure 2-1 Adding a new configuration database

Note

DS-5 processes the databases from top to bottom with the information in the lower databases replacing information in the higher databases. For example, if you want to produce a modified Cortex-A15 processor definition with different registers then those changes can be added to a new database that resides lower down in the list.

Related concepts

[2.9 About the target configuration import utility on page 2-51.](#)

Related tasks

[2.10 Adding a new platform on page 2-53.](#)

Related references

[10.43 Configuration database panel on page 10-293.](#)

2.12 Exporting an existing launch configuration

Describes how to export an existing launch configuration.

Procedure

1. Select **Export...** from the **File** menu.
2. In the Export dialog box, expand the **Run/Debug** group and select **Launch Configurations**.

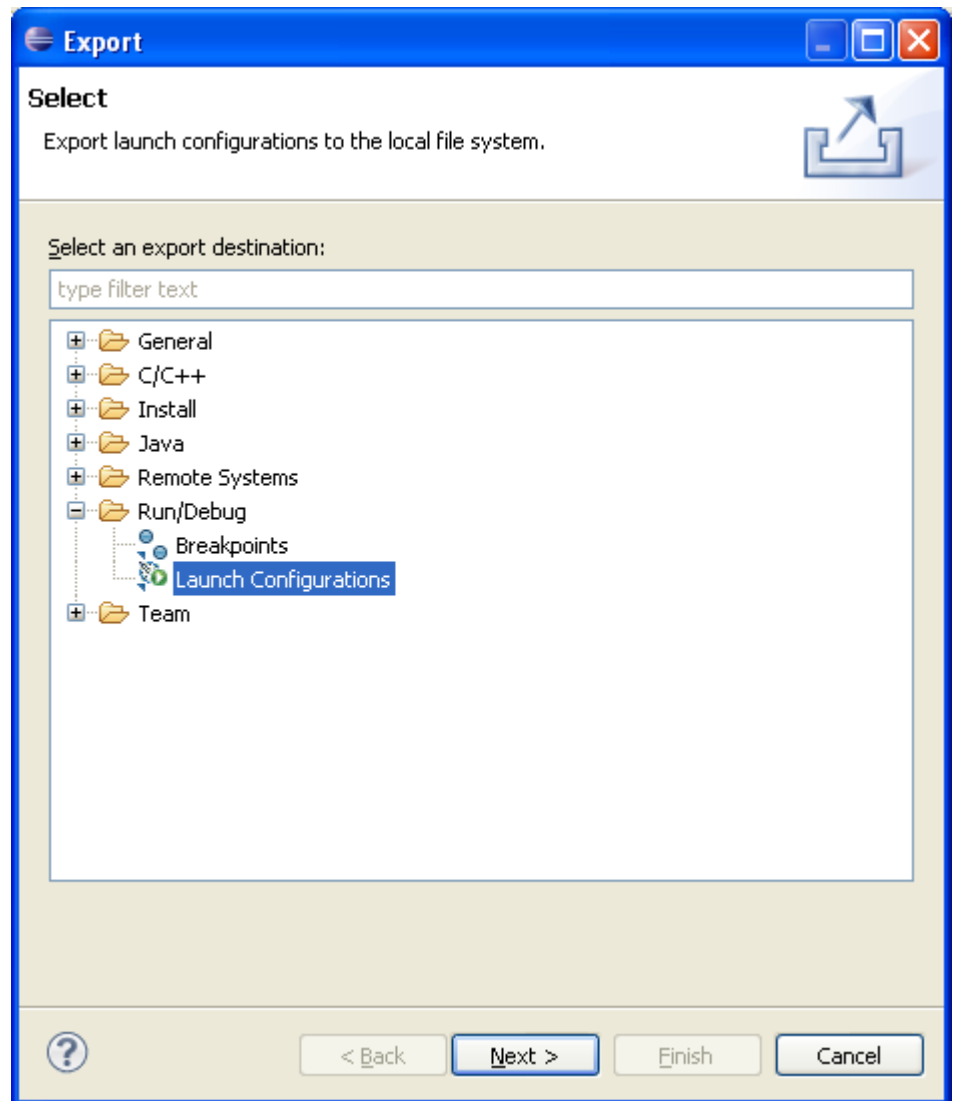


Figure 2-2 Export launch configuration dialog box

3. Click on **Next**.
4. In the Export Launch Configurations dialog box:
 - a) Expand the **DS-5 Debugger** group and then select one or more launch configurations.
 - b) Click on **Browse...** to select the required location in the local file system.
 - c) Select the folder and then click **OK**.

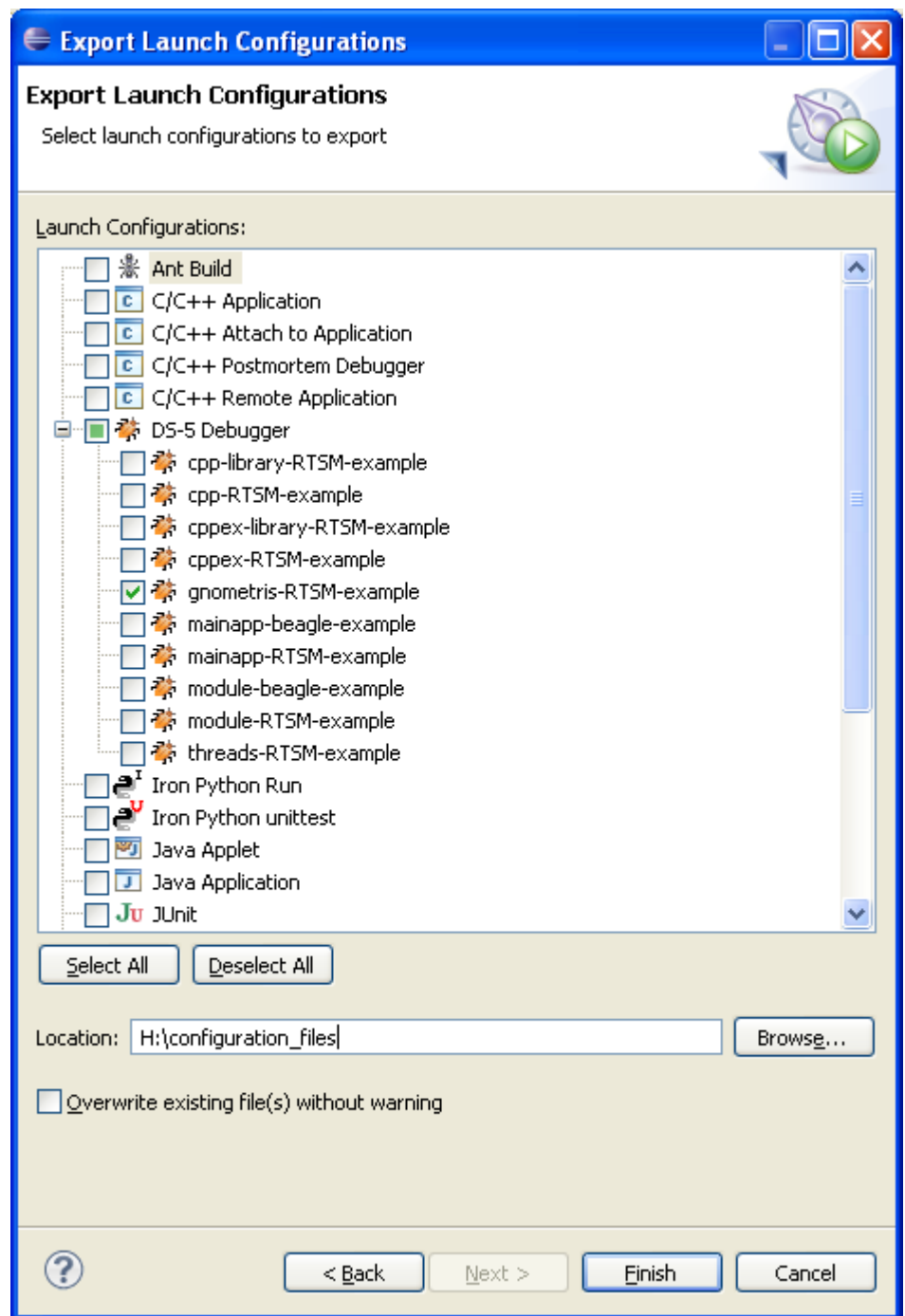


Figure 2-3 Launch configuration selection panels

5. If required, select **Overwrite existing file(s) without warning**.
6. Click on **Finish**.

Related tasks

- [2.13 Importing an existing launch configuration on page 2-60.](#)
- [2.2 Configuring a connection to a Fixed Virtual Platform \(FVP\) on page 2-33.](#)
- [2.3 Configuring a connection to a Linux target using gdbserver on page 2-35.](#)
- [2.4 Configuring a connection to a Linux Kernel on page 2-37.](#)

2.7 Configuring a connection to a bare-metal target on page 2-47.

2.8 Configuring an Event Viewer connection to a bare-metal target on page 2-49.

2.13 Importing an existing launch configuration

Describes how to import an existing launch configuration into Eclipse.

Procedure

1. Select **Import...** from the **File** menu.
2. In the Import dialog box, expand the **Run/Debug** group and select **Launch Configurations**.

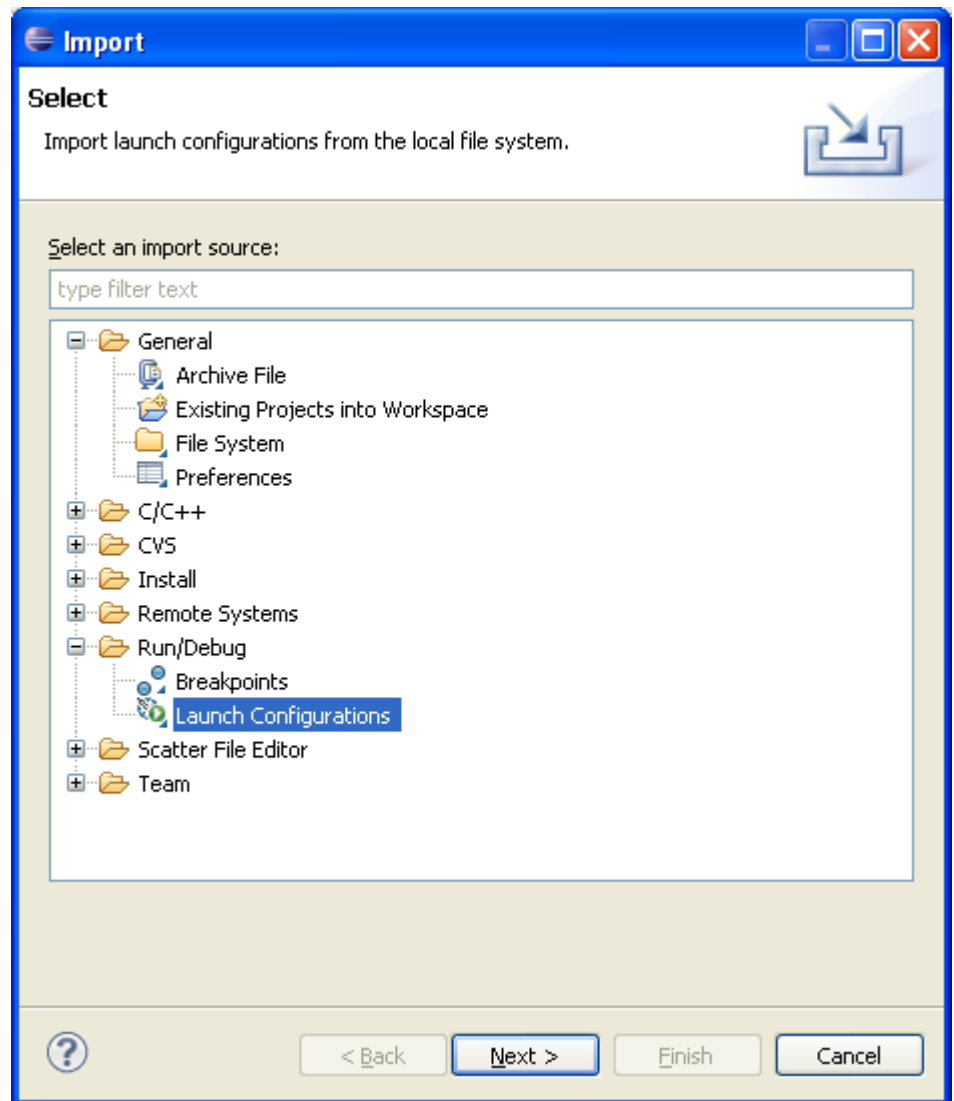


Figure 2-4 Import launch configuration dialog box

3. Click on **Next**.
4. Click on **Browse...** to select the required location in the local file system.
5. Select the folder containing the launch files and then click **OK**.
6. Select the checkboxes for the required folder and launch file(s).

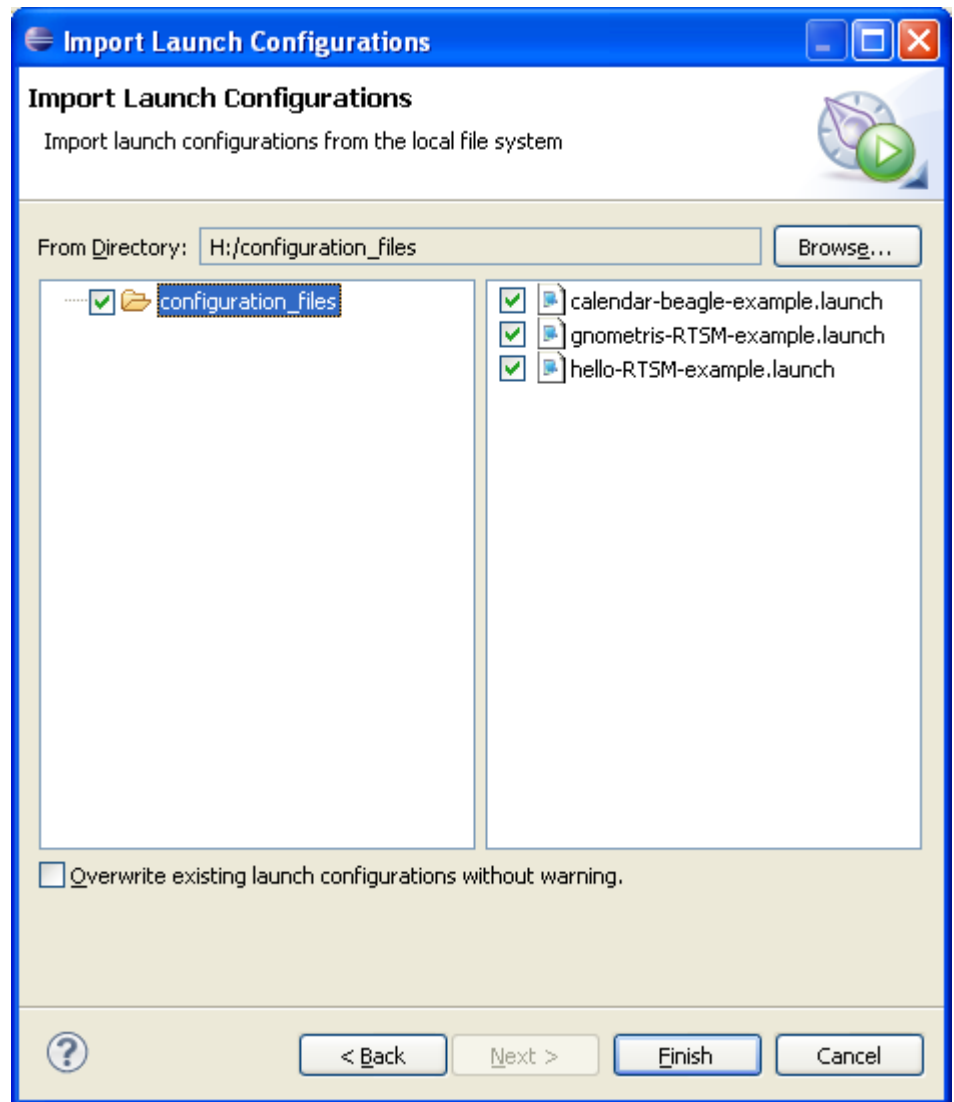


Figure 2-5 Launch configuration file selection panels

7. If you are replacing an existing configuration with the same name then select **Overwrite existing launch configurations without warning.**
8. Click on **Finish.**

Related tasks

- [2.12 Exporting an existing launch configuration on page 2-57.](#)
- [2.2 Configuring a connection to a Fixed Virtual Platform \(FVP\) on page 2-33.](#)
- [2.3 Configuring a connection to a Linux target using gdbserver on page 2-35.](#)
- [2.4 Configuring a connection to a Linux Kernel on page 2-37.](#)
- [2.7 Configuring a connection to a bare-metal target on page 2-47.](#)
- [2.8 Configuring an Event Viewer connection to a bare-metal target on page 2-49.](#)

2.14 Disconnecting from a target

Describes how to disconnect from a target using the DS-5 Debug perspective.

Procedure

You can use either the **Debug Control** or **Commands** view as follows:

- Click on the **Disconnect from Target** toolbar icon in the **Debug Control** view.
- Alternatively, in the **Commands** view you can:
 1. Enter `quit` in the Command field.
 2. Click **Submit**.

Related references

[10.6 Commands view on page 10-209.](#)

[10.7 Debug Control view on page 10-212.](#)

[10.51 DS-5 Debugger menu and toolbar icons on page 10-303.](#)

Related information

[DS-5 Debugger commands.](#)

Chapter 3

Working with the target configuration editor

Describes how to use the editor when developing a project for an ARM target. It contains the following:

- *3.1 About the target configuration editor on page 3-64.*
- *3.2 Target configuration editor - Overview tab on page 3-65.*
- *3.3 Target configuration editor - Memory tab on page 3-67.*
- *3.4 Target configuration editor - Peripherals tab on page 3-69.*
- *3.5 Target configuration editor - Registers tab on page 3-71.*
- *3.6 Target configuration editor - Group View tab on page 3-73.*
- *3.7 Target configuration editor - Enumerations tab on page 3-76.*
- *3.8 Target configuration editor - Configurations tab on page 3-78.*
- *3.9 Scenario demonstrating how to create a new target configuration file on page 3-80.*
- *3.10 Creating a power domain for a target on page 3-92.*
- *3.11 Creating a Group list on page 3-94.*
- *3.12 Importing an existing target configuration file on page 3-96.*
- *3.13 Exporting a target configuration file on page 3-98.*

3.1 About the target configuration editor

The target configuration editor provides forms and graphical views to easily create and edit *Target Configuration Files* (TCF) describing memory mapped peripheral registers present on a device. It also provides import and export wizards for compatibility with the file formats used in μ Vision System Viewer.

TCF files must have the file extension `.tcf` to invoke this editor.

If this is not the default editor, right-click on your source file in the **Project Explorer** view and select **Open With > Target Configuration Editor** from the context menu.

The target configuration editor also provides a hierarchical tree using the **Outline** view. Click on an entry in the **Outline** view to move the focus of the editor to the relevant tab and selected field. If this view is not visible, select **Window > Show View > Outline** from the main menu.

Directories containing TCF files can be specified in DS-5 Debugger launch configurations.

Related references

- [3.2 Target configuration editor - Overview tab on page 3-65.](#)
- [3.3 Target configuration editor - Memory tab on page 3-67.](#)
- [3.4 Target configuration editor - Peripherals tab on page 3-69.](#)
- [3.5 Target configuration editor - Registers tab on page 3-71.](#)
- [3.6 Target configuration editor - Group View tab on page 3-73.](#)
- [3.7 Target configuration editor - Enumerations tab on page 3-76.](#)
- [3.8 Target configuration editor - Configurations tab on page 3-78.](#)
- [10.37 Debug Configurations - Files tab on page 10-279.](#)
- [3.9 Scenario demonstrating how to create a new target configuration file on page 3-80.](#)

3.2 Target configuration editor - Overview tab

A graphical view showing general information about the current target and summary information for all the tabs.

General Information

Unique Name

Unique board name (mandatory).

Category

Name of the manufacturer.

Inherits

Name of the board, memory region or peripheral to inherit data from. You must use the **Includes** panel to populate this drop-down menu.

Endianness

Byte order of the target.

TrustZone

TrustZone support for the target. If supported, the **Memory** and **Peripheral** tabs are displayed with a TrustZone **Address Type** field.

Power Domain

Power Domain support for the target. If supported, the **Memory** and **Peripheral** tabs are displayed with a Power Domain **Address Type** field. Also, the **Configurations** tab includes an additional **Power Domain Configurations** group.

Description

Board description.

Includes

Include files for use when inheriting target data that is defined in an external file.
Populates the **Inherits** drop-down menu.

The **Overview** tab also provides a summary of the other tabs available in this view, together with the total number of items defined in that view.

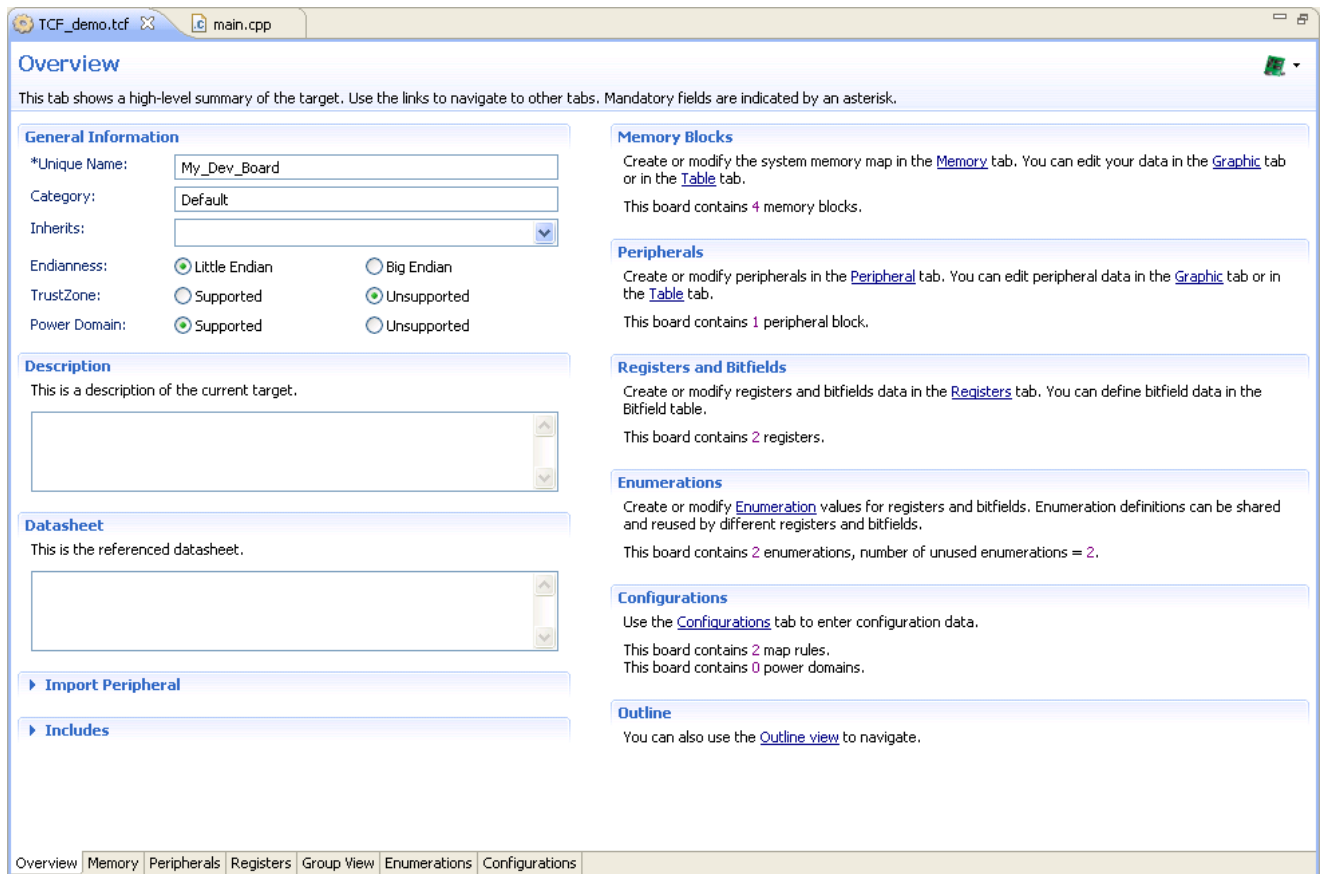


Figure 3-1 Target configuration editor - Overview tab

Mandatory fields are indicated by an asterisk. Toolbar buttons and error messages are displayed in the header panel as appropriate.

Related concepts

[3.1 About the target configuration editor on page 3-64.](#)

Related tasks

[3.10 Creating a power domain for a target on page 3-92.](#)

Related references

[3.3 Target configuration editor - Memory tab on page 3-67.](#)

[3.4 Target configuration editor - Peripherals tab on page 3-69.](#)

[3.5 Target configuration editor - Registers tab on page 3-71.](#)

[3.6 Target configuration editor - Group View tab on page 3-73.](#)

[3.7 Target configuration editor - Enumerations tab on page 3-76.](#)

[3.8 Target configuration editor - Configurations tab on page 3-78.](#)

[3.9 Scenario demonstrating how to create a new target configuration file on page 3-80.](#)

3.3 Target configuration editor - Memory tab

A graphical view or tabular view that enables you to define the attributes for each of the block of memory on your target. These memory blocks are used to ensure that your debugger accesses the memory on your target in the right way.

Graphical view

In the graphical view, the following display options are available:

View by Map Rule

Filter the graphical view based on the selected rule.

View by Address Type

Filter the graphical view based on secure or non-secure addresses. Available only when TrustZone is supported. You can select TrustZone support in the **Overview** tab.

View by Power Domain

Filter the graphical view based on the power domain. Available only when Power Domain is supported. You can select Power Domain support in the **Overview** tab.

Add button

Add a new memory region.

Remove button

Remove the selected memory region.

Graphical and tabular views

In both the graphical view and the tabular view, the following settings are available:

Unique Name

Name of the selected memory region (mandatory).

Name

User-friendly name for the selected memory region.

Description

Detailed description of the selected memory region.

Base Address

Absolute address or the Name of the memory region to use as a base address. The default is an absolute starting address of 0×0 .

Offset

Offset that is added to the base address (mandatory).

Size

Size of the selected memory region in bytes (mandatory).

Width

Access width of the selected memory region.

Access

Access mode for the selected memory region.

Apply Map Rule (graphical view) Map Rule (tabular view)

Mapping rule to be applied to the selected memory region. You can use the **Map Rules** tab to create and modify rules for control registers.

More... (tabular view)

In the tabular view, the ... button is displayed when you select **More...** cell. Click the ... button to display the Context and Parameters dialog box.

Context

Debugger plug-in. If you want to pass parameters to a specific debugger, select a plug-in and enter the associated parameters.

Parameters

Parameters associated with the selected debugger plug-in. Select the required debugger plug-in from the **Context** drop-down menu to enter parameters for that debugger plug-in.

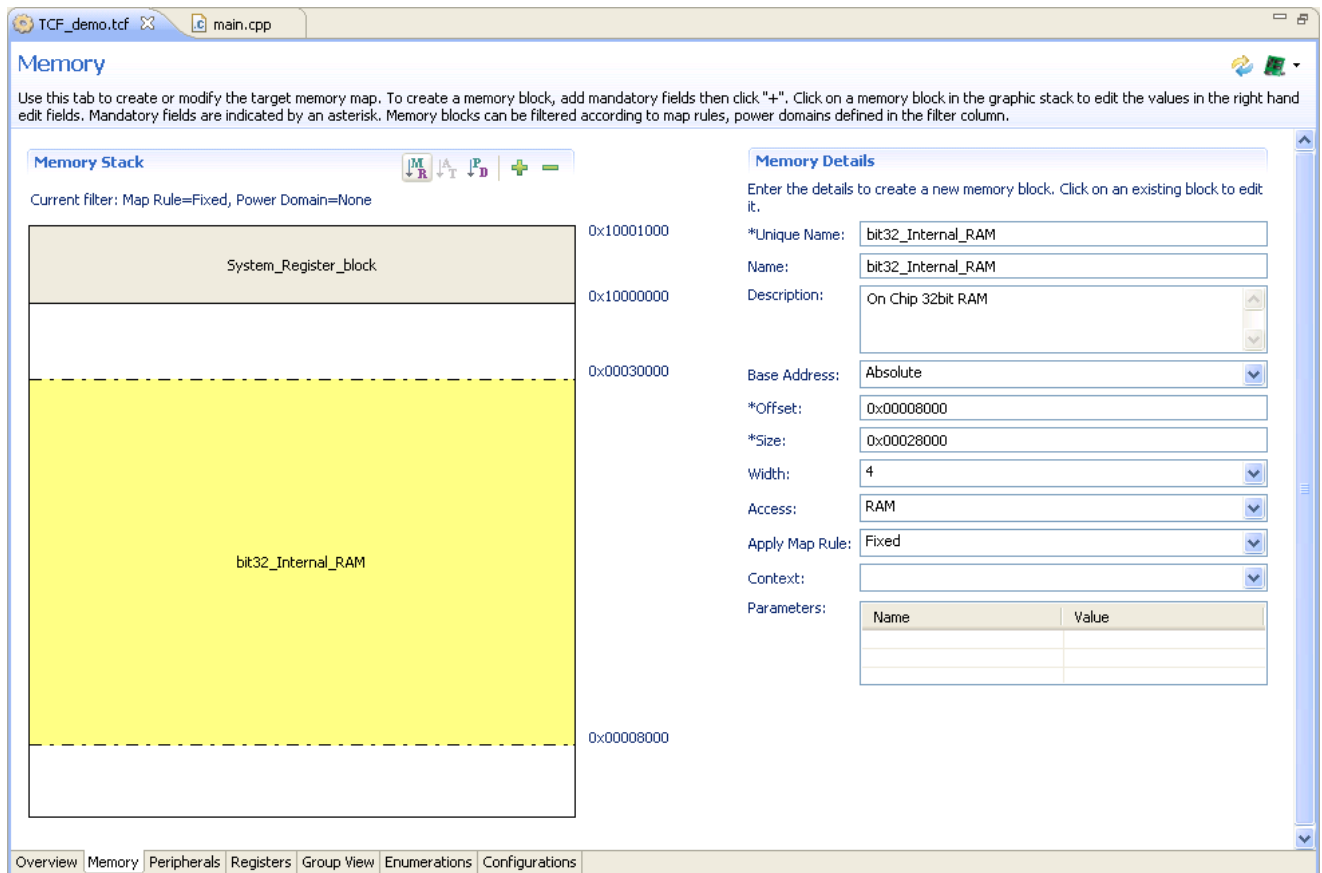


Figure 3-2 Target configuration editor - Memory tab

Mandatory fields are indicated by an asterisk. Toolbar buttons and error messages are displayed in the header panel as appropriate.

Related concepts

[3.1 About the target configuration editor on page 3-64.](#)

Related tasks

[3.9.1 Creating a memory map on page 3-81.](#)

[3.9.8 Creating a memory region for remapping by a control register on page 3-88.](#)

[3.9.9 Applying the map rules to the overlapping memory regions on page 3-89.](#)

Related references

[3.2 Target configuration editor - Overview tab on page 3-65.](#)

[3.4 Target configuration editor - Peripherals tab on page 3-69.](#)

[3.5 Target configuration editor - Registers tab on page 3-71.](#)

[3.6 Target configuration editor - Group View tab on page 3-73.](#)

[3.7 Target configuration editor - Enumerations tab on page 3-76.](#)

[3.8 Target configuration editor - Configurations tab on page 3-78.](#)

3.4 Target configuration editor - Peripherals tab

A graphical view or tabular view that enables you to define peripherals on your target. They can then be mapped in memory, for display and control, and accessed for block data, when available. You define the peripheral in terms of the area of memory it occupies.

Graphical view

In the graphical view, the following display options are available:

View by Address Type

Filter the graphical view based on secure or non-secure addresses. Available only when TrustZone is supported. You can select TrustZone support in the **Overview** tab.

View by Power Domain

Filter the graphical view based on the power domain. Available only when Power Domain is supported. You can select Power Domain support in the **Overview** tab.

Add button

Add a new peripheral.

Remove button

Remove the selected peripheral and, if required, the associated registers.

Graphical and tabular views

In both the graphical view and the tabular view, the following settings are available:

Unique Name

Name of the selected peripheral (mandatory).

Name

User-friendly name for the selected peripheral.

Description

Detailed description of the selected peripheral.

Base Address

Absolute address or the Name of the memory region to use as a base address. The default is an absolute starting address of 0×0 .

Offset

Offset that is added to the base address (mandatory).

Size

Size of the selected peripheral in bytes.

Width

Access width of the selected peripheral in bytes.

Access

Access mode for the selected peripheral.

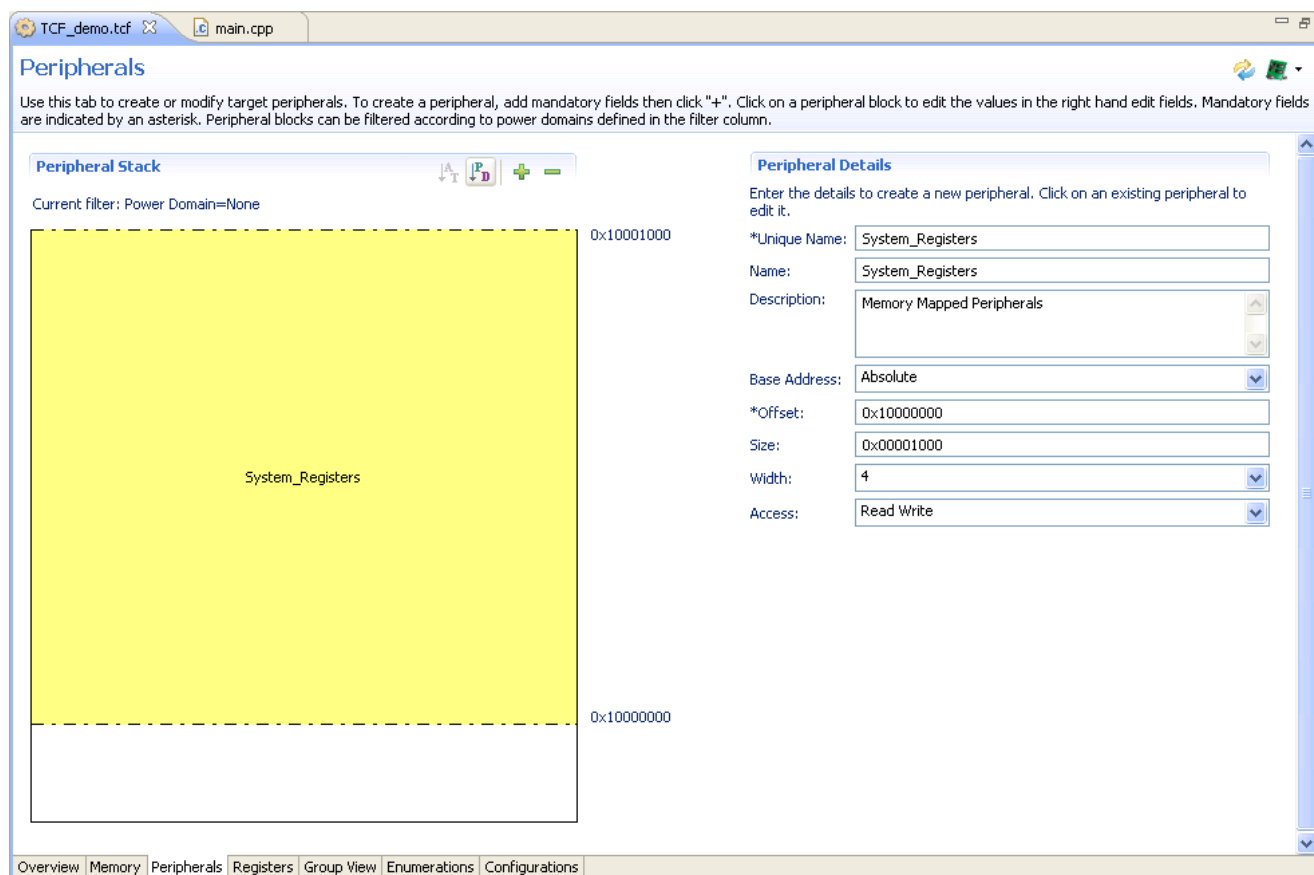


Figure 3-3 Target configuration editor - Peripherals tab

Mandatory fields are indicated by an asterisk. Toolbar buttons and error messages are displayed in the header panel as appropriate.

Related concepts

[3.1 About the target configuration editor on page 3-64.](#)

Related tasks

[3.9.2 Creating a peripheral on page 3-82.](#)

Related references

[3.2 Target configuration editor - Overview tab on page 3-65.](#)

[3.3 Target configuration editor - Memory tab on page 3-67.](#)

[3.5 Target configuration editor - Registers tab on page 3-71.](#)

[3.6 Target configuration editor - Group View tab on page 3-73.](#)

[3.7 Target configuration editor - Enumerations tab on page 3-76.](#)

[3.8 Target configuration editor - Configurations tab on page 3-78.](#)

3.5 Target configuration editor - Registers tab

A tabular view that enables you to define memory mapped registers for your target. Each register is named and typed and can be subdivided into bit fields (any number of bits) which act as subregisters.

Unique Name

Name of the register (mandatory).

Name

User-friendly name for the register.

Base Address

Absolute address or the Name of the memory region to use as a base address. The default is an absolute starting address of 0x0.

Offset

Offset that is added to the base address (mandatory).

Size

Size of the register in bytes (mandatory).

Access size

Access width of the register in bytes.

Access

Access mode for the selected register.

Description

Detailed description of the register.

Peripheral

Associated peripheral, if applicable.

The **Bitfield** button opens a table displaying the following information:

Unique Name

Name of the selected bitfield (mandatory).

Name

User-friendly name for the selected bitfield.

Low Bit

Zero indexed low bit number for the selected bitfield (mandatory).

High Bit

Zero indexed high bit number for the selected bitfield (mandatory).

Access

Access mode for the selected bitfield.

Description

Detailed description of the selected bitfield.

Enumeration

Associated enumeration for the selected bitfield, if applicable.

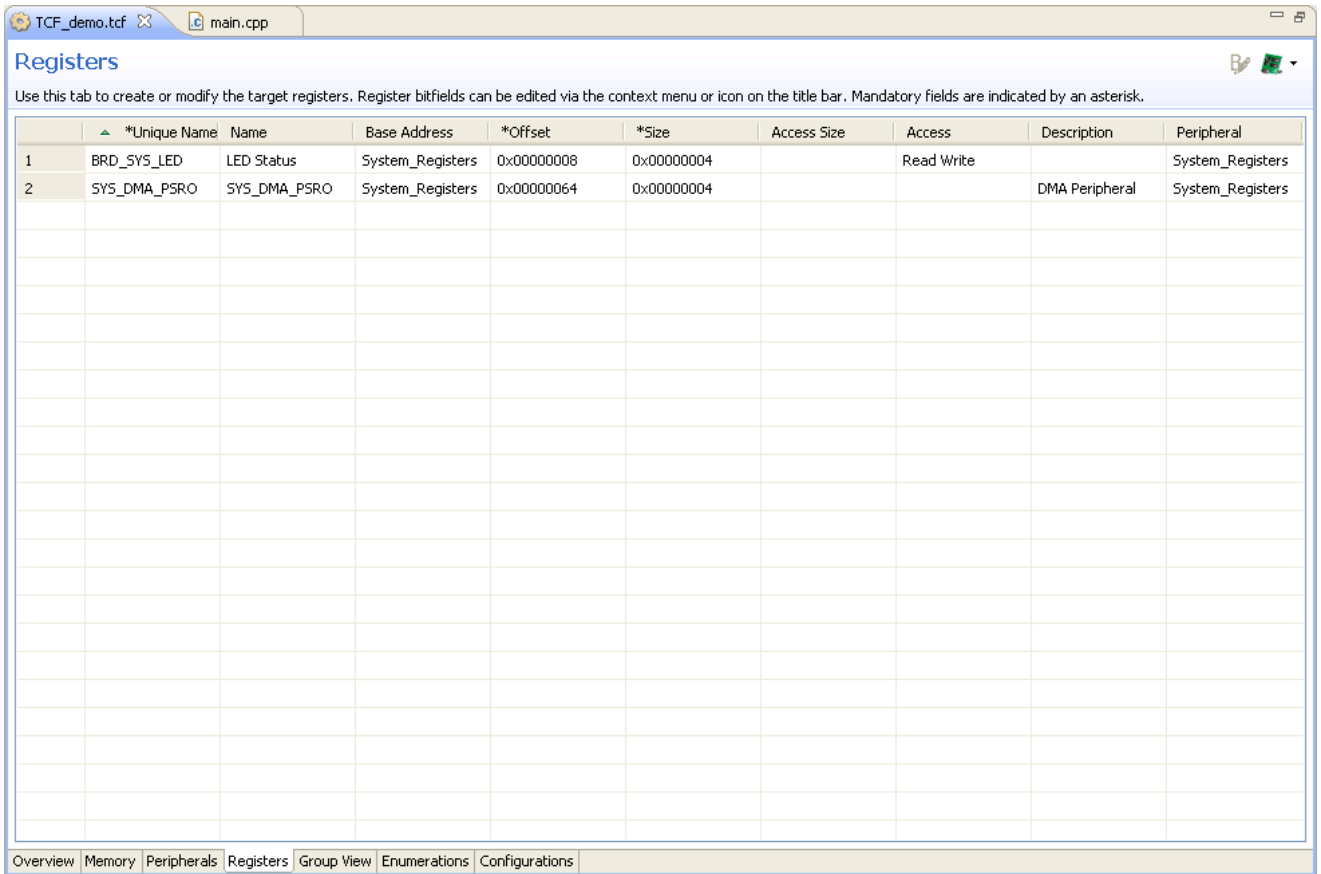


Figure 3-4 Target configuration editor - Registers tab

Mandatory fields are indicated by an asterisk. Toolbar buttons and error messages are displayed in the header panel as appropriate.

Related concepts

3.1 About the target configuration editor on page 3-64.

Related tasks

3.9.3 Creating a standalone register on page 3-83.

3.9.4 Creating a peripheral register on page 3-84.

3.9.6 Assigning enumerations to a peripheral register on page 3-86.

3.9.5 Creating enumerations for use with a peripheral register on page 3-85.

3.9.6 Assigning enumerations to a peripheral register on page 3-86.

Related references

3.2 Target configuration editor - Overview tab on page 3-65.

3.3 Target configuration editor - Memory tab on page 3-67.

3.4 Target configuration editor - Peripherals tab on page 3-69.

3.6 Target configuration editor - Group View tab on page 3-73.

3.7 Target configuration editor - Enumerations tab on page 3-76.

3.8 Target configuration editor - Configurations tab on page 3-78.

3.6 Target configuration editor - Group View tab

A list view that enables you to select peripherals for use by the debugger.

Group View List

Empty list that enables you to add frequently used peripherals to the debugger.

Add a new group

Creates a group that you can personalize with peripherals.

Remove the selected group

Removes a group from the list.

Available Peripheral List

A list of the available peripherals. You can select peripherals from this view to add to the **Group View List**.

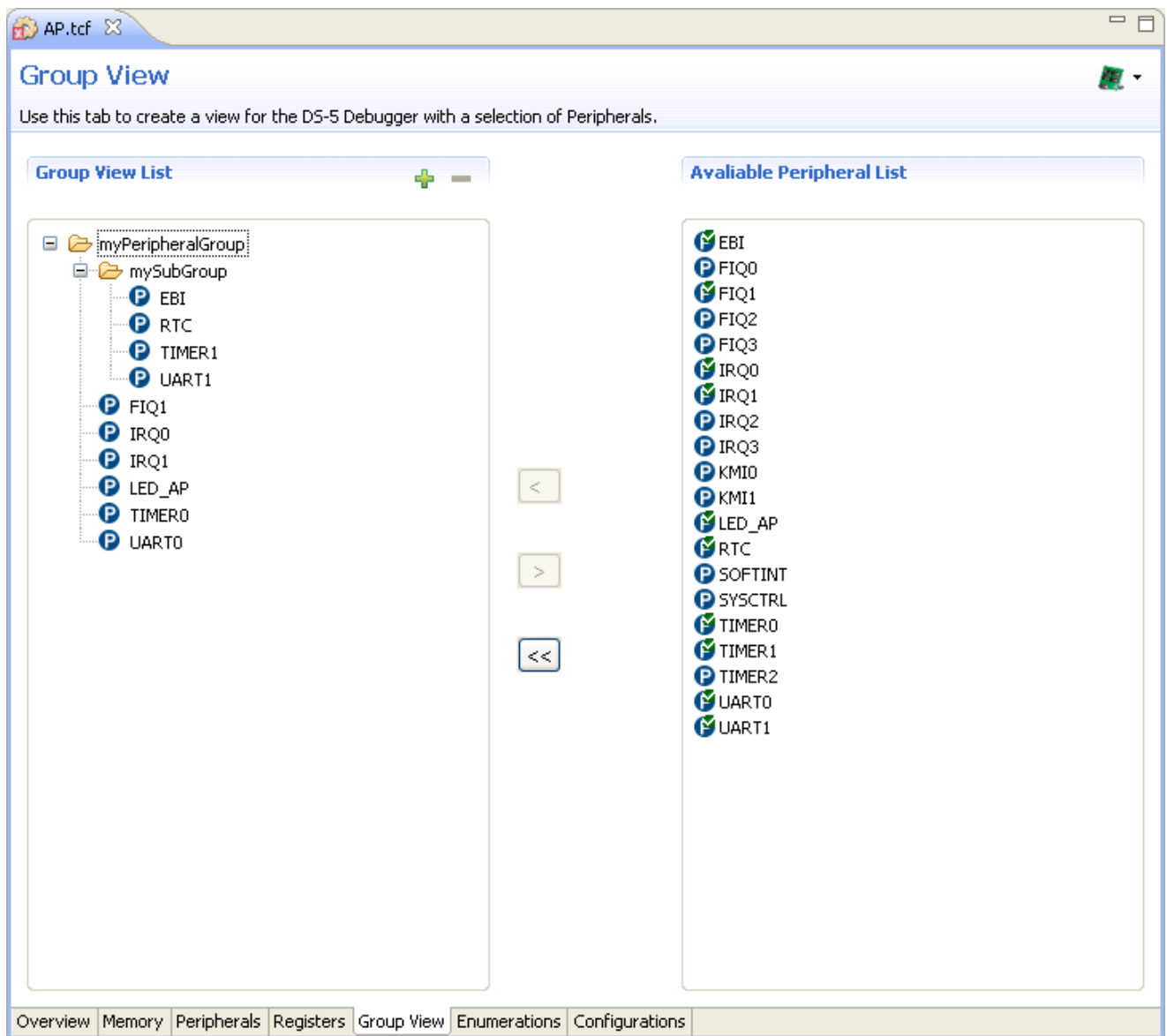


Figure 3-5 Target configuration editor - Group View tab

Mandatory fields are indicated by an asterisk. Toolbar buttons and error messages are displayed in the header panel as appropriate.

Related concepts

[3.1 About the target configuration editor on page 3-64.](#)

Related tasks

[3.11 Creating a Group list on page 3-94.](#)

Related references

[3.2 Target configuration editor - Overview tab on page 3-65.](#)

[3.3 Target configuration editor - Memory tab on page 3-67.](#)

[3.4 Target configuration editor - Peripherals tab on page 3-69.](#)

[3.5 Target configuration editor - Registers tab on page 3-71.](#)

[3.7 Target configuration editor - Enumerations tab on page 3-76.](#)

[3.8 Target configuration editor - Configurations tab on page 3-78.](#)

3.7 Target configuration editor - Enumerations tab

A tabular view that enables you to assign values to meaningful names for use by registers you have defined. Enumerations can be used, instead of values, when a register is displayed in the **Registers** view. This setting enables you to define the names associated with different values. Names defined in this group are displayed in the **Registers** view, and can be used to change register values.

Register bit fields are numbered 0, 1, 2,... regardless of their position in the register.

For example, you might want to define ENABLED as 1 and DISABLED as 0.

The following settings are available:

Unique Name

Name of the selected enumeration (mandatory).

Value

Definitions specified as comma separated values for selection in the **Registers** tab (mandatory).

Description

Detailed description of the selected enumeration.

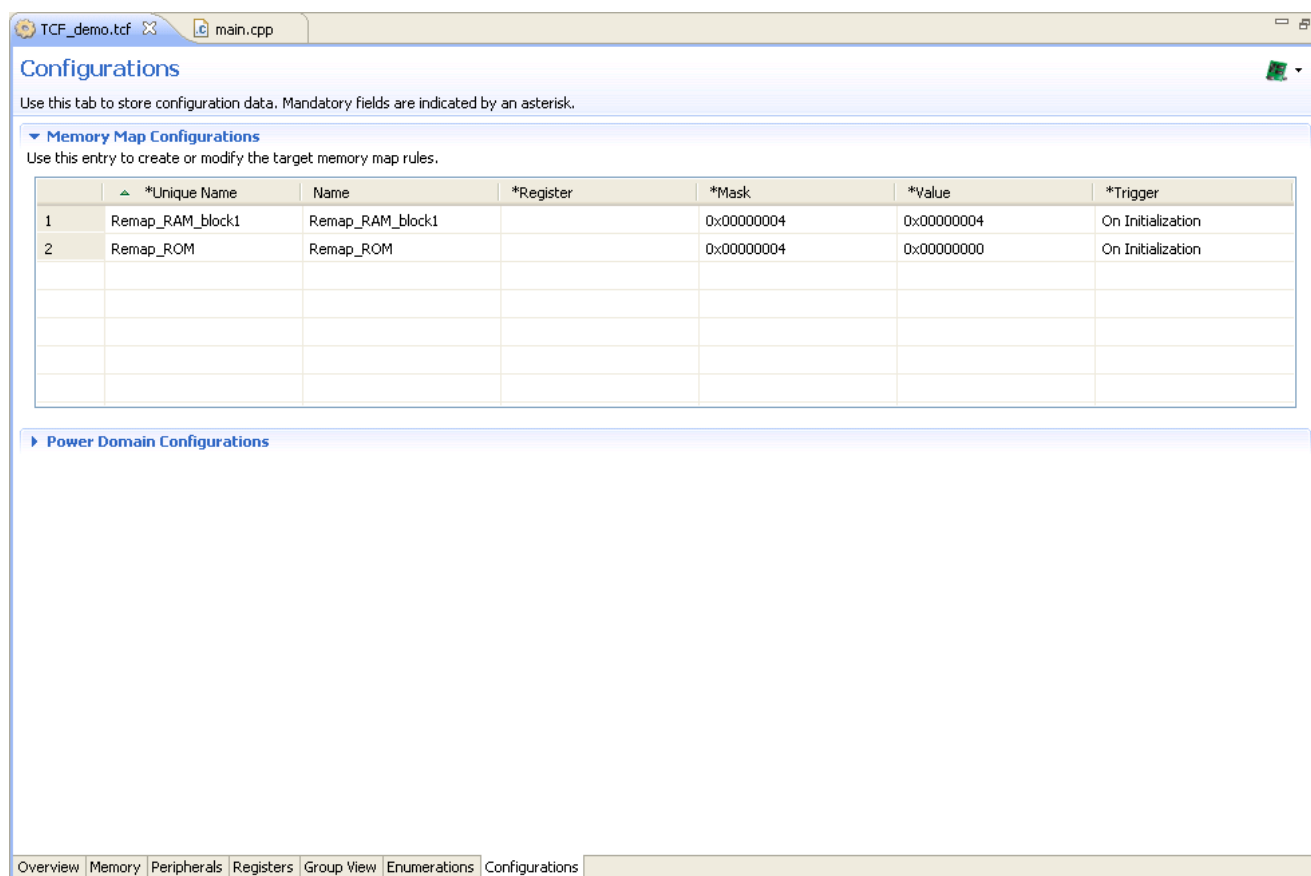


Figure 3-6 Target configuration editor - Enumerations tab

Mandatory fields are indicated by an asterisk. Toolbar buttons and error messages are displayed in the header panel as appropriate.

Related concepts

3.1 About the target configuration editor on page 3-64.

Related tasks

3.9.5 Creating enumerations for use with a peripheral register on page 3-85.

3.9.6 Assigning enumerations to a peripheral register on page 3-86.

Related references

3.2 Target configuration editor - Overview tab on page 3-65.

3.3 Target configuration editor - Memory tab on page 3-67.

3.4 Target configuration editor - Peripherals tab on page 3-69.

3.5 Target configuration editor - Registers tab on page 3-71.

3.6 Target configuration editor - Group View tab on page 3-73.

3.8 Target configuration editor - Configurations tab on page 3-78.

3.8 Target configuration editor - Configurations tab

A tabular view that enables you to:

- Define rules to control the enabling and disabling of memory blocks using target registers. You specify a register to be monitored, and when the contents match a given value, a set of memory blocks is enabled. You can define several map rules, one for each of several memory blocks.
- Define power domains that are supported on your target.

Memory Map Configurations group

The following settings are available in the **Memory Map Configurations** group:

Unique Name

Name of the rule (mandatory).

Name

User-friendly name for the rule.

Register

Associated control register (mandatory).

Mask

Mask value (mandatory).

Value

Value for a condition (mandatory).

Trigger

Condition that changes the control register mapping (mandatory).

Power Domain Configurations group

The **Power Domain Configurations** group

The following settings are available in this group, and all are mandatory:

Unique Name

Name of the power domain.

Wake-up Conditions

User-friendly name for the rule:

Register

An associated control register that you have previously created.

Mask

Mask value.

Value

Value for a condition.

Power State

The power state of the power domain:

- Active.
- Inactive.
- Retention.
- Off.

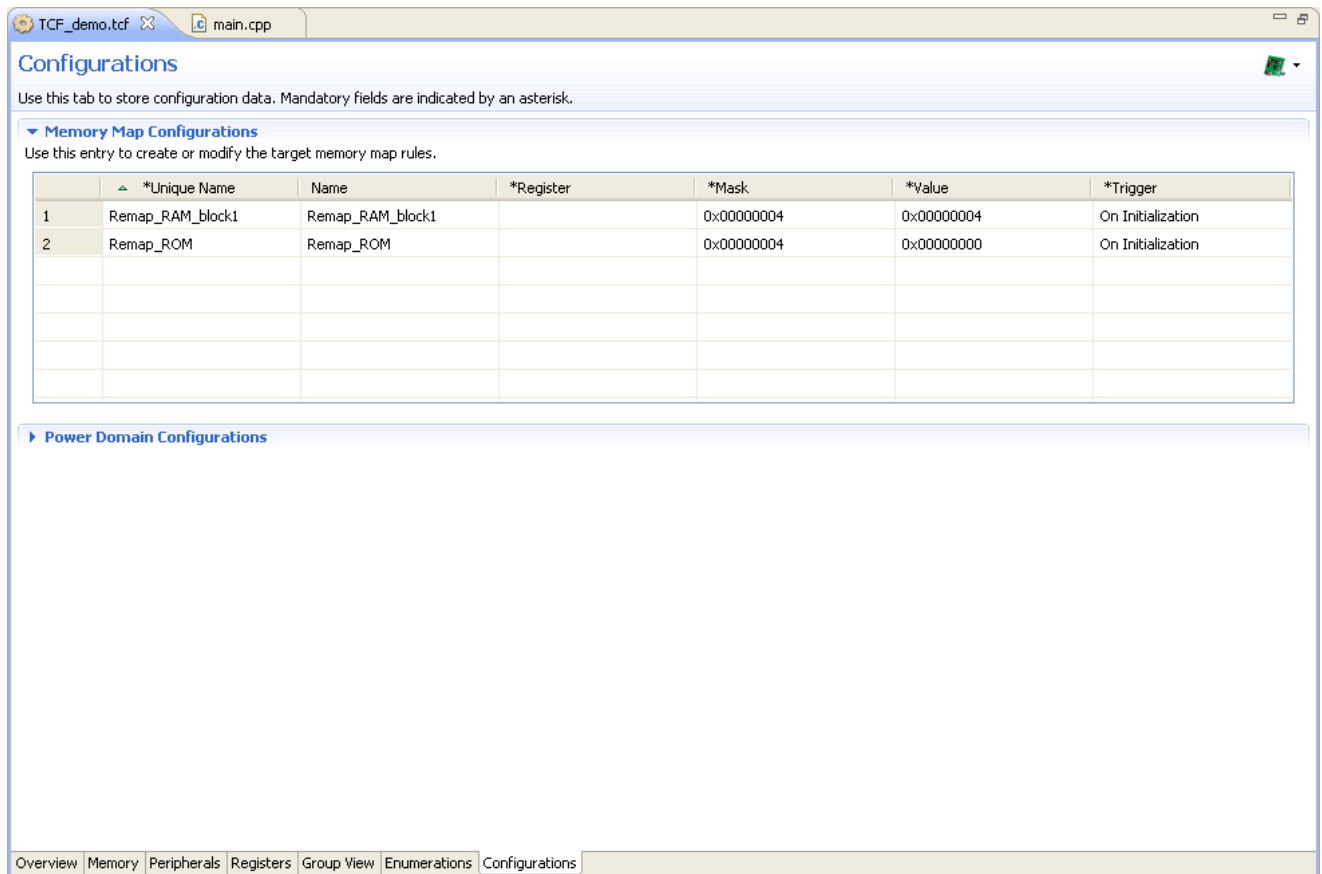


Figure 3-7 Target configuration editor - Configuration tab

Mandatory fields are indicated by an asterisk. Toolbar buttons and error messages are displayed in the header panel as appropriate.

Related concepts

[3.1 About the target configuration editor on page 3-64.](#)

Related tasks

[3.9.3 Creating a standalone register on page 3-83.](#)

[3.9.7 Creating remapping rules for a control register on page 3-87.](#)

[3.10 Creating a power domain for a target on page 3-92.](#)

Related references

[3.2 Target configuration editor - Overview tab on page 3-65.](#)

[3.3 Target configuration editor - Memory tab on page 3-67.](#)

[3.4 Target configuration editor - Peripherals tab on page 3-69.](#)

[3.5 Target configuration editor - Registers tab on page 3-71.](#)

[3.6 Target configuration editor - Group View tab on page 3-73.](#)

[3.7 Target configuration editor - Enumerations tab on page 3-76.](#)

3.9 Scenario demonstrating how to create a new target configuration file

This is a fictitious scenario to demonstrate how to create a new Target Configuration File (TCF) containing the following memory map and register definitions. The individual tasks required to complete each step of this tutorial are listed below.

- Boot ROM: 0x0 - 0x8000
- SRAM: 0x0 - 0x8000
- Internal RAM: 0x8000 - 0x28000
- System Registers that contain memory mapped peripherals:

0x10000000 - 0x10001000.

- A basic standalone LED register. This register is located at 0x10000008 and is used to write a hexadecimal value that sets the corresponding bits to 1 to illuminate the respective LEDs.

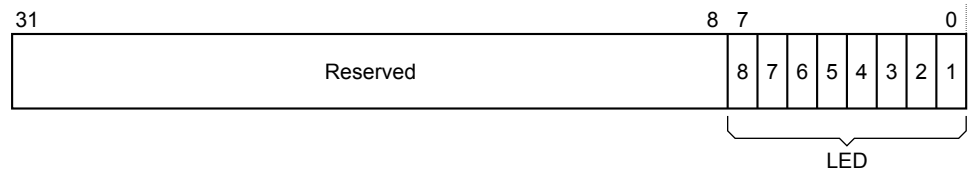


Figure 3-8 LED register and bitfields

- DMA map register. This register is located at 0x10000064 and controls the mapping of external peripheral DMA request and acknowledge signals to DMA channel 0.

Table 3-1 DMA map register SYS_DMPSR0

Bits [31:8]	-	Reserved. Use read-modify-write to preserve value
Bit [7]	Read/Write	Set to 1 to enable mapping of external peripheral DMA signals to the DMA controller channel.
Bits [6:5]	-	Reserved. Use read-modify-write to preserve value
Bits [4:0]	Read/Write	FPGA peripheral mapped to this channel

b00000 = AACI Tx
b00001 = AACI Rx
b00010 = USB A
b00011 = USB B
b00100 = MCI 0

- The core module and LCD control register. This register is located at 0x1000000C and controls a number of user-configurable features of the core module and the display interface on the baseboard.

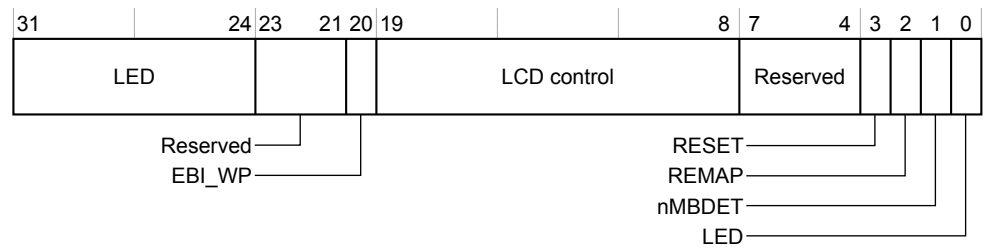


Figure 3-9 Core module and LCD control register

This register uses bit 2 to control the remapping of an area of memory as shown in the following table.

Table 3-2 Control bit that remaps an area of memory

Bits	Name	Access	Function
[2]	REMAP	Read/Write	0 = Flash ROM at address 0 1 = SRAM at address 0.

- Clearing bit 2 (CM_CTRL = 0) generates the following memory map:

0x0000 - 0x8000 Boot_ROM

0x8000 - 0x28000 32bit_RAM

- Setting bit 2 (CM_CTRL = 1) generates the following memory map:

0x0000 - 0x8000 32bit_RAM_block1_alias

0x8000 - 0x28000 32bit_RAM

It contains the following:

- [3.9.1 Creating a memory map on page 3-81.](#)
- [3.9.2 Creating a peripheral on page 3-82.](#)
- [3.9.3 Creating a standalone register on page 3-83.](#)
- [3.9.4 Creating a peripheral register on page 3-84.](#)
- [3.9.5 Creating enumerations for use with a peripheral register on page 3-85.](#)
- [3.9.6 Assigning enumerations to a peripheral register on page 3-86.](#)
- [3.9.7 Creating remapping rules for a control register on page 3-87.](#)
- [3.9.8 Creating a memory region for remapping by a control register on page 3-88.](#)
- [3.9.9 Applying the map rules to the overlapping memory regions on page 3-89.](#)

3.9.1 Creating a memory map

Describes how to create a new memory map.

Procedure

1. Add a new file with the **.tcf** file extension to an open project.
The editor opens with the **Overview** tab activated.
2. Click on the **Overview** tab, enter a unique board name, for example: **My-Dev-Board**.
3. Click on the **Memory** tab.
4. Click the **Switch to table** button in the top right of the view.
5. Enter the data as shown in the following figure.

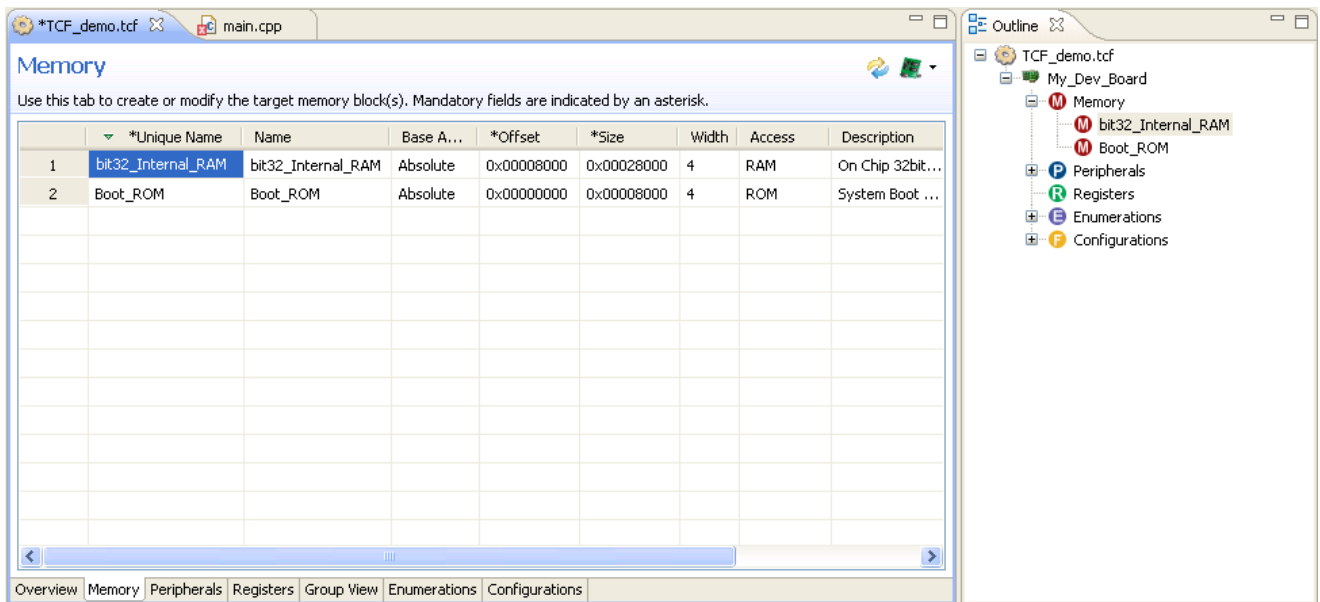


Figure 3-10 Creating a Memory map

On completion, you can switch back to the graphical view to see the color coded stack of memory regions.

Related tasks

- [3.9.2 Creating a peripheral on page 3-82.](#)
- [3.9.3 Creating a standalone register on page 3-83.](#)
- [3.9.4 Creating a peripheral register on page 3-84.](#)
- [3.9.5 Creating enumerations for use with a peripheral register on page 3-85.](#)
- [3.9.6 Assigning enumerations to a peripheral register on page 3-86.](#)
- [3.9.7 Creating remapping rules for a control register on page 3-87.](#)
- [3.9.8 Creating a memory region for remapping by a control register on page 3-88.](#)
- [3.9.9 Applying the map rules to the overlapping memory regions on page 3-89.](#)

Related references

- [3.9 Scenario demonstrating how to create a new target configuration file on page 3-80.](#)
- [3.3 Target configuration editor - Memory tab on page 3-67.](#)

3.9.2 Creating a peripheral

Describes how to create a peripheral.

Procedure

1. Click on the **Peripherals** tab.
2. Click the **Switch to table** button in the top right of the view.
3. Enter the data as shown in the following figure.

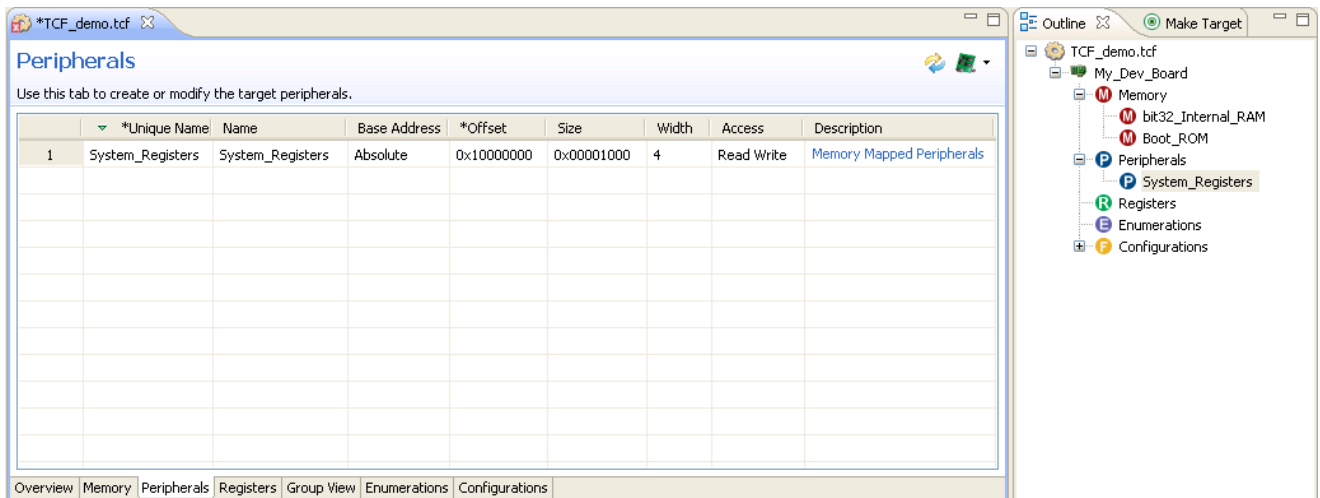


Figure 3-11 Creating a peripheral

On completion, you can switch back to the graphical view to see the color coded stack of peripherals.

Related tasks

- [3.9.1 Creating a memory map on page 3-81.](#)
- [3.9.3 Creating a standalone register on page 3-83.](#)
- [3.9.4 Creating a peripheral register on page 3-84.](#)
- [3.9.5 Creating enumerations for use with a peripheral register on page 3-85.](#)
- [3.9.6 Assigning enumerations to a peripheral register on page 3-86.](#)
- [3.9.7 Creating remapping rules for a control register on page 3-87.](#)
- [3.9.8 Creating a memory region for remapping by a control register on page 3-88.](#)
- [3.9.9 Applying the map rules to the overlapping memory regions on page 3-89.](#)

Related references

- [3.9 Scenario demonstrating how to create a new target configuration file on page 3-80.](#)
- [3.4 Target configuration editor - Peripherals tab on page 3-69.](#)

3.9.3 Creating a standalone register

Describes how to create a basic standalone register.

Procedure

1. Click on the **Registers** tab.
2. Enter the register data as shown in the following figure.
3. Bitfield data is entered in a floating table associated with the selected register. Select the Unique name field containing the register name, BRD_SYS_LED.
4. Click on the **Edit Bitfield** button in the top right corner of the view.
5. In the floating Bitfield table, enter the data as shown in the following figure. If required, you can dock this table below the register table by clicking on the title bar of the Bitfield table and dragging it to the base of the register table.

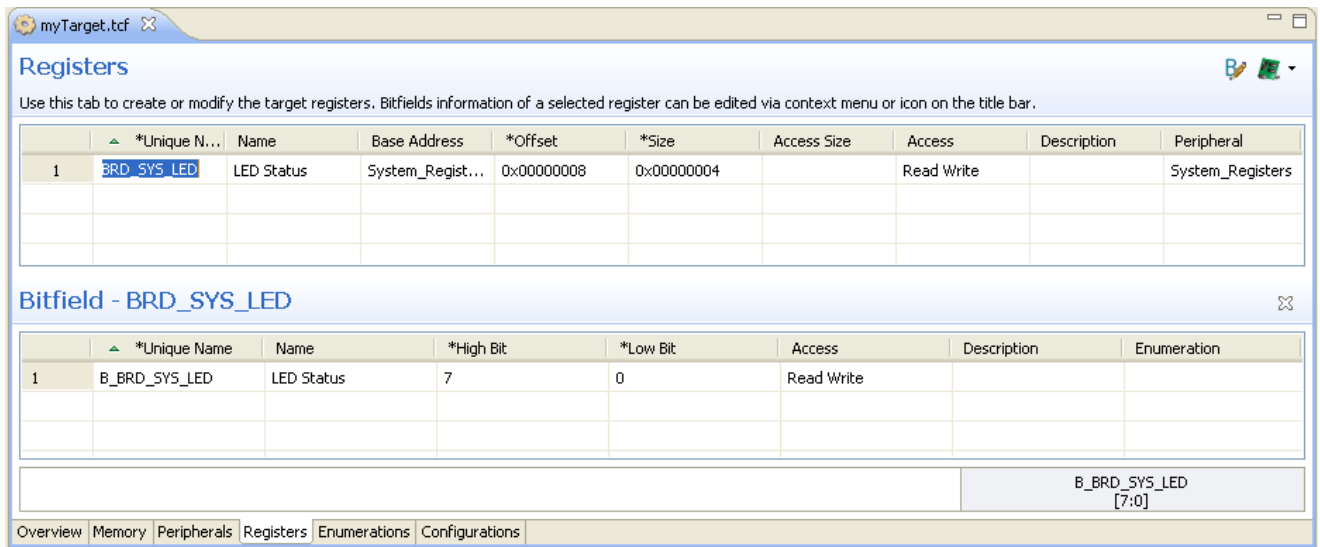


Figure 3-12 Creating a standalone register

On completion, close the floating table.

Related tasks

- [3.9.1 Creating a memory map on page 3-81.](#)
- [3.9.2 Creating a peripheral on page 3-82.](#)
- [3.9.4 Creating a peripheral register on page 3-84.](#)
- [3.9.5 Creating enumerations for use with a peripheral register on page 3-85.](#)
- [3.9.6 Assigning enumerations to a peripheral register on page 3-86.](#)
- [3.9.7 Creating remapping rules for a control register on page 3-87.](#)
- [3.9.8 Creating a memory region for remapping by a control register on page 3-88.](#)
- [3.9.9 Applying the map rules to the overlapping memory regions on page 3-89.](#)

Related references

- [3.9 Scenario demonstrating how to create a new target configuration file on page 3-80.](#)
- [3.5 Target configuration editor - Registers tab on page 3-71.](#)
- [3.8 Target configuration editor - Configurations tab on page 3-78.](#)

3.9.4 Creating a peripheral register

Describes how to create a peripheral register.

Procedure

1. Click on the **Registers** tab, if it is not already active.
2. Enter the peripheral register and associated bitfield data as shown in the following figure.

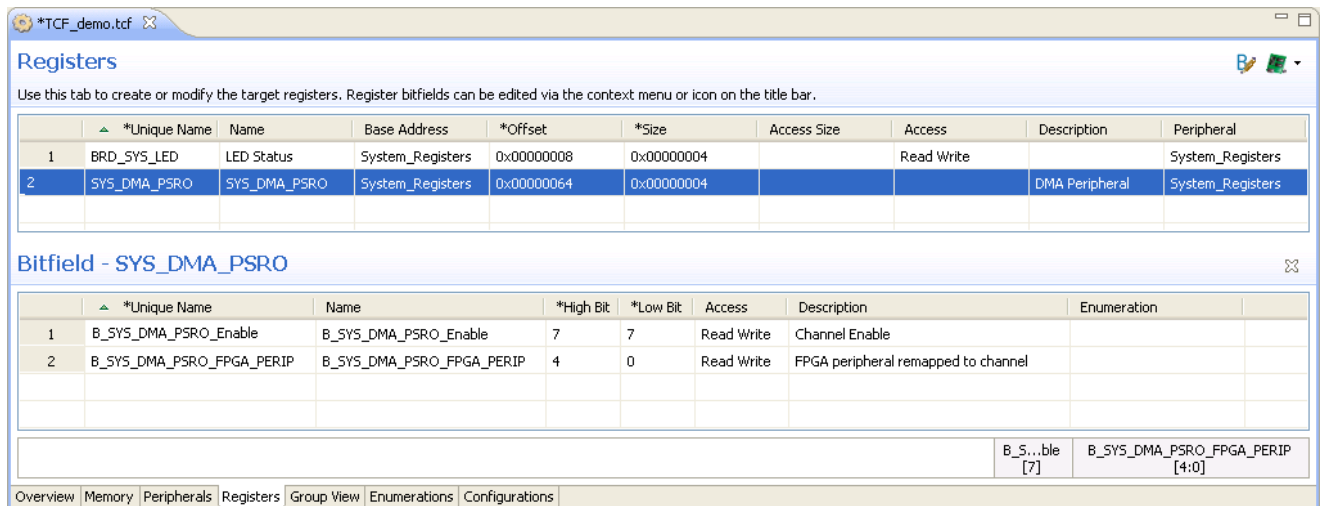


Figure 3-13 Creating a peripheral register

Related tasks

- [3.9.1 Creating a memory map on page 3-81.](#)
- [3.9.2 Creating a peripheral on page 3-82.](#)
- [3.9.3 Creating a standalone register on page 3-83.](#)
- [3.9.5 Creating enumerations for use with a peripheral register on page 3-85.](#)
- [3.9.6 Assigning enumerations to a peripheral register on page 3-86.](#)
- [3.9.7 Creating remapping rules for a control register on page 3-87.](#)
- [3.9.8 Creating a memory region for remapping by a control register on page 3-88.](#)
- [3.9.9 Applying the map rules to the overlapping memory regions on page 3-89.](#)

Related references

- [3.9 Scenario demonstrating how to create a new target configuration file on page 3-80.](#)
- [3.5 Target configuration editor - Registers tab on page 3-71.](#)

3.9.5 Creating enumerations for use with a peripheral register

Describes how to create enumerations for use with a peripheral.

With more complex peripherals it can be useful to create and assign enumerations to particular peripheral bit patterns so that you can select from a list of enumerated values rather than write the equivalent hexadecimal value. (For example: Enabled/Disabled, On/Off).

Procedure

1. Click on the **Enumerations** tab.
2. Enter the data as shown in the following figure.

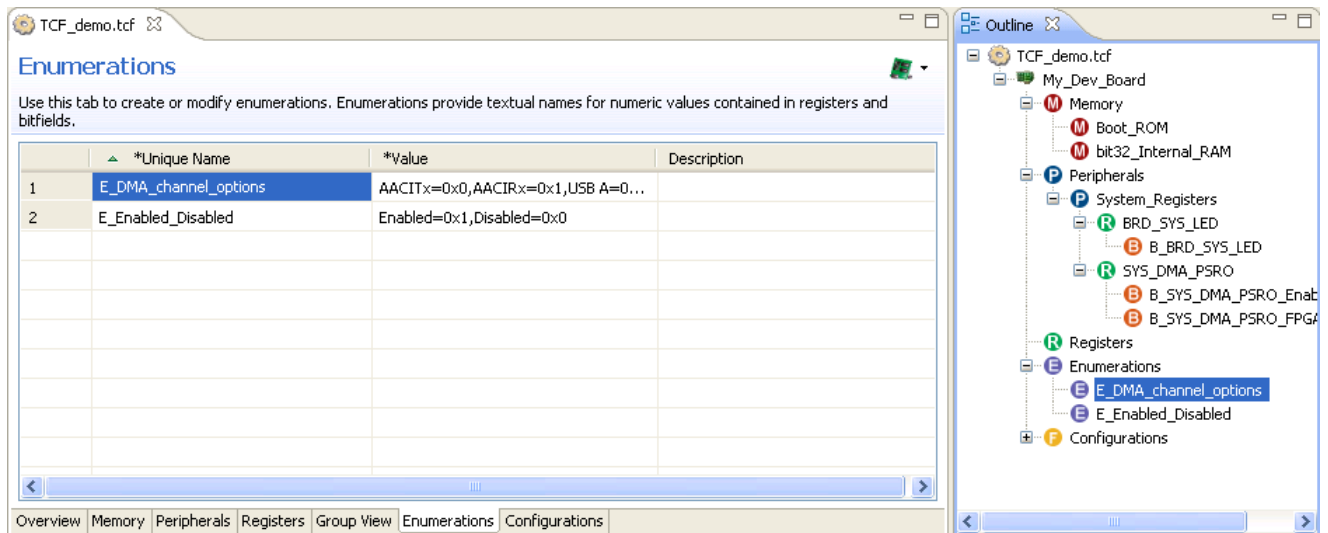


Figure 3-14 Creating enumerations

Related tasks

- [3.9.1 Creating a memory map on page 3-81.](#)
- [3.9.2 Creating a peripheral on page 3-82.](#)
- [3.9.3 Creating a standalone register on page 3-83.](#)
- [3.9.4 Creating a peripheral register on page 3-84.](#)
- [3.9.6 Assigning enumerations to a peripheral register on page 3-86.](#)
- [3.9.7 Creating remapping rules for a control register on page 3-87.](#)
- [3.9.8 Creating a memory region for remapping by a control register on page 3-88.](#)
- [3.9.9 Applying the map rules to the overlapping memory regions on page 3-89.](#)

Related references

- [3.9 Scenario demonstrating how to create a new target configuration file on page 3-80.](#)
- [3.5 Target configuration editor - Registers tab on page 3-71.](#)
- [3.7 Target configuration editor - Enumerations tab on page 3-76.](#)

3.9.6 Assigning enumerations to a peripheral register

Describes how to assign enumerations to a peripheral register.

Procedure

1. Click on the **Registers** tab
2. Open the relevant Bitfield table for the DMA peripheral.
3. Assign enumerations as shown in the following figure.

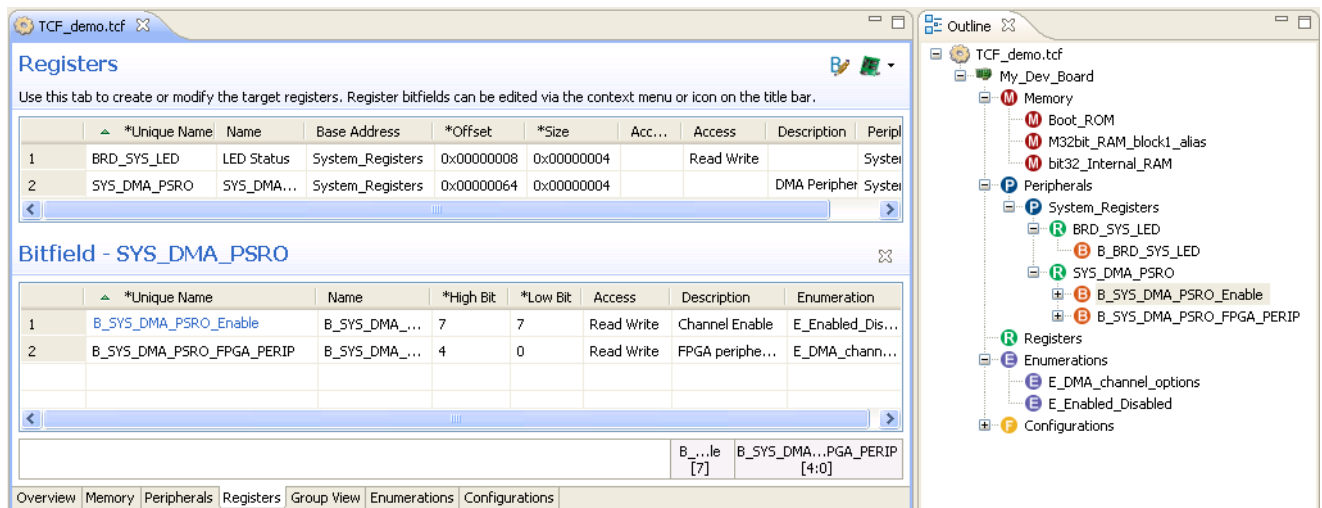


Figure 3-15 Assigning enumerations

Related tasks

- [3.9.1 Creating a memory map on page 3-81.](#)
- [3.9.2 Creating a peripheral on page 3-82.](#)
- [3.9.3 Creating a standalone register on page 3-83.](#)
- [3.9.4 Creating a peripheral register on page 3-84.](#)
- [3.9.5 Creating enumerations for use with a peripheral register on page 3-85.](#)
- [3.9.7 Creating remapping rules for a control register on page 3-87.](#)
- [3.9.8 Creating a memory region for remapping by a control register on page 3-88.](#)
- [3.9.9 Applying the map rules to the overlapping memory regions on page 3-89.](#)

Related references

- [3.9 Scenario demonstrating how to create a new target configuration file on page 3-80.](#)
- [3.5 Target configuration editor - Registers tab on page 3-71.](#)
- [3.5 Target configuration editor - Registers tab on page 3-71.](#)
- [3.7 Target configuration editor - Enumerations tab on page 3-76.](#)

3.9.7 Creating remapping rules for a control register

Describes how to create remapping rules for the core module and LCD control register.

Procedure

1. Click on the **Configurations** tab.
2. Enter the data as shown in the following figure.

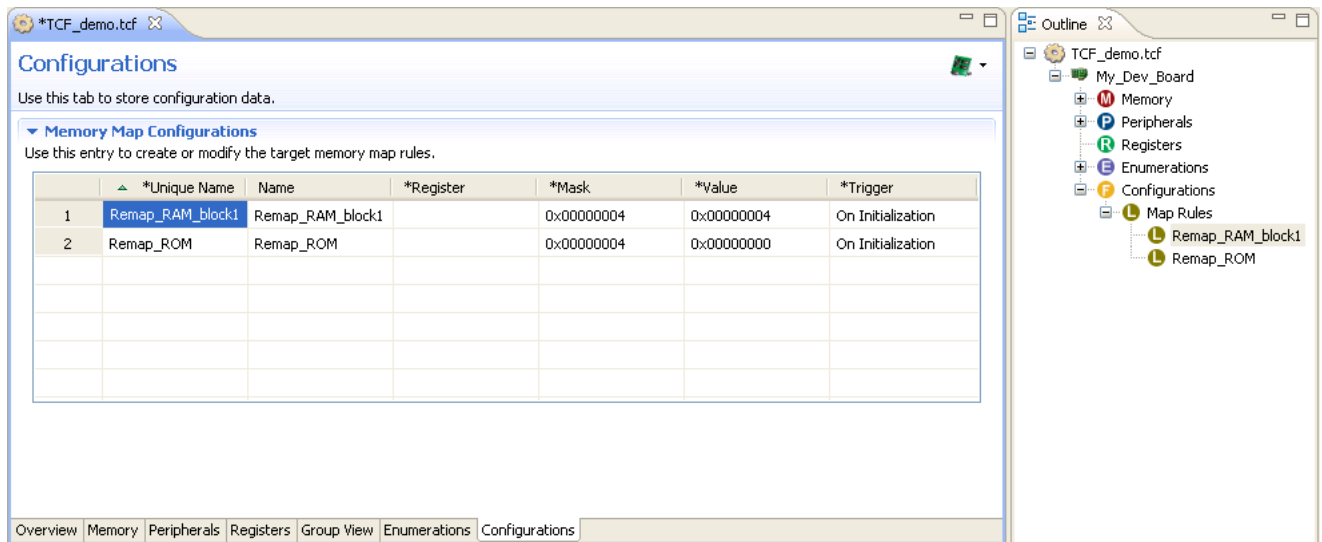


Figure 3-16 Creating remapping rules

Related tasks

- [3.9.1 Creating a memory map on page 3-81.](#)
- [3.9.2 Creating a peripheral on page 3-82.](#)
- [3.9.3 Creating a standalone register on page 3-83.](#)
- [3.9.4 Creating a peripheral register on page 3-84.](#)
- [3.9.5 Creating enumerations for use with a peripheral register on page 3-85.](#)
- [3.9.6 Assigning enumerations to a peripheral register on page 3-86.](#)
- [3.9.8 Creating a memory region for remapping by a control register on page 3-88.](#)
- [3.9.9 Applying the map rules to the overlapping memory regions on page 3-89.](#)

Related references

- [3.9 Scenario demonstrating how to create a new target configuration file on page 3-80.](#)
- [3.8 Target configuration editor - Configurations tab on page 3-78.](#)

3.9.8 Creating a memory region for remapping by a control register

Describes how to create a new memory region that can be used for remapping when bit 2 of the control register is set.

Procedure

1. Click on the **Memory** tab.
2. Switch to the table view by clicking on the relevant button in the top corner.
3. Enter the data as shown in the following figure.

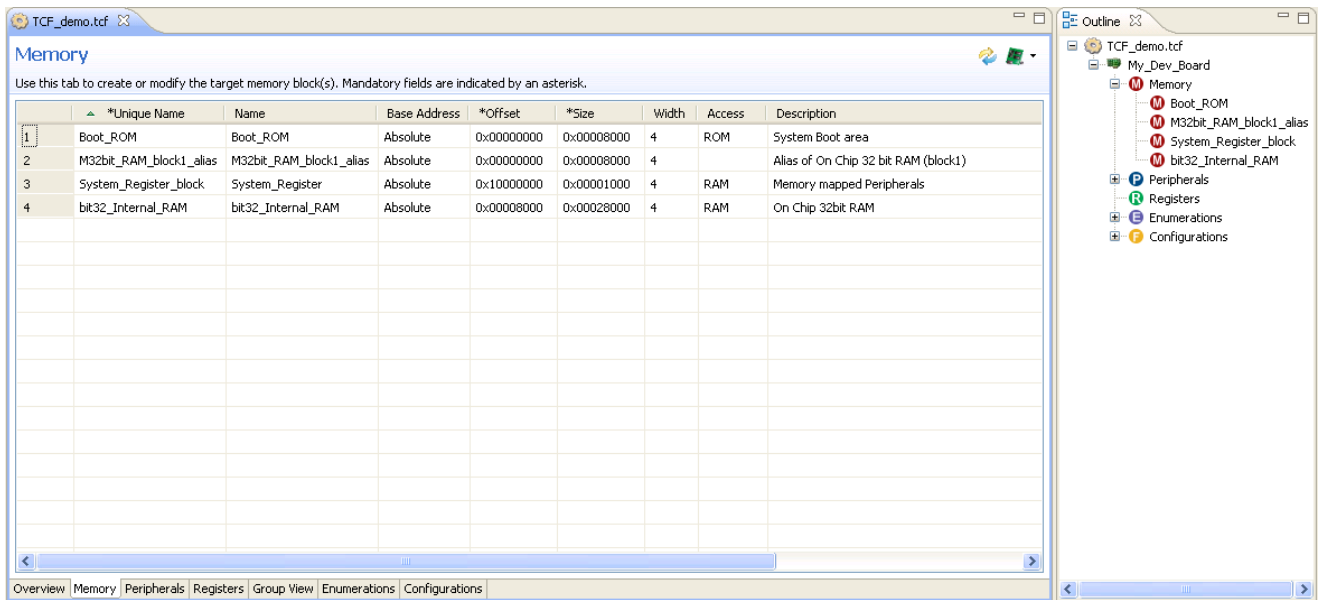


Figure 3-17 Creating a memory region for remapping by a control register

Related tasks

- [3.9.1 Creating a memory map on page 3-81.](#)
- [3.9.2 Creating a peripheral on page 3-82.](#)
- [3.9.3 Creating a standalone register on page 3-83.](#)
- [3.9.4 Creating a peripheral register on page 3-84.](#)
- [3.9.5 Creating enumerations for use with a peripheral register on page 3-85.](#)
- [3.9.6 Assigning enumerations to a peripheral register on page 3-86.](#)
- [3.9.7 Creating remapping rules for a control register on page 3-87.](#)
- [3.9.9 Applying the map rules to the overlapping memory regions on page 3-89.](#)

Related references

- [3.9 Scenario demonstrating how to create a new target configuration file on page 3-80.](#)
- [3.3 Target configuration editor - Memory tab on page 3-67.](#)

3.9.9 Applying the map rules to the overlapping memory regions

Describes how to apply the map rules to the overlapping memory regions.

Procedure

1. Switch back to the graphic view by clicking on the relevant button in the top corner.
2. Select the overlapping memory region **M32bit_RAM_block1_alias** and then select **Remap_RAM_block1** from the Apply Map Rule drop-down menu as shown in the following figure.

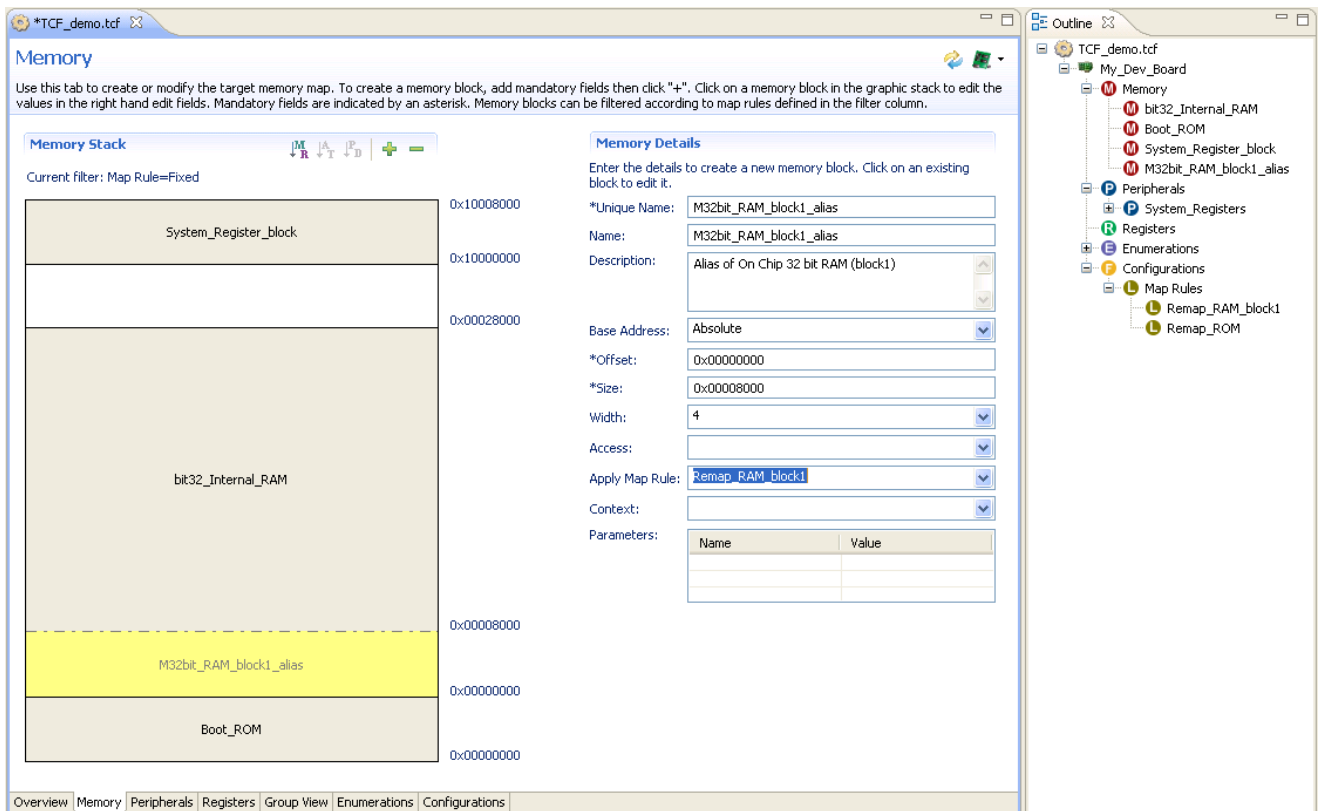


Figure 3-18 Applying the Remap_RAM_block1 map rule

3. To apply the other map rule you must select **Remap_ROM** in the the View by Map Rule drop-down menu at the top of the stack view.
4. Select the overlapping memory region **Boot_ROM** and then select **Remap_ROM** from the Apply Map Rule drop-down menu as shown in the following figure.

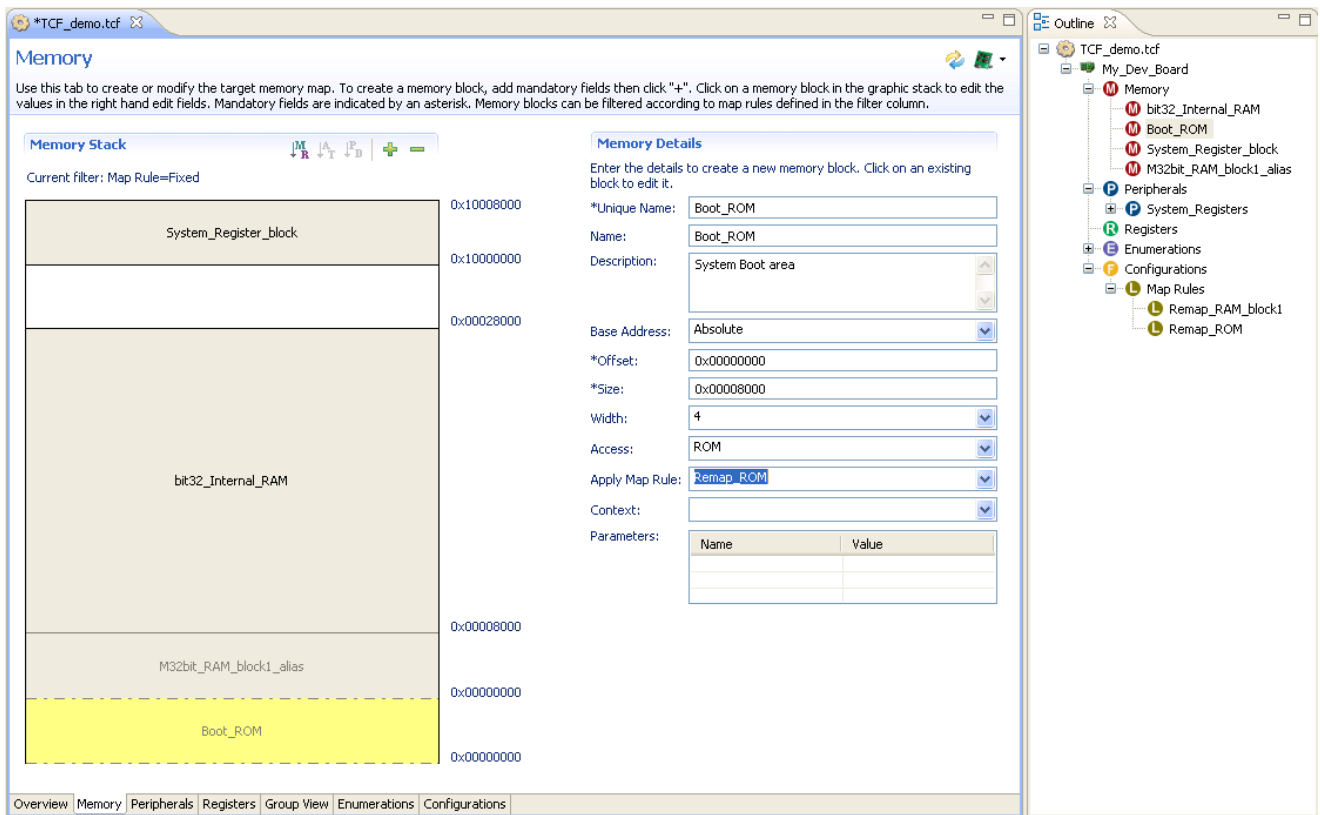


Figure 3-19 Applying the Remap_ROM map rule

5. Save the file.

Related tasks

- [3.9.1 Creating a memory map on page 3-81.](#)
- [3.9.2 Creating a peripheral on page 3-82.](#)
- [3.9.3 Creating a standalone register on page 3-83.](#)
- [3.9.4 Creating a peripheral register on page 3-84.](#)
- [3.9.5 Creating enumerations for use with a peripheral register on page 3-85.](#)
- [3.9.6 Assigning enumerations to a peripheral register on page 3-86.](#)
- [3.9.7 Creating remapping rules for a control register on page 3-87.](#)
- [3.9.8 Creating a memory region for remapping by a control register on page 3-88.](#)

Related references

- [3.9 Scenario demonstrating how to create a new target configuration file on page 3-80.](#)
- [3.3 Target configuration editor - Memory tab on page 3-67.](#)

3.10 Creating a power domain for a target

Describes how to create a power domain configuration for your target.

Prerequisites

Before you create a power domain configuration, you must first create a control register.

Procedure

1. Click on the **Overview** tab.
2. Select **Supported** for the Power Domain setting.
3. Click on the **Configurations** tab.
4. Expand the **Power Domain Configurations** group.

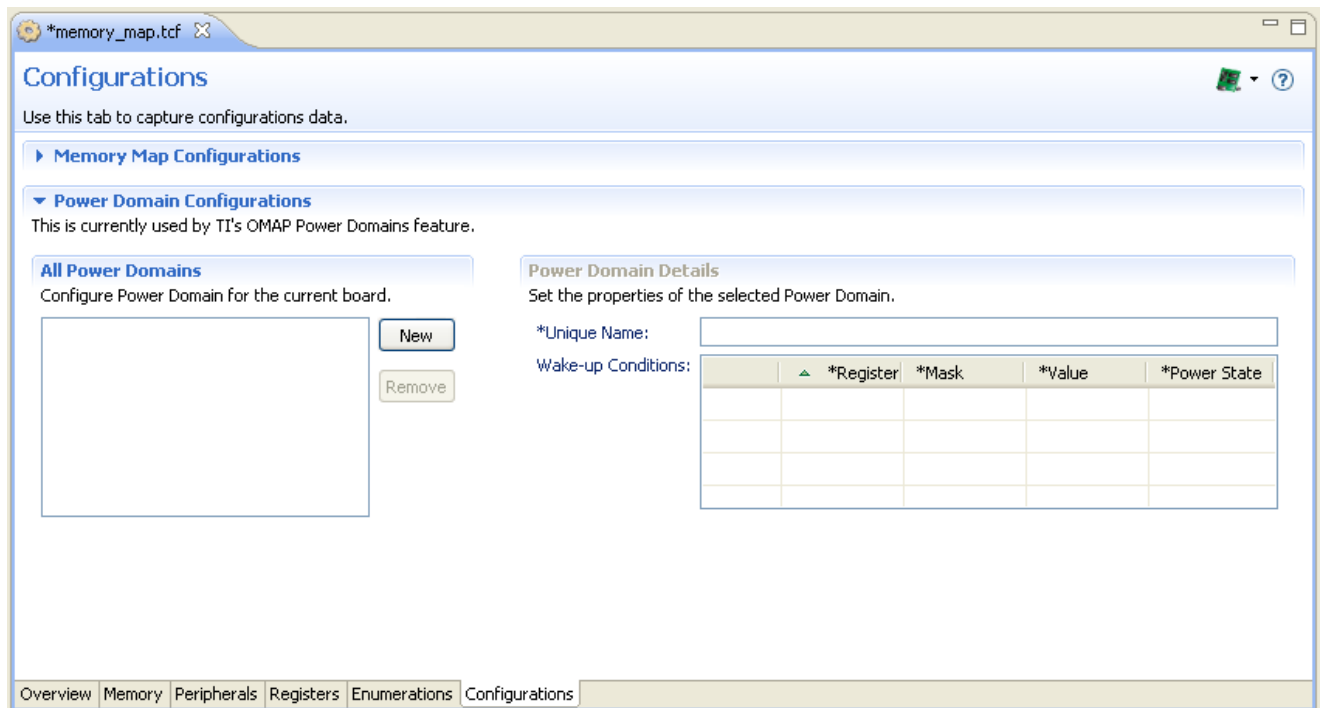


Figure 3-20 Power Domain Configurations

5. Click **New** to create a new power domain.
6. Enter a name in the **Unique Name** field.
7. Set the following **Wake-up Conditions** for the power domain:
 - **Register** - a list of registers you have previously created
 - **Mask**
 - **Value**
 - **Power State**.

All settings are mandatory.

Related tasks

[3.9.3 Creating a standalone register on page 3-83.](#)

Related references

3.2 Target configuration editor - Overview tab on page 3-65.

3.8 Target configuration editor - Configurations tab on page 3-78.

3.11 Creating a Group list

Describes how to create a new group list.

Procedure

1. Click on the **Group View** tab.
2. Click **Add a new group** in the **Group View List**.
3. Select the new group.

———— **Note** ————

You can create a subgroup by selecting a group and clicking **Add**.

4. Select peripherals and registers from the **Available Peripheral List**.
5. Press the << **Add** button to add the selected peripherals to the **Group View List**.
6. Click the **Save** icon in the toolbar.

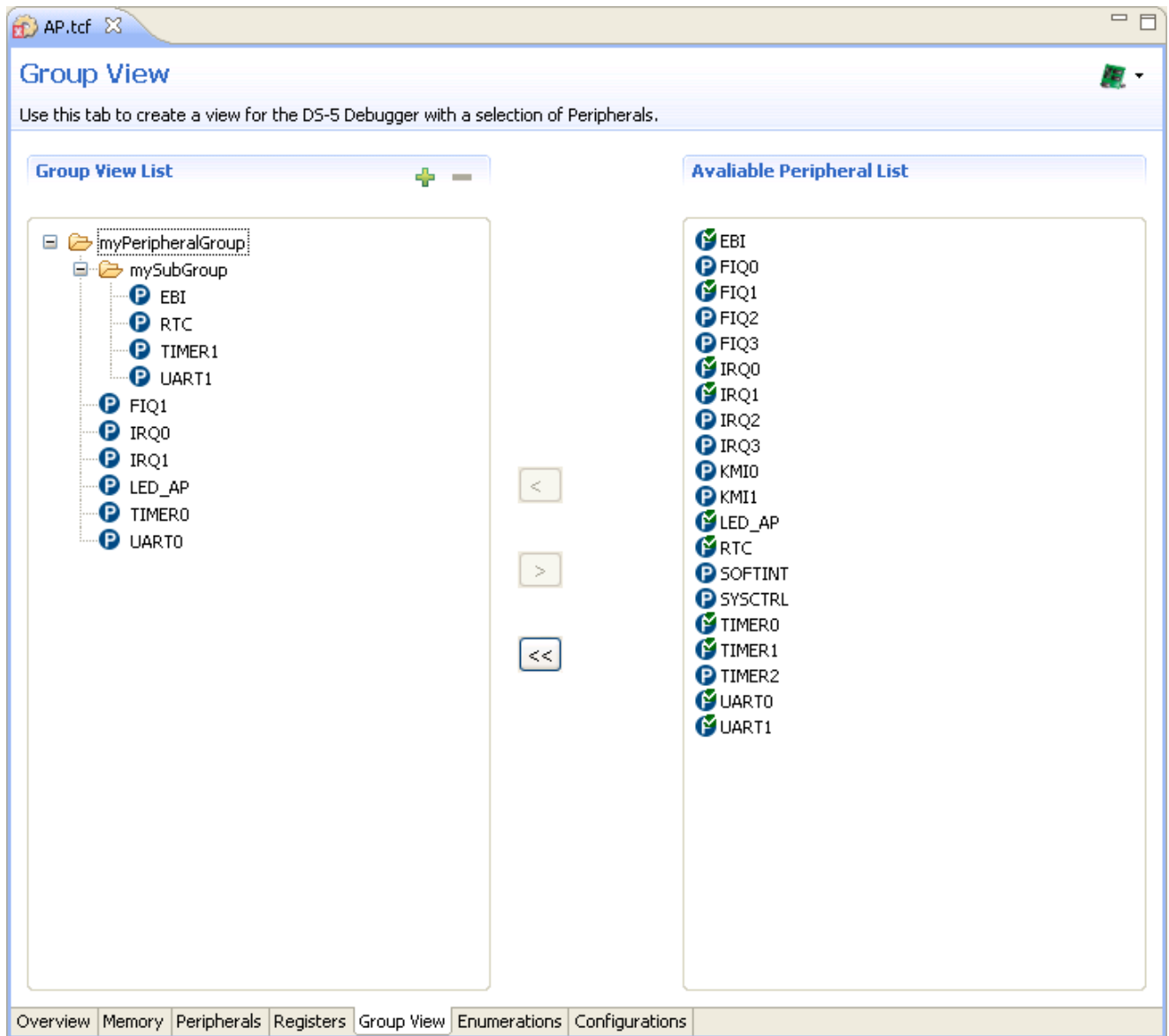


Figure 3-21 Creating a group list

Related references

[3.6 Target configuration editor - Group View tab on page 3-73.](#)

3.12 Importing an existing target configuration file

Describes how to import an existing target configuration file into the workspace.

Procedure

1. Select **Import** from the **File** menu.
2. Expand the **Target Configuration Editor** group.
3. Select the required file type.

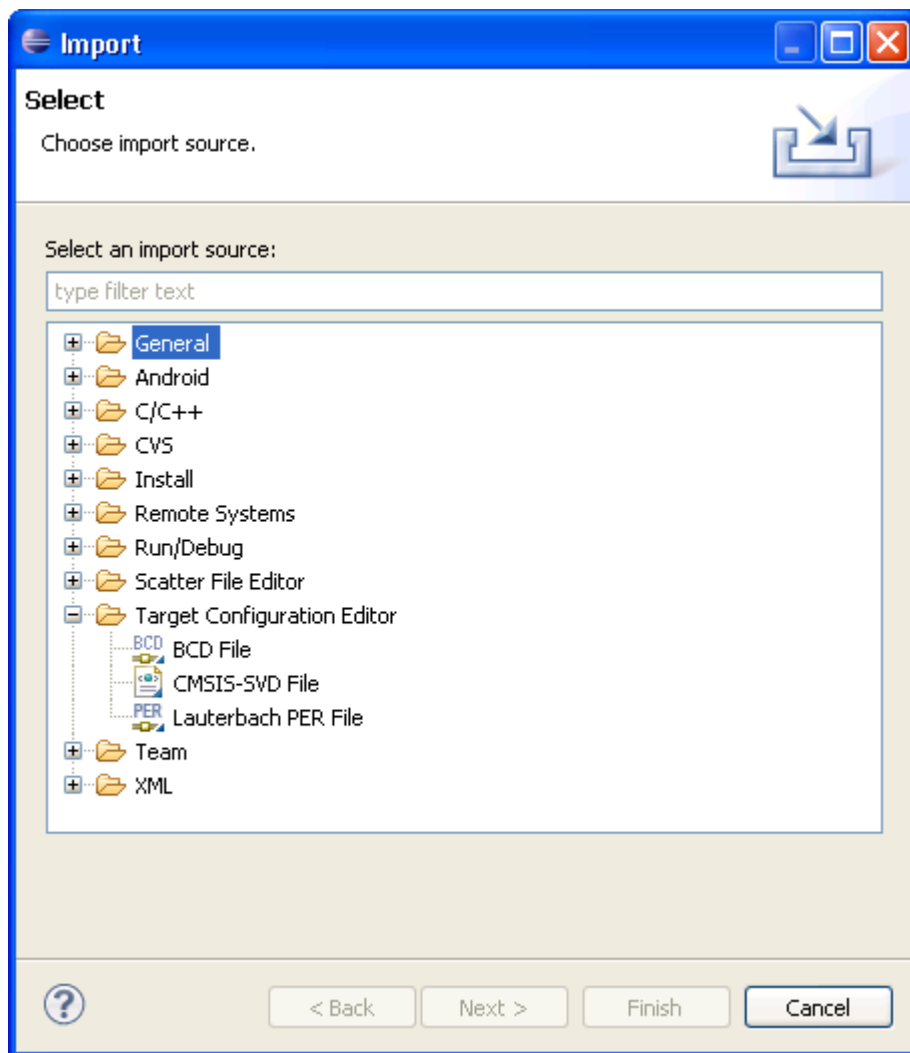


Figure 3-22 Selecting an existing target configuration file

4. Click on **Next**.
5. In the Import dialog box, click **Browse...** to select the folder containing the file.

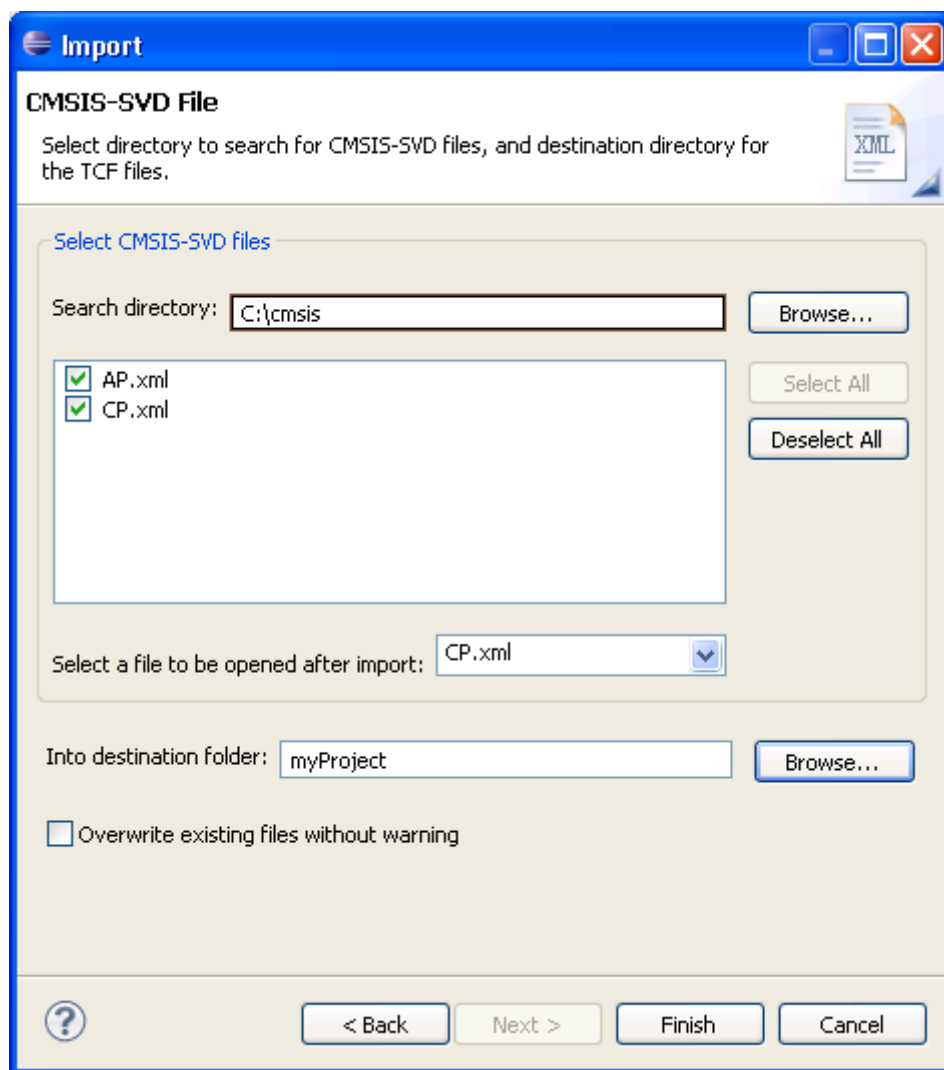


Figure 3-23 Importing the target configuration file

6. By default, all the files that can be imported are displayed. If the selection panel shows more than one file, select the files that you want to import.
7. Select the file that you want to automatically open in the editor.
8. In the Into destination folder field, click **Browse...** to select an existing project.
9. Click **Finish**.

The new *Target Configuration Files* (TCF) is visible in the **Project Explorer** view.

Related tasks

[3.13 Exporting a target configuration file on page 3-98.](#)

3.13 Exporting a target configuration file

Describes how to export a target configuration file from a project in the workspace to a C header file.

———— **Note** ————

Before using the export wizard, you must ensure that the Target Configuration File (TCF) is open in the editor view.

Procedure

1. Select **Export** from the **File** menu.
2. Expand the **Target Configuration Editor** group.
3. Select **C Header file**.

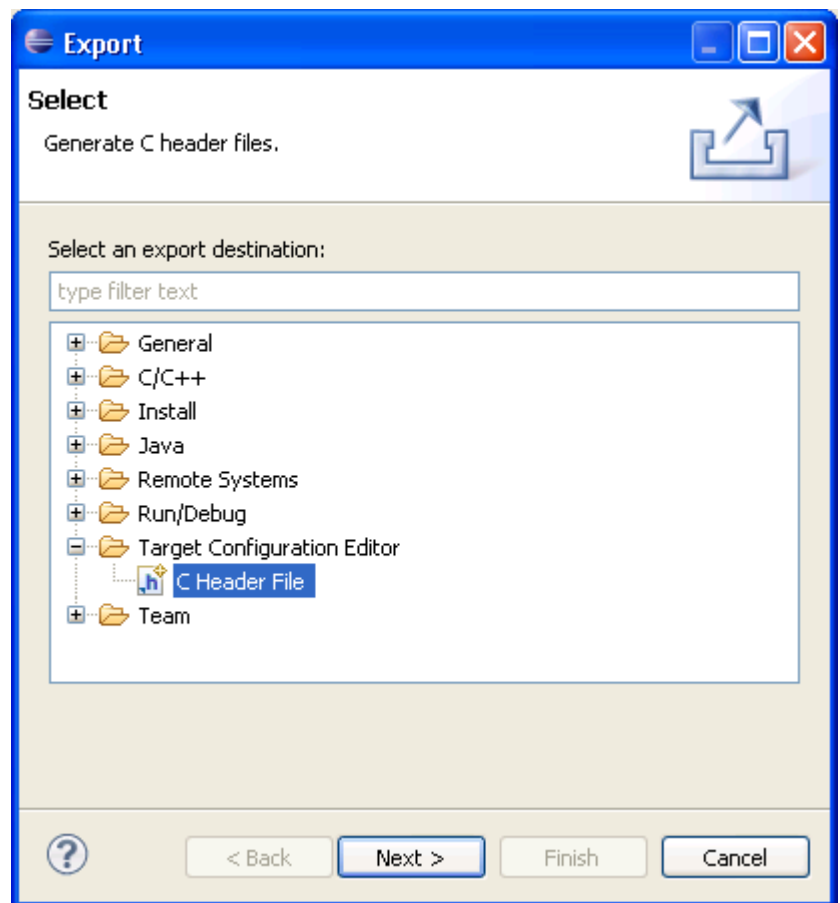


Figure 3-24 Exporting to C header file

4. Click on **Next**.
5. By default, the active files that are open in the editor are displayed. If the selection panel shows more than one file, select the files that you want to export.
6. Click **Browse...** to select a destination path.
7. If required, select **Overwrite existing files without warning**.
8. Click on **Next**.

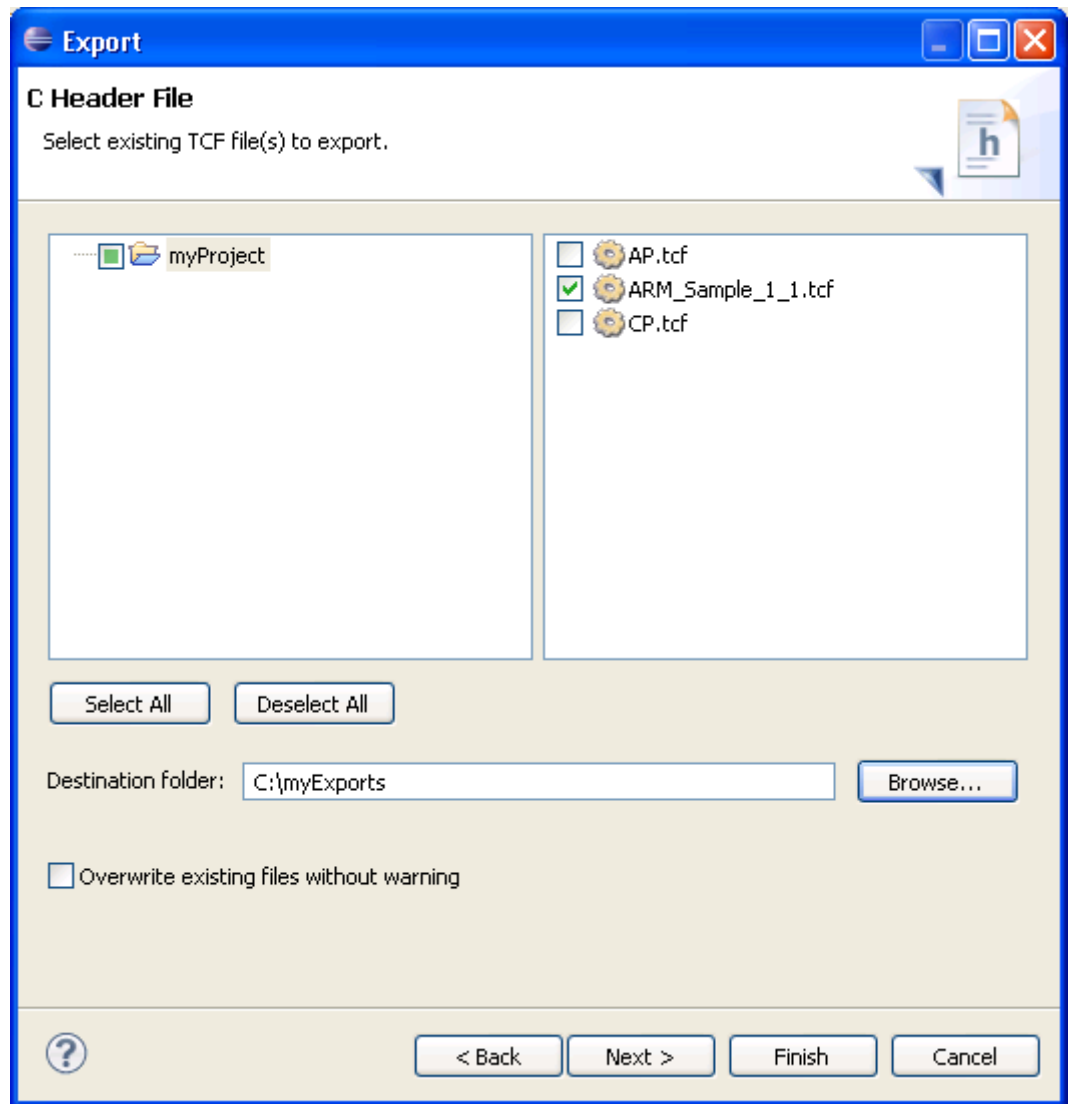


Figure 3-25 Selecting the files

9. If the TCF file has multiple boards, select the board that you want to configure the data for.
10. Select the data that you want to export.
11. Select required export options.
12. Click **Finish** to create the C header file.

Related tasks

[3.12 Importing an existing target configuration file on page 3-96.](#)

Chapter 4

Controlling execution

Describes how to stop the target execution when certain events occur, and when certain conditions are met.

It contains the following:

- *4.1 About loading an image on to the target on page 4-101.*
- *4.2 About loading debug information into the debugger on page 4-103.*
- *4.3 About passing arguments to main() on page 4-105.*
- *4.4 Running an image on page 4-106.*
- *4.5 Working with breakpoints and watchpoints on page 4-107.*
- *4.6 Working with conditional breakpoints on page 4-114.*
- *4.7 About pending breakpoints and watchpoints on page 4-118.*
- *4.8 Setting a tracepoint on page 4-119.*
- *4.9 Setting Streamline start and stop points on page 4-120.*
- *4.10 Stepping through an application on page 4-121.*
- *4.11 Handling Unix signals on page 4-123.*
- *4.12 Handling processor exceptions on page 4-125.*
- *4.13 Configuring the debugger path substitution rules on page 4-127.*

4.1 About loading an image on to the target

Before you can start debugging your application image, you must load the files on to the target. The files on your target must be the same as those on your local host workstation. The code layout must be identical, but the files on your target do require debug information.

You can manually load the files on to the target or you can configure a debugger connection to automatically do this after a connection is established. Some target connections do not support load operations and the relevant menu options are therefore disabled.

After connecting to the target you can also use the **Debug Control** view menu entry **Load...** to load files as required. The following options for loading an image are available:

Load Image Only

Loads the application image on to the target.

Load Image and Debug Info

Loads the application image on to the target and debug information from the same image into the debugger.

Load Offset

Specifies a decimal or hexadecimal offset that is added to all addresses within the image. A hexadecimal offset must be prefixed with 0x.

Enable on-demand loading

Specifies how you want the debugger to load debug information. Enabling this option can provide a faster load and use less memory but debugging might be slower.

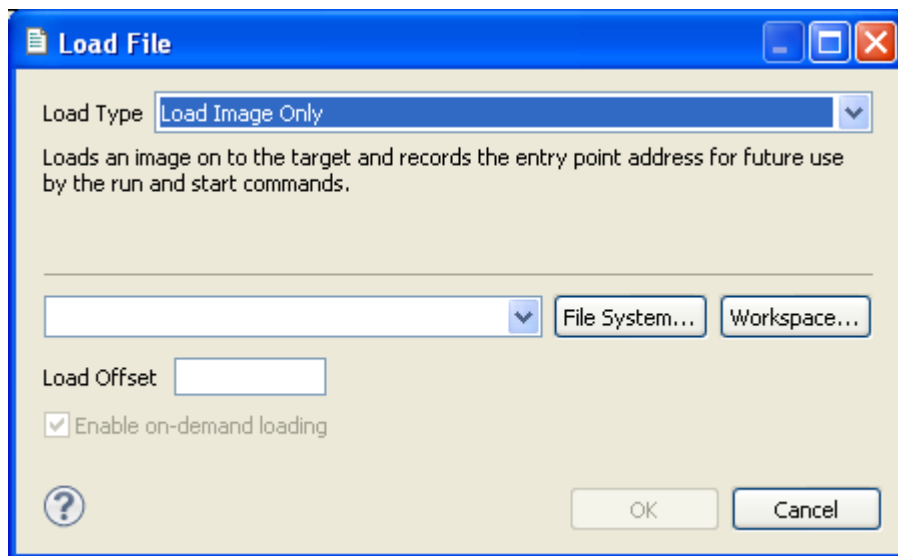


Figure 4-1 Load File dialog box

Related concepts

[4.2 About loading debug information into the debugger on page 4-103.](#)

Related tasks

[2.2 Configuring a connection to a Fixed Virtual Platform \(FVP\) on page 2-33.](#)

[2.3 Configuring a connection to a Linux target using gdbserver on page 2-35.](#)

[2.4 Configuring a connection to a Linux Kernel on page 2-37.](#)

[2.7 Configuring a connection to a bare-metal target on page 2-47.](#)

[2.8 Configuring an Event Viewer connection to a bare-metal target on page 2-49.](#)

Related references

10.6 Commands view on page 10-209.
10.7 Debug Control view on page 10-212.

Related information

DS-5 Debugger commands.

4.2 About loading debug information into the debugger

An executable image contains symbolic references, such as function and variable names, in addition to the application code and data. These symbolic references are generally referred to as debug information. Without this information the debugger is unable to debug at the source level.

To debug an application at source level, the image file and shared object files must be compiled with debug information, and a suitable level of optimization. For example, when compiling with either the ARM or the GNU compiler you can use the following options:

```
-g -O0
```

Debug information is not loaded when a file is loaded, but is a separate action. A typical load sequence is:

1. Load the main application image.
2. Load any shared objects.
3. Load the symbols for the main application image
4. Load the symbols for shared objects on-demand.

Images and shared objects might be preloaded onto the target, such as an image in a ROM device or an OS-aware target. The corresponding image file and any shared object files must contain debug information, and be accessible from your local host workstation. You can then configure a connection to the target loading only the debug information from these files. Use the **Load symbols from file** option on the debug configuration **Files** tab as appropriate for the target environment.

After connecting to the target you can also use the view menu entry **Load...** in the **Debug Control** view to load files as required. The following options for loading debug information are available:

Add Symbols File

Loads additional debug information into the debugger.

Load Debug Info

Loads debug information into the debugger.

Load Image and Debug Info

Loads the application image on to the target and debug information from the same images into the debugger.

Load Offset

Specifies a decimal or hexadecimal offset that is added to all addresses within the image. A hexadecimal offset must be prefixed with **0x**.

Enable on-demand loading

Specifies how you want the debugger to load debug information. Enabling this option can provide a faster load and use less memory but debugging might be slower.

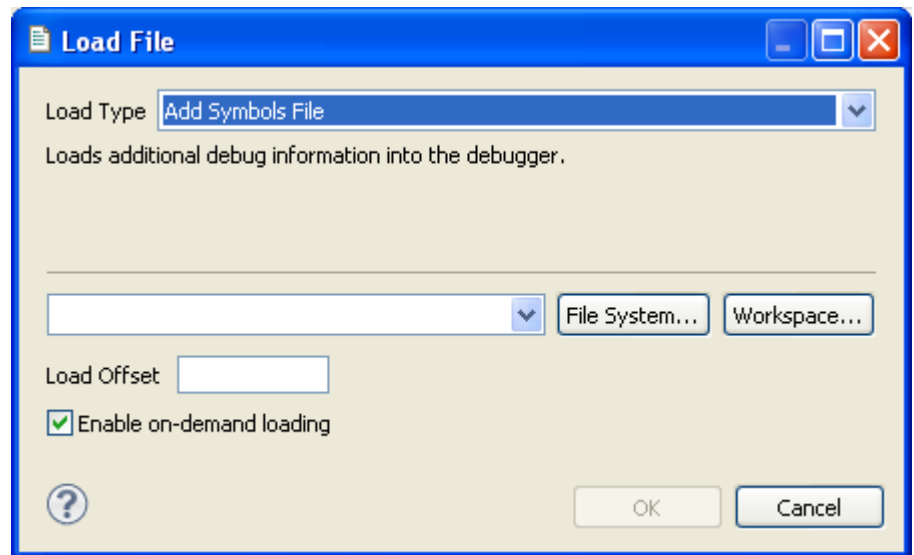


Figure 4-2 Load additional debug information dialog box

The debug information in an image or shared object also contains the path of the sources used to build it. When execution stops at an address in the image or shared object, the debugger attempts to open the corresponding source file. If this path is not present or the required source file is not found, then you must inform the debugger where the source file is located. You do this by setting up a substitution rule to associate the path obtained from the image with the path to the required source file that is accessible from your local host workstation.

Related concepts

4.1 About loading an image on to the target on page 4-101.

Related tasks

2.2 Configuring a connection to a Fixed Virtual Platform (FVP) on page 2-33.

2.3 Configuring a connection to a Linux target using gdbserver on page 2-35.

2.4 Configuring a connection to a Linux Kernel on page 2-37.

2.7 Configuring a connection to a bare-metal target on page 2-47.

2.8 Configuring an Event Viewer connection to a bare-metal target on page 2-49.

Related references

10.6 Commands view on page 10-209.

10.7 Debug Control view on page 10-212.

4.13 Configuring the debugger path substitution rules on page 4-127.

Related information

DS-5 Debugger commands.

4.3 About passing arguments to main()

ARM DS-5 Debugger enables you to pass arguments to the `main()` function of your application with one of the following methods:

- using the **Arguments** tab in the **Debug Configuration** dialog box.
- on the command-line (or in a script), you can use either:
 - `set semihosting args <arguments>`
 - `run <arguments>`.

Note

Semihosting must be active for these to work with bare-metal images.

Related references

[7.1 About semihosting and top of memory on page 7-162.](#)

[7.2 Working with semihosting on page 7-163.](#)

[7.3 Enabling automatic semihosting support in the debugger on page 7-164.](#)

[7.4 Controlling semihosting messages using the command-line console on page 7-165.](#)

[10.40 Debug Configurations - Arguments tab on page 10-287.](#)

Related information

[DS-5 Debugger commands.](#)

4.4 Running an image

Describes how to run an application image so that you can monitor how it executes on a target.

Use the Debug Configurations dialog box to set up a connection and define the run control options that you want the debugger to do after connection. To do this select **Debug Configurations...** from the **Run** menu.

After connection, you can control the debug session by using the toolbar icons in the **Debug Control** view.

Prerequisites

Before you can run an image it must be loaded onto the target. An image can either be preloaded on a target or loaded onto the target as part of the debug session.

————— **Note** —————

The files that resides on the target do not have to contain debug information, however, to be able to debug them you must have the corresponding files with debug information on your local host workstation.

Related references

10.6 Commands view on page 10-209.

10.7 Debug Control view on page 10-212.

10.36 Debug Configurations - Connection tab on page 10-276.

10.37 Debug Configurations - Files tab on page 10-279.

10.38 Debug Configurations - Debugger tab on page 10-283.

10.40 Debug Configurations - Arguments tab on page 10-287.

10.41 Debug Configurations - Environment tab on page 10-289.

Debug Configurations - Event Viewer tab.

Related information

DS-5 Debugger commands.

4.5 Working with breakpoints and watchpoints

Breakpoints and watchpoints enable you to stop the target when certain events occur, and when certain conditions are met. When execution stops, you can choose to examine the contents of memory, registers, or variables, or you specify other actions to be taken before resuming execution.

Breakpoints

A breakpoint enables you to interrupt your application when execution reaches a specific address. A breakpoint is always related to a particular memory address, regardless of what might be stored there. When execution reaches the breakpoint, normal execution stops before any instruction stored there is performed.

You can set:

- Software breakpoints that trigger when a particular instruction is executed at a specific address.
- Hardware breakpoints that trigger when the processor attempts to execute an instruction that is fetched from a specific memory address.
- Conditional breakpoints that trigger when an expression evaluates to true or when an ignore counter is reached.
- Temporary software or hardware breakpoints that are subsequently deleted when the breakpoint is hit.

Note

The type of breakpoints you can set depends on the:

- Memory region and the related access attributes.
 - Hardware support provided by your target processor.
 - Debug interface used to maintain the target connection.
 - Running state if you are debugging an OS-aware application.
-

Watchpoints

A watchpoint is similar to a breakpoint, but it is the address or value of a data access that is monitored rather than an instruction being executed from a specific address. You specify a register or a memory address to identify a location that is to have its contents tested. Watchpoints are sometimes known as data breakpoints, emphasizing that they are data dependent. Execution of your application stops when the address being monitored is accessed by your application.

You can set:

- Watchpoints that trigger when a particular memory location is accessed in a particular way.
- Conditional watchpoints that trigger when an expression evaluates to true or when an ignore counter is reached.

Note

- Depending on the target, it is possible that a few additional instructions, after the instruction that accessed the variable, may also be executed.
 - Watchpoints are only supported on scalar values.
 - The number of watchpoints that can be set at the same time depends on the target and the debug connection being used. Some targets do not support watchpoints.
-

Considerations when setting breakpoints and watchpoints

Be aware of the following when setting breakpoints and watchpoints:

- The number of hardware breakpoints available depends on the target.
- If an image is compiled with a high optimization level or perhaps contains C++ templates then the effect of setting a breakpoint in the source code depends on where you set the breakpoint. For example, if you set a breakpoint on an inlined function or a C++ template, then a breakpoint is created for each instance of that function or template. Therefore the target can run out of breakpoint resources.
- Enabling a *Memory Management Unit* (MMU) might set a memory region to read-only. If that memory region contains a software breakpoint, then that software breakpoint cannot be removed. Therefore, make sure you clear software breakpoints before enabling the MMU.
- Watchpoints are only supported on global/static data symbols because they are always in scope. Local variables are not available when you step out of a function.
- Some targets do not support watchpoints. Currently you can only use watchpoint commands on a hardware target using a debug hardware adapter.
- The address of the instruction that triggers the watchpoint might not be the address shown in the PC register. This is because of pipelining effects in the processor.
- When debugging an application that uses shared objects, breakpoints that are set within a shared object are re-evaluated when the shared object is unloaded. Those with addresses that can be resolved are set and the others remain pending.
- If a breakpoint is set by function name then only inline instances that have been already demand loaded are found. To find all the inline instances of a function you must disable on-demand loading.

It contains the following:

- [4.5.1 Setting or deleting an execution breakpoint on page 4-108.](#)
- [4.5.2 Setting or removing a data watchpoint on page 4-110.](#)
- [4.5.3 Viewing the properties of a data watchpoint on page 4-110.](#)
- [4.5.4 Importing DS-5™ breakpoint settings from a file on page 4-111.](#)
- [4.5.5 Exporting DS-5™ breakpoint settings to a file on page 4-112.](#)

4.5.1 Setting or deleting an execution breakpoint

The debugger enables you to set software or hardware breakpoints depending on your target memory type.

Software breakpoints are implemented by the debugger replacing the instruction at the breakpoint address with a special instruction opcode. Because the debugger requires write access to application memory, software breakpoints can only be set in RAM.

Hardware breakpoints are implemented by EmbeddedICE® logic that monitors the address and data buses of your processor. For simulated targets, hardware breakpoints are implemented by your simulator software.

Procedure

- To set an execution breakpoint, double-click in the left-hand marker bar of the C/C++ editor or the **Disassembly** view at the position where you want to set the breakpoint.
- To delete a breakpoint, double-click on the breakpoint marker.

The following figure shows how breakpoints are displayed in the C/C++ editor, the **Disassembly** view, and the **Breakpoints** view.

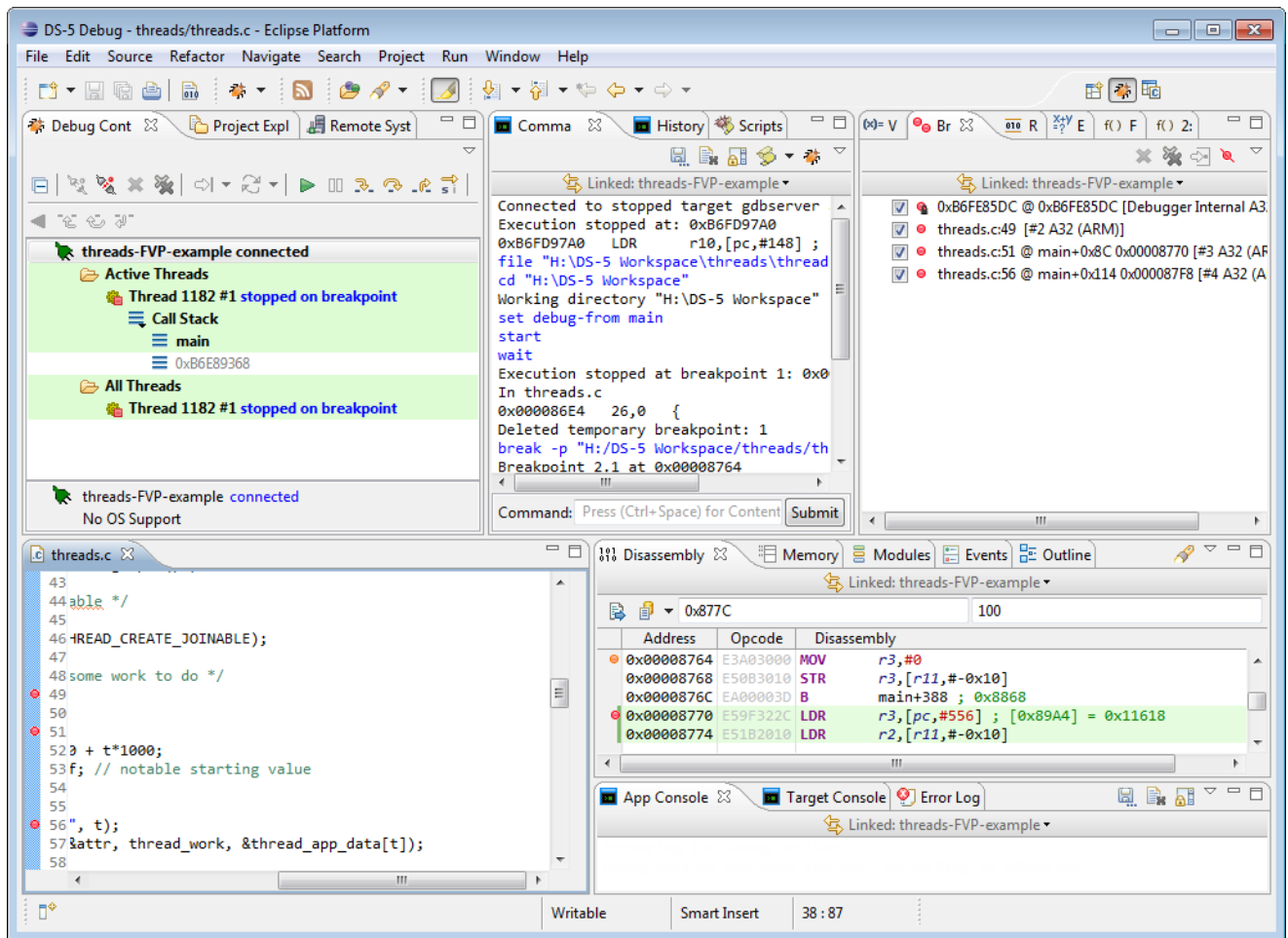


Figure 4-3 Setting an execution breakpoint

Related references

- [4.5 Working with breakpoints and watchpoints on page 4-107.](#)
- [4.5.3 Viewing the properties of a data watchpoint on page 4-110.](#)
- [Working with data watchpoints.](#)
- [4.8 Setting a tracepoint on page 4-119.](#)
- [4.9 Setting Streamline start and stop points on page 4-120.](#)
- [4.6 Working with conditional breakpoints on page 4-114.](#)
- [4.6.1 Assigning conditions to an existing breakpoint on page 4-114.](#)
- [4.7 About pending breakpoints and watchpoints on page 4-118.](#)
- [4.5.5 Exporting DS-5™ breakpoint settings to a file on page 4-112.](#)
- [4.5.4 Importing DS-5™ breakpoint settings from a file on page 4-111.](#)

Related information

[DS-5 Debugger commands.](#)

4.5.2 Setting or removing a data watchpoint

Like breakpoints, watchpoints can be used to stop the target. Watchpoints stop the target when a particular variable is accessed no matter which function is executing.

Procedure

1. To set a data watchpoint, in the **Variables** view, right-click on a data symbol and select **Toggle Watchpoint** to display the Add Watchpoint dialog.

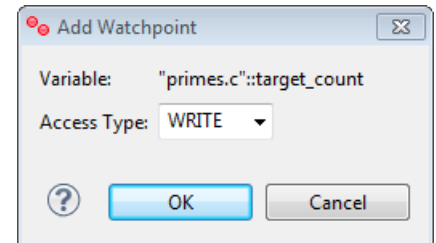


Figure 4-4 Setting a data watchpoint

2. Select the required **Access Type**, and then click **OK**. You can view the created watchpoint in the **Variables** view and also in the **Breakpoints** view. You can view the created watchpoint in the **Variables** view and also in the **Breakpoints** view.
 - To remove a data watchpoint, in the **Variables** view, right-click a watchpoint and select **Toggle Watchpoint**.

4.5.3 Viewing the properties of a data watchpoint

Once a data watchpoint is set, you can view its properties.

To view the properties of a data watchpoint, either:

Procedure

- In the **Variables** view, right-click a watchpoint and select **Watchpoint Properties**.
- In the **Breakpoints** view, right-click a watchpoint and select **Properties...**

This displays the Watchpoint Properties dialog:

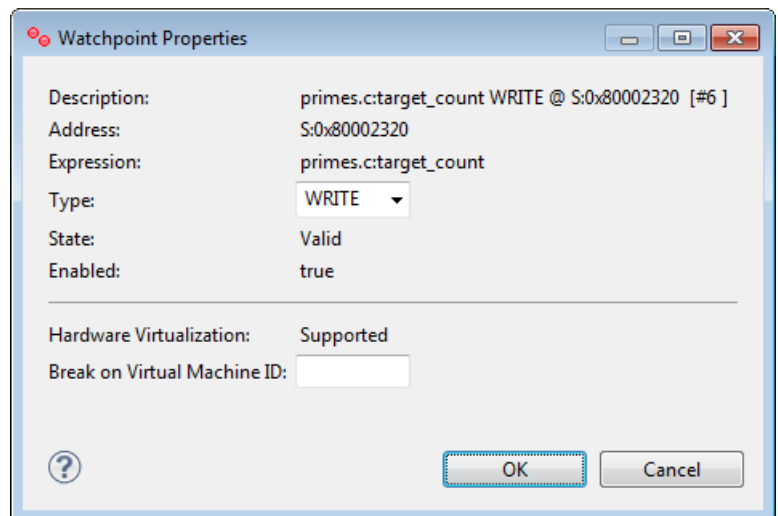


Figure 4-5 Viewing the properties of a data watchpoint

- Use the options available in the **Type** options to change the watchpoint type.

- If the target supports virtualization, you can use the Break on Virtual Machine ID field to enter a virtual machine ID. This allows the watchpoint to stop only at the virtual machine ID you specify.

Related references

[4.5 Working with breakpoints and watchpoints on page 4-107.](#)
[4.5.1 Setting or deleting an execution breakpoint on page 4-108.](#)
[Working with data watchpoints.](#)
[4.8 Setting a tracepoint on page 4-119.](#)
[4.9 Setting Streamline start and stop points on page 4-120.](#)
[4.6 Working with conditional breakpoints on page 4-114.](#)
[4.6.1 Assigning conditions to an existing breakpoint on page 4-114.](#)
[4.7 About pending breakpoints and watchpoints on page 4-118.](#)
[4.5.5 Exporting DS-5™ breakpoint settings to a file on page 4-112.](#)
[4.5.4 Importing DS-5™ breakpoint settings from a file on page 4-111.](#)

Related information

[DS-5 Debugger commands.](#)

4.5.4 Importing DS-5™ breakpoint settings from a file

You can import DS-5 breakpoints and watchpoints using the Breakpoints view. This makes it possible to work with breakpoints and watchpoints created in a different workspace.

To import breakpoint settings from a file:

Procedure

1. In the DS-5 debug perspective, select **Import Breakpoints** from the Breakpoints view menu.
2. In the Open window which appears, browse and select the file which contains the breakpoints settings.
3. Click **Open**.

Note

Existing breakpoints and watchpoints settings for the current connection is deleted and replaced by the settings from the imported file.

Related references

[4.5 Working with breakpoints and watchpoints on page 4-107.](#)
[4.5.1 Setting or deleting an execution breakpoint on page 4-108.](#)
[4.5.3 Viewing the properties of a data watchpoint on page 4-110.](#)
[Working with data watchpoints.](#)
[4.8 Setting a tracepoint on page 4-119.](#)
[4.9 Setting Streamline start and stop points on page 4-120.](#)
[4.6 Working with conditional breakpoints on page 4-114.](#)
[4.6.1 Assigning conditions to an existing breakpoint on page 4-114.](#)
[4.7 About pending breakpoints and watchpoints on page 4-118.](#)
[4.5.5 Exporting DS-5™ breakpoint settings to a file on page 4-112.](#)
[10.3 ARM assembler editor on page 10-199.](#)
[10.4 Breakpoints view on page 10-202.](#)
[10.5 C/C++ editor on page 10-206.](#)
[10.6 Commands view on page 10-209.](#)

[10.8 Disassembly view on page 10-216.](#)
[10.10 Expressions view on page 10-221.](#)
[10.13 Memory view on page 10-228.](#)
[10.15 Registers view on page 10-236.](#)
[10.23 Variables view on page 10-255.](#)

Related information

[DS-5 Debugger commands.](#)

4.5.5 Exporting DS-5™ breakpoint settings to a file

You can export DS-5 breakpoints and watchpoints from the Breakpoints view. This makes it possible to export your current breakpoints and watchpoints to a different workspace.

To export the breakpoint settings to a file:

Procedure

1. In the DS-5 debug perspective, select **Export Breakpoints** from the Breakpoints view menu.
2. In the Save As window which appears, enter a filename and browse and select the location where you want to save the file.
3. Click **Save**.

Note

All breakpoints and watchpoints shown in the Breakpoints view are saved.

Related references

[4.5 Working with breakpoints and watchpoints on page 4-107.](#)
[4.5.1 Setting or deleting an execution breakpoint on page 4-108.](#)
[4.5.3 Viewing the properties of a data watchpoint on page 4-110.](#)
[Working with data watchpoints.](#)
[4.8 Setting a tracepoint on page 4-119.](#)
[4.9 Setting Streamline start and stop points on page 4-120.](#)
[4.6 Working with conditional breakpoints on page 4-114.](#)
[4.6.1 Assigning conditions to an existing breakpoint on page 4-114.](#)
[4.7 About pending breakpoints and watchpoints on page 4-118.](#)
[4.5.4 Importing DS-5™ breakpoint settings from a file on page 4-111.](#)
[10.3 ARM assembler editor on page 10-199.](#)
[10.4 Breakpoints view on page 10-202.](#)
[10.5 C/C++ editor on page 10-206.](#)
[10.6 Commands view on page 10-209.](#)
[10.8 Disassembly view on page 10-216.](#)
[10.10 Expressions view on page 10-221.](#)
[10.13 Memory view on page 10-228.](#)
[10.15 Registers view on page 10-236.](#)
[10.23 Variables view on page 10-255.](#)

Related information

[DS-5 Debugger commands.](#)

Related references

[4.5.1 Setting or deleting an execution breakpoint on page 4-108.](#)

4.5.3 Viewing the properties of a data watchpoint on page 4-110.

Working with data watchpoints.

4.8 Setting a tracepoint on page 4-119.

4.9 Setting Streamline start and stop points on page 4-120.

4.6 Working with conditional breakpoints on page 4-114.

4.6.1 Assigning conditions to an existing breakpoint on page 4-114.

4.7 About pending breakpoints and watchpoints on page 4-118.

4.5.5 Exporting DS-5™ breakpoint settings to a file on page 4-112.

4.5.4 Importing DS-5™ breakpoint settings from a file on page 4-111.

Related information

DS-5 Debugger commands.

4.6 Working with conditional breakpoints

Conditional breakpoints have properties assigned to test for conditions that must be satisfied to trigger the breakpoint.

For example, using conditional breakpoints, you can:

- Test a variable for a given value.
- Execute a function a set number of times.
- Trigger a breakpoint only on a specific thread or processor.

During execution, the specified condition is checked and if it evaluates to true, then the target remains in the stopped state, otherwise execution resumes.

Note

- Conditional breakpoints can be very intrusive and lower the performance if they are hit frequently since the debugger stops the target every time the breakpoint triggers.
 - You must not assign a script to a breakpoint that has sub-breakpoints. If you do, the debugger attempts to execute the script for each sub-breakpoint. If this occurs, an error message is displayed.
-

Breakpoints that are set on a single line of source code with multiple statements are assigned as sub-breakpoints to a parent breakpoint. You can enable, disable, and view the properties of each sub-breakpoint in the same way as a single statement breakpoint. Conditions are assigned to top level breakpoints only and therefore affect both the parent breakpoint and sub-breakpoints.

Considerations when setting multiple conditions on a breakpoint

Be aware of the following when setting multiple conditions on a breakpoint:

- If you set a Stop Condition and an Ignore Count, then the Ignore Count is not decremented until the Stop Condition is met. For example, you might have a breakpoint in a loop that is controlled by the variable `c` and has 10 iterations. If you set the Stop Condition `c==5` and the Ignore Count to 3, then the breakpoint might not activate until it has been hit with `c==5` for the fourth time. It subsequently activates every time it is hit with `c==5`.
- If you choose to break on a selected thread or processor, then the Stop Condition and Ignore Count are checked only for the selected thread or processor.
- Conditions are evaluated in the following order:
 1. Thread or processor.
 2. Condition.
 3. Ignore count.

It contains the following:

- [4.6.1 Assigning conditions to an existing breakpoint on page 4-114.](#)

4.6.1 Assigning conditions to an existing breakpoint

Using the options available on the Breakpoint Properties dialog, you can specify different conditions for a specific breakpoint.

For example, you can set a breakpoint to be applicable to only specific threads or processors, schedule to run a script when a selected breakpoint is triggered, delay hitting a breakpoint, or specify a conditional expression for a specific breakpoint.

Procedure

1. In the **Breakpoints** view, select the breakpoint that you want to modify and right-click to display the context menu.
2. Select **Properties...** to display the Breakpoint Properties dialog box.

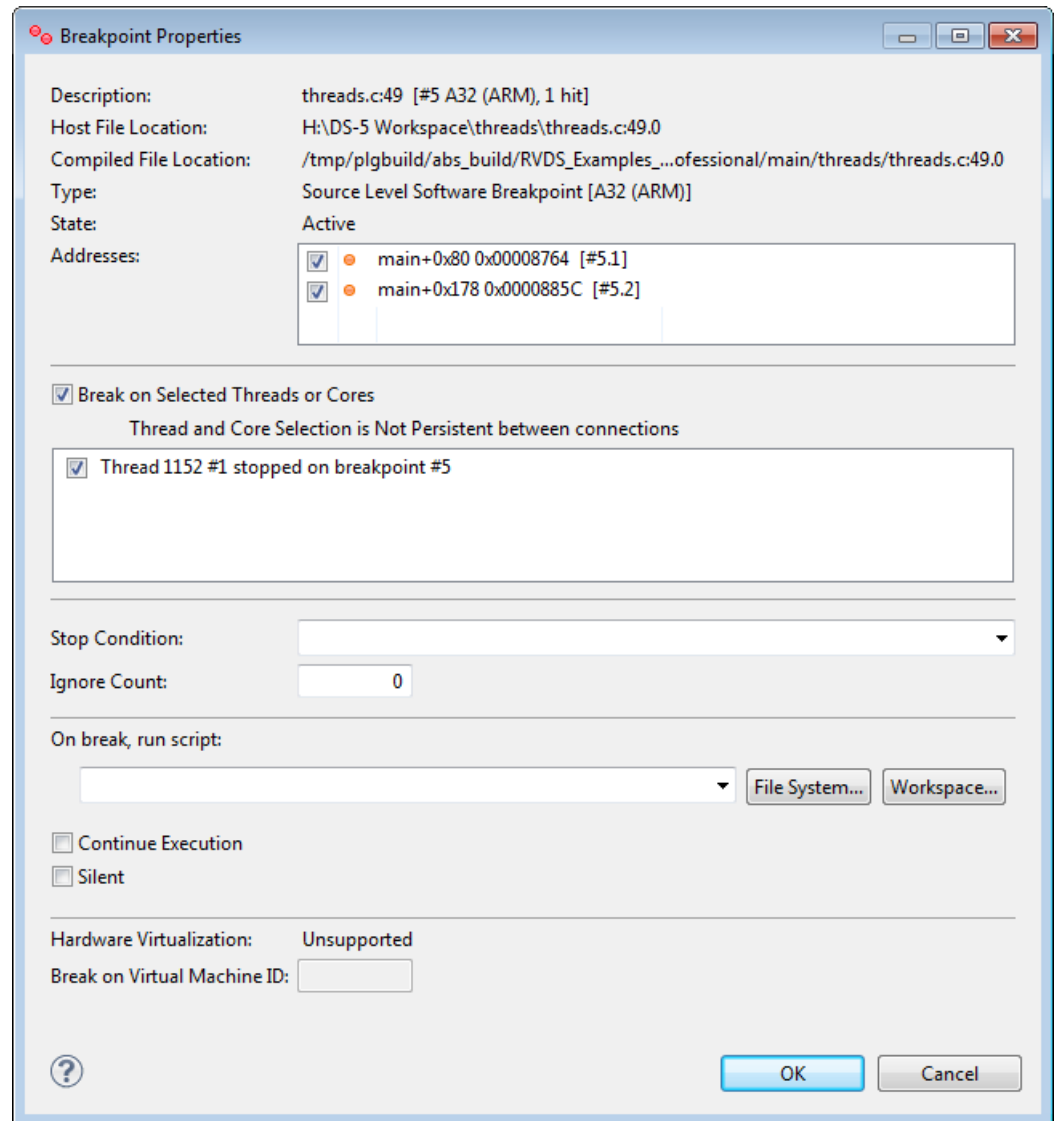


Figure 4-6 Breakpoint Properties dialog

3. Breakpoints apply to all threads by default, but you can modify the properties for a breakpoint to restrict it to a specific thread.
 - a) Select the **Break on Selected Threads** option to view and select individual threads.
 - b) Select the checkbox for each thread that you want to assign the breakpoint to.

Note

If you set a breakpoint for a specific thread, then any conditions you set for the breakpoint are checked only for that thread.

4. If you want to set a conditional expression for a specific breakpoint, then:
 - a) In the Stop Condition field, enter a C-style expression. For example, if your application has a variable `x`, then you can specify: `x == 10`

5. If you want the debugger to delay hitting the breakpoint until a specific number of passes has occurred, then:
 - a) In the **Ignore Count** field, enter the number of passes. For example, if you have a loop that performs 100 iterations, and you want a breakpoint in that loop to be hit after 50 passes, then enter **50**.
 6. If you want to run a script when the selected breakpoint is triggered, then:
 - a) In the On break, run script field, specify the script file.
Click **File System...** to locate the file in an external directory from the workspace or click **Workspace...** to locate the file within the workspace.
- **Note** ————
- Take care with commands used in a script file that is attached to a breakpoint. For example, if the script file contains the `quit` command, the debugger disconnects from the target when the breakpoint is hit.
- b) Select **Continue Execution** if you want to enable the debugger to automatically continue running the application on completion of all the breakpoint actions. Alternatively, you can enter the `continue` command as the last command in a script file, that is attached to a breakpoint.
 7. Once you have selected the required options, click **OK** to save your changes.

Related references

[4.5 Working with breakpoints and watchpoints on page 4-107.](#)
[4.5.1 Setting or deleting an execution breakpoint on page 4-108.](#)
[4.5.3 Viewing the properties of a data watchpoint on page 4-110.](#)
[Working with data watchpoints.](#)
[4.8 Setting a tracepoint on page 4-119.](#)
[4.9 Setting Streamline start and stop points on page 4-120.](#)
[4.6 Working with conditional breakpoints on page 4-114.](#)
[4.7 About pending breakpoints and watchpoints on page 4-118.](#)
[4.5.5 Exporting DS-5™ breakpoint settings to a file on page 4-112.](#)
[4.5.4 Importing DS-5™ breakpoint settings from a file on page 4-111.](#)
[10.3 ARM assembler editor on page 10-199.](#)
[10.4 Breakpoints view on page 10-202.](#)
[10.5 C/C++ editor on page 10-206.](#)
[10.6 Commands view on page 10-209.](#)
[10.8 Disassembly view on page 10-216.](#)
[10.10 Expressions view on page 10-221.](#)
[10.13 Memory view on page 10-228.](#)
[10.15 Registers view on page 10-236.](#)
[10.23 Variables view on page 10-255.](#)

Related information

[DS-5 Debugger commands.](#)

Related references

[4.5 Working with breakpoints and watchpoints on page 4-107.](#)
[4.5.1 Setting or deleting an execution breakpoint on page 4-108.](#)
[4.5.3 Viewing the properties of a data watchpoint on page 4-110.](#)
[Working with data watchpoints.](#)

- 4.8 Setting a tracepoint on page 4-119.*
- 4.9 Setting Streamline start and stop points on page 4-120.*
- 4.6.1 Assigning conditions to an existing breakpoint on page 4-114.*
- 4.7 About pending breakpoints and watchpoints on page 4-118.*
- 4.5.5 Exporting DS-5™ breakpoint settings to a file on page 4-112.*
- 4.5.4 Importing DS-5™ breakpoint settings from a file on page 4-111.*
- 10.3 ARM assembler editor on page 10-199.*
- 10.4 Breakpoints view on page 10-202.*
- 10.5 C/C++ editor on page 10-206.*
- 10.6 Commands view on page 10-209.*
- 10.8 Disassembly view on page 10-216.*
- 10.10 Expressions view on page 10-221.*
- 10.13 Memory view on page 10-228.*
- 10.15 Registers view on page 10-236.*
- 10.23 Variables view on page 10-255.*

Related information

DS-5 Debugger commands.

4.7 About pending breakpoints and watchpoints

Breakpoints and watchpoints are typically set when debug information is available. Pending breakpoints and watchpoints however, enable you to set breakpoints and watchpoints before the associated debug information is available.

The debugger automatically re-evaluates all pending breakpoints and watchpoints when debug information changes. Those with addresses that can be resolved are set and the others remain pending.

In the **Breakpoints** view, you can force the resolution of a pending breakpoint or watchpoint. For example, this might be useful if you have manually modified the shared library search paths. To do this:

1. Right-click on the pending breakpoint or watchpoint that you want to resolve.
2. Click on **Resolve** to attempt to find the address and set the breakpoint or watchpoint.

Examples

To manually set a pendable breakpoint or watchpoint you can use the `-p` option with any of these commands, `advance`, `awatch`, `break`, `hbreak`, `rwatch`, `tbreak`, `thbreak`, and `watch`. You can enter debugger commands in the **Commands** view.

```
break -p lib.c:20      # Sets a pending breakpoint at line 20 in lib.c
awatch -p *0x80D4      # Sets a pending read/write watchpoint on address 0x80D4
```

Related references

[4.5 Working with breakpoints and watchpoints on page 4-107.](#)
[4.5.1 Setting or deleting an execution breakpoint on page 4-108.](#)
[4.5.3 Viewing the properties of a data watchpoint on page 4-110.](#)
[Working with data watchpoints.](#)
[4.8 Setting a tracepoint on page 4-119.](#)
[4.9 Setting Streamline start and stop points on page 4-120.](#)
[4.6 Working with conditional breakpoints on page 4-114.](#)
[4.6.1 Assigning conditions to an existing breakpoint on page 4-114.](#)
[4.5.5 Exporting DS-5™ breakpoint settings to a file on page 4-112.](#)
[4.5.4 Importing DS-5™ breakpoint settings from a file on page 4-111.](#)
[10.3 ARM assembler editor on page 10-199.](#)
[10.4 Breakpoints view on page 10-202.](#)
[10.5 C/C++ editor on page 10-206.](#)
[10.6 Commands view on page 10-209.](#)
[10.8 Disassembly view on page 10-216.](#)
[10.10 Expressions view on page 10-221.](#)
[10.13 Memory view on page 10-228.](#)
[10.15 Registers view on page 10-236.](#)
[10.23 Variables view on page 10-255.](#)

Related information

[DS-5 Debugger commands.](#)

4.8 Setting a tracepoint

Tracepoints are memory locations that are used to trigger behavior in a trace capture device when running an application. A tracepoint is hit when the processor executes an instruction at a specific address. Depending on the type, trace capture is either enabled or disabled.

Tracepoints can be set from the following:

- ARM Assembler editor.
- C/C++ editor.
- Disassembly view.
- Functions view.
- Memory view.
- Disassembly panel of the Trace view.

To set a tracepoint, right-click in the left-hand marker bar at the position where you want to set the tracepoint and select either **Toggle Trace Start Point**, **Toggle Trace Stop Point**, or **Toggle Trace Trigger Point** from the context menu. To remove a tracepoint, repeat this procedure on the same tracepoint or delete it from the **Breakpoints** view.

Tracepoints are stored on a per connection basis. If the active connection is disconnected then tracepoints can only be created from the source editor.

All tracepoints are visible in the **Breakpoints** view.

Related references

4.5 Working with breakpoints and watchpoints on page 4-107.
4.5.1 Setting or deleting an execution breakpoint on page 4-108.
4.5.3 Viewing the properties of a data watchpoint on page 4-110.
Working with data watchpoints.
4.9 Setting Streamline start and stop points on page 4-120.
4.6 Working with conditional breakpoints on page 4-114.
4.6.1 Assigning conditions to an existing breakpoint on page 4-114.
4.7 About pending breakpoints and watchpoints on page 4-118.
4.5.5 Exporting DS-5™ breakpoint settings to a file on page 4-112.
4.5.4 Importing DS-5™ breakpoint settings from a file on page 4-111.
10.3 ARM assembler editor on page 10-199.
10.4 Breakpoints view on page 10-202.
10.5 C/C++ editor on page 10-206.
10.6 Commands view on page 10-209.
10.8 Disassembly view on page 10-216.
10.10 Expressions view on page 10-221.
10.13 Memory view on page 10-228.
10.15 Registers view on page 10-236.
10.23 Variables view on page 10-255.

Related information

DS-5 Debugger commands.

4.9 Setting Streamline start and stop points

Streamline start and stop points are locations in source or assembly code that are used to enable or disable Streamline capture in a running application. A Streamline start and stop point is hit when the processor executes an instruction at a specific address.

Streamline start and stop points can be set from the following views:

- ARM Assembler editor.
- C/C++ editor.

To set a Streamline start and stop point, right-click in the left-hand marker bar at the position where you want to set the start and stop point and select either **Toggle Streamline Start** or **Toggle Streamline Stop** from the **DS-5 Breakpoints** context menu. To remove a start and stop point, repeat this procedure on the same start and stop point.

Related references

4.5 Working with breakpoints and watchpoints on page 4-107.
4.5.1 Setting or deleting an execution breakpoint on page 4-108.
4.5.3 Viewing the properties of a data watchpoint on page 4-110.
Working with data watchpoints.
4.8 Setting a tracepoint on page 4-119.
4.6 Working with conditional breakpoints on page 4-114.
4.6.1 Assigning conditions to an existing breakpoint on page 4-114.
4.7 About pending breakpoints and watchpoints on page 4-118.
4.5.5 Exporting DS-5™ breakpoint settings to a file on page 4-112.
4.5.4 Importing DS-5™ breakpoint settings from a file on page 4-111.
10.3 ARM assembler editor on page 10-199.
10.5 C/C++ editor on page 10-206.

Related information

DS-5 Debugger commands.
Using ARM Streamline.

4.10 Stepping through an application

The debugger enables you to finely control the execution of an image by sequentially stepping through an application at the source level or the instruction level.

———— Note ————

You must compile your code with debug information to use the source level stepping commands. By default, source level calls to functions with no debug information are stepped over. Use the `set step-mode` command to change the default setting.

There are several ways to step through an application. You can choose to step:

- Into or over all function calls.
- At source level or instruction level.
- Through multiple statements in a single line of source code, for example a `for` loop.

Be aware that when stepping at the source level, the debugger uses temporary breakpoints to stop execution at the specified location. These temporary breakpoints might require the use of hardware breakpoints, especially when stepping through code in ROM or Flash. If there are not enough hardware breakpoint resources available, then the debugger displays an error message.

You can use the stepping toolbar in the **Debug Control** view to step through the application either by source line or instruction.

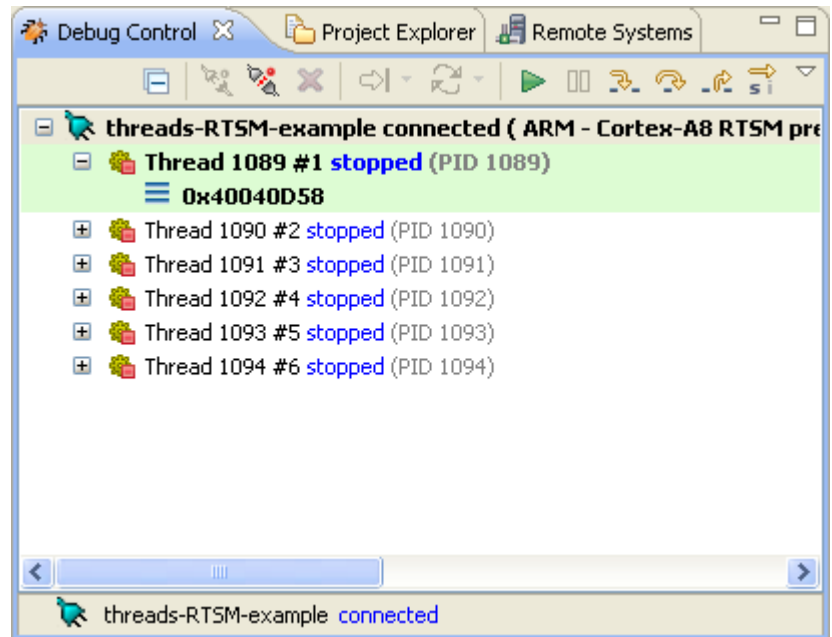


Figure 4-7 Debug Control view

Examples

To step a specified number of times you must use the **Commands** view to manually execute one of the stepping commands with a number. For example:

```
steps 5          # Execute five source statements
stepli 5         # Execute five instructions
```

Related concepts

- 6.8 About debugging shared libraries on page 6-146.*
- 6.9 About debugging a Linux kernel on page 6-148.*
- 6.10 About debugging Linux kernel modules on page 6-150.*

Related references

- 5.1 Examining the target execution environment on page 5-130.*
- 5.2 Examining the call stack on page 5-132.*
- 4.11 Handling Unix signals on page 4-123.*
- 4.12 Handling processor exceptions on page 4-125.*

4.11 Handling Unix signals

For Linux applications, ARM processors have the facility to trap Unix signals. These are managed in the debugger by selecting **Manage Signals** from the **Breakpoints** view menu or you can use the `handle` command. You can also use the `info signals` command to display the current handler settings.

The default handler settings are dependent on the type of debug activity. For example, by default on a Linux kernel connection, all signals are handled by Linux on the target.

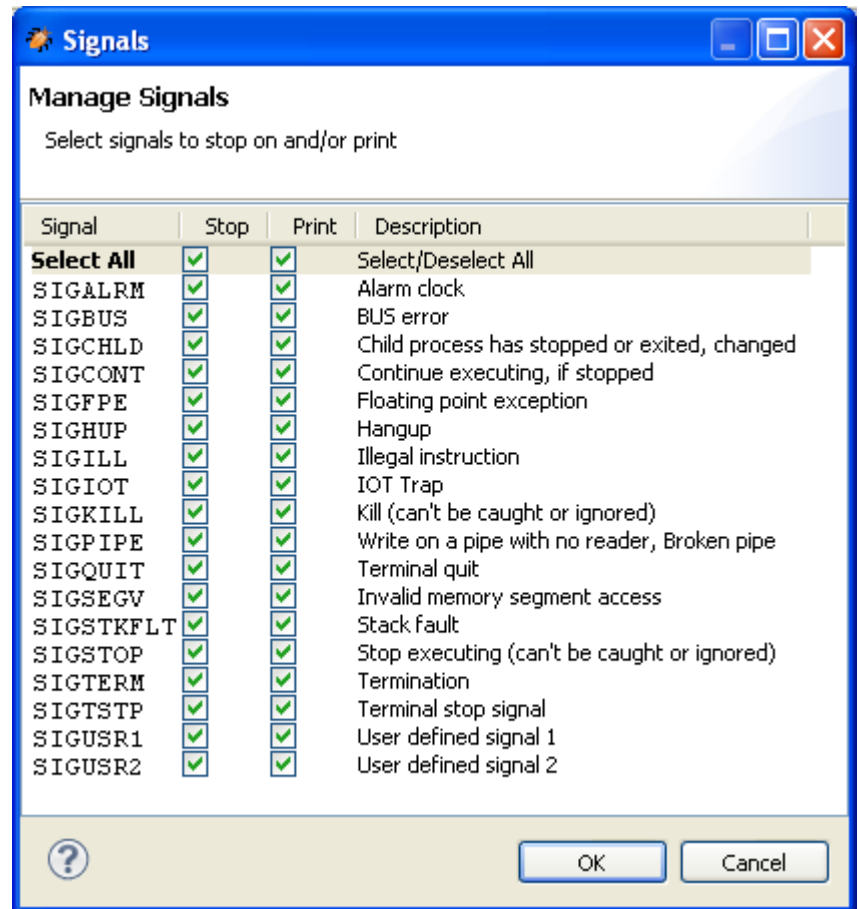


Figure 4-8 Managing signal handler settings

Note

Unix signals `SIGINT` and `SIGTRAP` cannot be debugged in the same way as other signals because they are used internally by the debugger for asynchronous stopping of the process and breakpoints respectively.

Examples

If you want the application to ignore a signal but log the event when it is triggered then you must enable stopping on a signal. In the following example, a `SIGHUP` signal occurs causing the debugger to stop and print a message. No signal handler is invoked when using this setting and the application being debugged ignores the signal and continues.

Ignoring a SIGHUP signal

```
handle SIGHUP stop print      # Enable stop and print on SIGHUP
signal
```

The following example shows how to debug a signal handler. To do this you must disable stopping on a signal and then set a breakpoint in the signal handler. This is because if stopping on a signal is disabled then the handling of that signal is performed by the process that passes signal to the registered handler. If no handler is registered then the default handler runs and the application generally exits.

Debugging a SIGHUP signal

```
handle SIGHUP nostop noprint  # Disable stop and print on SIGHUP
signal
```

Related concepts

- [6.8 About debugging shared libraries on page 6-146.](#)
- [6.9 About debugging a Linux kernel on page 6-148.](#)
- [6.10 About debugging Linux kernel modules on page 6-150.](#)

Related references

- [4.10 Stepping through an application on page 4-121.](#)
- [5.1 Examining the target execution environment on page 5-130.](#)
- [5.2 Examining the call stack on page 5-132.](#)
- [4.12 Handling processor exceptions on page 4-125.](#)
- [10.4 Breakpoints view on page 10-202.](#)
- [10.6 Commands view on page 10-209.](#)
- [10.32 Manage Signals dialog box on page 10-271.](#)

Related information

- [DS-5 Debugger commands.](#)

4.12 Handling processor exceptions

ARM processors handle exceptional events by jumping to one of a set of fixed addresses known as exception vectors. Except for a *Supervisor Call* (SVC), these events are not part of normal program flow and can happen unexpectedly because of a software bug. For this reason most ARM processors include a vector catch feature to trap these exceptions. This is most useful for bare-metal projects, or projects at an early stage of development. When an OS is running it might use these exceptions for legitimate purposes, for example virtual memory.

When vector catch is enabled, the effect is similar to placing a breakpoint on the selected vector table entry, except that vector catches use dedicated hardware in the processor and do not use up valuable breakpoint resources. To manage vector catch in the debugger either select **Manage Signals** from the **Breakpoints** view menu or use the `handle` command. You can also use the `info signals` command to display the current handler settings.

The vector catch events that are available are dependent on the exact processor that you are connected to.

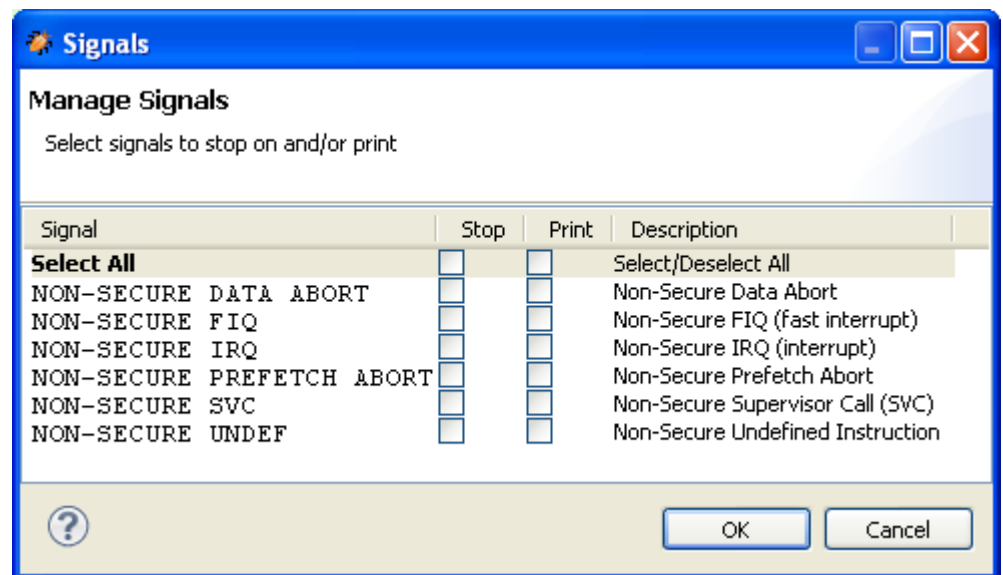


Figure 4-9 Manage exception handler settings

Examples

If you want the debugger to catch the exception, log the event, and stop the application when the exception occurs then you must enable stopping on an exception. In the following example, a `NON-SECURE_FIQ` exception occurs causing the debugger to stop and print a message. You can then step or run to the handler, if present.

Debugging an exception handler

```
handle NON-SECURE_FIQ stop      # Enable stop and print on a NON-SECURE_FIQ
exception
```

If you want the exception to invoke the handler without stopping then you must disable stopping on an exception.

Ignoring an exception

```
handle NON-SECURE_FIQ nostop    # Disable stop on a NON-SECURE_FIQ exception
```

Related concepts

- 6.8 About debugging shared libraries on page 6-146.*
- 6.9 About debugging a Linux kernel on page 6-148.*
- 6.10 About debugging Linux kernel modules on page 6-150.*

Related references

- 4.10 Stepping through an application on page 4-121.*
- 5.1 Examining the target execution environment on page 5-130.*
- 5.2 Examining the call stack on page 5-132.*
- 4.11 Handling Unix signals on page 4-123.*
- 10.4 Breakpoints view on page 10-202.*
- 10.6 Commands view on page 10-209.*
- 10.32 Manage Signals dialog box on page 10-271.*

Related information

- DS-5 Debugger commands.*

4.13 Configuring the debugger path substitution rules

The debugger might not be able to locate the source file when debug information is loaded because:

- The path specified in the debug information is not present on your workstation, or that path does not contain the required source file.
- The source file is not in the same location on your workstation as the image containing the debug information. The debugger attempts to use the same path as this image by default.

Therefore, you must modify the search paths used by the debugger when it executes any of the commands that look up and display source code.

To modify the search paths:

1. Open the Path Substitution dialog box:
 - If a source file cannot be located, a warning is displayed in the C/C++ editor. Click on **Set Path Substitution**.
 - In the **Debug Control** view, select **Path Substitution** from the view menu.

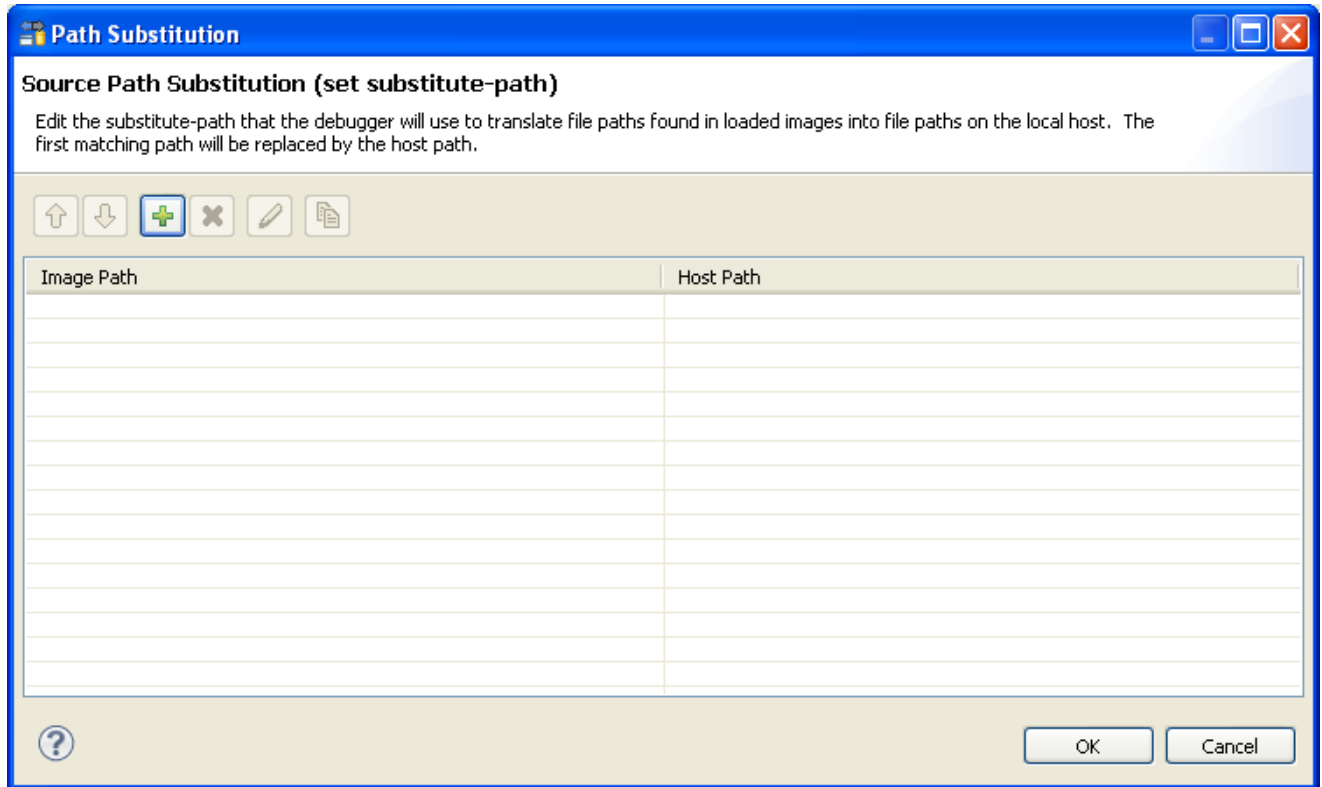


Figure 4-10 Path Substitution dialog box

2. Click on the toolbar icons in the Path Substitution dialog box to add, edit, or duplicate substitution rules:
 - a. Enter the original path for the source files in the Image Path field or click on **Select...** to select from the compilation paths.
 - b. Enter the current location of the source files in the Host Path field or click on:
 - **File System...** to locate the source files in an external folder.
 - **Workspace...** to locate the source files in a workspace project.

- c. Click **OK**.

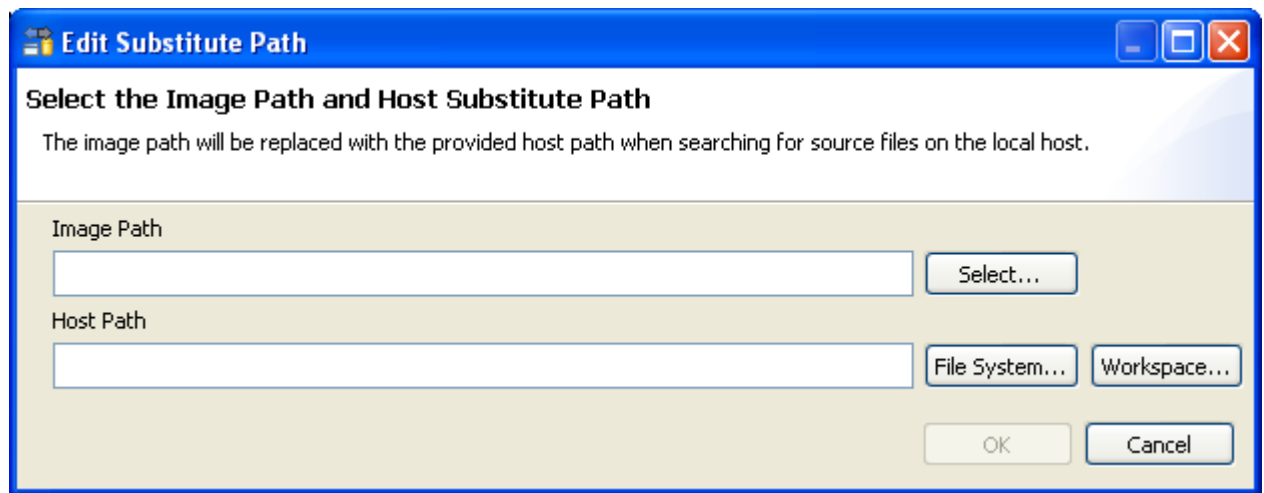


Figure 4-11 Edit Substitute Path dialog box

3. If required, you can use the toolbar icons in the Path Substitution dialog box to change the order of the substitution rules or delete rules that are no longer required.
4. Click **OK** to pass the substitution rules to the debugger and close the dialog box.

Related concepts

[4.2 About loading debug information into the debugger on page 4-103.](#)

Chapter 5

Examining the target

This chapter describes how to examine registers, variables, memory, and the call stack. It contains the following:

- *5.1 Examining the target execution environment on page 5-130.*
- *5.2 Examining the call stack on page 5-132.*
- *5.3 About trace support on page 5-133.*
- *5.4 About post-mortem debugging of trace data on page 5-136.*

5.1 Examining the target execution environment

During a debug session, you might want to display the value of a register or variable, the address of a symbol, the data type of a variable, or the content of memory. The DS-5 Debug perspective provides essential debugger views showing the current values.

As you step through the application, all the views associated with the active connection are updated. In the perspective, you can move any of the views to a different position by clicking on the tab and dragging the view to a new position. You can also double-click on a tab to maximize or reset a view for closer analysis of the contents in the view.

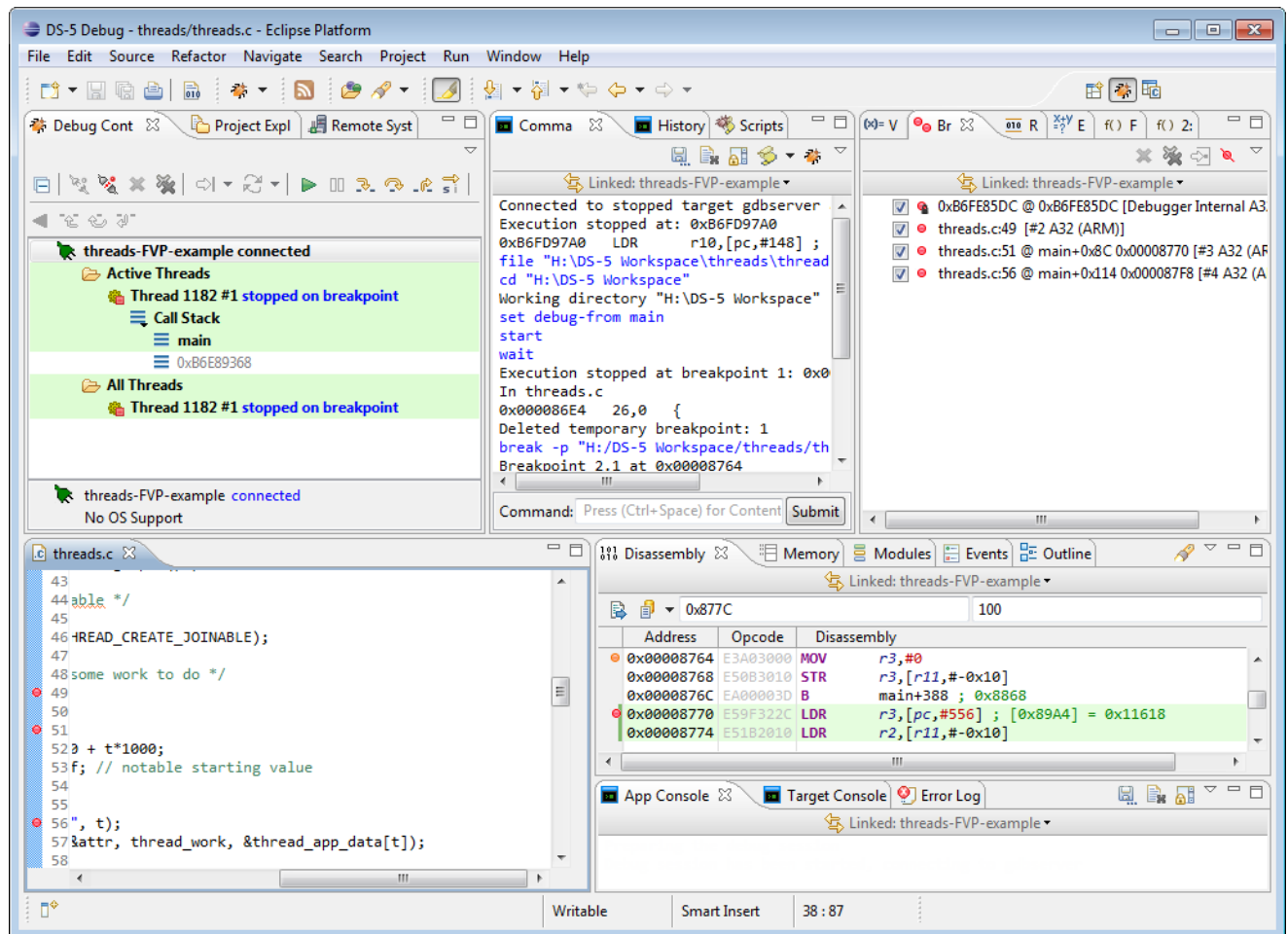


Figure 5-1 Target execution environment

Alternatively, you can use debugger commands to display the required information. In the Commands view, you can execute individual commands or you can execute a sequence of commands by using a script file.

Related concepts

[6.8 About debugging shared libraries on page 6-146.](#)

[6.9 About debugging a Linux kernel on page 6-148.](#)

[6.10 About debugging Linux kernel modules on page 6-150.](#)

Related references

[4.10 Stepping through an application on page 4-121.](#)

5.2 Examining the call stack on page 5-132.

4.11 Handling Unix signals on page 4-123.

4.12 Handling processor exceptions on page 4-125.

5.2 Examining the call stack

The call stack, or runtime stack, is an area of memory used to store function return information and local variables. As each function is called, a record is created on the call stack. This record is commonly known as a stack frame.

The debugger can display the calling sequence of any functions that are still in the execution path because their calling addresses are still on the call stack. However:

- When a function completes execution the associated stack frame is removed from the call stack and the information is no longer available to the debugger.
- If the call stack contains a function for which there is no debug information, the debugger might not be able to trace back up the calling stack frames. Therefore you must compile all your code with debug information to successfully view the full call stack.

If you are debugging multi-threaded applications, a separate call stack is maintained for each thread.

All the views in the DS-5 Debug perspective are associated with the current stack frame and are updated when you select another frame. The current stack frame is shown in bold text.

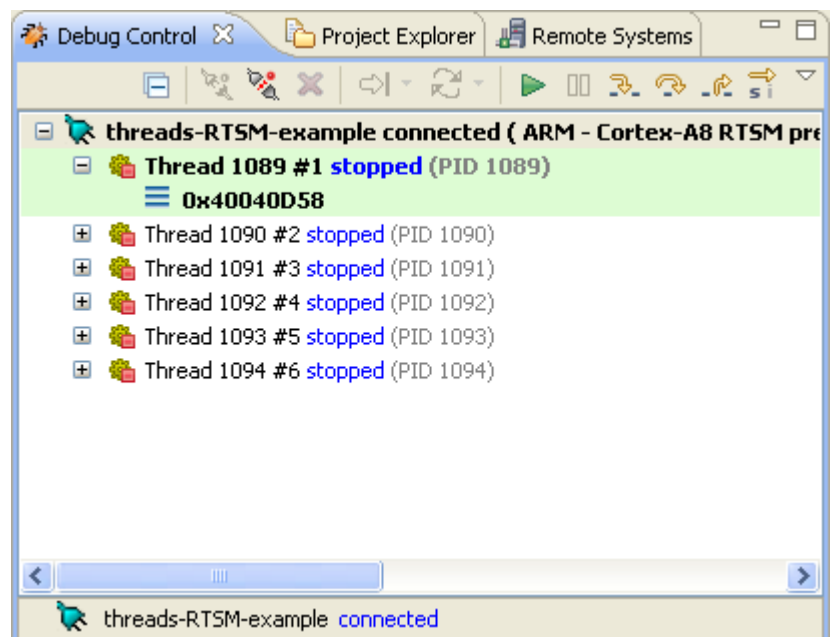


Figure 5-2 Debug Control view

Related concepts

- [6.8 About debugging shared libraries on page 6-146.](#)
- [6.9 About debugging a Linux kernel on page 6-148.](#)
- [6.10 About debugging Linux kernel modules on page 6-150.](#)

Related references

- [4.10 Stepping through an application on page 4-121.](#)
- [5.1 Examining the target execution environment on page 5-130.](#)
- [4.11 Handling Unix signals on page 4-123.](#)
- [4.12 Handling processor exceptions on page 4-125.](#)

5.3 About trace support

ARM DS-5 enables you to perform tracing on your application or system. Tracing is the ability to capture in real-time a historical, non-invasive debug of instructions and data accesses. It is a powerful tool that enables you to investigate problems while the system runs at full speed. These problems can be intermittent, and are difficult to identify through traditional debugging methods that require starting and stopping the processor. Tracing is also useful when trying to identify potential bottlenecks or to improve performance-critical areas of your application.

Before the debugger can trace function executions in your application you must ensure that:

- you have a debug hardware agent, such as an ARM DSTREAM \VSTREAM unit with a connection to a trace stream
- the debugger is connected to the debug hardware agent.

Trace hardware

Trace is typically provided by an external hardware block connected to the processor. This is known as an *Embedded Trace Macrocell* (ETM) or *Program Trace Macrocell* (PTM) and is an optional part of an ARM architecture-based system. System-on-chip designers might omit this block from their silicon to reduce costs. These blocks observe (but do not affect) the processor behavior and are able to monitor instruction execution and data accesses.

There are two main problems with capturing trace. The first is that with very high processor clock speeds, even a few seconds of operation can mean billions of cycles of execution. Clearly, to look at this volume of information would be extremely difficult. The second, related problem is that modern processors could potentially perform one or more 64-bit cache accesses per cycle and to record both the data address and data values might require large bandwidth. This presents a problem in that typically, only a few pins are provided on the chip and these outputs might be able to be switched at significantly lower rates than the processor can be clocked at. It is very easy to exceed the capacity of the trace port. To solve this latter problem, the trace macrocell tries to compress information to reduce the bandwidth required. However, the main method to deal with these issues is to control the trace block so that only selected trace information is gathered. For example, trace only execution, without recording data values, or trace only data accesses to a particular peripheral or during execution of a particular function.

In addition, it is common to store trace information in an on-chip memory buffer, the *Embedded Trace Buffer* (ETB). This alleviates the problem of getting information off-chip at speed, but has an additional cost in terms of silicon area and also provides a fixed limit on the amount of trace that can be captured.

The ETB stores the compressed trace information in a circular fashion, continuously capturing trace information until stopped. The size of the ETB varies between chip implementations, but a buffer of 8 or 16kB is typically enough to hold a few thousand lines of program trace.

When a program fails, and the trace buffer is enabled, you can see a portion of program history. With this program history, it is easier to walk back through your program to see what happened just before the point of failure. This is particularly useful for investigating intermittent and real-time failures, which can be difficult to identify through traditional debug methods that require stopping and starting the processor. The use of hardware tracing can significantly reduce the amount of time required to find these failures, because the trace shows exactly what was executed.

Trace Ranges

Trace ranges enable you to restrict the capture of trace to a linear range of memory. A trace range has a start and end address in virtual memory, and any execution within this address range is captured. In contrast to trace start and end points, any function calls made within a trace range are

only captured if the target of the function call is also within the specified address range. The number of trace ranges that can be enabled is determined by the debug hardware in your processor.

Trace capture is enabled by default when no trace ranges are set. Trace capture is disabled by default when any trace ranges are set, and is only enabled when within the defined ranges.

You can configure trace ranges using the **Ranges** tab in the Trace view. The start and end address for each range can either be an absolute address or an expression, such as the name of a function. Be aware that optimizing compilers might rearrange or minimize code in memory from that in the associated source code. This can lead to code being unexpectedly included or excluded from the trace capture.

Trace Points

Trace points enable you to control precisely where in your program trace is captured. Trace points are non-intrusive and do not require stopping the system to process. The maximum number of trace points that can be set is determined by the debug hardware in your processor. The following types of trace points are available: To set trace points in the source view, right-click in the margin and select the required option from the **DS-5 Breakpoints** context menu. To set trace points in the Disassembly view, right-click on an instruction and select the required option from the **DS-5 Breakpoints** context menu. Trace points are listed in the Breakpoints view.

Trace Start Point

Enables trace capture when execution reaches the selected address.

Trace Stop Point

Disables trace capture when execution reaches the selected address

Trace Trigger Point

Marks this point in the trace so that you can more easily locate it in the Trace view.

Trace Start Points and Trace Stop Points enable and disable capture of trace respectively. Trace points do not take account of nesting. For example, if you hit two Trace Start Points in a row, followed by two Trace Stop Points, then the trace is disabled immediately when the first Trace Stop Point is reached, not the second. With no Trace Start Points set then trace is enabled all the time by default. If you have any Trace Start Points set, then trace is disabled by default and is only enabled when the first Trace Start Point is hit.

Trace trigger points enable you to mark interesting locations in the trace so that you can easily find them later in the Trace view. The first time a Trigger Point is hit a Trace Trigger Event record is inserted into the trace buffer. Only the first Trigger Point to be hit inserts the trigger event record. To configure the debugger so that it stops collecting trace when a trace trigger point is hit, use the **Stop Trace Capture On Trigger** checkbox in the **Properties** tab of the Trace view.

———— Note ————

This does not stop the target. It only stops the trace capture. The target continues running normally until it hits a breakpoint or until you click the **Interrupt** icon in the Debug Control view.

When this is set you can configure the amount of trace that is captured before and after a trace trigger point using the Post-Trigger Capture Size field in the **Properties** tab of the Trace view. If you set this field to:

0%

The trace capture stops as soon as possible after the first trigger point is hit. The trigger event record can be found towards the end of the trace buffer.

50%

The trace capture stops after the first trigger point is hit and an additional 50% of the buffer is filled. The trigger event record can be found towards the middle of the trace buffer.

99%

The trace capture stops after the first trigger point is hit and an additional 99% of the buffer is filled. The trigger event record can be found towards the beginning of the trace buffer.

Note

Due to target timing constraints the trigger event record might get pushed out of the trace buffer.

Being able to limit trace capture to the precise areas of interest is especially helpful when using a capture device such as an ETB, where the quantity of trace that can be captured is very small.

Select the **Find Trigger Event record** option in the view menu to locate Trigger Event record in the trace buffer.

Note

Trace trigger functionality is dependent on the target platform being able to signal to the trace capture hardware, such as ETB or VSTREAM, that a trigger condition has occurred. If this hardware signal is not present or not configured correctly then it might not be possible to automatically stop trace capture around trigger points.

Related concepts

[5.4 About post-mortem debugging of trace data on page 5-136.](#)

[1.4 About DS-5™ headless command-line debugger on page 1-22.](#)

Related tasks

[1.6 Specifying a custom configuration database using the headless command-line debugger on page 1-26.](#)

Related references

[1.5 Headless command-line debugger options on page 1-23.](#)

5.4 About post-mortem debugging of trace data

You can decode previously captured trace data. You must have files available containing the captured trace, as well as any other files, such as configuration and images, that are needed to process and decode that trace data.

Once the trace data and other files are ready, you configure the headless command-line debugger to connect to the post-mortem debug configuration from the configuration database.

You can then inspect the state of the data at the time of the trace capture.

Note

- The memory and registers are read-only.
- You can add more debug information using additional files.
- You can also decode trace and dump the output to files.

The basic steps for post-mortem debugging using the headless command-line debugger are:

1. Generate trace data files.
2. Use **--cdb-list** to list the platforms and parameters available in the configuration database.
3. Use **--cdb-entry** to specify a platform entry in the configuration database.
4. If you need to specify additional parameters, use the **--cdb-entry-param** option to specify the parameters.

Note

At the DS-5 command prompt, enter **debugger --help** to view the list of available options.

Related concepts

[5.3 About trace support on page 5-133.](#)

[1.4 About DS-5™ headless command-line debugger on page 1-22.](#)

Related tasks

[1.6 Specifying a custom configuration database using the headless command-line debugger on page 1-26.](#)

Related references

[1.5 Headless command-line debugger options on page 1-23.](#)

Chapter 6

Debugging embedded systems

Gives an introduction to debugging embedded systems.
It contains the following:

- *6.1 About endianness on page 6-138.*
- *6.2 About accessing AHB, APB, and AXI buses on page 6-139.*
- *6.3 About virtual and physical memory on page 6-140.*
- *6.4 About debugging hypervisors on page 6-141.*
- *6.5 About debugging big.LITTLE systems on page 6-142.*
- *6.6 About debugging bare-metal symmetric multiprocessing systems on page 6-143.*
- *6.7 About debugging multi-threaded applications on page 6-145.*
- *6.8 About debugging shared libraries on page 6-146.*
- *6.9 About debugging a Linux kernel on page 6-148.*
- *6.10 About debugging Linux kernel modules on page 6-150.*
- *6.11 About debugging FreeRTOS™ on page 6-152.*
- *6.12 About debugging TrustZone enabled targets on page 6-153.*
- *6.13 About debugging a Unified Extensible Firmware Interface (UEFI) on page 6-155.*
- *6.14 About application rewind on page 6-156.*
- *6.15 About debugging ThreadX on page 6-158.*
- *6.16 About DTSL (Debug and Trace Service Layer) on page 6-159.*
- *6.17 About CoreSight™ Target Access Library on page 6-160.*

6.1 About endianness

The term endianness is used to describe the ordering of individually addressable quantities, which means bytes and halfwords in the ARM architecture. The term byte-ordering can also be used rather than endian.

If an image is loaded to the target on connection, the debugger automatically selects the endianness of the image otherwise it selects the current endianness of the target. If the debugger detects a conflict then a warning message is generated.

You can use the `set endian` command to modify the default debugger setting.

Related information

[*DS-5 Debugger commands.*](#)

6.2 About accessing AHB, APB, and AXI buses

ARM-based systems connect the processors, memories and peripherals using buses. Examples of common bus types include AMBA High-performance Bus (AHB), Advanced Peripheral Bus (APB), and Advanced eXtensible Interface (AXI).

In some systems these buses are accessible from the debug interface. Where this is the case then DS-5 Debugger provides access to these buses when performing bare-metal or kernel debugging. Buses are exposed within the debugger as additional address spaces. Accesses to these buses are available when the processor is running.

Within a debug session in DS-5 Debugger you can discover which buses are available using the `info memory` command. The address and description columns in the output of this command explain what each address space represents and how the debugger accesses it.

You can use `AHB:`, `APB:`, and `AXI:` address prefixes for these buses anywhere in the debugger where you normally enter an address or expression. For example, assuming that the debugger provides an APB address space, then you can print the contents of address zero using the following command:

```
x/1 APB:0x0
```

The exact topology of the buses and their connection to the debug interface is dependent on your system. See the CoreSight specifications for your hardware for more information. Typically, the debug access to these buses bypass the processor, and so does not take into account memory mappings or caches within the processor itself. It is implementation dependent on whether access to the buses occur before or after any other caches in the system, such as L2 or L3 caches. The debugger does not attempt to achieve coherency between caches in your system when accessing these buses and it is your responsibility to take this into account and manually perform any clean or flush operations as required.

For example, to achieve cache coherency when debugging an image with the processors level 1 cache enabled, you must clean and invalidate portions of the L1 cache prior to modifying any of your application code or data using the AHB address space. This ensures that any existing changes in the cache are written out to memory before writing to that address space, and that the processor correctly reads your modification when execution resumes.

The behavior when accessing unallocated addresses is undefined, and depending on your system can lead to locking up the buses. It is recommended that you only access those specific addresses that are defined in your system. You can use the `memory` command to redefine the memory regions within the debugger and modifying access rights to control the addresses. You can use the `x` command with the `count` option to limit the amount of memory that is read.

Related references

[10.6 Commands view on page 10-209.](#)

[10.7 Debug Control view on page 10-212.](#)

Related information

[DS-5 Debugger commands.](#)

6.3 About virtual and physical memory

Processors that contain a Memory Management Unit (MMU) provide two views of memory, virtual and physical. The virtual address is the address prior to address translation in the MMU and the physical address is the address after translation. Normally when the debugger accesses memory, it uses virtual addresses. However, if the MMU is disabled then the mapping is flat and the virtual address is the same as the physical address. To force the debugger to use physical addresses prefix an addresses with **P:**. For example:

```
P:0x8000
P:0+main creates a physical address with the address offset of main()
```

If your processor additionally contains TrustZone technology, then you have access to Secure and Normal worlds, each with their own separate virtual and physical address mappings. In this case, the address prefix **P:** is not available, and instead you must use **NP:** for normal physical and **SP:** for secure physical.

———— **Note** ————

Physical address access is not enabled for all operations. For example, the ARM hardware does not provide support for setting breakpoints via a physical address.

When memory is accessed via a physical address the caches are not flushed. Hence, results might differ depending on whether you view memory through the physical or virtual addresses (assuming they are addressing the same memory addresses).

Related references

[10.6 Commands view on page 10-209.](#)

[10.7 Debug Control view on page 10-212.](#)

Related information

[DS-5 Debugger commands.](#)

6.4 About debugging hypervisors

ARM processors that support virtualization extensions have the ability to run multiple guest operating systems beneath a hypervisor. The hypervisor is the software that arbitrates amongst the guest operating systems and controls access to the hardware.

DS-5 Debugger provides basic support for bare-metal hypervisor debugging. When connected to a processor that supports virtualization extensions, the debugger enables you to distinguish between hypervisor and guest memory, and to set breakpoints that only apply when in hypervisor mode or within a specific guest operating system.

A hypervisor typically provides separate address spaces for itself as well as for each guest operating system. Unless informed otherwise, all memory accesses by the debugger occur in the current context. If you are stopped in hypervisor mode then memory accesses use the hypervisor memory space, and if stopped in a guest operating system then memory accesses use the address space of the guest operating system. To force access to a particular address space you must prefix the address with either **H:** for hypervisor or **N:** for guest operating system. Note that it is only possible to access the address space of the guest operating system that is currently scheduled to run within the hypervisor. It is not possible to specify a different guest operating system.

Similarly, hardware and software breakpoints can be configured to match on hypervisor or guest operating systems using the same address prefixes. If no address prefix is used then the breakpoint applies to the address space that is current when the breakpoint is first set. For example, if a software breakpoint is set in memory that is shared between hypervisor and a guest operating system, then the possibility exists for the breakpoint to be hit from the wrong mode, and in this case the debugger may not recognize your breakpoint as the reason for stopping.

For hardware breakpoints only, not software breakpoints, you can additionally configure them to match only within a specific guest operating system. This feature uses the architecturally defined Virtual Machine ID (VMID) register to spot when a specific guest operating system is executing. The hypervisor is responsible for assigning unique VMIDs to each guest operating system setting this in the VMID register when that guest operating system executes. In using this feature, it is your responsibility to understand which VMID is associated with each guest operating system that you want to debug. Assuming a VMID is known, you can apply a breakpoint to it within the **Breakpoints** view or by using the **break-stop-on-vmid** command.

When debugging a system that is running multiple guest operating systems, you can optionally enable the **set print current-vmid** setting to receive notifications in the console when the debugger stops and the current VMID changes. You can also obtain the VMID within DS-5 scripts using the **\$vmid** debugger variable.

Related references

[10.6 Commands view on page 10-209.](#)

[10.7 Debug Control view on page 10-212.](#)

Related information

[DS-5 Debugger commands.](#)

6.5 About debugging big.LITTLE systems

A big.LITTLE system is designed to optimize both high performance and low power consumption over a wide variety of workloads. It achieves this by including one or more high performance processors alongside one or more low power processors. The system transitions the workload between the processors as necessary to achieve this goal.

big.LITTLE systems are typically configured in a *Symmetric MultiProcessing* (SMP) configuration. An operating system or hypervisor controls which processors are powered up or down at any given time and assists in migrating tasks between them.

For bare-metal debugging on big.LITTLE systems, you can establish an SMP connection within DS-5 Debugger. In this case all the processors in the system are brought under the control of the debugger. The debugger monitors the power state of each processor as it runs and displays it in the **Debug Control** view and on the command -line. Processors that are powered-down are visible to the debugger but cannot be accessed.

For Linux application debugging on big.LITTLE systems, you can establish a **gdbserver** connection within DS-5 Debugger. Linux applications are typically unaware of whether they are running on a big processor or a little processor because this is hidden by the operating system. There is therefore no difference within the debugger when debugging a Linux application on a big.LITTLE system as compared to application debug on any other system.

Related concepts

[6.6 About debugging bare-metal symmetric multiprocessing systems on page 6-143.](#)

Related references

[10.6 Commands view on page 10-209.](#)

[10.7 Debug Control view on page 10-212.](#)

Related information

[DS-5 Debugger commands.](#)

6.6 About debugging bare-metal symmetric multiprocessing systems

DS-5 Debugger supports debugging bare-metal *Symmetric MultiProcessing* (SMP) systems. The debugger expects an SMP system to meet the following requirements:

- The same ELF image running on all processors.
- All processors must have identical debug hardware. For example, the number of hardware breakpoint and watchpoint resources must be identical.
- Breakpoints and watchpoints must only be set in regions where all processors have identical memory maps, both physical and virtual. Processors with different instance of identical peripherals mapped at the same address are considered to meet this requirement, as in the case of the private peripherals of ARM multicore processors.

Configuring and connecting

To enable SMP support in the debugger you must first configure a debug session in the Debug Configurations dialog. Targets that support SMP debugging are identified by having SMP mentioned in the Debug operation drop-down list.

Configuring a single SMP connection is all you require to enable SMP support in the debugger. On connection, you can then debug all of the SMP processors in your system by selecting them in the **Debug Control** view.

Image and symbol loading

When debugging an SMP system, image and symbol loading operations apply to all the SMP processors. For image loading, this means that the image code and data are written to memory once through one of the processors, and are assumed to be accessible through the other processors at the same address because they share the same memory. For symbol loading, this means that debug information is loaded once and is available when debugging any of the processors.

Running, stopping and stepping

When debugging an SMP system, attempting to run one processor automatically starts running all the other processors in the system. Similarly, when one processor stops (either because you requested it or because of an event such as a breakpoint being hit), then all processors in the system stop.

For instruction level single-stepping (**stepi** and **nexti** commands), then the currently selected processor steps one instruction. The exception to this is when a **nexti** operation is required to step over a function call in which case the debugger sets a breakpoint and then runs all processors. All other stepping commands affect all processors.

Depending on your system, there might be a delay between one processor running or stopping and another processor running or stopping. This delay can be very large because the debugger must manually run and stop all the processors individually.

In rare cases, one processor might stop and one or more of the others fails to stop in response. This can occur, for example, when a processor running code in secure mode has temporarily disabled debug ability. When this occurs, the **Debug Control** view displays the individual state of each processor (running or stopped), so that you can see which ones have failed to stop. Subsequent run and step operations might not operate correctly until all the processors stop.

Breakpoints, watchpoints, and signals

By default, when debugging an SMP system, breakpoint, watchpoint, and signal (vector catch) operations apply to all processors. This means that you can set one breakpoint to trigger when any of the processors execute code that meets the criteria. When the debugger stops due to a

breakpoint, watchpoint, or signal, then the processor that causes the event is listed in the **Commands** view.

Breakpoints or watchpoints can be configured for one or more processors by selecting the required processor in the relevant Properties dialog box. Alternatively, you can use the **break-stop-on-cores** command. This feature is not available for signals.

Examining target state

Views of the target state, including registers, call stack, memory, disassembly, expressions, and variables contain content that is specific to a processor.

Views such as breakpoints, signals and commands are shared by all the processors in the SMP system, and display the same contents regardless of which processor is currently selected.

Trace

When you are using a connection that enables trace support then you are able to view trace for each of the processors in your system. By default, the **Trace** view shows trace for the processor that is currently selected in the **Debug Control** view. Alternatively, you can choose to link a **Trace** view to a specific processor by using the **Linked: context** toolbar option for that **Trace** view. Creating multiple **Trace** views linked to specific processors enables you to view the trace from multiple processors at the same time. The indexes in the **Trace** views do not necessarily represent the same point in time for different processors.

Related concepts

[6.5 About debugging big.LITTLE systems on page 6-142.](#)

[4.1 About loading an image on to the target on page 4-101.](#)

[4.2 About loading debug information into the debugger on page 4-103.](#)

Related references

[4.5 Working with breakpoints and watchpoints on page 4-107.](#)

[Working with data watchpoints.](#)

[4.8 Setting a tracepoint on page 4-119.](#)

[4.9 Setting Streamline start and stop points on page 4-120.](#)

[4.6 Working with conditional breakpoints on page 4-114.](#)

[4.6.1 Assigning conditions to an existing breakpoint on page 4-114.](#)

[4.7 About pending breakpoints and watchpoints on page 4-118.](#)

[4.10 Stepping through an application on page 4-121.](#)

[10.4 Breakpoints view on page 10-202.](#)

[10.6 Commands view on page 10-209.](#)

[10.7 Debug Control view on page 10-212.](#)

[10.8 Disassembly view on page 10-216.](#)

[10.13 Memory view on page 10-228.](#)

[10.14 Modules view on page 10-232.](#)

[10.15 Registers view on page 10-236.](#)

[10.23 Variables view on page 10-255.](#)

Related information

[DS-5 Debugger commands.](#)

6.7 About debugging multi-threaded applications

The debugger tracks the current thread using the debugger variable, `$thread`. You can use this variable in print commands or in expressions. Threads are displayed in the **Debug Control** view with a unique ID that is used by the debugger and a unique ID from the *Operating System* (OS) :

Thread 1086 #1 stopped (PID 1086)

where #1 is the unique ID used by the debugger and PID 1086 is the ID from the OS.

A separate call stack is maintained for each thread and the selected stack frame is shown in bold text. All the views in the DS-5 Debug perspective are associated with the selected stack frame and are updated when you select another frame.

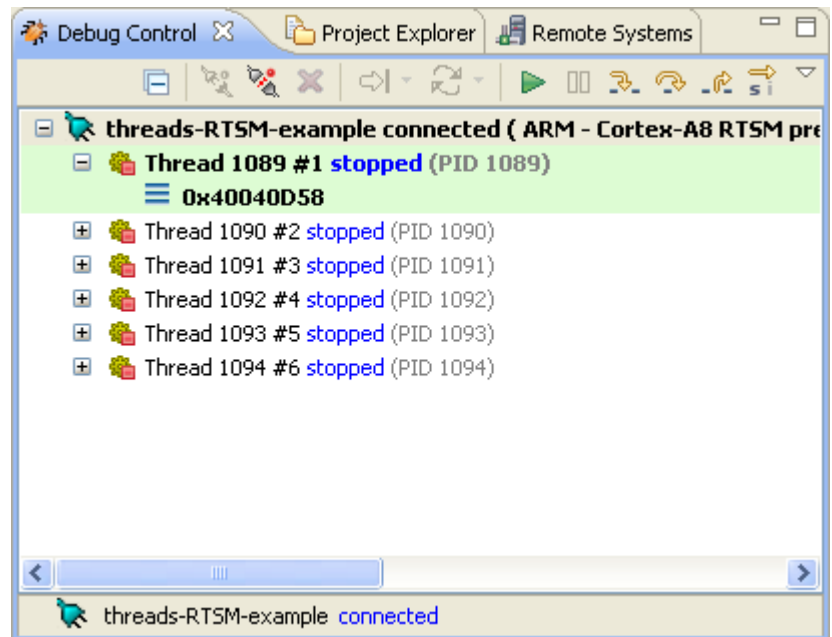


Figure 6-1 Threading call stacks in the Debug Control view

Related references

- [10.4 Breakpoints view on page 10-202.](#)
- [10.6 Commands view on page 10-209.](#)
- [10.7 Debug Control view on page 10-212.](#)
- [10.8 Disassembly view on page 10-216.](#)
- [10.13 Memory view on page 10-228.](#)
- [10.14 Modules view on page 10-232.](#)
- [10.15 Registers view on page 10-236.](#)
- [10.23 Variables view on page 10-255.](#)

6.8 About debugging shared libraries

Shared libraries enable parts of your application to be dynamically loaded at runtime. You must ensure that the shared libraries on your target are the same as those on your host. The code layout must be identical, but the shared libraries on your target do not require debug information.

You can set standard execution breakpoints in a shared library but not until it is loaded by the application and the debug information is loaded into the debugger. Pending breakpoints however, enable you to set execution breakpoints in a shared library before it is loaded by the application.

When a new shared library is loaded the debugger re-evaluates all pending breakpoints, and those with addresses that it can resolve are set as standard execution breakpoints. Unresolved addresses remain as pending breakpoints.

The debugger automatically changes any breakpoints in a shared library to a pending breakpoint when the library is unloaded by your application.

You can load shared libraries in the Debug Configurations dialog box. If you have one library file then you can use the **Load symbols from file** option in the **Files** tab.

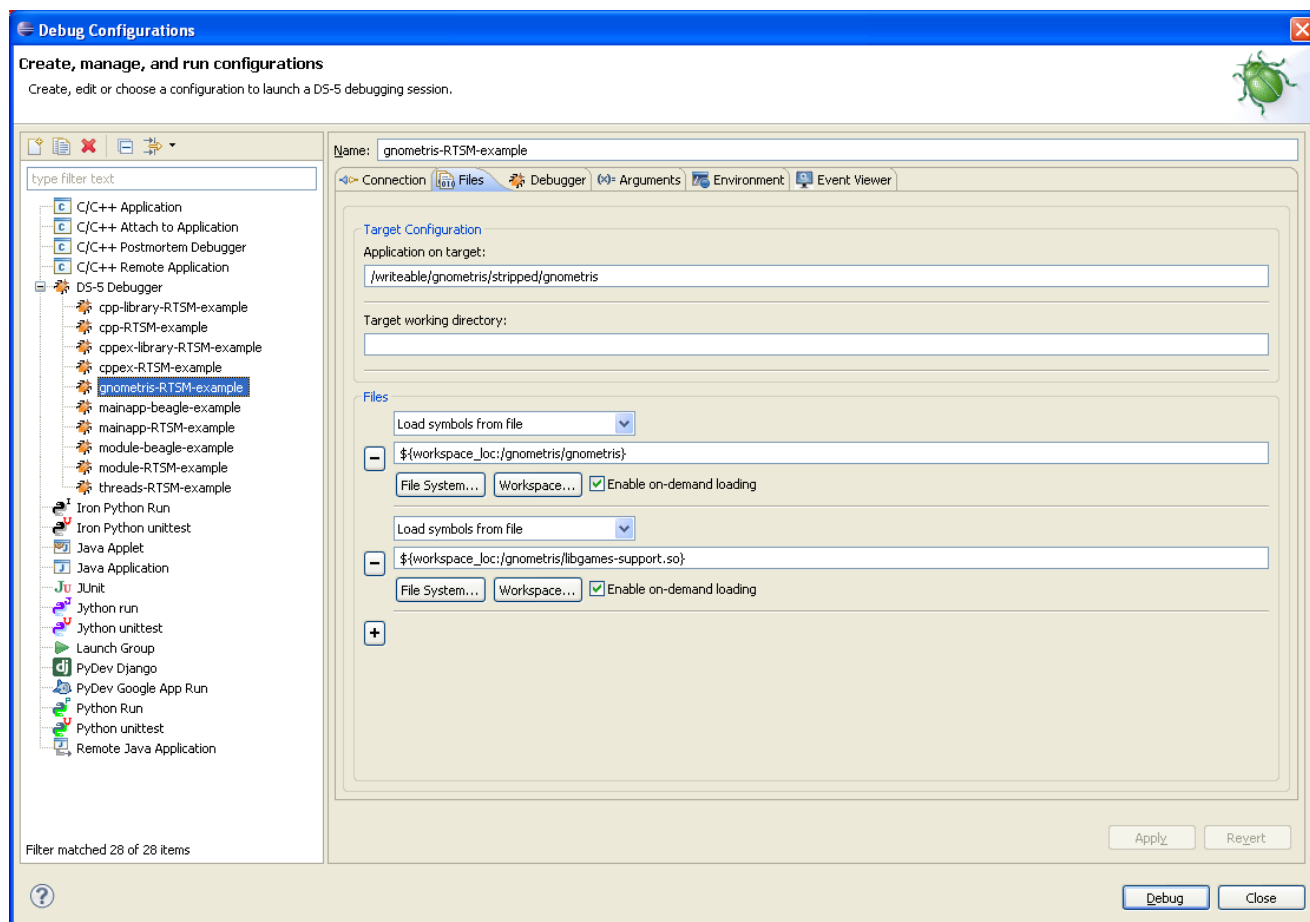


Figure 6-2 Adding individual shared library files

Alternatively if you have multiple library files then it is probably more efficient to modify the search paths in use by the debugger when searching for shared libraries. To do this you can use the **Shared library search directory** option in the **Paths** panel of the **Debugger** tab.

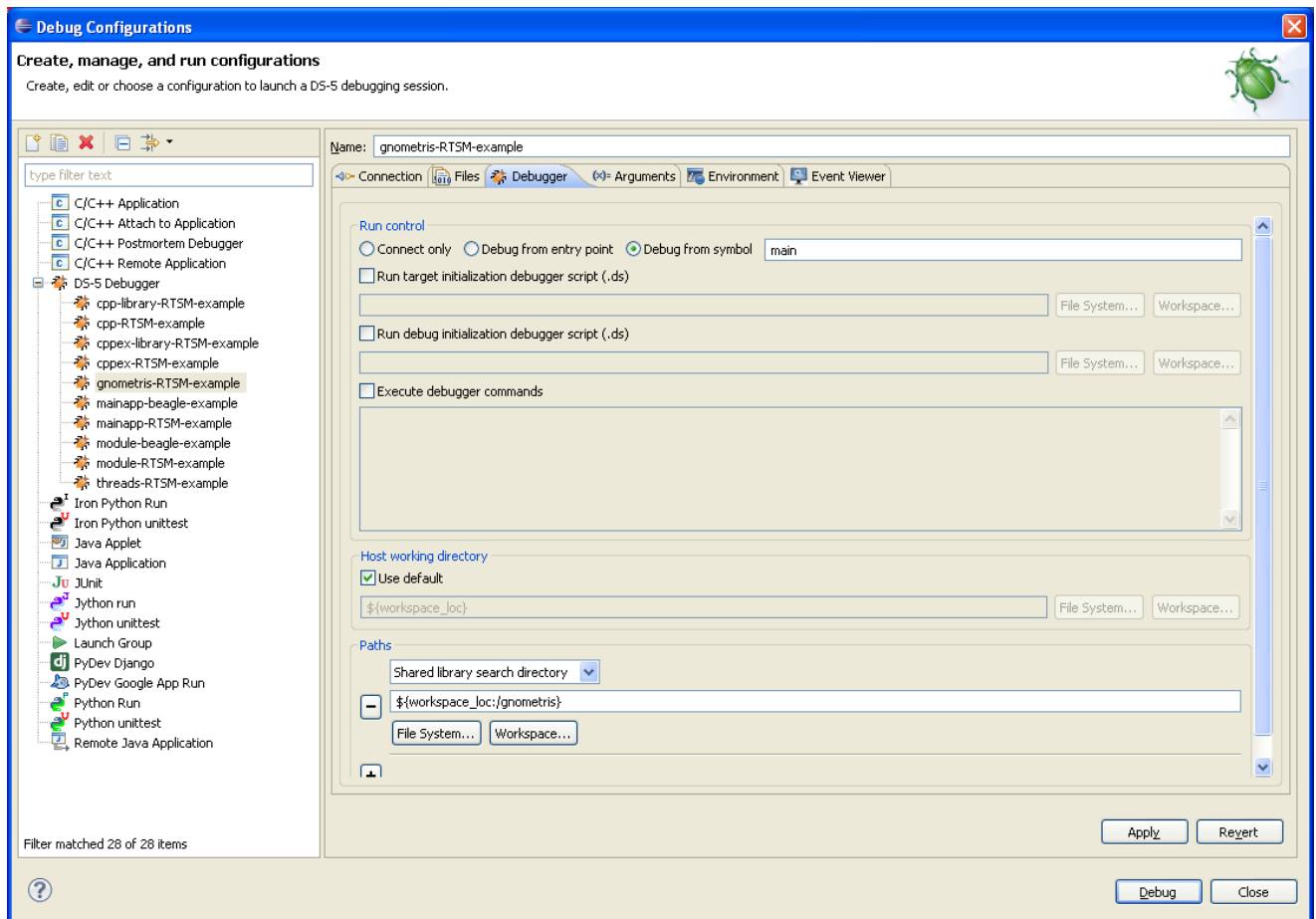


Figure 6-3 Modifying the shared library search paths

For more information on the options in the Debug Configurations dialog box, use the dynamic help.

Related references

- [4.10 Stepping through an application on page 4-121.](#)
- [5.1 Examining the target execution environment on page 5-130.](#)
- [5.2 Examining the call stack on page 5-132.](#)
- [4.11 Handling Unix signals on page 4-123.](#)
- [4.12 Handling processor exceptions on page 4-125.](#)
- [10.4 Breakpoints view on page 10-202.](#)
- [10.6 Commands view on page 10-209.](#)
- [10.7 Debug Control view on page 10-212.](#)
- [10.8 Disassembly view on page 10-216.](#)
- [10.13 Memory view on page 10-228.](#)
- [10.14 Modules view on page 10-232.](#)
- [10.15 Registers view on page 10-236.](#)
- [10.23 Variables view on page 10-255.](#)

6.9 About debugging a Linux kernel

DS-5 supports source level debugging of a Linux kernel. The Linux kernel (and associated device drivers) can be debugged in the same way as a standard ELF format executable. For example, you can set breakpoints in the kernel code, step through the source, inspect the call stack, and watch variables.

Note

User space parameters (marked `__user`) that are not currently mapped in cannot be read by the debugger.

To debug the kernel:

1. Compile the kernel source using the following options:

- `CONFIG_DEBUG_KERNEL=y`
- `CONFIG_DEBUG_INFO=y`
- Other options might be required depending on the type of debugging you want to perform.

Compiling the kernel source generates a Linux kernel image and symbol files containing debug information.

Note

Be aware that a Linux kernel is always compiled with full optimizations and inlining enabled, therefore:

- stepping through code might not work as expected due to the possible reordering of some instructions
- some variables might be optimized out by the compiler and therefore not be available for the debugger.

-
2. Load the Linux kernel on to the target
 3. Load kernel debug information into the debugger
 4. Debug the kernel as required.

Related concepts

[6.10 About debugging Linux kernel modules on page 6-150.](#)

Related tasks

[2.4 Configuring a connection to a Linux Kernel on page 2-37.](#)

Related references

[4.10 Stepping through an application on page 4-121.](#)

[5.1 Examining the target execution environment on page 5-130.](#)

[5.2 Examining the call stack on page 5-132.](#)

[4.11 Handling Unix signals on page 4-123.](#)

[4.12 Handling processor exceptions on page 4-125.](#)

[10.37 Debug Configurations - Files tab on page 10-279.](#)

[10.38 Debug Configurations - Debugger tab on page 10-283.](#)

[10.4 Breakpoints view on page 10-202.](#)

[10.6 Commands view on page 10-209.](#)

[10.7 Debug Control view on page 10-212.](#)

10.8 Disassembly view on page 10-216.

10.13 Memory view on page 10-228.

10.14 Modules view on page 10-232.

10.15 Registers view on page 10-236.

10.23 Variables view on page 10-255.

Related information

Debugging a loadable kernel module.

6.10 About debugging Linux kernel modules

Linux kernel modules provide a way to extend the functionality of the kernel, and are typically used for things such as device and file system drivers. Modules can either be built into the kernel or can be compiled as a loadable module and then dynamically inserted and removed from a running kernel during development without having to frequently recompile the kernel. However, some modules must be built into the kernel and are not suitable for loading dynamically. An example of a built-in module is one that is required during kernel boot and must be available prior to the root file system being mounted.

You can set source-level breakpoints in a module provided that the debug information is loaded into the debugger. Attempts to set a breakpoint in a module before it is inserted into the kernel results in the breakpoint being pended.

When debugging a module, you must ensure that the module on your target is the same as that on your host. The code layout must be identical, but the module on your target does not require debug information.

Built-in module

To debug a module that has been built into the kernel, the procedure is the same as for debugging the kernel itself:

1. Compile the kernel together with the module.
2. Load the kernel image on to the target.
3. Load the related kernel image with debug information into the debugger
4. Debug the module as you would for any other kernel code.

Built-in (statically linked) modules are indistinguishable from the rest of the kernel code, so are not listed by the `info os-modules` command and do not appear in the **Modules** view.

Loadable module

The procedure for debugging a loadable kernel module is more complex. From a Linux terminal shell, you can use the `insmod` and `rmmmod` commands to insert and remove a module. Debug information for both the kernel and the loadable module must be loaded into the debugger. When you insert and remove a module the debugger automatically resolves memory locations for debug information and existing breakpoints. To do this, the debugger intercepts calls within the kernel to insert and remove modules. This introduces a small delay for each action whilst the debugger stops the kernel to interrogate various data structures. For more information on debugging a loadable kernel module, see the tutorial in *Getting Started with DS-5*.

————— Note —————

A connection must be established and *Operating System* (OS) support enabled within the debugger before a loadable module can be detected. OS support is automatically enabled when a Linux kernel image is loaded into the debugger. However, you can manually control this by using the `set os` command.

Related concepts

[6.9 About debugging a Linux kernel on page 6-148.](#)

Related tasks

[2.4 Configuring a connection to a Linux Kernel on page 2-37.](#)

Related references

- 4.10 Stepping through an application on page 4-121.*
- 5.1 Examining the target execution environment on page 5-130.*
- 5.2 Examining the call stack on page 5-132.*
- 4.11 Handling Unix signals on page 4-123.*
- 4.12 Handling processor exceptions on page 4-125.*
- 10.4 Breakpoints view on page 10-202.*
- 10.6 Commands view on page 10-209.*
- 10.7 Debug Control view on page 10-212.*
- 10.8 Disassembly view on page 10-216.*
- 10.13 Memory view on page 10-228.*
- 10.14 Modules view on page 10-232.*
- 10.15 Registers view on page 10-236.*
- 10.23 Variables view on page 10-255.*

Related information

Debugging a loadable kernel module.

6.11 About debugging FreeRTOS™

FreeRTOS is an open-source real-time operating system.

DS-5 Debugger provides the following support for debugging FreeRTOS:

- Supports FreeRTOS on Cortex-M cores.
- View FreeRTOS tasks in the Debug Control view.
- View FreeRTOS tasks and queues in the RTOS Data view.

To enable FreeRTOS support in DS-5 Debugger, in the Debug Configuration dialog, select FreeRTOS in the OS tab. Debugger support is activated when FreeRTOS is initialized on the target device.

Note

Operating system support in the debugger is activated only when OS-specific debug symbols are loaded. Ensure that the debug symbols for the operating system are loaded before using any of the OS-specific views and commands.

When building your FreeRTOS image, ensure that the following compiler flags are set:

- -DportREMOVE_STATIC_QUALIFIER
- -DINCLUDE_xTaskGetIdleTaskHandle
- -DconfigQUEUE_REGISTRY_SIZE=n (where n >= 1)

If these flags are set incorrectly, FreeRTOS support might fail to activate in DS-5 Debugger. See the documentation supplied with FreeRTOS to view the details of these flags.

6.12 About debugging TrustZone enabled targets

ARM TrustZone® is a security technology designed into some ARM processors. For example, the Cortex-A class processors. It segments execution and resources such as memory and peripherals into secure and normal worlds.

When connected to a target that supports TrustZone and where access to the secure world is permitted, then the debugger provides access to both secure and normal worlds. In this case, all addresses and address-related operations are specific to a single world. This means that any commands you use that require an address or expression must also specify the world that they apply to. : 0x1000

Where:

N:

For an address in Normal World memory.

S:

For an address in Secure World memory.

If you want to specify an address in the current world, then you can omit the prefix.

When loading images and debug information it is important that you load them into the correct world. The debug launcher panel does not provide a way to directly specify an address world for images and debug information, so to achieve this you must use scripting commands instead. The **Debugger** tab in the debugger launcher panel provides an option to run a debug initialization script or a set of arbitrary debugger commands on connection. Here are some example commands:

- Load image only to normal world (applying zero offset to addresses in the image)

```
load myimage.axf N:0
```

- Load debug information only to secure world (applying zero offset to addresses in the debug information)

```
file myimage.axf S:0
```

- Load image and debug information to secure world (applying zero offset to addresses)

```
loadfile myimage.axf S:0
```

When an operation such as loading debug symbols or setting a breakpoint needs to apply to both normal and secure worlds then you must perform the operation twice, once for the normal world and once for the secure world.

Registers such as \$PC have no world. To access the content of memory from an address in a register that is not in the current world, you can use an expression, N:0+\$PC. This is generally not necessary for expressions involving debug information, because these are associated with a world when they are loaded.

Related references

- [10.4 Breakpoints view on page 10-202.](#)
- [10.6 Commands view on page 10-209.](#)
- [10.7 Debug Control view on page 10-212.](#)
- [10.8 Disassembly view on page 10-216.](#)
- [10.13 Memory view on page 10-228.](#)
- [10.14 Modules view on page 10-232.](#)

10.15 Registers view on page 10-236.

10.23 Variables view on page 10-255.

Related information

DS-5 Debugger commands.

ARM Security Technology Building a Secure System using TrustZone Technology.

Technical Reference Manual.

Architecture Reference Manual.

6.13 About debugging a Unified Extensible Firmware Interface (UEFI)

UEFI defines a software interface to control the start-up of complex microprocessor systems. UEFI on ARM allows you to control the booting of ARM-based servers and client computing devices.

DS-5 provides a complete UEFI development environment which enables you to:

- Fetch the UEFI source code via the Eclipse Git plug-in (available as a separate download from the Eclipse website).
- Build the source code using the ARM Compiler.
- Download the executables to a software model (a Cortex-A9x4 FVP is provided with DS-5) or to a hardware target (available separately).
- Run/debug the code using the DS-5 Debugger.
- Debug dynamically loaded modules at source-level using Jython scripts.

To download the UEFI source code and Jython scripts, search for "SourceForge.net: ArmPlatformPkg/ArmVExpressPkg" in your preferred search engine.

For more information, see this blog: [UEFI Debug Made Easy](#)

6.14 About application rewind

Application rewind is a DS-5 Debugger feature that allows you to debug backwards as well as forwards through the execution of Linux and Android applications.

Note

The application rewind feature in DS-5 Debugger is license managed. Contact your support representative for details about this feature.

Debugging backwards is useful to help track down how an application reached a particular state, without having to repeatedly rerun your application from the beginning. Using this feature, you can both run and step, including hitting breakpoints and watchpoints. You can also view the contents of recorded memory, registers, and variables at any point in your application's execution.

Note

- Application rewind does not follow forked processes.
 - When debugging backwards, you can only view the contents of recorded memory, registers, or variables. You cannot edit or change them.
 - Application rewind supports architecture ARMv5TE targets and later, except for the 64-bit ARMv8 architecture.
-

Application rewind uses a custom debug agent that records the execution of your application as it runs. This custom debug agent implements a buffer on the target to store history for recorded executions. The default is a straight buffer, which records events until the buffer limit is reached, and then stops the execution. At this point, you can either increase the size of the buffer or change the buffer to be circular. When using a circular buffer, once the limit of a circular buffer is reached, instead of stopping execution, the data wraps around and overwrites old content. A circular buffer ensures that execution does not stop when the buffer limit is reached, but you lose the execution history beyond the point where data wrapped around.

- To change buffer limits, use the command **set debug-agent history-buffer-size "size"** Where **size** specifies the amount of memory. You can specify the value in kilobytes (**K**), megabytes (**M**), or gigabytes (**G**).

For example, **set debug-agent history-buffer-size "256.00 M"**

- To change buffer type, use the command **set debug-agent history-buffer-type "type"** Where **type** specifies the type of buffer, which is either **straight** or **circular**.

For example, **set debug-agent history-buffer-type "circular"**

Note

Debugging your application using application rewind results in increased memory consumption on your target and might slow down your application. The exact impact is dependent on the behavior of your application. Applications that perform large amounts of I/O are likely to experience increased memory consumption during the recording process.

Related tasks

[2.5.1 Connecting to an existing application and application rewind session on page 2-39.](#)

[2.5.2 Downloading your application and application rewind server on the target system on page 2-41.](#)

[2.5.3 Starting the application rewind server and debugging the target-resident application on page 2-42.](#)

2.6.1 Attaching to a running Android application on page 2-44.

2.6.2 Downloading and debugging an Android application on page 2-45.

Related references

2.1 Types of target connections on page 2-32.

6.15 About debugging ThreadX

ThreadX is a real-time operating system from Express Logic, Inc.

DS-5 Debugger provides the following ThreadX RTOS visibility:

- Comprehensive thread list with thread status and objects on which the threads are blocked/suspended.
- All major ThreadX objects including semaphores, mutexes, memory pools, message queues, event flags, and timers.
- Stack usage for individual threads.
- Call frames and local variables for all threads.

To enable ThreadX support in DS-5 Debugger, in the Debug Configuration dialog, select **ThreadX** in the **OS Awareness** tab. ThreadX OS awareness is activated when ThreadX is initialized on the target device.

6.16 About DTSL (Debug and Trace Service Layer)

DTSL (Debug and Trace Service Layer) is a software layer within the DS-5 Debugger stack.

DTSL is implemented as a set of Java classes which are typically implemented (and possibly extended) by Jython scripts. A typical DTSL instance is a combination of Java and Jython.

Note

DTSL Jython Scripting should not be confused with DS-5 Debugger Jython Scripting. They both use Jython but operate at different levels within the software stack. It is however possible for a debugger Jython Script to use DTSL functionality.

DTSL takes responsibility for:

- Low level debugger component creation and configuration. For example, CoreSight component configuration (and sometimes live re-configuration).
- Target access and debug control.
- Capture and control of trace data - both in-target trace capture components, like ETB and any off-target trace capture device, like DSTREAM.
- Delivery of trace streams to the debugger or other 3rd party trace consumers.

ARM has made DTSL available for your own use so that you can create programs (Java or Jython) to access/control the target platform.

For details, see the DTSL documents and files provided with DS-5 here:

<DS-5 Install folder>\sw\DTSL

6.17 About CoreSight™ Target Access Library

CoreSight on-target access library allows you to interact directly with CoreSight devices. This supports use-cases such as enabling flight-recorder trace in a production system without the need to connect an external debugger.

The library offers a flexible programming interface allowing a variety of use cases and experimentation.

It also offers some advantages compared to a register-level interface. For example, it can:

- Manage any unlocking and locking of CoreSight devices via the lock register, OS Lock register, programming bit, power-down bit.
- Attempt to ensure that the devices are programmed correctly and in a suitable sequence.
- Handle variations between devices, and where necessary, work around known issues. For example, between variants of ETM/PTMs.
- Become aware of the trace bus topology and will generally manage trace links automatically. For example enabling only funnel ports in use
- Manage “claim bits” that coordinate internal and external use of CoreSight devices.

For details, see the CoreSight example provided with DS-5 here:

`<installdir>/examples/CoreSight_Access_Library.zip`

Chapter 7

Controlling runtime messages

Describes semihosting and how to control runtime messages.
It contains the following:

- *7.1 About semihosting and top of memory on page 7-162.*
- *7.2 Working with semihosting on page 7-163.*
- *7.3 Enabling automatic semihosting support in the debugger on page 7-164.*
- *7.4 Controlling semihosting messages using the command-line console on page 7-165.*
- *7.5 Controlling the output of logging messages on page 7-166.*
- *7.6 About Log4j configuration files on page 7-167.*
- *7.7 Customizing the output of logging messages from the debugger on page 7-168.*

7.1 About semihosting and top of memory

Describes a typical memory layout for an ARM target.

Semihosting is typically used when debugging an application that is using the C library and running without an operating system. This enables functions in the C library, such as `printf()` and `scanf()`, to use the screen and keyboard on the host workstation instead of having a screen and keyboard on the target system.

Semihosting uses stack base and heap base addresses to determine the location and size of the stack and heap.

The stack base, also known as the top of memory, is an address that is by default 64K from the end of the heap base.

The heap base is by default contiguous to the application code.

The following figure shows a typical layout for an ARM target.

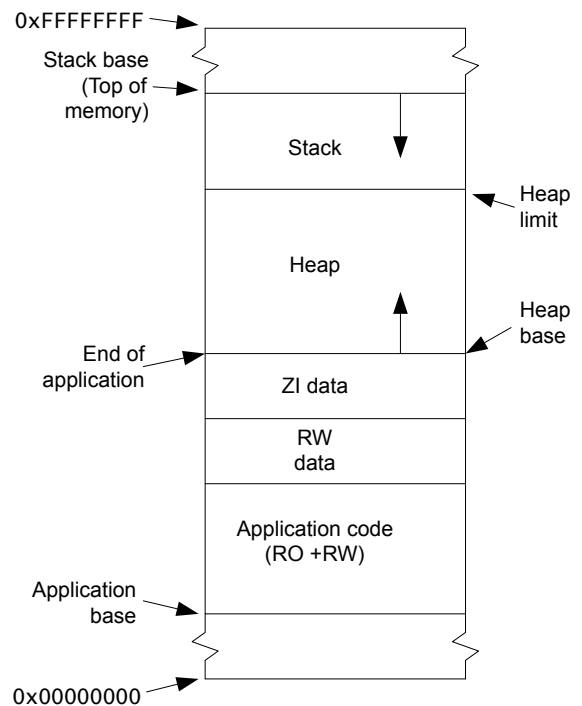


Figure 7-1 Typical layout between top of memory, stack, and heap

Related references

- [4.3 About passing arguments to main\(\) on page 4-105.](#)
- [7.2 Working with semihosting on page 7-163.](#)
- [7.3 Enabling automatic semihosting support in the debugger on page 7-164.](#)
- [7.4 Controlling semihosting messages using the command-line console on page 7-165.](#)
- [10.40 Debug Configurations - Arguments tab on page 10-287.](#)
- [10.1 App Console view on page 10-196.](#)

Related information

[DS-5 Debugger commands.](#)

7.2 Working with semihosting

Gives an overview of semihosting support in the debugger.

Semihosting is supported by the debugger in both the command-line console and in Eclipse.

Command-line console

By default all semihosting messages (`stdout` and `stderr`) are output to the console. When using this console interactively with debugger commands you must use the `stdin` command to send input messages (`stdin`) to the application.

Alternatively, you can disable semihosting in the console and use a separate telnet session to interact directly with the application. During start up, the debugger creates a semihosting server socket and displays the port number to use for the telnet session.

Eclipse

The **App Console** view within the DS-5 Debug perspective controls all the semihosting input/output requests (`stdin`, `stdout`, and `stderr`) between the application code and the debugger.

Related references

[4.3 About passing arguments to main\(\) on page 4-105.](#)

[7.1 About semihosting and top of memory on page 7-162.](#)

[7.3 Enabling automatic semihosting support in the debugger on page 7-164.](#)

[7.4 Controlling semihosting messages using the command-line console on page 7-165.](#)

[10.40 Debug Configurations - Arguments tab on page 10-287.](#)

[10.1 App Console view on page 10-196.](#)

Related information

[DS-5 Debugger commands.](#)

7.3 Enabling automatic semihosting support in the debugger

Describes how create an ELF symbol to enable semihosting support in the debugger.

By default, semihosting support is disabled in the debugger. However, the debugger can automatically enable semihosting on supported targets when you load debug information that contains the ELF symbol **__auto_semihosting**.

In C code you can easily create the ELF symbol by defining a function with the name **__auto_semihosting**. To prevent this function generating any additional code or data in your image you can define it as an alias of another function. This places the required ELF symbol in the debug information but does not affect the code and data in the application image.

———— Note —————

Creating a special semihosting ELF symbol is not required if you build your application image using ARM Compiler 5.0 and later. The linker automatically adds this symbol if required.

Create a special semihosting ELF symbol with an alias to main()

```
#include <stdio.h>
void __auto_semihosting(void) __attribute__((alias("main")));
                                     //mark as alias for main() to declare
                                     //semihosting ELF symbol in debug information only

int main(void)
{
    printf("Hello world\n");
    return 0;
}
```

Related references

- [4.3 About passing arguments to main\(\) on page 4-105.](#)
- [7.1 About semihosting and top of memory on page 7-162.](#)
- [7.2 Working with semihosting on page 7-163.](#)
- [7.4 Controlling semihosting messages using the command-line console on page 7-165.](#)
- [10.40 Debug Configurations - Arguments tab on page 10-287.](#)
- [10.1 App Console view on page 10-196.](#)

Related information

[DS-5 Debugger commands.](#)

7.4 Controlling semihosting messages using the command-line console

Describes the debugger commands you can use to control semihosting messages.

You can control input/output requests from application code to a host workstation running the debugger. These are called semihosting messages.

By default, all messages are output to the command-line console but you can choose to redirect them when launching the debugger by using one or more of the following:

--disable_semihosting

Disables all semihosting operations.

--disable_semihosting_console

Disables all semihosting operations to the debugger console.

--semihosting_error=filename

Specifies a file to write `stderr` for semihosting operations.

--semihosting_input=filename

Specifies a file to read `stdin` for semihosting operations.

--semihosting_output=filename

Specifies a file to write `stdout` for semihosting operations.

Related references

[4.3 About passing arguments to main\(\) on page 4-105.](#)

[7.1 About semihosting and top of memory on page 7-162.](#)

[7.2 Working with semihosting on page 7-163.](#)

[7.3 Enabling automatic semihosting support in the debugger on page 7-164.](#)

[10.40 Debug Configurations - Arguments tab on page 10-287.](#)

[10.1 App Console view on page 10-196.](#)

Related information

[DS-5 Debugger commands.](#)

7.5 Controlling the output of logging messages

Describes the debugger commands you can use to control logging messages.

You can control logging messages from the debugger. By default, all messages are output to the **App Console** view but you can control the output and redirection of logging messages by using the `log config` and `log file` debugger commands:

`log config=option`

Specifies the type of logging configuration to output runtime messages from the debugger:

Where:

option

Specifies a predefined logging configuration or a user-defined logging configuration file:

info

Output messages using the predefined INFO level configuration. This is the default.

debug

Output messages using the predefined DEBUG level configuration.

filename

Specifies a user-defined logging configuration file to customize the output of messages. The debugger supports log4j configuration files.

`log file=filename`

Output messages to a file in addition to the console.

Related concepts

[7.6 About Log4j configuration files on page 7-167.](#)

Related tasks

[7.7 Customizing the output of logging messages from the debugger on page 7-168.](#)

Related information

[Log4j in Apache Logging Services.](#)

7.6 About Log4j configuration files

Lists the log4j logging levels you can use to control messages.

In general, the predefined logging configurations provided by the debugger are sufficient for most debugging tasks. However, if you want finer control then you can specify your own customized logging configuration by creating a log4j configuration file. Log4j is an open source logging system for the Java platform and the debugger currently uses version 1.2.

Log4j uses a hierarchy of logging levels to control messages with each level inheriting all lower levels. The following logging levels are currently supported by the debugger:

- DEBUG
- INFO
- WARN
- ERROR
- FATAL

Messages are assigned to a specific logging level and can be redirected to different output locations using one or more of the following log4j components:

Table 7-1 Log4j Components

Component	Description
Logger	Defines the level of logging.
Appender	Defines the output destination.
Layout	Defines the message format.

Related tasks

[7.7 Customizing the output of logging messages from the debugger on page 7-168.](#)

Related references

[7.5 Controlling the output of logging messages on page 7-166.](#)

Related information

[Log4j in Apache Logging Services.](#)

7.7 Customizing the output of logging messages from the debugger

Describes how to create a customized log4j configuration file.

Procedure

1. Create an **Appender** instance for the required logging type. The following types are supported:
 - **ConsoleAppender**
 - **RollingFileAppender**.
2. Suppress the **Threshold** logging level, if required.
3. If the **Appender** instance outputs to a file, define the layout for the **Appender** instance. The following layouts are supported:

PatternLayout

Textual format.

HTMLayout

HTML format.

4. If the **Appender** instance outputs to a file, define the file components. The following components are supported:

File

File name

MaxFileSize

Long integer or string, 10KB.

MaxBackupIndex

Maximum number of log files to use. The default is 1.

5. If you use the layout **PatternLayout**, you can enhance the format of a message by using an additional **ConversionPattern** component. The following patterns are supported:

%c

Logging category

%C

Class name

%d

Date

%F

Filename

%l

Caller location

%L

Line number

%m

Logging message

%M

Method name

%n

End of line character

%p

Logging level. For alignment, you can also supply the number of characters, for example: **%-5p**.

%r

Elapsed time (milliseconds)

%t

Thread name.

6. Define the `name` component for the `Appender` instance, if required.
7. Define the logging level for the `rootLogger` and assign to the required `Appender` instance.
8. To pass the `log4j` configuration file to the debugger you can use:
 - `--log_config=filename` command-line option when launching the debugger from the command-line console.
 - `log config filename` debugger command if the debugger is already running.

Example showing how to log messages to the console

The following example shows how to log messages to the console. This sets the default logging level to `DEBUG`. All the logging for this example is output to the console. However the output of error and warning messages are sent to the error stream, and debug and info messages are sent to the output stream.

```
# Setup logConsole to be a ConsoleAppender
log4j.appender.logConsole=org.apache.log4j.ConsoleAppender
log4j.appender.logConsole.layout=org.apache.log4j.PatternLayout
log4j.appender.logConsole.layout.ConversionPattern=%m%n
log4j.appender.logConsole.name=Console
# Send all DEBUG level logs to the console
log4j.rootLogger=DEBUG, console
```

Example showing how to log messages to a file

The following example shows how to log messages to a file. This sets the default logging level to `DEBUG`. However some packages only write logs at the `INFO` level. All the logging for this example is output to a file. When the file reaches 10MB, it is renamed by adding `.1` file extension and logging continues to write to a new file with the original name. This happens multiple times, but only ten backup files are stored.

```
# Setup logFile to be a RollingFileAppender
log4j.appender.logFile=org.apache.log4j.RollingFileAppender
log4j.appender.logFile.File=output.log
log4j.appender.logFile.MaxFileSize=10MB
log4j.appender.logFile.MaxBackupIndex=10
log4j.appender.logFile.layout=org.apache.log4j.PatternLayout
log4j.appender.logFile.layout.ConversionPattern=%d %-5p %t %c - %m%n
# Send all DEBUG level logs to a file: logFile
log4j.rootLogger=DEBUG, logFile
# Send all INFO level logs in the debug packages to the file: logFile
log4j.logger.com.arm.debug.logging=INFO, logFile
```

Example showing how to combine the logging of messages to the console and a file

The following example shows a combination of the previous examples. This sets the default logging level to `INFO`. All the `INFO` level logging for this example is output to the console. However, a selection of messages are also sending output to two files.

```
# Setup logConsole to be a ConsoleAppender
log4j.appender.logConsole=org.apache.log4j.ConsoleAppender
# Suppress all logs to the console that are lower than the threshold
log4j.appender.logConsole.Threshold=INFO
log4j.appender.logConsole.layout=org.apache.log4j.PatternLayout
log4j.appender.logConsole.layout.ConversionPattern=%m%n
log4j.appender.logConsole.name=Console
# Setup logConnFile to be a RollingFileAppender
log4j.appender.logConnFile=org.apache.log4j.RollingFileAppender
# Suppress all logs to the file that are lower than the threshold
log4j.appender.logConnFile.Threshold.DEBUG
log4j.appender.logConnFile.File=connection.log
log4j.appender.logConnFile.MaxFileSize=10MB
log4j.appender.logConnFile.MaxBackupIndex=10
log4j.appender.logConnFile.layout=org.apache.log4j.PatternLayout
log4j.appender.logConnFile.layout.ConversionPattern=%d %-5p %t %c - %m%n
# Setup logTAccessFile to be a RollingFileAppender
log4j.appender.logTAccessFile=org.apache.log4j.RollingFileAppender
# Suppress all logs to the file that are lower than the threshold
log4j.appender.logTAccessFile.Threshold.DEBUG
```

```
log4j.appender.logTAccessFile.File=target_access.log
log4j.appender.logTAccessFile.MaxFileSize=10MB
log4j.appender.logTAccessFile.MaxBackupIndex=10
log4j.appender.logTAccessFile.layout=org.apache.log4j.PatternLayout
log4j.appender.logTAccessFile.layout.ConversionPattern=%d %-5p %t %c - %m%n
# Send all INFO logs to the console
log4j.rootLogger=INFO, logConsole
# Send all DEBUG logs in the connection package to the file: logConnFile
log4j.logger.com.arm.debug.core.engine.connection=DEBUG, logConnFile
# Send all DEBUG logs in the targetaccess package to the file: logTAccessFile
log4j.logger.com.arm.debug.core.targetaccess.rvi=DEBUG, logTAccessFile
```

Related concepts

[7.6 About Log4j configuration files on page 7-167.](#)

Related references

[7.5 Controlling the output of logging messages on page 7-166.](#)

Related information

[Log4j in Apache Logging Services.](#)

Chapter 8

Debugging with scripts

Describes how to use scripts containing debugger commands to enable you to automate debugging operations.

It contains the following:

- *8.1 Exporting DS-5 Debugger commands generated during a debug session on page 8-172.*
- *8.2 Creating a DS-5 Debugger script on page 8-173.*
- *8.3 Creating a CMM-style script on page 8-174.*
- *8.4 About Jython scripts on page 8-175.*
- *8.5 Jython script concepts and interfaces on page 8-177.*
- *8.6 Creating Jython projects in Eclipse for DS-5™ on page 8-179.*
- *8.7 Creating a Jython script on page 8-182.*
- *8.8 Running a script on page 8-184.*

8.1 Exporting DS-5 Debugger commands generated during a debug session

Shows a typical example of the commands generated by the debugger during a debug sessions.

You can work through a debug session using all the toolbar icons and menu options as required. A full list of all the DS-5 Debugger commands generated during the current debug session is recorded in the **History** view. Before closing Eclipse, you can select the commands that you want in your script file and click on **Export the selected lines as a script file** to save them to a file.

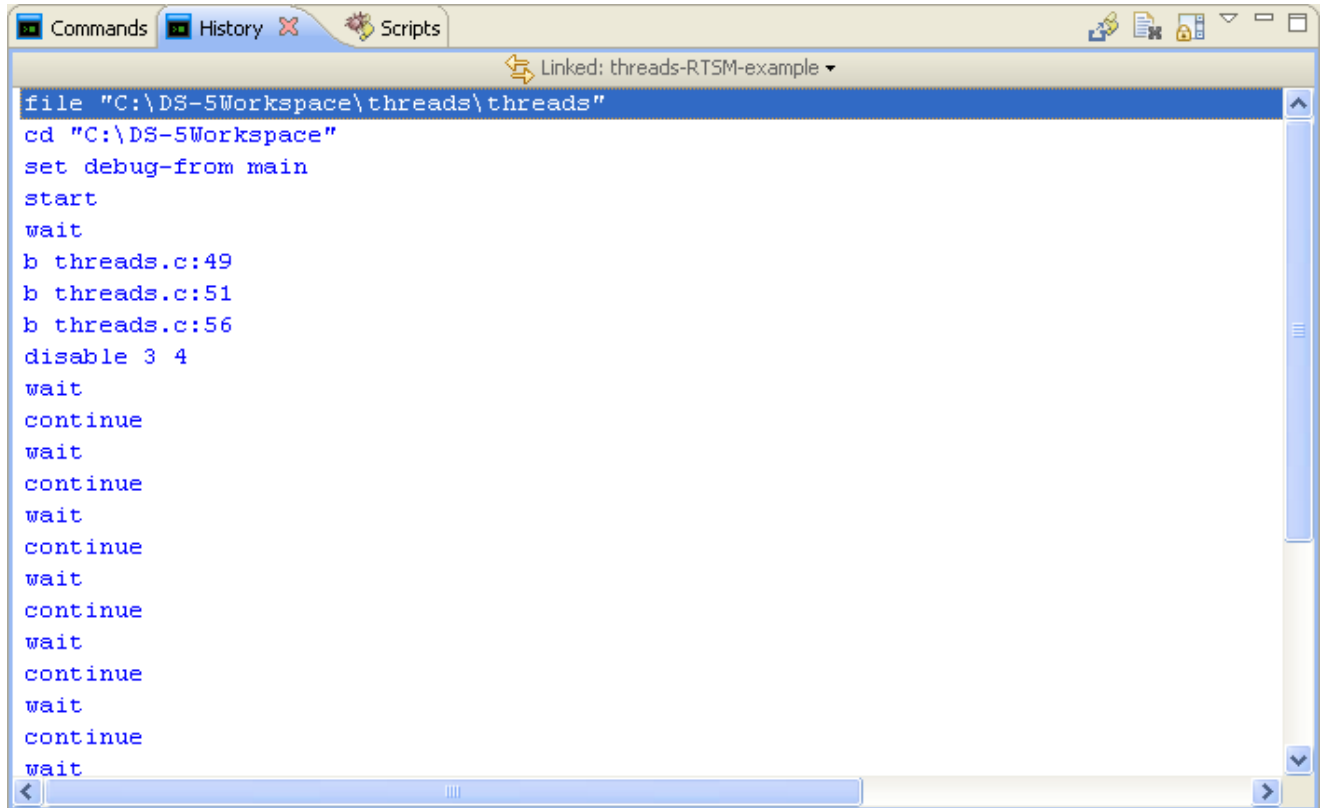


Figure 8-1 Commands generated during a debug session

8.2 Creating a DS-5 Debugger script

Shows a typical example of a DS-5 Debugger script.

The script file must contain only one command on each line. Each command can be identified with comments if required. The `.ds` file extension must be used to identify this type of script.

```
# Filename: myScript.ds
# Initialization commands
load "struct_array.axf"      # Load image
file "struct_array.axf"     # Load symbols
break main                   # Set breakpoint at main()
break *0x814C                 # Set breakpoint at address 0x814C
# Run to breakpoint and print required values
run                           # Start running device
wait 0.5s                    # Wait or time-out after half a second
info stack                   # Display call stack
info registers                # Display info for all registers
# Continue to next breakpoint and print required values
continue                     # Continue running device
wait 0.5s                    # Wait or time-out after half a second
info functions                # Displays info for all functions
info registers                # Display info for all registers
x/3wx 0x8000                 # Display 3 words of memory from 0x8000 (hex)
...
# Shutdown commands
delete 1                     # Delete breakpoint assigned number 1
delete 2                     # Delete breakpoint assigned number 2
```

8.3 Creating a CMM-style script

Shows a typical example of a CMM-style script.

The script file must contain only one command on each line. Each command can be identified with comments if required. The `.cmm` or `.t32` file extension must be used to identify this type of script.

```
// Filename: myScript.cmm
system.up                ; Connect to target and device
data.load.elf "hello.axf" ; Load image and symbols
// Setup breakpoints and registers
break.set main /disable  ; Set breakpoint and immediately disabled
break.set 0x8048          ; Set breakpoint at specified address
break.set 0x8060          ; Set breakpoint at specified address
register.set R0 15         ; Set register R0
register.set PC main       ; Set PC register to symbol address
...
break.enable main         ; Enable breakpoint at specified symbol
// Run to breakpoint and display required values
go                        ; Start running device
var.print "Value is: " myVar ; Display string and variable value
print %h r(R0)            ; Display register R0 in hexadecimal
// Run to breakpoint and print stack
go                        ; Run to next breakpoint
var.frame /locals /caller ; Display all variables and function callers
...
// Shutdown commands
break.delete main         ; Delete breakpoint at address of main()
break.delete 0x8048       ; Delete breakpoint at address
break.delete 0x8060       ; Delete breakpoint at specified address
system.down              ; Disconnect from target
```

8.4 About Jython scripts

Shows a typical example of a Jython script.

Jython is a Java implementation of the Python scripting language. It provides extensive support for data types, conditional execution, loops and organization of code into functions, classes and modules, as well as access to the standard Jython libraries. Jython is an ideal choice for larger or more complex scripts. These are important concepts that are required in order to write a debugger Jython script.

The .py file extension must be used to identify this type of script.

```
# Filename: myScript.py
import sys
from arm_ds.debugger_v1 import Debugger
from arm_ds.debugger_v1 import DebugException
# Debugger object for accessing the debugger
debugger = Debugger()
# Initialisation commands
ec = debugger.getCurrentExecutionContext()
ec.getExecutionService().stop()
ec.getExecutionService().waitForStop()
# in case the execution context reference is out of date
ec = debugger.getCurrentExecutionContext()
# load image if provided in script arguments
if len(sys.argv) == 2:
    image = sys.argv[1]
    ec.getImageService().loadImage(image)
    ec.getExecutionService().setExecutionAddressToEntryPoint()
    ec.getImageService().loadSymbols(image)
    # we can use all the DS commands available
    print "Entry point: ",
    print ec.executeDSCommand("print $ENTRYPOINT")
    # Sample output:
    #      Entry point: $8 = 32768
else:
    pass # assuming image and symbols are loaded
# sets a temporary breakpoint at main and resumes
ec.getExecutionService().resumeTo("main") # this method is non-blocking
try:
    ec.getExecutionService().waitForStop(500) # wait for 500ms
except DebugException, e:
    if e.getErrorCode() == "JYI31": # code of "Wait for stop timed out" message
        print "Waiting timed out!"
        sys.exit()
    else:
        raise # re-raise the exception if it is a different error
ec = debugger.getCurrentExecutionContext()
def getRegisterValue(executionContext, name):
    """Get register value and return string with unsigned hex and signed
    integer, possibly string "error" if there was a problem reading
    the register.
    """
    try:
        value = executionContext.getRegisterService().getValue(name)
        # the returned value behaves like a numeric type,
        # and even can be accessed like an array of bytes, e.g. 'print value[:]'
        return "%s (%d)" % (str(value), int(value))
    except DebugException, e:
        return "error"
# print Core registers on all execution contexts
for i in range(debugger.getExecutionContextCount()):
    ec = debugger.getExecutionContext(i)
    # filter register names starting with "Core::"
    coreRegisterNames = filter(lambda name: name.startswith("Core::"),
                               ec.getRegisterService().getRegisterNames())
    # using Jython list comprehension get values of all these registers
    registerInfo = ["%s = %s" % (name, getRegisterValue(ec, name))
                    for name in coreRegisterNames]
    registers = ", ".join(registerInfo[:3]) # only first three
    print "Identifier: %s, Registers: %s" % (ec.getIdentfier(), registers)
# Output:
#      Identifier: 1, Registers: Core::R0 = 0x00000010 (16), Core::R1 = 0x00000000
#      (0), Core::R2 = 0x0000A4A4 (42148)
#      ...
```

Related tasks

- 8.6.1 Creating a new Jython project in Eclipse for DS-5™ on page 8-179.*
- 8.6.2 Configuring an existing project to use the DS-5™ Jython interpreter on page 8-180.*
- 8.7 Creating a Jython script on page 8-182.*
- 8.8 Running a script on page 8-184.*

Related references

- 8.5 Jython script concepts and interfaces on page 8-177.*
- 10.18 Scripts view on page 10-244.*
- 10.35 Jython Script Parameters dialog box on page 10-275.*

8.5 Jython script concepts and interfaces

Summary of the important debugger interfaces and concepts.

Imports

The debugger module provides a `Debugger` class for initial access to the DS-5 Debugger, with further classes, known as services, to access registers and memory. Here is an example showing the full set of module imports that are typically placed at the top of the jython script:

```
from arm_ds.debugger_v1 import Debugger
from arm_ds.debugger_v1 import DebugException
```

Execution Contexts

Most operations on DS-5 Debugger Jython interfaces require an execution context. The execution context represents the state of the target system. Separate execution contexts exist for each process, thread, or processor that is accessible in the debugger. You can obtain an execution context from the `Debugger` class instance, for example:

```
# Obtain the first execution context
debugger = Debugger()
ec = debugger.getCurrentExecutionContext()
```

Registers

You can access processor registers, coprocessor registers and peripheral registers using the debugger Jython interface. To access a register you must know its name. The name can be obtained from the **Registers** view in the graphical debugger. The `RegisterService` enables you to read and write register values, for a given execution context, for example:

```
# Print the Program Counter (PC) from execution context ec
value = ec.getRegisterService().getValue('PC')
print 'The PC is %s' %value
```

Memory

You can access memory using the debugger Jython interface. You must specify an address and the number of bytes to access. The address and size can be an absolute numeric value or a string containing an expression to be evaluated within the specified execution context. Here is an example:

```
# Print 16 bytes at address 0x0 from execution context ec
print ec.getMemoryService().read(0x0, 16)
```

DS Commands

The debugger jython interface enables you to execute arbitrary DS-5 commands. This is useful when the required functionality is not directly provided in the Jython interface. You must specify the execution context, the command and any arguments that you want to execute. The return value includes the textual output from the command and any errors. Here is an example:

```
# Execute the DS-5 command 'print $ENTRYPOINT' and print the result
print ec.executeDSCommand('print $ENTRYPOINT')
```

Error Handling

The methods on the debugger Jython interfaces throw `DebugException` whenever an error occurs. You can catch exceptions to handle errors in order to provide more information. Here is an example:

```
# Catch a DebugException and print the error message
try:
    ec.getRegisterService().getValue('ThisRegisterDoesNotExist')
except DebugException, de:
    print "Caught DebugException: %s" % (de.getMessage())
```

For more information on the DS-5 Debugger Jython API documentation select **Help Contents** from the **Help** menu.

8.6 Creating Jython projects in Eclipse for DS-5™

To work with Jython scripts in DS-5, the project must use DS-5 Jython as the interpreter. You can either create a new Jython project in Eclipse for DS-5 with DS-5 Jython set as the interpreter or configure an existing project to use DS-5 Jython as the interpreter.

It contains the following:

- [8.6.1 Creating a new Jython project in Eclipse for DS-5™ on page 8-179.](#)
- [8.6.2 Configuring an existing project to use the DS-5™ Jython interpreter on page 8-180.](#)

8.6.1 Creating a new Jython project in Eclipse for DS-5™

Use these instructions to create a new Jython project and select DS-5 Jython as the interpreter.

Procedure

1. Select **File > New > Project...** from the main menu.
2. Expand the **PyDev** group.
3. Select **PyDev Project**.

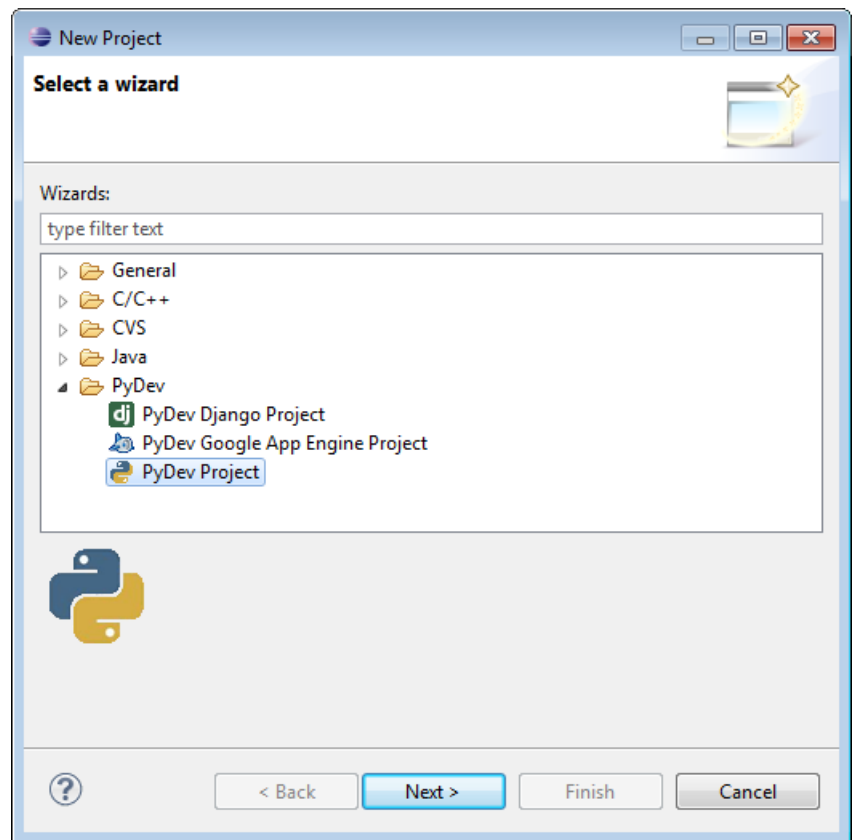


Figure 8-2 PyDev project wizard

4. Click **Next**.
5. Enter the project name and select relevant details:
 - a) In Project name, enter a suitable name for the project.
 - b) In Choose the project type, select **Jython**.
 - c) In Interpreter, select **DS-5 Jython**.

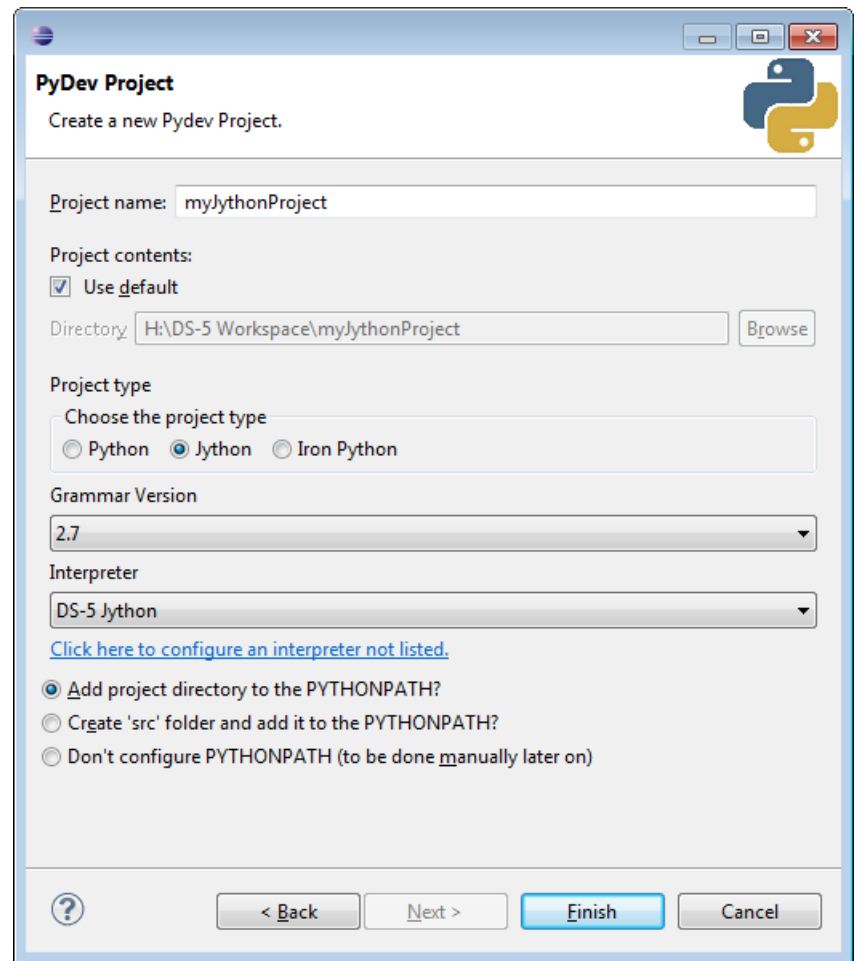


Figure 8-3 PyDev project settings

6. Click **Finish** to create the project.

Related concepts

8.4 About Jython scripts on page 8-175.

Related tasks

8.6.2 Configuring an existing project to use the DS-5™ Jython interpreter on page 8-180.

8.7 Creating a Jython script on page 8-182.

8.8 Running a script on page 8-184.

Related references

8.5 Jython script concepts and interfaces on page 8-177.

10.18 Scripts view on page 10-244.

10.35 Jython Script Parameters dialog box on page 10-275.

8.6.2 Configuring an existing project to use the DS-5™ Jython interpreter

Use these instructions to configure an existing project to use DS-5 Jython as the interpreter.

Procedure

1. In the **Project Explorer** view, right-click the project and select **PyDev > Set as PyDev Project** from the context menu.
2. From the **Project** menu, select **Properties** to display the properties for the selected project.

Note

You can also right-click a project and select **Properties** to display the properties for the selected project.

-
3. In the Properties dialog box, select **PyDev - Interpreter/Grammar**.
 4. In Choose the project type, select **Jython**.
 5. In Interpreter, select **DS-5 Jython**.
 6. Click **OK** to apply these settings and close the dialog box.
 7. Add a Python source file to the project.

Note

The `.py` file extension must be used to identify this type of script.

Related concepts

[8.4 About Jython scripts on page 8-175.](#)

Related tasks

[8.6.1 Creating a new Jython project in Eclipse for DS-5™ on page 8-179.](#)

[8.7 Creating a Jython script on page 8-182.](#)

[8.8 Running a script on page 8-184.](#)

Related references

[8.5 Jython script concepts and interfaces on page 8-177.](#)

[10.18 Scripts view on page 10-244.](#)

[10.35 Jython Script Parameters dialog box on page 10-275.](#)

8.7 Creating a Jython script

Shows a typical workflow for creating and running a Jython script in the debugger.

Procedure

1. Create an empty Jython script file.
2. Right-click the Jython script file and select **Open**.
3. Add the following code to your file in the editor:

```
from arm_ds.debugger_v1 import Debugger
from arm_ds.debugger_v1 import DebugException
```

Note

With this minimal code saved in the file you have access to auto-completion list and online help. ARM recommends the use of this code to explore the Jython interface.

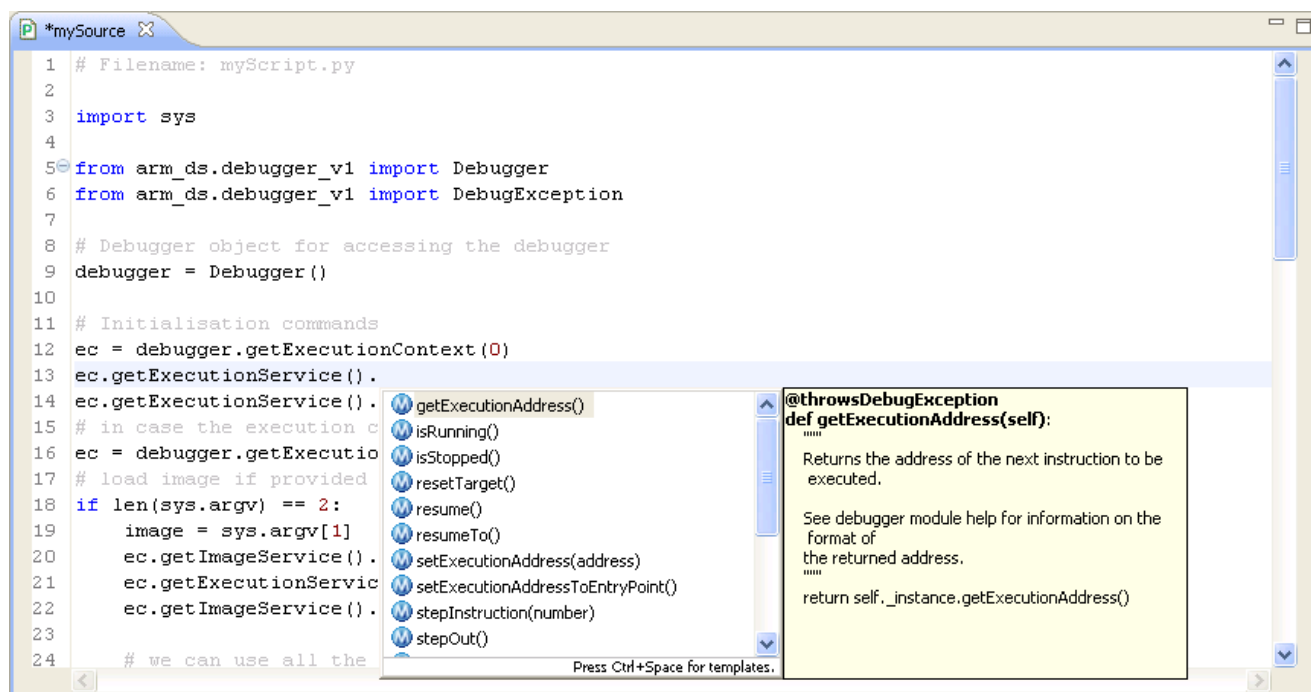


Figure 8-4 Jython auto-completion and help

4. Edit the file to contain the desired scripting commands.
5. Run the script in the debugger.

You can also view an entire Jython interface in the debugger by selecting a debugger object or interface followed by the keyboard and mouse combination **Ctrl+Click**. This opens the source code that implements it.

Related concepts

[8.4 About Jython scripts on page 8-175.](#)

Related tasks

[8.6.1 Creating a new Jython project in Eclipse for DS-5™ on page 8-179.](#)

[8.6.2 Configuring an existing project to use the DS-5™ Jython interpreter on page 8-180.](#)

[8.8 Running a script on page 8-184.](#)

Related references

8.5 Jython script concepts and interfaces on page 8-177.

10.18 Scripts view on page 10-244.

10.35 Jython Script Parameters dialog box on page 10-275.

8.8 Running a script

Describes how to run a script from Eclipse.

Procedure

To run a script from Eclipse:

- You can run a script file immediately after the debugger connects to the target.
 1. Launch Eclipse.
 2. Configure a connection to the target. A DS-5 Debugger configuration can include the option to run a script file immediately after the debugger connects to the target. To do this select the script file in the **Debugger** tab of the DS-5 Debug configuration dialog box.
 3. Connect to the target.
- Run a script file whilst a debug session is in progress.
 - In the **Scripts** view you can use script files:
 1. Import one or more script files in the order that you want them to be executed.
 2. Select the scripts that you want to execute.
 3. Click on the **Execute Selected Scripts** toolbar icon.

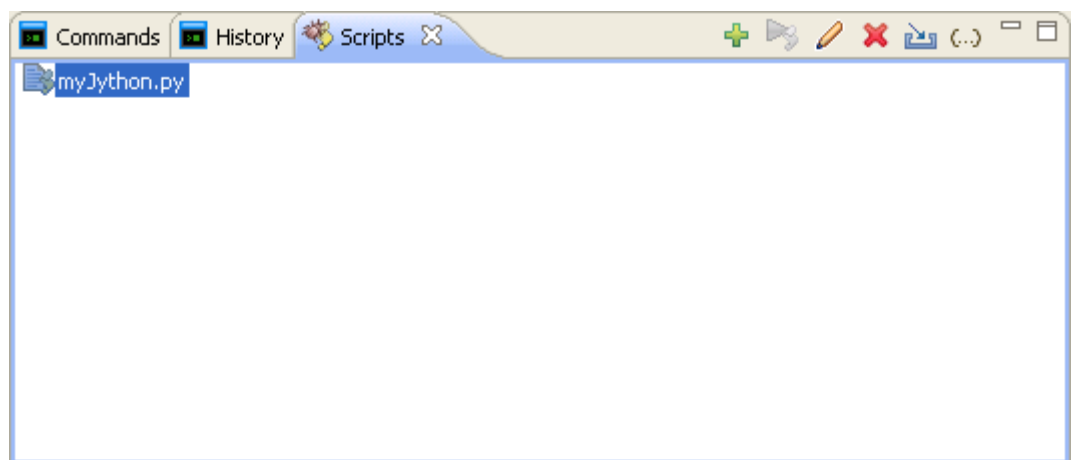


Figure 8-5 Scripts view

— In the **Commands** view you can use the DS-5 Debugger source command.

Related concepts

[8.4 About Jython scripts on page 8-175.](#)

Related tasks

[8.6.1 Creating a new Jython project in Eclipse for DS-5™ on page 8-179.](#)

[8.6.2 Configuring an existing project to use the DS-5™ Jython interpreter on page 8-180.](#)

[8.7 Creating a Jython script on page 8-182.](#)

Related references

[8.1 Exporting DS-5 Debugger commands generated during a debug session on page 8-172.](#)

[8.2 Creating a DS-5 Debugger script on page 8-173.](#)

[8.3 Creating a CMM-style script on page 8-174.](#)

[10.6 Commands view on page 10-209.](#)

[10.12 History view on page 10-226.](#)

10.18 Scripts view on page 10-244.

8.5 Jython script concepts and interfaces on page 8-177.

10.18 Scripts view on page 10-244.

10.35 Jython Script Parameters dialog box on page 10-275.

Related information

DS-5 Debugger commands.

Chapter 9

Working with the Snapshot Viewer

This chapter describes how to work with the Snapshot Viewer.
It contains the following:

- *9.1 About the Snapshot Viewer on page 9-187.*
- *9.2 Components of a Snapshot Viewer initialization file on page 9-189.*
- *9.3 Connecting to the Snapshot Viewer on page 9-192.*
- *9.4 Considerations when creating debugger scripts for the Snapshot Viewer on page 9-193.*

9.1 About the Snapshot Viewer

Use the Snapshot Viewer to analyze a snapshot representation of the application state of a single processor in scenarios where interactive debugging is not possible.

To enable debugging of an application using the Snapshot Viewer, you must have the following data:

- Register Values.
- Memory Values
- Debug Symbols.

If you are unable to provide all of this data, then the level of debug that is available is compromised. Capturing this data is specific to your application, and no tools are provided to help with this. You might have to install exception or signal handlers to catch erroneous situations in your application and dump the required data out.

You must also consider how to get the dumped data from your device onto a workstation that is accessible by the debugger. Some suggestions on how to do this are to:

- Write the data to a file on the host workstation using semihosting.
- Send the data over a UART to a terminal.
- Send the data over a socket using TCP/IP.

Note

Currently DS-5 only supports DS-5 Debugger connections to the Snapshot Viewer using the command-line console.

Register values

Register values are used to emulate the state of the original system at a particular point in time. The most important registers are those in the current processor mode. For example, on an ARMv4 architecture processor these registers are R0-R15 and also the Program Status Registers (PSRs):

- Current Program Status Register (CPSR)
- Application Program Status Register (APSR)
- Saved Program Status Register (SPSR).

Be aware that on many ARM processors, an exception, a data abort, causes a switch to a different processor mode. In this case, you must ensure that the register values you use reflect the correct mode in which the exception occurred, rather than the register values within your exception handler.

If your application uses floating-point data and your device contains vector floating-point hardware, then you must also provide the Snapshot Viewer with the contents of the vector floating-point registers. The important registers to capture are:

- Floating-point Status and Control Register (FPSCR)
- Floating-Point EXception register (FPEXC)
- Single precision registers (S_n)
- Double precision registers (D_n)
- Quad precision registers (Q_n).

Memory values

The majority of the application state is usually stored in memory in the form of global variables, the heap and the stack. Due to size constraints, it is often difficult to provide the Snapshot Viewer

with a copy of the entire contents of memory. In this case, you must carefully consider the areas of memory that are of particular importance.

If you are debugging a crash, the most useful information to find out is often the call stack, because this shows the calling sequence of each function prior to the exception and the values of all the respective function parameters. To show the call stack, the debugger must know the current stack pointer and have access to the contents of the memory that contains the stack. By default, on ARM processors, the stack grows downwards, you must provide the memory starting from the current stack pointer and going up in memory until the beginning of the stack is reached. If you are unable to provide the entire contents of the stack, then a smaller portion starting at the current stack pointer is still useful because it provides the most recent function calls.

If your application uses global (**extern** or file **static**) data, then providing the corresponding memory values enables you to view the variables within the debugger.

If you have local or global variables that point to heap data, then you might want to follow the relevant pointers in the debugger to examine the data. To do this you must have provided the contents of the heap to the Snapshot Viewer. Be aware that heaps can often occupy a large memory range, so it might not be possible to capture the entire heap. The layout of the heap in memory and the data structures that control heap allocation are often specific to the application or the C library, see the relevant documentation for more information.

To debug at the disassembly level, the debugger must have access to the memory values where the application code is located. It is often not necessary to capture the contents of the memory containing the code, because identical data can often be extracted directly from the image using processing tools such as `fromelf`. However, some complications to be aware of are:

- Self-modifying code where the values in the image and memory can vary.
- Dynamic relocation of the memory address within the image at runtime.

Debug symbols

The debugger requires debug information to display high-level information about your application, for example:

- Source code.
- Variable values and types.
- Structures.
- Call stack.

This information is stored by the compiler and linker within the application image, so you must ensure that you have a local debug copy of the same image that you are running on your device. The amount of debug information that is stored in the image, and therefore the resulting quality of your debug session, can be affected by the debug and optimization settings passed to the compiler and linker.

It is common to strip out as much of the debug information as possible when running an image on an embedded device. In such cases, try to use the original unstripped image for debugging purposes.

Related tasks

[9.3 Connecting to the Snapshot Viewer on page 9-192.](#)

Related references

[9.2 Components of a Snapshot Viewer initialization file on page 9-189.](#)

[9.4 Considerations when creating debugger scripts for the Snapshot Viewer on page 9-193.](#)

9.2 Components of a Snapshot Viewer initialization file

Describes the groups and sections used to create a Snapshot Viewer initialization file.

The Snapshot Viewer initialization file is a simple text file consisting of one or more sections that emulate the state of the original system. Each section uses an *option=value* structure.

Note

Currently DS-5 only supports DS-5 Debugger connections to the Snapshot Viewer using the command-line console.

Before creating a Snapshot Viewer initialization file you must ensure that you have:

- One or more binary files containing a snapshot of the application that you want to analyze.

Note

The binary files must be formatted correctly in accordance with the following restrictions.

- Details of the type of processor.
- Details of the memory region addresses and offset values.
- Details of the last known register values.

To create a Snapshot Viewer initialization file, you must add grouped sections as required from the following list and save the file with `.ini` for the file extension.

[global]

A section for global settings. The following option can be used:

core

The selected processor, for example, `core=Cortex-M3`.

[dump]

One or more sections for contiguous memory regions stored in a binary file. The following options can be used:

file

Location of the binary file.

address

Memory start address for the specified region.

length

Length of the region. If none specified then the default is the rest of file from the offset value.

offset

Offset of the specified region from the start of the file. If none specified then the default is zero.

[regs]

A section for standard ARM register names and values, for example, 0x0.

Banked registers can be explicitly specified using their names from the *ARM Architecture Reference Manual*, for example, R13_fiq. In addition, the current mode is determined from the Program Status Registers (PSRs), enabling register names without mode suffixes to be identified with the appropriate banked registers.

The values of the PSRs and PC registers must always be provided. The values of other registers are only required if it is intended to read them from the debugger.

Consider:

```
[regs]
CPSR=0x600000D2 ; IRQ
SP=0x8000
R14_irq=0x1234
```

Reading the registers named SP, R13, or R13_irq all yield the value 0x8000.

Reading the registers named LR, R14, or R14_irq all yield the value 0x1234.

Note

All registers are 32-bits.

Restrictions

The following restrictions apply:

- If you require a global section then it must be the first in the file.
- Consecutive bytes of memory must appear as consecutive bytes in one or more dump files.
- Address ranges representing memory regions must not overlap.

Examples

```
; All sections are optional
[global]
core=Cortex-M3           ; Selected processor
; Location of a contiguous memory region stored in a dump file
[dump]
file="path/dumpfile1.bin" ; File location (full path must be specified)
address=0x8000             ; Memory start address for specific region
length=0x0090             ; Length of region
                        ; (optional, default is rest of file from offset)
; Location of another contiguous memory region stored in a dump file
[dump]
file="path/dumpfile2.bin" ; File location
address=0x8090             ; Memory start address for specific region
offset=0x0024             ; Offset of region from start of file
                        ; (optional, default is 0)

; ARM registers
[regs]
R0=0x000080C8
R1=0x0007C000
R2=0x0007C000
R3=0x0007C000
R4=0x00000363
R5=0x00008EEC
R6=0x00000000
R7=0x00000000
R8=0x00000000
R9=0xB3532737
R10=0x00008DE8
R11=0x00000000
R12=0x00000000
SP=0x0007FFF8
LR=0x0000808D
PC=0x000080B8
```

Related concepts

9.1 About the Snapshot Viewer on page 9-187.

Related tasks

9.3 Connecting to the Snapshot Viewer on page 9-192.

Related references

9.4 Considerations when creating debugger scripts for the Snapshot Viewer on page 9-193.

Related information

ARM Architecture Reference Manual.

9.3 Connecting to the Snapshot Viewer

Describes how to launch the debugger from a command-line console and connect to the Snapshot Viewer.

A Snapshot Viewer provides a virtual target that you can use to analyze a snapshot of a known system state using the debugger.

Prerequisites

Before connecting you must ensure that you have a Snapshot Viewer initialization file containing static information about a target at a specific point in time. For example, the contents of registers, memory and processor state.

Procedure

- Launch the debugger in the command-line console using `--target` command-line option to pass the Snapshot Viewer initialization file to the debugger.

```
debugger --target=int.ini --script=int.cmm
```

Note

Currently DS-5 only supports DS-5 Debugger connections to the Snapshot Viewer using the command-line console.

Related concepts

[9.1 About the Snapshot Viewer on page 9-187.](#)

Related tasks

[1.5 Headless command-line debugger options on page 1-23.](#)

Related references

[9.2 Components of a Snapshot Viewer initialization file on page 9-189.](#)

[9.4 Considerations when creating debugger scripts for the Snapshot Viewer on page 9-193.](#)

[1.8 DS-5 Debugger command-line console keyboard shortcuts on page 1-29.](#)

9.4 Considerations when creating debugger scripts for the Snapshot Viewer

Shows a typical example of an initialization file for use with the Snapshot Viewer.

The Snapshot Viewer uses an initialization file that emulates the state of the original system. The symbols are loaded from the image using the `data.load.elf` command with the `/nocode /noreg` arguments.

The snapshot data and registers are read-only and so the commands you can use are limited.

———— Note ————

Currently DS-5 only supports DS-5 Debugger connections to the Snapshot Viewer using the command-line console.

The following example shows a script using CMM-style commands to analyze the contents of the `types_m3.axf` image.

```
var.print "Connect and load symbols:"
system.up
data.load.elf "types_m3.axf" /nocode /noreg
;Arrays and pointers to arrays
var.print ""
var.print "Arrays and pointers to arrays:"
var.print "Value of i_array[9999] is " i_array[9999]
var.print "Value of *(i_array+9999) is " (*(i_array+9999))
var.print "Value of d_array[1][5] is " d_array[1][5]
var.print "Values of *((*d_array)+9) is " *((*d_array)+9)
var.print "Values of *((*d_array)) is " *((*d_array))
var.print "Value of &d_array[5][5] is " &d_array[5][5]
;Display 0x100 bytes from address in register PC
var.print ""
var.print "Display 0x100 bytes from address in register PC:"
data.dump r(PC)+0x100
;Structures and bit-fields
var.print ""
var.print "Structures and bit-fields:"
var.print "Value of values2.no is " values2.no
var.print "Value of ptr_values->no is " ptr_values->no
var.print "Value of values2.name is " values2.name
var.print "Value of ptr_values->name is " ptr_values->name
var.print "Value of values2.name[0] is " values2.name[0]
var.print "Value of (*ptr_values).name is " (*ptr_values).name
var.print "Value of values2.f1 is " values2.f1
var.print "Value of values2.f2 is " values2.f2
var.print "Value of ptr_values->f1 is " ptr_values->f1
var.print ""
var.print "Disconnect:"
system.down
```

Related concepts

[9.1 About the Snapshot Viewer on page 9-187.](#)

Related tasks

[9.3 Connecting to the Snapshot Viewer on page 9-192.](#)

Related references

[9.2 Components of a Snapshot Viewer initialization file on page 9-189.](#)

Chapter 10

DS-5 Debug perspectives and views

Describes the DS-5 Debug perspective and related views in the Eclipse *Integrated Development Environment* (IDE).

It contains the following:

- [10.1 App Console view on page 10-196.](#)
- [10.2 ARM Asm Info view on page 10-198.](#)
- [10.3 ARM assembler editor on page 10-199.](#)
- [10.4 Breakpoints view on page 10-202.](#)
- [10.5 C/C++ editor on page 10-206.](#)
- [10.6 Commands view on page 10-209.](#)
- [10.7 Debug Control view on page 10-212.](#)
- [10.8 Disassembly view on page 10-216.](#)
- [10.9 Events view on page 10-220.](#)
- [10.10 Expressions view on page 10-221.](#)
- [10.11 Functions view on page 10-224.](#)
- [10.12 History view on page 10-226.](#)
- [10.13 Memory view on page 10-228.](#)
- [10.14 Modules view on page 10-232.](#)
- [10.15 Registers view on page 10-236.](#)
- [10.16 RTOS Data view on page 10-239.](#)
- [10.17 Screen view on page 10-241.](#)
- [10.18 Scripts view on page 10-244.](#)
- [10.19 Target Console view on page 10-246.](#)
- [10.20 Target view on page 10-247.](#)

- *10.21 Trace view on page 10-249.*
- *10.22 Trace Control view on page 10-253.*
- *10.23 Variables view on page 10-255.*
- *10.24 Auto Refresh Properties dialog box on page 10-258.*
- *10.25 Memory Exporter dialog box on page 10-259.*
- *10.26 Memory Importer dialog box on page 10-260.*
- *10.27 Fill Memory dialog box on page 10-261.*
- *10.28 Export trace report dialog box on page 10-262.*
- *10.29 Breakpoint properties dialog box on page 10-264.*
- *10.30 Watchpoint properties dialog box on page 10-269.*
- *10.31 Tracepoint properties dialog box on page 10-270.*
- *10.32 Manage Signals dialog box on page 10-271.*
- *10.33 Event Viewer dialog box on page 10-273.*
- *10.34 Functions Filter dialog box on page 10-274.*
- *10.35 Jython Script Parameters dialog box on page 10-275.*
- *10.36 Debug Configurations - Connection tab on page 10-276.*
- *10.37 Debug Configurations - Files tab on page 10-279.*
- *10.38 Debug Configurations - Debugger tab on page 10-283.*
- *10.39 Debug Configurations - OS Awareness tab on page 10-286.*
- *10.40 Debug Configurations - Arguments tab on page 10-287.*
- *10.41 Debug Configurations - Environment tab on page 10-289.*
- *10.42 DTSL Configuration Editor dialog box on page 10-291.*
- *10.43 Configuration database panel on page 10-293.*
- *10.44 About the Remote System Explorer on page 10-295.*
- *10.45 Remote Systems view on page 10-296.*
- *10.46 Remote System Details view on page 10-297.*
- *10.47 Target management terminal for serial and SSH connections on page 10-298.*
- *10.48 Remote Scratchpad view on page 10-299.*
- *10.49 Remote Systems terminal for SSH connections on page 10-300.*
- *10.50 New Terminal Connection dialog box on page 10-301.*
- *10.51 DS-5 Debugger menu and toolbar icons on page 10-303.*

10.1 App Console view

Describes the view content.

This view enables you to interact with the console I/O capabilities provided by the semihosting implementation in the ARM C libraries. To use this feature, semihosting support must be enabled in the debugger.

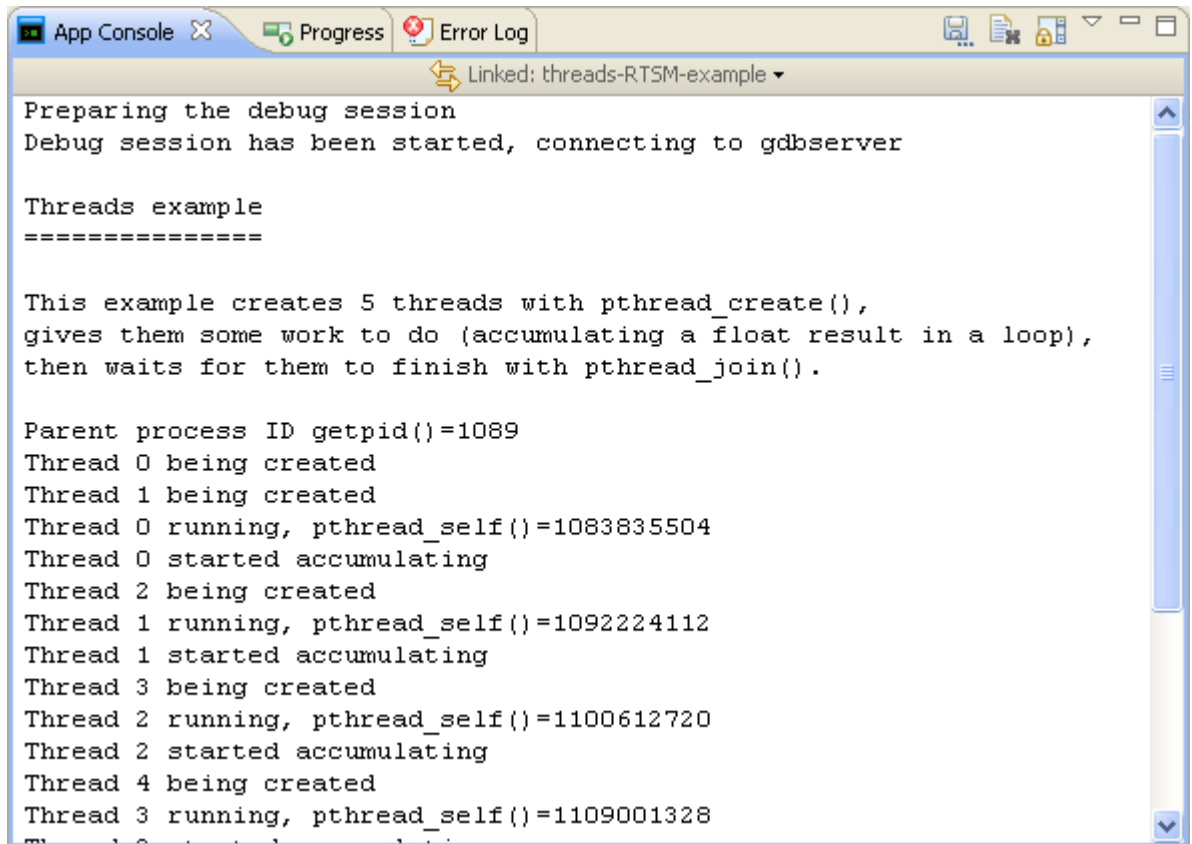


Figure 10-1 App Console view

———— Note ————

Default settings for this view are controlled by a DS-5 Debugger setting in the Preferences dialog box. For example, default locations for specific files or the maximum number of lines to display. You can access these settings by selecting **Preferences** from the **Window** menu.

Toolbar and context menu options

The following options are available from the toolbar or context menu:

Linked: context

Links this view to the selected connection in the **Debug Control** view. This is the default. Alternatively you can link the view to a specific connection. If the connection you want is not shown in the drop-down list you might have to select it first in the **Debug Control** view.

Save Console Buffer

Saves the contents of the **App Console** view to a text file.

Clear Console

Clears the contents of the **App Console** view.

Scroll Lock

Enables or disables the automatic scrolling of messages in the **App Console** view.

View Menu

This menu contains the following option:

New App Console View

Displays a new instance of the **App Console** view.

Bring to Front for Write

If enabled, the debugger automatically changes the focus to this view when a semihosting application prompts for input.

Copy

Copies the selected text.

Paste

Pastes text that you have previously copied. You can paste text only when the application displays a semihosting prompt.

Select All

Selects all text.

Related references

[7.1 About semihosting and top of memory on page 7-162.](#)

[7.2 Working with semihosting on page 7-163.](#)

[7.3 Enabling automatic semihosting support in the debugger on page 7-164.](#)

[10 DS-5 Debug perspectives and views on page 10-194.](#)

10.2 ARM Asm Info view

Describes the view content.

This view enables you to view more information on an ARM or Thumb® instruction or directive.

When you are editing assembly language source files (.s) using the ARM assembler editor, you can access more information by:

1. selecting an instruction or directive
2. pressing **F3**.

The related documentation is displayed in the **ARM Asm Info** view. The **ARM Asm Info** view is automatically displayed when you press **F3**.

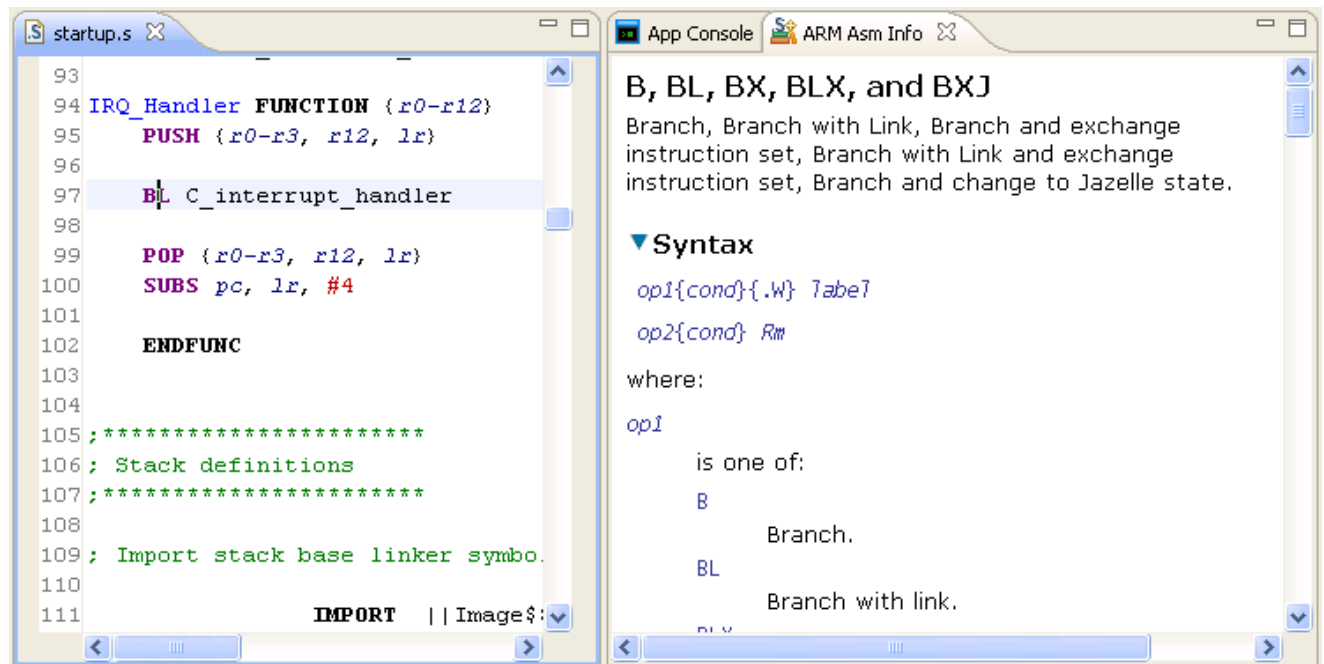


Figure 10-2 ARM Asm Info view

To manually add this view:

1. Ensure that you are in the DS-5 Debug perspective.
2. Select **Window > Show View > Other...** to open the Show View dialog box.
3. Select the **ARM Asm Info** view from the **DS-5 Debugger** group.

Related references

[10 DS-5 Debug perspectives and views on page 10-194.](#)

[10.3 ARM assembler editor on page 10-199.](#)

10.3 ARM assembler editor

Describes the view content.

The ARM assembler editor provides syntax highlighting, formatting of code and content assistance for labels in ARM assembly language source files. This editor enables you to:

- edit source code
- view the syntax highlighting
- select an instruction or directive and press **F3** to view the related ARM assembler reference information
- use content assist for auto-completion
- set, remove, enable or disable a breakpoint
- set or remove a trace start or stop point
- set or remove a Streamline start or stop capture.

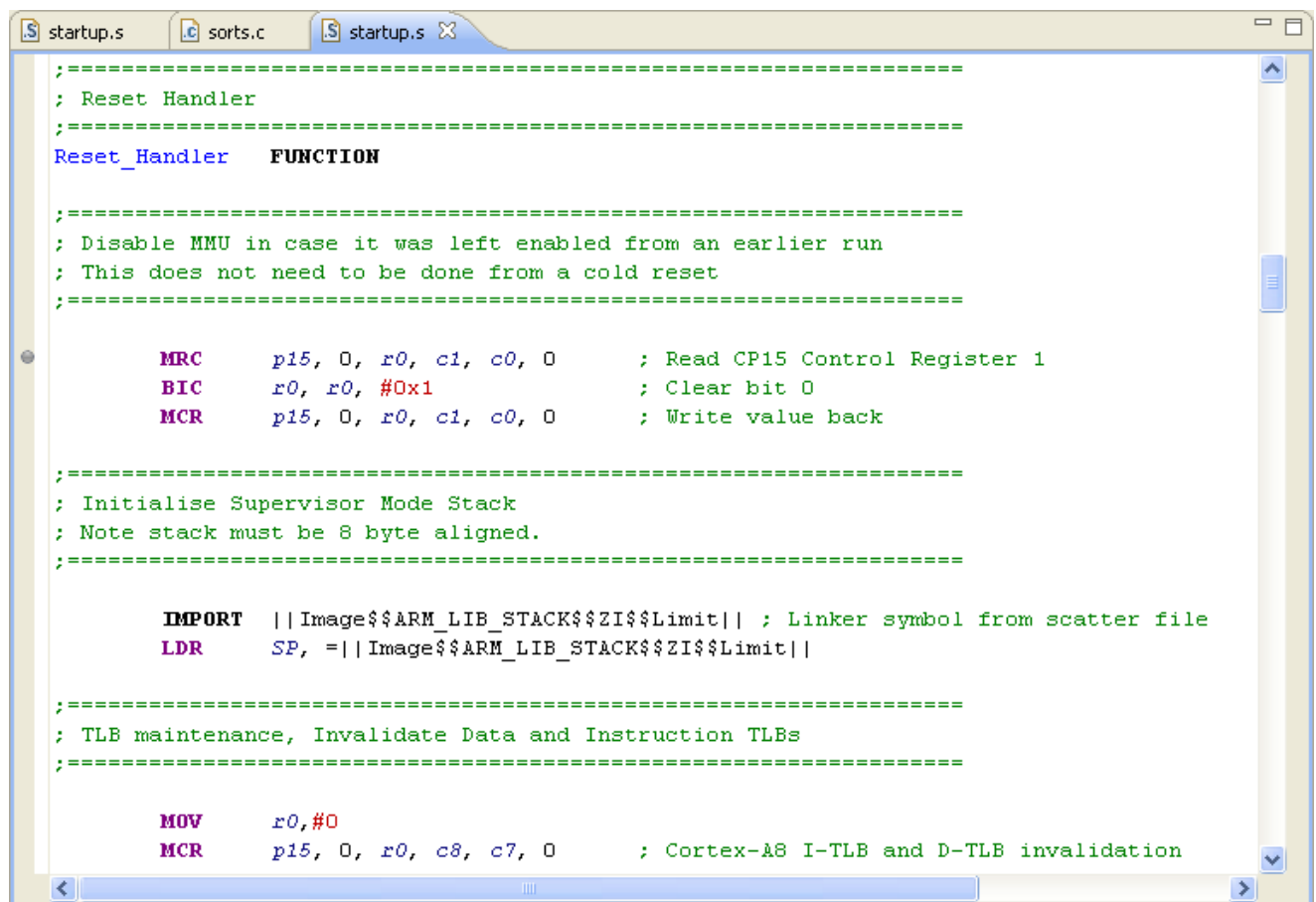


Figure 10-3 ARM assembler editor

In the left-hand margin of each editor tab you can find a marker bar that displays view markers associated with specific lines in the source code.

To set a breakpoint, double-click in the marker bar at the position where you want to set the breakpoint. To delete a breakpoint, double-click on the breakpoint marker.

Action context menu options

Right-click in the marker bar, or the line number column if visible, to display the action context menu for the ARM assembler editor. The options available include:

DS-5 Breakpoints menu

The following breakpoint options are available:

Toggle Breakpoint

Adds a new breakpoint, or remove a selected breakpoint.

Disable Breakpoint, Enable Breakpoint

Disables or enables the selected breakpoint.

Breakpoint Properties...

Displays the Breakpoint Properties dialog box for the selected breakpoint. This enables you to control breakpoint activation.

Toggle Trace Start Point

Sets or removes a trace start point at the selected address.

Toggle Trace Stop Point

Sets or removes a trace stop point at the selected address.

Toggle Trace Trigger Point

Starts a trace trigger point at the selected address.

Toggle Streamline Start

Sets or removes a Streamline start capture at the selected address.

Toggle Streamline Stop

Sets or removes a Streamline stop capture at the selected address.

Default Breakpoint Type

The following breakpoint options are available:

C/C++ Breakpoints

Select to use the C/C++ perspective breakpoint scheme.

DS-5 C/C++ Breakpoint

Select to use the DS-5 Debug perspective breakpoint scheme. This is the default for the DS-5 Debug perspective.

DS-5 breakpoint markers are red to distinguish them from the blue C/C++ perspective breakpoint markers.

————— Note —————

The **Default Breakpoint Type** selected causes the top-level **Toggle Breakpoint** menu in this context menu and the double-click action in the left-hand ruler to toggle either CDT Breakpoints or DS-5 Breakpoints. This menu is also available from the **Run** menu in the main menu bar at the top of the C/C++, Debug, and DS-5 Debug perspectives.

The menu options under **DS-5 Breakpoints** do not honor this setting and always refer to DS-5 Breakpoints.

Show Line Numbers

Show or hide line numbers.

For more information on the other options not listed here, see the dynamic help.

Related references

Working with data watchpoints.

4.8 Setting a tracepoint on page 4-119.

4.6 Working with conditional breakpoints on page 4-114.

4.6.1 Assigning conditions to an existing breakpoint on page 4-114.

4.7 About pending breakpoints and watchpoints on page 4-118.

- [*4.5.5 Exporting DS-5™ breakpoint settings to a file on page 4-112.*](#)
- [*4.5.4 Importing DS-5™ breakpoint settings from a file on page 4-111.*](#)
- [*4.9 Setting Streamline start and stop points on page 4-120.*](#)
- [*10 DS-5 Debug perspectives and views on page 10-194.*](#)
- [*5.1 Examining the target execution environment on page 5-130.*](#)
- [*5.2 Examining the call stack on page 5-132.*](#)
- [*10.2 ARM Asm Info view on page 10-198.*](#)

10.4 Breakpoints view

Describes the view content.

This view enables you to:

- disable, enable, or delete breakpoints and watchpoints
- import or export a list of breakpoints and watchpoints
- display the source file containing the line of code where the selected breakpoint is set
- display the disassembly where the selected breakpoint is set
- display the memory where the selected watchpoint is set
- delay breakpoint activation by setting properties for the breakpoint
- control the handling and output of messages for all Unix signals and processor exception handlers
- change the access type for the selected watchpoint.

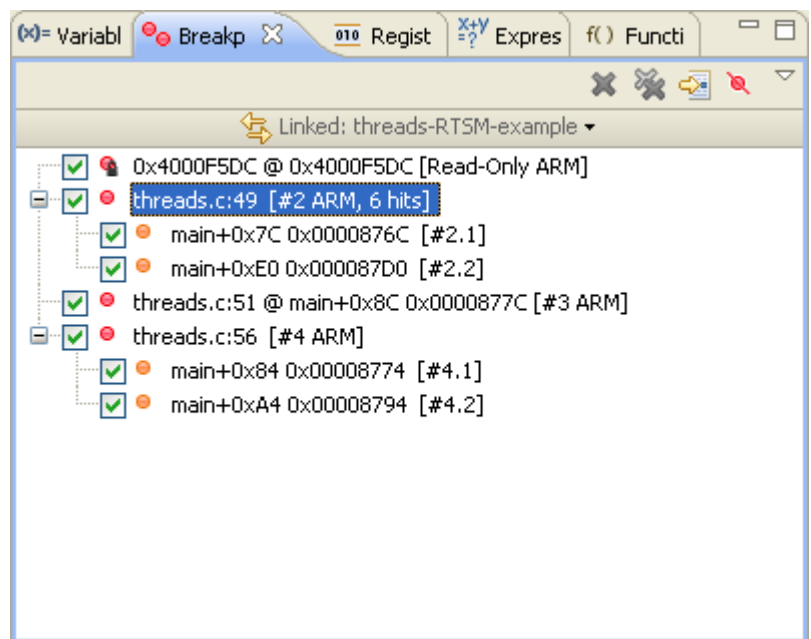


Figure 10-4 Breakpoints view

Syntax

Syntax of a breakpoint entry

A breakpoint entry has the following syntax:

```
source_file:linenum @ function+offset address [#ID instruction_type, ignore = num/
count, nHits hits, (condition)]
```

where:

source_file:linenum

If the source file is available, the file name and line number in the file where the breakpoint is set, `threads.c:115`.

function+offset

The name of the function in which the breakpoint is set and the number of bytes from the start of the function. For example, `accumulate()+52` shows that the breakpoint is 52 bytes from the start of the `accumulate()` function.

address

The address where the breakpoint is set.

ID

The breakpoint ID number, **#N**. In some cases, such as in a **for** loop, a breakpoint might comprise a number of sub-breakpoints. These are identified as **N.n**, where **N** is the number of the parent. The description of a sub-breakpoint in this dialog box is shown as

`main()+132sub-breakpoint ofmain()+132 @ threads.c:56 [#14 ARM]
(threads)`

instruction_type

The type of instruction at the address of the breakpoint, ARM or Thumb.

ignore = num/count

An **ignore** count if set, where:

num equals **count** initially, and decrements on each pass until it reaches zero.

count is the value you have specified for the **ignore** count.

nHits hits

A counter that increments each time the breakpoint is hit. This is not displayed until the first hit. If you set an **ignore** count, **hits** count does not start incrementing until the **ignore** count reaches zero.

condition

The stop condition you have specified, (**i==3**).

Syntax

Syntax of a watchpoint entry

A watchpoint entry has the following syntax:

```
name type[#ID]
```

where:

name

The name of the variable where the watchpoint is set.

type

The access type of the watchpoint.

ID

The watchpoint ID number.

Syntax

Syntax of a tracepoint entry

A tracepoint entry has the following syntax:

```
source_file:linenum address
```

where:

address

The address where the tracepoint is set.

source_file:linenum

If the source file is available, the file name and line number in the file where the tracepoint is set, `0x80000A72`.

Toolbar and context menu options

The following options are available from the toolbar or context menu:

Linked: context

Links this view to the selected connection in the **Debug Control** view. This is the default. Alternatively you can link the view to a specific connection. If the connection you want is not shown in the drop-down list you might have to select it first in the **Debug Control** view.

Delete

Removes the selected breakpoints and watchpoints.

Delete All

Removes all breakpoints and watchpoints.

Go to File

Displays the source file containing the line of code where the selected breakpoint is set. This option is disabled for a watchpoint.

Go to Disassembly

Displays the disassembly where the selected breakpoint is set. This option is disabled for a watchpoint.

Go to Memory

Displays the memory where the selected watchpoint is set. This option is disabled for a breakpoint.

Skip All Breakpoints

Deactivates all breakpoints or watchpoints that are currently set. The debugger remembers the enabled and disabled state of each breakpoint or watchpoint, and restores that state when you reactivate them again.

Enable Breakpoints

Enables the selected breakpoints and watchpoints.

Disable Breakpoints

Disables the selected breakpoints and watchpoints.

Resolve

Re-evaluates the address of the selected breakpoint or watchpoint. If the address can be resolved the breakpoint or watchpoint is set, otherwise it remains pending.

Properties...

Displays the Properties dialog box for the selected breakpoint, watchpoint or tracepoint. This enables you to control activation or change the access type for the selected watchpoint.

Copy

Copies the selected breakpoints and watchpoints. You can also use the standard keyboard shortcut to do this.

Paste

Pastes the copied breakpoints and watchpoints. The breakpoints or watchpoints are enabled by default. You can also use the standard keyboard shortcut to do this.

Select all

Selects all breakpoints or watchpoints. You can also use the standard keyboard shortcut to do this.

View Menu

The following **View Menu** options are available:

New Breakpoints View

Displays a new instance of the **Breakpoints** view.

Export Breakpoints

Exports the current list of breakpoints and watchpoints to a file.

Import Breakpoints

Imports a list of breakpoints and watchpoints from a file.

Alphanumeric Sort

Sorts the list alphanumerically based on the string displayed in the view.

Ordered Sort

Sorts the list in the order they have been set.

Manage Signals

Displays the Manage Signal dialog box.

Related concepts

- [6.7 About debugging multi-threaded applications on page 6-145.](#)
- [6.8 About debugging shared libraries on page 6-146.](#)
- [6.9 About debugging a Linux kernel on page 6-148.](#)
- [6.10 About debugging Linux kernel modules on page 6-150.](#)
- [6.12 About debugging TrustZone enabled targets on page 6-153.](#)

Related references

- [Working with data watchpoints.](#)
- [4.8 Setting a tracepoint on page 4-119.](#)
- [4.6 Working with conditional breakpoints on page 4-114.](#)
- [4.6.1 Assigning conditions to an existing breakpoint on page 4-114.](#)
- [4.7 About pending breakpoints and watchpoints on page 4-118.](#)
- [4.5.5 Exporting DS-5™ breakpoint settings to a file on page 4-112.](#)
- [4.5.4 Importing DS-5™ breakpoint settings from a file on page 4-111.](#)
- [10 DS-5 Debug perspectives and views on page 10-194.](#)
- [5.1 Examining the target execution environment on page 5-130.](#)
- [5.2 Examining the call stack on page 5-132.](#)

10.5 C/C++ editor

Describes the view content.

This editor enables you to:

- Edit source code.
- View the syntax highlighting.
- View interactive help when hovering over C library functions. For example, `printf()`.
- Use content assist (**Ctrl+Space**) for auto-completion.
- Set, remove, enable or disable a breakpoint.
- Set or remove a trace start or stop point.
- Set or remove a Streamline start or stop capture.

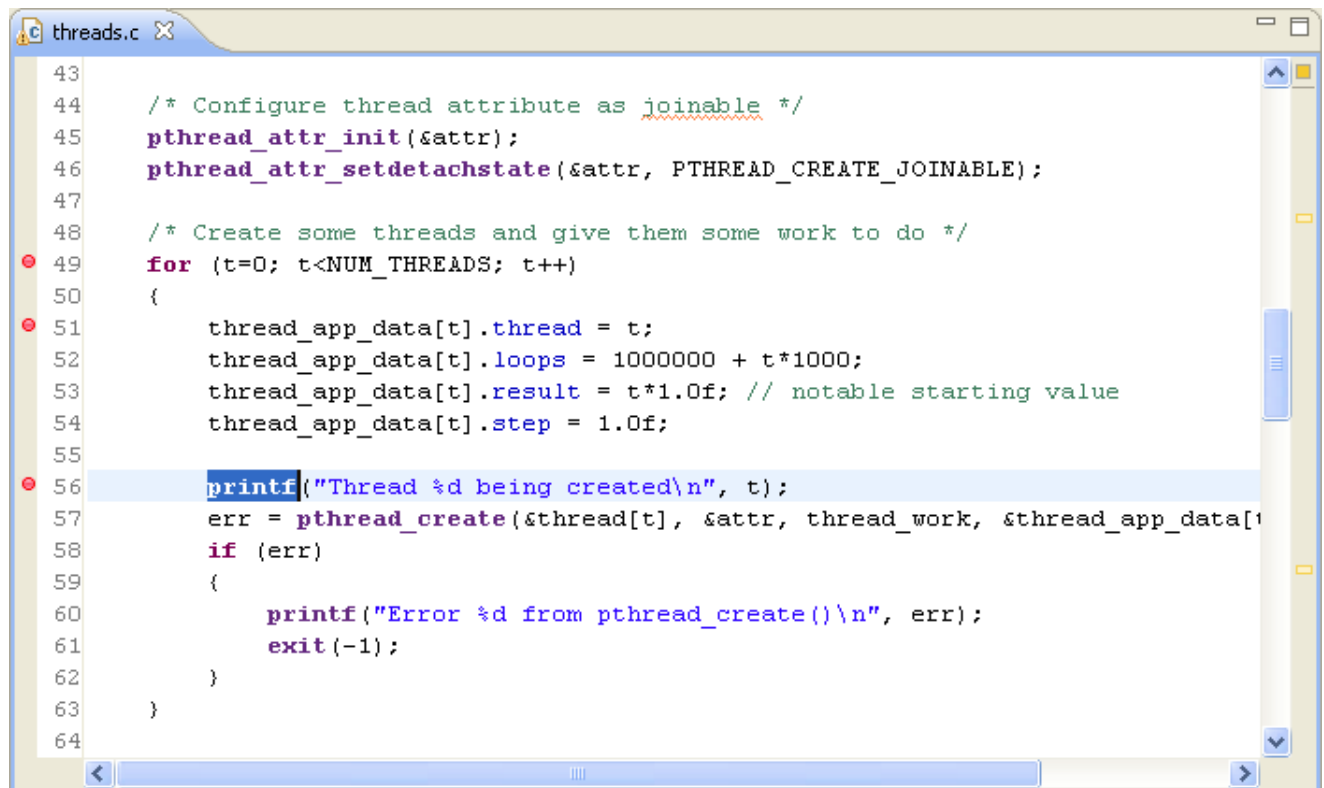


Figure 10-5 C/C++ editor

In the left-hand margin of each editor tab you can find a marker bar that displays view markers associated with specific lines in the source code.

To set a breakpoint, double-click in the marker bar at the position where you want to set the breakpoint. To delete a breakpoint, double-click on the breakpoint marker.

———— Note ————

If you have sub-breakpoints to a parent breakpoint then double-clicking on the marker also deletes the related sub-breakpoints.

Action context menu options

Right-click in the marker bar, or the line number column if visible, to display the action context menu for the C/C++ editor. The options available include:

DS-5 Breakpoints menu

The following breakpoint options are available:

Toggle Breakpoint

Sets or removes a breakpoint at the selected address.

Toggle Hardware Breakpoint

Sets or removes a hardware breakpoint at the selected address.

Resolve Breakpoint

Resolves a pending breakpoint at the selected address.

Enable Breakpoint

Enables the breakpoint at the selected address.

Disable Breakpoint

Disables the breakpoint at the selected address.

Breakpoint Properties...

Displays the Breakpoint Properties dialog box for the selected breakpoint. This enables you to control breakpoint activation.

Toggle Trace Start Point

Sets or removes a trace start point at the selected address.

Toggle Trace Stop Point

Sets or removes a trace stop point at the selected address.

Toggle Trace Trigger Point

Starts a trace trigger point at the selected address.

Toggle Streamline Start

Sets or removes a Streamline start capture at the selected address.

Toggle Streamline Stop

Sets or removes a Streamline stop capture at the selected address.

Default Breakpoint Type

The default type causes the top-level context menu entry, **Toggle Breakpoint** and the double-click action in the marker bar to toggle either CDT Breakpoints or DS-5 Breakpoints. When using DS-5 Debugger you must select **DS-5 C/C++ Breakpoint**. DS-5 breakpoint markers are red to distinguish them from the blue CDT breakpoint markers.

Show Line Numbers

Shows or hides line numbers.

For more information on the other options not listed here, see the dynamic help.

Editor context menu

Right-click on any line of source to display the editor context menu for the C/C++ editor. The following options are enabled when you connect to a target:

Set PC to Selection

Sets the PC to the address of the selected source line.

Run to Selection

Runs to the selected source line.

Show in Disassembly

This option:

1. Opens a new instance of the Disassembly view.
2. Highlights the addresses and instructions associated with the selected source line. A vertical bar and shaded highlight shows the related disassembly.

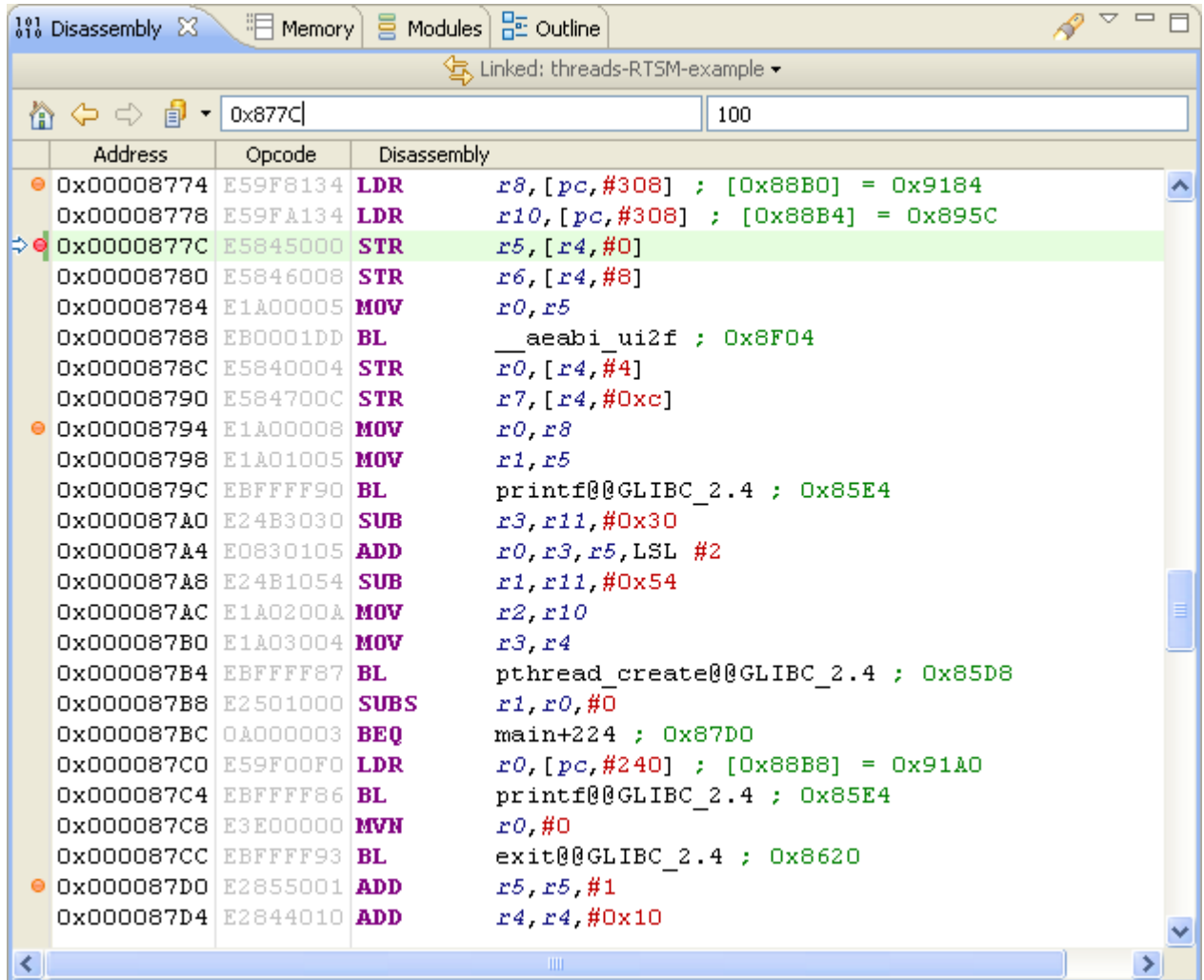


Figure 10-6 Show disassembly for selected source line

For more information on the other options not listed here, see the dynamic help.

Related references

Working with data watchpoints.

4.8 Setting a tracepoint on page 4-119.

4.6 Working with conditional breakpoints on page 4-114.

4.6.1 Assigning conditions to an existing breakpoint on page 4-114.

4.7 About pending breakpoints and watchpoints on page 4-118.

4.5.5 Exporting DS-5™ breakpoint settings to a file on page 4-112.

4.5.4 Importing DS-5™ breakpoint settings from a file on page 4-111.

4.9 Setting Streamline start and stop points on page 4-120.

10 DS-5 Debug perspectives and views on page 10-194.

10.6 Commands view

Describes the view content.

This view enables you to:

- Enter debugger commands.
- Run command scripts.
- See messages output by the debugger.
- Save the contents to a text file.

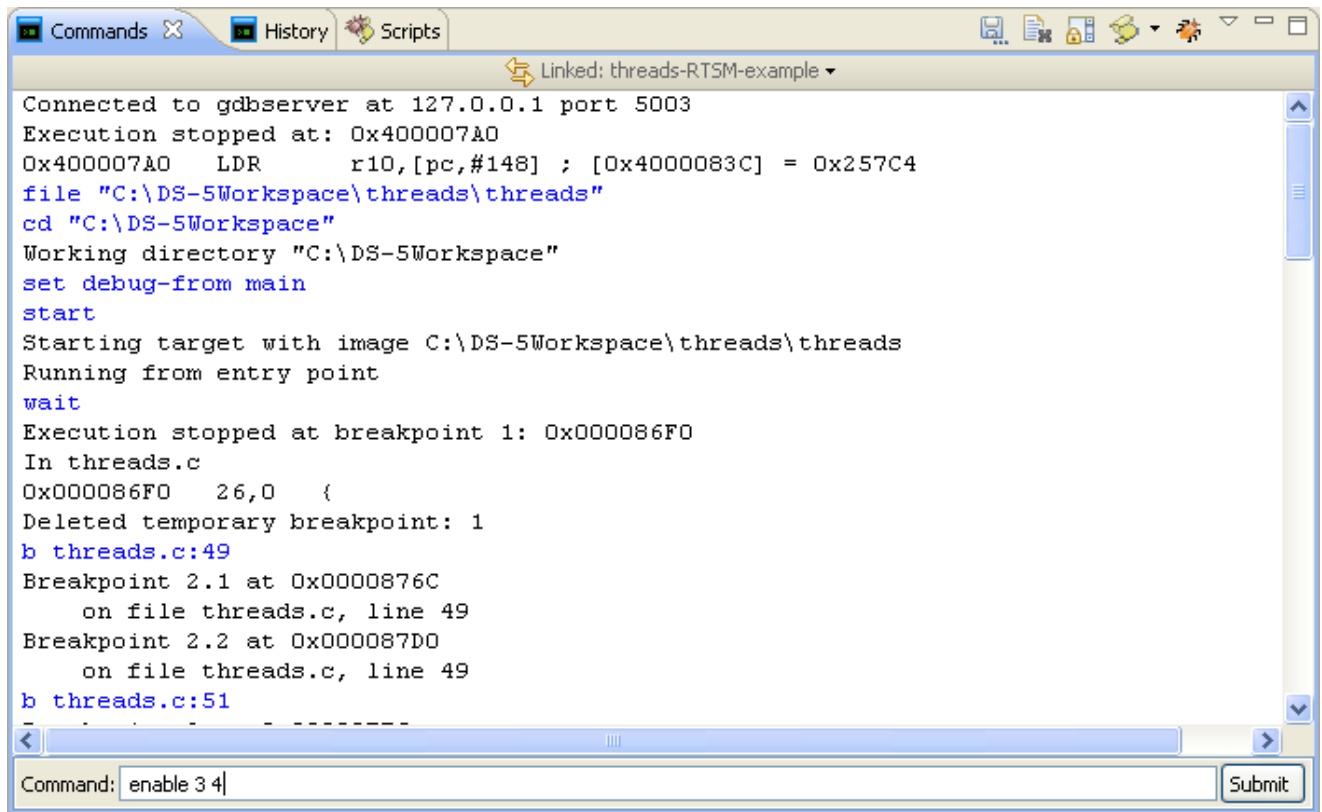


Figure 10-7 Commands view

You can execute DS-5 Debugger commands by entering the command in the field provided, then click **Submit**.

———— Note ————

This feature is not available until you connect to a target.

You can also use content assist keyboard combinations provided by Eclipse to display a list of DS-5 Debugger commands. Filtering is also possible by entering a partial command. For example, enter `pr` followed by the content assist keyboard combination to search for the `print` command.

To display sub-commands, you must filter on the top level command. For example, enter `info` followed by the content assist keyboard combination to display all the `info` sub-commands.

See DS-5 Debug perspective keyboard shortcuts in the Related reference section for details about specific content assist keyboard combinations available in DS-5 Debugger.

Note

Default settings for this view are controlled by a DS-5 Debugger setting in the Preferences dialog box. For example, default locations for specific files or the maximum number of lines to display. You can access these settings by selecting **Preferences** from the **Window** menu.

Toolbar and context menu options

The following options are available from the toolbar or context menu:

Linked: **context**

Links this view to the selected connection in the **Debug Control** view. This is the default. Alternatively you can link the view to a specific connection. If the connection you want is not shown in the drop-down list you might have to select it first in the **Debug Control** view.

Save Console Buffer

Saves the contents of the **Commands** view to a text file.

Clear Console

Clears the contents of the **Commands** view.

Scroll Lock

Enables or disables the automatic scrolling of messages in the **Commands** view.

Script menu

A menu of options that enable you to manage and run command scripts:

<Recent scripts list>

A list of the recently run scripts.

<Recent favorites list>

A list of the scripts you have added to your favorites list.

Run Script File...

Displays the Open dialog box to select and run a script file.

Organize Favorites...

Displays the **Scripts** view, where you can organize your scripts.

Show Command History View

Displays the **History** view.

Copy

Copies the selected commands. You can also use the standard keyboard shortcut to do this.

Paste

Pastes the command that you have previously copied into the Command field. You can also use the standard keyboard shortcut to do this.

Select all

Selects all output in the **Commands** view. You can also use the standard keyboard shortcut to do this.

Save the selected lines as a script...

Displays the Save As dialog box to save the selected commands to a script file.

When you click **Save** on the Save As dialog box, you are given the option to add the script file to your favorites list. Click **OK** to add the script to your favorites list. Favorites are displayed in the **Scripts** view.

Execute selected lines

Runs the selected commands.

New Commands View

Displays a new instance of the **Commands** view.

Related concepts

[6.7 About debugging multi-threaded applications on page 6-145.](#)

- 6.8 About debugging shared libraries on page 6-146.*
- 6.9 About debugging a Linux kernel on page 6-148.*
- 6.10 About debugging Linux kernel modules on page 6-150.*
- 6.12 About debugging TrustZone enabled targets on page 6-153.*

Related references

- 1.7 DS-5 Debug perspective keyboard shortcuts on page 1-28.*
- 10.7 Debug Control view on page 10-212.*
- Working with data watchpoints.*
- 4.8 Setting a tracepoint on page 4-119.*
- 4.6 Working with conditional breakpoints on page 4-114.*
- 4.6.1 Assigning conditions to an existing breakpoint on page 4-114.*
- 4.7 About pending breakpoints and watchpoints on page 4-118.*
- 4.5.5 Exporting DS-5™ breakpoint settings to a file on page 4-112.*
- 4.5.4 Importing DS-5™ breakpoint settings from a file on page 4-111.*
- 10 DS-5 Debug perspectives and views on page 10-194.*

Related information

- DS-5 Debugger commands.*

10.7 Debug Control view

Describes the view content.

This view enables you to: The **Debug Control** view displays target connections with a hierarchical layout of running threads, user space processes, and related call stacks. Call stack information is gathered when the system is stopped.

- view a list of running threads and user space processes as applicable
- view the call stack showing stack elements for each thread or process as applicable
- connect to and disconnect from a target
- load an application image on to the target
- load debug information when required by the debugger
- start the application
- run the application
- stop the application
- reset the target
- continue running the application after a breakpoint is hit or the target is suspended
- control the execution of an image by sequentially stepping through an application at the source or instruction level
- modify the search paths used by the debugger when it executes any of the commands that look up and display source code
- set the current working directory.

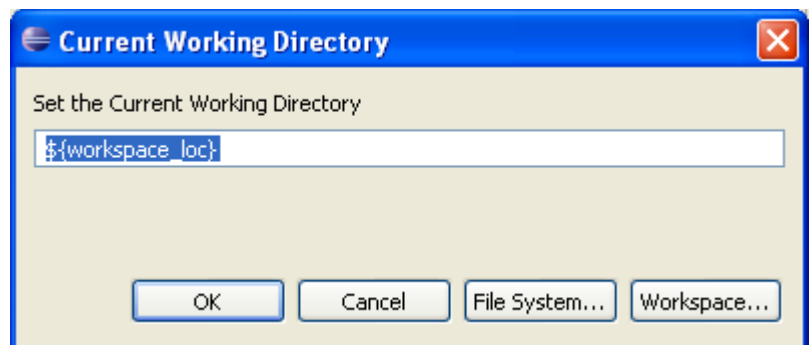


Figure 10-8 Set the current working directory

On Linux Kernel connections, the hierarchical nodes **Active Threads** and **All Threads** are displayed. **Active Threads** shows each thread that is currently scheduled on a processor. **All Threads** shows every thread in the system, including those presently scheduled on a processor. On **gdbserver** connections, the hierarchical nodes **Active Threads** and **All Threads** are displayed, but the scope is limited to the application under debug. **Active Threads** shows only application threads that are currently scheduled. **All Threads** shows all application threads, including ones that are currently scheduled.

Some of the views in the DS-5 Debug perspective are associated with the currently selected stack frame. Other views are associated with editors or target connections. Each associated view is synchronized accordingly.

Connection states are identified with different icons and background highlighting and are also displayed in the view status bar. The following example shows a connection in the connected state. If you want to add another configuration to the view then you can use the **Debug Control** view menu.

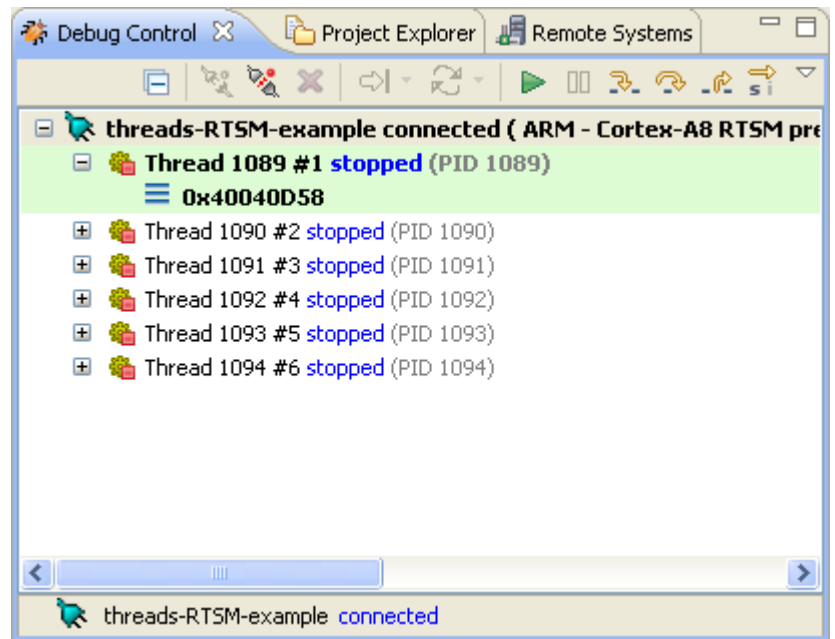


Figure 10-9 Debug Control view

Toolbar and context menu options

The following options are available from the toolbar or context menu:

Collapse All

Collapses all expanded stack trace configurations.

Connect to Target

Connects to the selected target using the same launch configuration settings as the previous connection.

Disconnect from Target

Disconnects from the selected target.

Debug Configurations...

Displays the Debug Configurations dialog box, with the configuration for the selected connection displayed.

Launch in background

Displays the Progress Information dialog box if disabled.

Remove Connection

Removes the selected target connection from the **Debug Control** view.

Debug from menu

This menu lists the different types of actions that you can perform when a connection is established.

Reset menu

This menu lists the different types of reset that are available on your target.

Continue

Continues running the target.

———— Note ————

A **Connect only** connection might require setting the PC register to the start of the image before running it.

Interrupt

Interrupts the target and stops the current application.

Step Source Line Step Instruction

This option depends on the stepping mode selected:

- If source line mode is selected, steps at the source level including stepping into all function calls where there is debug information.
- If instruction mode is selected, steps at the instruction level including stepping into all function calls.

Step Over Source Line Step Over Instruction

This option depends on the stepping mode selected:

- If source line mode is selected, steps at the source level but stepping over all function calls.
- If instruction mode is selected, steps at the instruction level but stepping over all function calls.

Step Out

Continues running to the next instruction after the selected stack frame finishes.

Stepping by Source Line (press to step by instruction) Stepping by Instruction (press to step by source line)

Toggles the stepping mode between source line and instruction.

The **Disassembly** view and the source editor view are automatically displayed when you step in instruction mode.

The source editor view is automatically displayed when you step in source line mode. If the target stops in code such as a shared library, and the corresponding source is not available, then the source editor view is not displayed.

Step Out to This Frame

Continues running to the selected stack frame.

Change Connection Color

Enables you to change the color of the connection icon.

View Menu

The following options are available:

Add Configuration (without connecting)...

Displays the Add Launch Configuration dialog box. The dialog box lists any configurations that are not already listed in the **Debug Control** view.

Select one or more configurations, then click **OK**. The selected configurations are added to the **Debug Control** view, but remain unconnected.

Load...

Displays a dialog box where you can select whether to load an image, debug information, an image and debug information, or additional debug information. This option might be disabled for targets where this functionality is not supported.

Set Working Directory...

Displays the Current Working Directory dialog box. Enter a new location for the current working directory, then click **OK**.

Path Substitution...

Displays the Path Substitution and Edit Substitute Path dialog box.

Use the Edit Substitute Path dialog box to associate the image path with a source file path on the host. Click **OK**. The image and host paths are added to the Path Substitution dialog box. Click **OK** when finished.

Reset DS-5 Views to 'Linked'

Resets DS-5 views to link to the selected connection in the **Debug Control** view.

Threads Presentation

Displays either a flat or hierarchical presentation of the threads in the stack trace.

Auto Expand Stack

Controls whether to automatically display an expanded stack when selecting a connection.

Always Show Cores

Displays the available processors.

Related concepts

[6.7 About debugging multi-threaded applications on page 6-145.](#)

[6.8 About debugging shared libraries on page 6-146.](#)

[6.9 About debugging a Linux kernel on page 6-148.](#)

[6.10 About debugging Linux kernel modules on page 6-150.](#)

[6.12 About debugging TrustZone enabled targets on page 6-153.](#)

Related references

[1.7 DS-5 Debug perspective keyboard shortcuts on page 1-28.](#)

[10.6 Commands view on page 10-209.](#)

[10 DS-5 Debug perspectives and views on page 10-194.](#)

10.8 Disassembly view

Describes the view content.

This view enables you to:

- See a disassembly of the target memory
- Specify the start address for the **Disassembly** view. You can use expressions in this field, \$r3, or drag and drop a register from the **Registers** view into the **Disassembly** view to see the disassembly at the address in that register.
- Select the instruction set for the **Disassembly** view
- Create, delete, enable or disable a breakpoint or watchpoint at a memory location
- Freeze the selected view to prevent the values being updated by a running target.

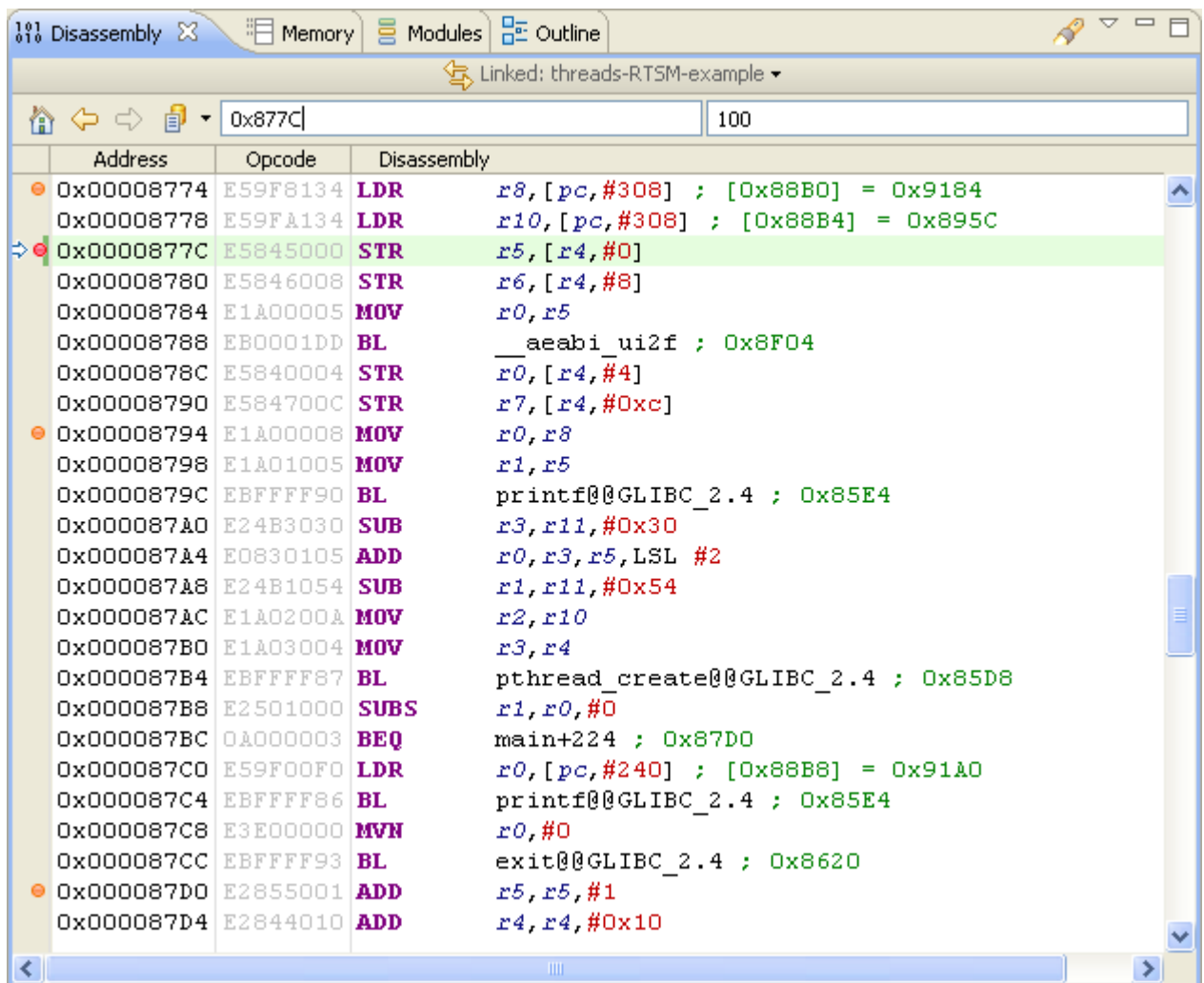


Figure 10-10 Disassembly view

Gradient shading in the **Disassembly** view shows the start of each function.

Solid shading in the **Disassembly** view shows the instruction at the address of the current PC register followed by any related instructions that correspond to the current source line.

In the left-hand margin of the **Disassembly** view you can find a marker bar that displays view markers associated with specific locations in the disassembly code.

To set a breakpoint, double-click in the marker bar at the position where you want to set the breakpoint. To delete a breakpoint, double-click on the breakpoint marker.

Note

If you have sub-breakpoints to a parent breakpoint then double-clicking on the marker also deletes the related sub-breakpoints.

Toolbar and context menu options

The following options are available from the toolbar or context menu:

Linked: *context*

Links this view to the selected connection in the **Debug Control** view. This is the default. Alternatively you can link the view to a specific connection. If the connection you want is not shown in the drop-down list you might have to select it first in the **Debug Control** view.

Back, Forward

Navigates through the history list.

<Next Instruction>

Navigates to the selected stack frame in the **Debug Control** view.

\$LR

Navigates to the LR register.

expression

Navigates to the address specified by an expression. `$PC+256`.

address

Navigates to the specified address.

History

Addresses and expressions you specify in the Address field are added to the drop down box, and persist until you clear the history list or exit Eclipse. If you want to keep an expression for later use, add it to the **Expressions** view.

Address field

Enter the address where you want to view the disassembly.

Context menu options are available for editing this field.

Size field

The number of instructions to display before and after the location pointed to by the program counter.

Context menu options are available for editing this field.

Search

Searches through debug information for symbols.

View Menu

The following **View Menu** options are available:

New Disassembly View

Displays a new instance of the **Disassembly** view.

Instruction Set

The instruction set to show in the view by default. Select one of the following:

[AUTO]

Auto detect the instruction set from the image.

ARM

ARM instruction set.

Thumb

Thumb instruction set.

Byte Order

Selects the byte order of the memory. The default is **Auto(LE)**.

Clear History

Clears the list of addresses and expressions in the History drop-down box.

Refresh

Refreshes the view.

Freeze Data

Toggles the freezing of data in the current view. This also disables and enables the Size and Type fields and the **Refresh** option.

Action context menu

When you right-click in the left margin, the corresponding address and instruction is selected and this context menu is displayed. The available options are:

Copy

Copies the selected address.

Paste

Pastes into the Address field the last address that you copied.

Select All

Selects all disassembly in the range specified by the Size field.

If you want to copy the selected lines of disassembly, you cannot use the **Copy** option on this menu. Instead, use the copy keyboard shortcut for your host, Ctrl +C on Windows.

Run to Selection

Runs to the selected address

Set PC to Selection

Sets the PC register to the selected address.

Show in source

If source code is available:

1. Opens the corresponding source file in the C/C++ source editor view, if necessary.
2. Highlights the line of source associated with the selected address.

Show in registers

If the memory address corresponds to a register, then displays the **Registers** view with the related register selected.

Show in functions

If the memory address corresponds to a function, then displays the **Functions** view with the related function selected.

Toggle Breakpoint

Sets or removes a breakpoint at the selected address.

Toggle Hardware Breakpoint

Sets or removes a hardware breakpoint at the selected address.

Resolve Breakpoint

Resolves a pending breakpoint at the selected address.

Enable Breakpoint

Enables the breakpoint at the selected address.

Disable Breakpoint

Disables the breakpoint at the selected address.

Toggle Trace Start Point

Sets or removes a trace start point at the selected address.

Toggle Trace Stop Point

Sets or removes a trace stop point at the selected address.

Toggle Trace Trigger Point

Starts a trace trigger point at the selected address.

Editing context menu options

The following options are available on the context menu when you select the Address field or Size field for editing:

Cut

Copies and deletes the selected text.

Copy

Copies the selected text.

Paste

Pastes text that you previously cut or copied.

Delete

Deletes the selected text.

Undo

Reverts the last change.

Select All

Selects all the text.

Related concepts

[6.7 About debugging multi-threaded applications on page 6-145.](#)

[6.8 About debugging shared libraries on page 6-146.](#)

[6.9 About debugging a Linux kernel on page 6-148.](#)

[6.10 About debugging Linux kernel modules on page 6-150.](#)

[6.12 About debugging TrustZone enabled targets on page 6-153.](#)

Related references

[Working with data watchpoints.](#)

[4.8 Setting a tracepoint on page 4-119.](#)

[4.6 Working with conditional breakpoints on page 4-114.](#)

[4.6.1 Assigning conditions to an existing breakpoint on page 4-114.](#)

[4.7 About pending breakpoints and watchpoints on page 4-118.](#)

[4.5.5 Exporting DS-5™ breakpoint settings to a file on page 4-112.](#)

[4.5.4 Importing DS-5™ breakpoint settings from a file on page 4-111.](#)

[10 DS-5 Debug perspectives and views on page 10-194.](#)

10.9 Events view

This section describes the **Events** view.

The **Events** view enables you to view the output generated by the *System Trace Macrocell* (STM) and *Instruction Trace Macrocell* (ITM) events.

Data is captured from your application only when it runs. However, no data appears in the view until you stop the application. You can stop the target by clicking the **Interrupt** icon in the **Debug Control** view, or by entering the **stop** command in the **Commands** view. When your application stops then any captured logging information is automatically appended to the open views.

The Configuration dialog enables you to select a trace source as well as masters and channels to display in the view.

Toolbar and context menu options

The following options are available from the toolbar or context menu:

Linked: **context**

Links this view to the selected connection in the **Debug Control** view. This is the default. Alternatively you can link the view to a specific connection. If the connection you want is not shown in the drop-down list you might have to select it first in the **Debug Control** view.

Search by **Timestamp**

Searches through debug information for the nearest record to a given time stamp.

Configure **Masters and Channels**

Enables you to select a trace source in addition to masters and channels to display in the view.

View **Menu**

The following **View Menu** options are available:

New Events View

Displays a new instance of the **Events** view.

Refresh

Refreshes the view.

Freeze Data

Toggles the freezing of data in the current view.

Display Format

Select the display format of the events.

Data

Displays data in the data column as hex values with widths depending on the underlying data packets. This is the default.

Text

Displays character wrapped text data.

Related references

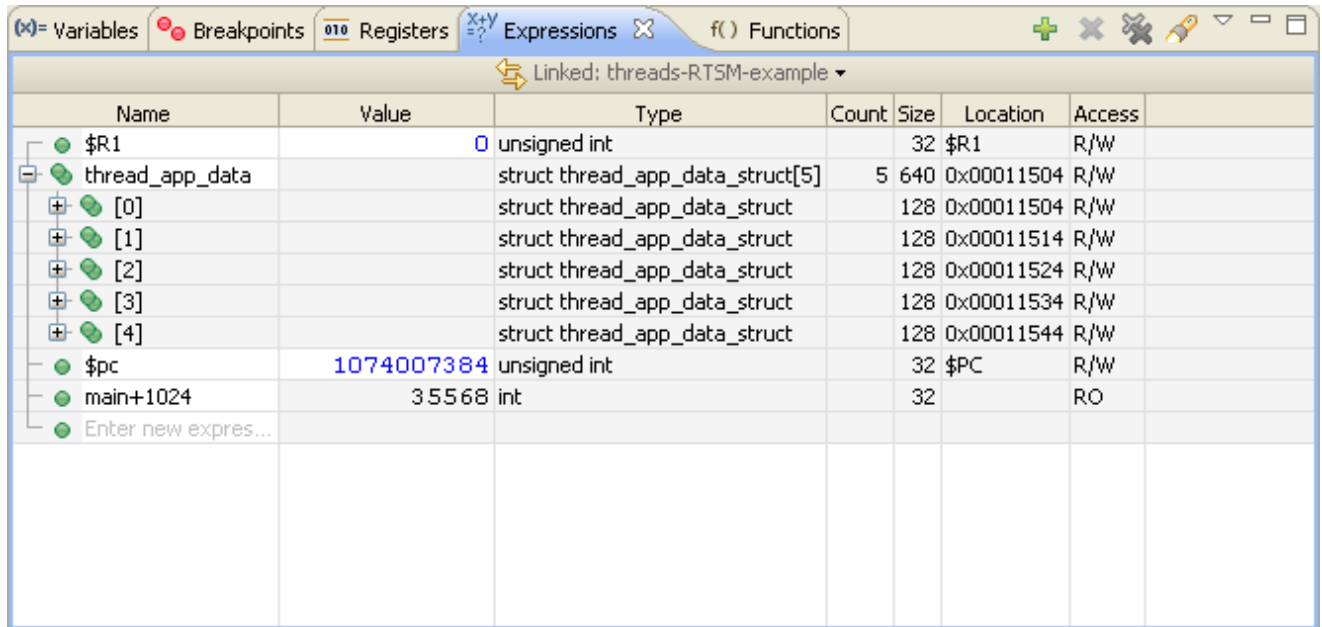
[10 DS-5 Debug perspectives and views on page 10-194.](#)

10.10 Expressions view

Describes the view content.

This view enables you to:

- add expressions that you use regularly or that you want to examine in more detail
- edit, and delete expressions
- freeze the selected view to prevent the values being updated by a running target.



Name	Value	Type	Count	Size	Location	Access
\$R1		unsigned int		32	\$R1	R/W
thread_app_data		struct thread_app_data_struct[5]	5	640	0x00011504	R/W
[0]		struct thread_app_data_struct		128	0x00011504	R/W
[1]		struct thread_app_data_struct		128	0x00011514	R/W
[2]		struct thread_app_data_struct		128	0x00011524	R/W
[3]		struct thread_app_data_struct		128	0x00011534	R/W
[4]		struct thread_app_data_struct		128	0x00011544	R/W
\$pc	1074007384	unsigned int		32	\$PC	R/W
main+1024	35568	int		32		RO
Enter new expres...						

Figure 10-11 Expressions view

Note

If your expression contains side-effects when evaluating the expression, the results are unpredictable. Side-effects occur when the state of one or more inputs to the expression changes when the expression is evaluated.

For example, instead of `x++` or `x+=1` you must use `x+1`.

Right-click on the column headers to select the columns that you want displayed:

Name

An expression that resolves to an address, such as `main+1024`.

Value

The value of the expression. You can modify a value that has a white background. A yellow background indicates the value has changed.

If you freeze the view, then you cannot change a value.

Type

The type associated with the value at the address identified by the expression.

Count

The number of array or pointer elements. You can edit a pointer element count.

Size

The size of expression in bits.

Location

The address in hexadecimal identified by the expression, or the name of a register, if expression contains only a single register name.

Access

The access type of expression.

All columns are displayed by default.

Toolbar and context menu options

The following options are available from the toolbar or context menu:

Linked: **context**

Links this view to the selected connection in the Debug Control view. This is the default. Alternatively you can link the view to a specific connection. If the connection you want is not shown in the drop-down list you might have to select it first in the **Debug Control** view.

Add New Expression

Adds a new expression to the expression list.

Remove Selected Expression

Removes the selected expression from the list.

Remove All Expressions

Removes all expressions from the list.

Search

Searches the data in the current view for an expression.

Cut

Copies and removes the selected expression.

Copy

Copies the selected expression.

To copy an expression for use in the **Disassembly** view or **Memory** view, first select the expression in the Name field.

Paste

Pastes expressions that you have previously cut or copied.

Delete

Deletes the selected expression.

Select All

Selects all expressions.

Show in memory view

Where enabled, displays the **Memory** view with the address set to either:

- the value of the selected expression, if the expression translates to an address, the address of an array, **&name**
- the location of the expression, the name of an array, **name**.

The memory size is set to the size of the variable, using the **sizeof** keyword.

Show in register view

If the expression corresponds to a register, then displays the **Registers** view with that register selected. This might be:

- an expression that consists only of a single register, **\$pc**
- a variable that is currently held in a register, For example, the variable **t** might be held in register R5.

Send to Selection

Enables you to add register filters to an **Expression** view. Displays a sub menu that enables you to add to a specific **Expressions** view.

format list

A list of formats you can use for the expression value.

View Menu

The following **View Menu** options are available:

New Expression View

Displays a new instance of the **Expressions** view.

Refresh

Refreshes the view.

Freeze Data

Toggles the freezing of data in the current view. This also disables and enables the **Refresh** option.

Related references

Working with data watchpoints.

4.8 Setting a tracepoint on page 4-119.

4.6 Working with conditional breakpoints on page 4-114.

4.6.1 Assigning conditions to an existing breakpoint on page 4-114.

4.7 About pending breakpoints and watchpoints on page 4-118.

4.5.5 Exporting DS-5™ breakpoint settings to a file on page 4-112.

4.5.4 Importing DS-5™ breakpoint settings from a file on page 4-111.

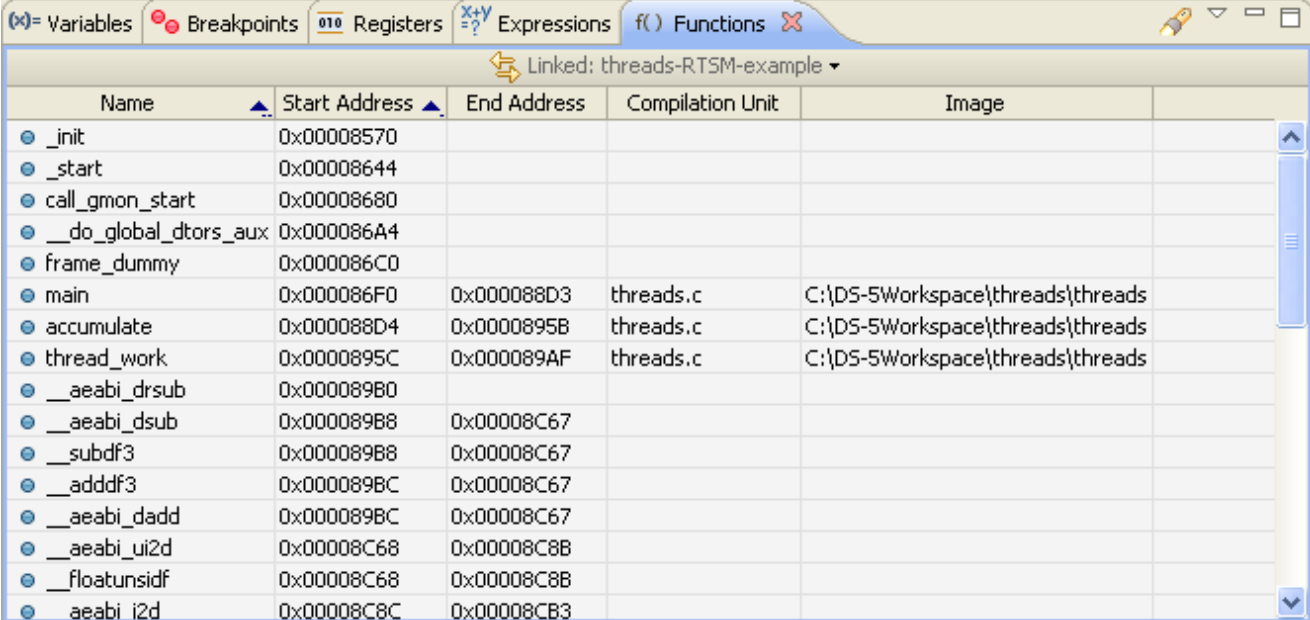
10 DS-5 Debug perspectives and views on page 10-194.

10.11 Functions view

This section describes the **Functions** view.

This view enables you to:

- See the ELF data associated with function symbols for all loaded images.
- Freeze the selected view to prevent the information being updated by a running target.



Name	Start Address	End Address	Compilation Unit	Image
• _init	0x00008570			
• _start	0x00008644			
• call_gmon_start	0x00008680			
• __do_global_ctors_aux	0x000086A4			
• frame_dummy	0x000086C0			
• main	0x000086F0	0x000088D3	threads.c	C:\DS-5Workspace\threads\threads
• accumulate	0x000088D4	0x0000895B	threads.c	C:\DS-5Workspace\threads\threads
• thread_work	0x0000895C	0x000089AF	threads.c	C:\DS-5Workspace\threads\threads
• __aeabi_drsub	0x000089B0			
• __aeabi_dsub	0x000089B8	0x00008C67		
• _subdf3	0x000089B8	0x00008C67		
• _adddf3	0x000089BC	0x00008C67		
• __aeabi_dadd	0x000089BC	0x00008C67		
• __aeabi_ui2d	0x00008C68	0x00008C8B		
• _floatunsidf	0x00008C68	0x00008C8B		
• __aeabi_i2d	0x00008C8C	0x00008CB3		

Figure 10-12 Functions view

Right-click on the column headers to select the columns that you want displayed:

Name

The name of the function.

Mangled Name

The C++ mangled name of the function.

Base Address

The function entry point.

Start Address

The start address of the function.

End Address

The end address of the function.

Size

The size of the function in bytes.

Compilation Unit

The location of the compilation unit containing the function.

Image

The location of the ELF image containing the function.

The Name, Start Address, End Address, Compilation Unit, and Image columns are displayed by default.

Toolbar and context menu options

The following options are available from the toolbar or context menu:

Linked: context

Links this view to the selected connection in the **Debug Control** view. This is the default. Alternatively you can link the view to a specific connection. If the connection you want is not shown in the drop-down list you might have to select it first in the **Debug Control** view.

Search

Searches the data in the current view for a function.

Copy

Copies the selected functions.

Select All

Selects all the functions in the view.

Run to Selection

Runs to the selected address

Set PC to Selection

Sets the PC register to the start address of the selected function.

Show in Source

If source code is available:

1. Opens the corresponding source file in the C/C++ editor view, if necessary.
2. Highlights the line of source associated with the selected address.

Show in Memory

Displays the **Memory** view starting at the address of the selected function.

Show in Disassembly

Displays the **Disassembly** view starting at the address of the selected function.

Toggle Breakpoint

Sets or removes a breakpoint at the selected address.

Resolve Breakpoint

Resolves a pending breakpoint at the selected address.

Enable Breakpoint

Enables the breakpoint at the selected address.

Disable Breakpoint

Disables the breakpoint at the selected address.

Toggle Trace Start Point

Sets or removes a trace start point at the selected address.

Toggle Trace Stop Point

Sets or removes a trace stop point at the selected address.

Toggle Trace Trigger Point

Starts a trace trigger point at the selected address.

View Menu

The following **View Menu** options are available:

New Function View

Displays a new instance of the **Functions** view.

Refresh

Refreshes the view.

Freeze Data

Toggles the freezing of data in the current view. This also disables or enables the **Refresh** option.

Filter...

Displays the Filter dialog box.

Related references

[10 DS-5 Debug perspectives and views on page 10-194.](#)

10.12 History view

Describes the view content.

This view enables you to:

- See a full list of commands generated during the current debug session.
- Clear the contents of the view.
- Save the selected commands to a script file. You can also add the script file to your favorites list when you click **Save**. Favorites are displayed in the **Scripts** view.
- Enable or disable the automatic scrolling of messages in the **History** view.

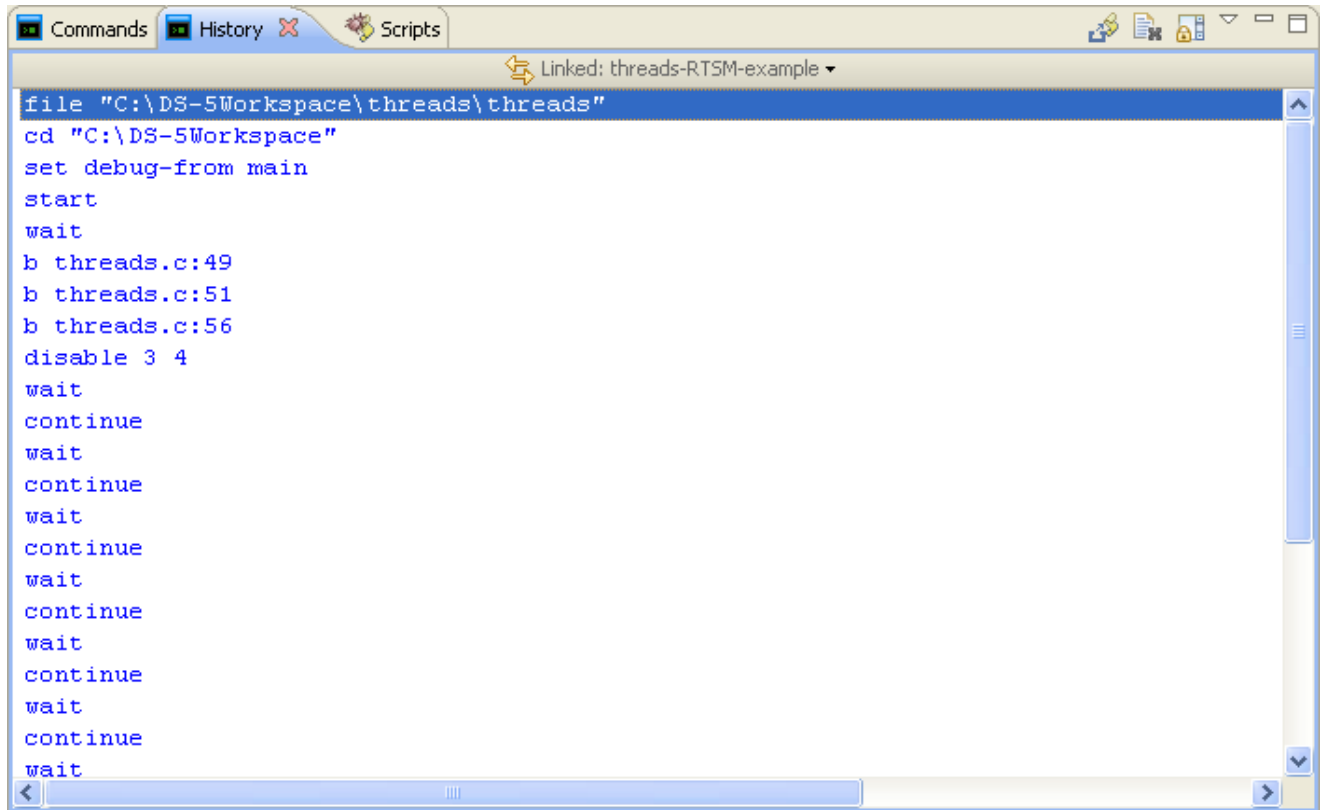


Figure 10-13 History view

———— Note ————

Default settings for this view are controlled by a DS-5 Debugger setting in the Preferences dialog box. For example, default locations for specific files. You can access these settings by selecting **Preferences** from the **Window** menu.

Toolbar and context menu options

The following options are available from the toolbar or context menu:

Linked: context

Links this view to the selected connection in the **Debug Control** view. This is the default. Alternatively you can link the view to a specific connection. If the connection you want is not shown in the drop-down list you might have to select it first in the **Debug Control** view.

Exports the selected lines as a script

Displays the Save As dialog box to save the selected commands to a script file.

When you click **Save** on the Save As dialog box, you are given the option to add the script file to your favorites list. Click **OK** to add the script to your favorites list. Favorites are displayed in the **Scripts** view.

Clear Console

Clears the contents of the **History** view.

Toggles Scroll Lock

Enables or disables the automatic scrolling of messages in the **History** view.

Copy

Copies the selected commands.

Select All

Selects all commands.

Save the selected lines as a script...

Displays the Save As dialog box to save the selected commands to a script file.

When you click **Save** on the Save As dialog box, you are given the option to add the script file to your favorites list. Click **OK** to add the script to your favorites list. Favorites are displayed in the **Scripts** view.

Execute selected lines

Runs the selected commands.

New History View

Displays a new instance of the **History** view.

Related references

[*10 DS-5 Debug perspectives and views on page 10-194.*](#)

10.13 Memory view

Describes the view content.

This view enables you to:

- Modify memory content.
- Specify the start address for the **Memory** view, either as an absolute address or as an expression, `$pc`. Previous entries are listed in the drop-down list. This list is cleared when you exit Eclipse.
- Specify the display size of the **Memory** view in bytes, either as an offset value from the start address, or as an address held in a register by dragging and dropping the register from the **Registers** view into the **Memory** view.
- Specify the format of the memory cell values. The default is hexadecimal.
- Set the width of the memory cells in the **Memory** view. The default is four bytes.
- Display the ASCII character equivalent of the memory values.
- Freeze the selected view to prevent the view being updated by a running target.

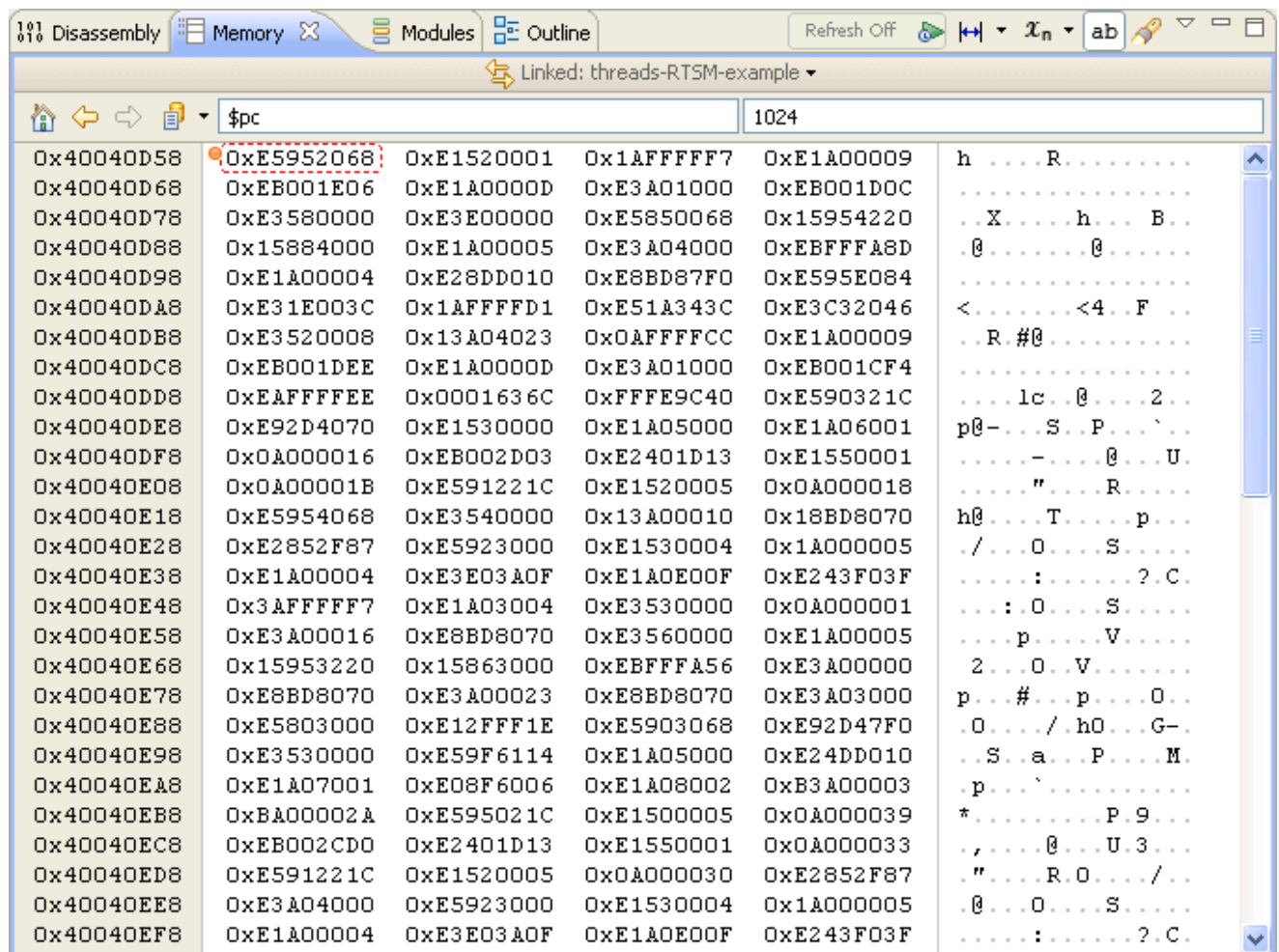


Figure 10-14 Memory view

The **Memory** view only provides the facility to modify how memory is displayed in this view. It is not possible to specify the use of byte, half-word, word or double read/write instructions to access memory from the **Memory** view. To control the memory access width you can use:

- the **memory** command to configure access widths for a region of memory, followed by the **x** command to read memory according to those access widths and display the contents
- the **memory set** command to write to memory with an explicit access width.

Toolbar and context menu options

The following options are available from the toolbar or context menu:

Linked: **context**

Links this view to the selected connection in the **Debug Control** view. This is the default. Alternatively you can link the view to a specific connection. If the connection you want is not shown in the drop-down list you might have to select it first in the **Debug Control** view.

Back, Forward

Navigates through the history list.

History

Addresses and expressions you specify in the Address field are added to the drop down box, and persist until you clear the history list or exit Eclipse. If you want to keep an expression for later use, add it to the **Expressions** view.

Timed auto refresh is off Cannot update

This option opens a dialog box where you can specify refresh intervals:

- If timed auto refresh is off mode is selected, the auto refresh is off.
- If the cannot update mode is selected, the auto refresh is blocked.

Display Width

Click to cycle through the memory cell widths in the Memory view, or select a width from the drop-down menu. The default is four bytes.

Format

Click to cycle through the memory cell formats, or select a format from the drop-down menu. The default is hexadecimal.

Showing characters - click to hide the character display Not showing characters - click to show the character display

Toggles the display of ASCII character equivalents for the memory values.

Address field

Enter the address where you want to start viewing the target memory. Alternatively, you can enter an expression that evaluates to an address. **\$PC+256**.

Addresses and expressions you specify are added to the drop down list, and persist until you exit Eclipse. If you want to keep an expression for later use, add it to the **Expressions** view.

Context menu options are available for editing this field.

Size field

The number of bytes to display.

Context menu options are available for editing this field.

Search

Searches through debug information for symbols.

View Menu

The following **View Menu** options are available:

New Memory View

Displays a new instance of the **Memory** view.

Show Tooltips

Toggles the display of tooltips on memory cell values.

Byte Order

Selects the byte order of the memory. The default is **Auto(LE)**.

Clear History

Clears the list of addresses and expressions in the History drop-down box.

Import Memory

Reads data from a file and writes it to memory.

Export Memory

Reads data from memory and writes it to a file.

Fill Memory

Writes a specific pattern of bytes to memory.

Refresh

Refreshes the view.

Freeze Data

Toggles the freezing of data in the current view. This also disables or enables the Address and Size fields and the **Refresh** option.

Editing context menu options

The following options are available on the context menu when you select a memory cell value, the Address field, or the Size field for editing:

Cut

Copies and deletes the selected value.

Copy

Copies the selected value.

Paste

Pastes a value that you have previously cut or copied into the selected memory cell or field.

Delete

Deletes the selected value.

Undo

Reverts the last change you made to the selected memory cell or field. This is disabled for the Address field.

Select All

Selects all the address.

Toggle Breakpoint

Sets or removes a breakpoint at the selected address.

Resolve Breakpoint

Resolves a pending breakpoint at the selected address.

Enable Breakpoint

Enables the breakpoint at the selected address.

Disable Breakpoint

Disables the breakpoint at the selected address.

Toggle Trace Start Point

Sets or removes a trace start point at the selected address.

Toggle Trace Stop Point

Sets or removes a trace stop point at the selected address.

Toggle Trace Trigger Point

Starts a trace trigger point at the selected address.

Related concepts

[6.7 About debugging multi-threaded applications on page 6-145.](#)

[6.8 About debugging shared libraries on page 6-146.](#)

[6.9 About debugging a Linux kernel on page 6-148.](#)

[6.10 About debugging Linux kernel modules on page 6-150.](#)

[6.12 About debugging TrustZone enabled targets on page 6-153.](#)

Related references

[Working with data watchpoints.](#)

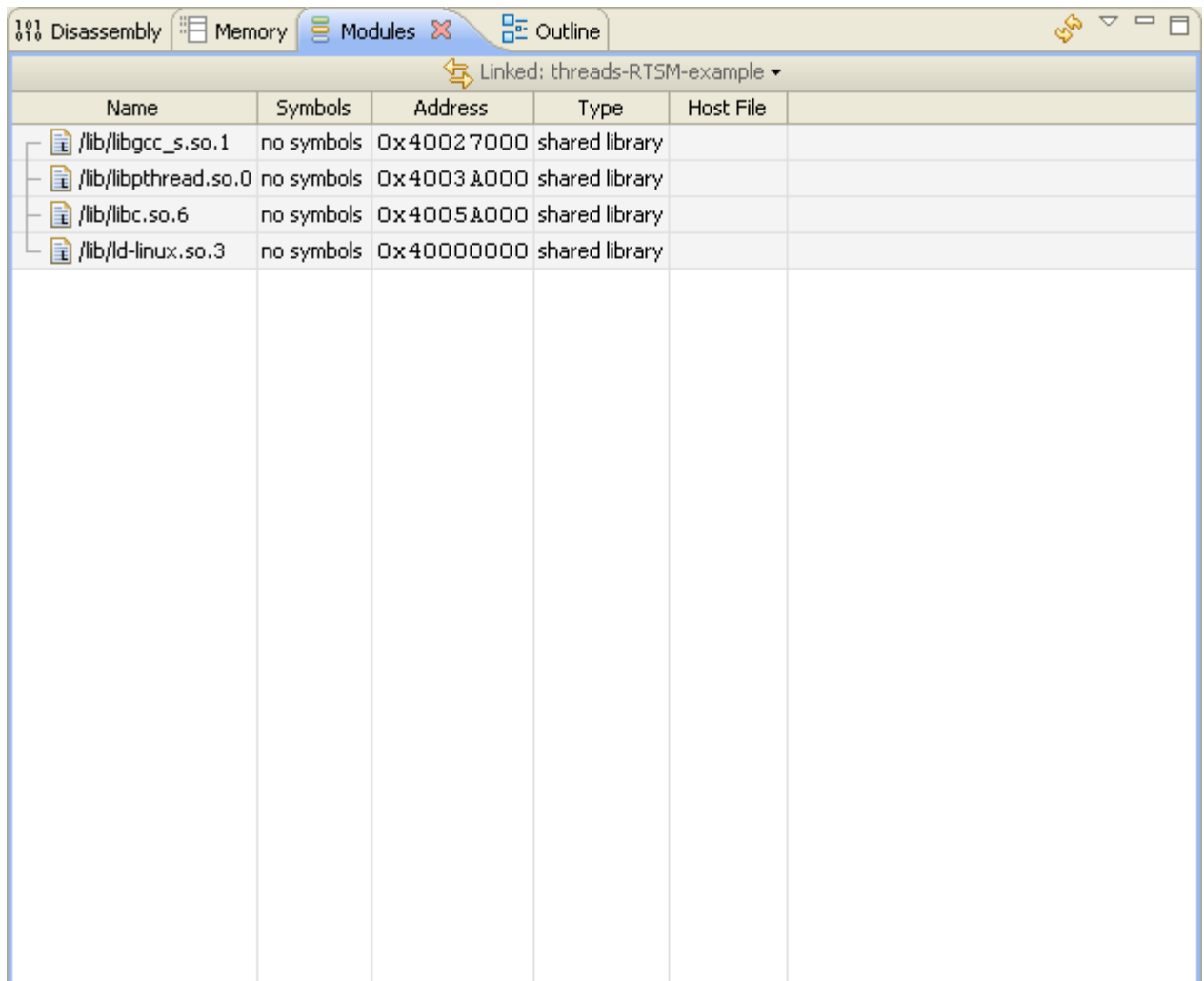
4.8 Setting a tracepoint on page 4-119.
4.6 Working with conditional breakpoints on page 4-114.
4.6.1 Assigning conditions to an existing breakpoint on page 4-114.
4.7 About pending breakpoints and watchpoints on page 4-118.
4.5.5 Exporting DS-5™ breakpoint settings to a file on page 4-112.
4.5.4 Importing DS-5™ breakpoint settings from a file on page 4-111.
10 DS-5 Debug perspectives and views on page 10-194.

10.14 Modules view

Describes the view content.

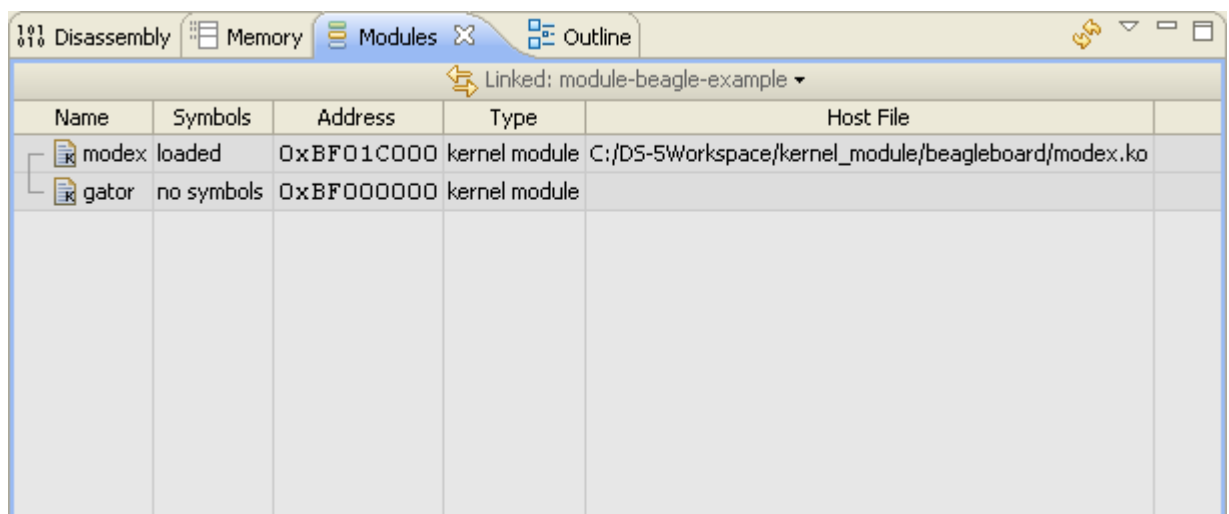
This view is only populated when connected to a Linux target. It enables you to:

- see a tabular view of the shared libraries used by the application
- see a tabular view of dynamically loaded *Operating System* (OS) modules
- load and unload debug information for a specific module or shared library.



Name	Symbols	Address	Type	Host File
/lib/libgcc_s.so.1	no symbols	0x40027000	shared library	
/lib/libpthread.so.0	no symbols	0x4003A000	shared library	
/lib/libc.so.6	no symbols	0x4005A000	shared library	
/lib/ld-linux.so.3	no symbols	0x40000000	shared library	

Figure 10-15 Modules view showing shared libraries



Name	Symbols	Address	Type	Host File
modex	loaded	0xBF01C000	kernel module	C:/DS-5Workspace/kernel_module/beagleboard/modex.ko
gator	no symbols	0xBF000000	kernel module	

Figure 10-16 Modules view showing operating system modules

————— Note —————

A connection must be established and OS support enabled within the debugger before a loadable module can be detected. OS support is automatically enabled when a Linux kernel image is loaded into the debugger. However, you can manually control this by using the `set os` command.

Right-click on the column headers to select the columns that you want displayed:

Name

Displays the name and location of the component on the target.

Symbols

Displays whether the symbols are currently loaded for each object.

Address

Displays the load address of the object.

Size

Displays the size of the object.

Kind

Displays the component type. For example, shared library or OS module.

Host File

Displays the name and location of the component on the host workstation.

The Name, Symbols, Address, Kind, and Host File columns are displayed by default.

Toolbar and context menu options

The following options are available from the toolbar or context menu:

Linked: *context*

Links this view to the selected connection in the **Debug Control** view. This is the default. Alternatively you can link the view to a specific connection. If the connection you want is not shown in the drop-down list you might have to select it first in the **Debug Control** view.

Refresh

Refreshes the view.

Copy

Copies the selected data.

Select All

Selects all the displayed data.

Load Symbols

Loads debug information into the debugger from the source file displayed in the Host File column. This option is disabled if the host file is unknown, before the file is loaded.

Add Symbol File...

Opens a dialog box where you can select a file from the host workstation containing the debug information required by the debugger.

Discard Symbols

Discards debug information relating to the selected file.

Show in Memory

Displays the **Memory** view starting at the load address of the selected object.

Show in Disassembly

Displays the **Disassembly** view starting at the load address of the selected object.

Related concepts

[6.7 About debugging multi-threaded applications on page 6-145.](#)

[6.8 About debugging shared libraries on page 6-146.](#)

[6.9 About debugging a Linux kernel on page 6-148.](#)

[6.10 About debugging Linux kernel modules on page 6-150.](#)

[6.12 About debugging TrustZone enabled targets on page 6-153.](#)

Related references

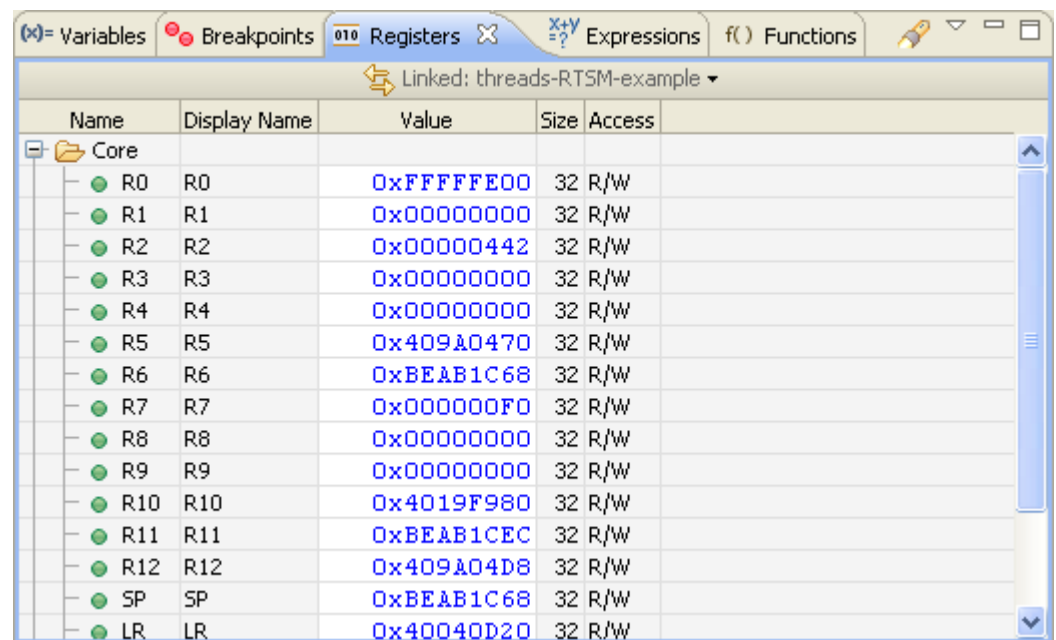
10 DS-5 Debug perspectives and views on page 10-194.

10.15 Registers view

Describes the view content.

This view enables you to:

- See the contents of target registers.
- Change the values for registers that have write access. When a register value changes, the register value background changes to yellow.
- Change the display format of register values. The *Program Status Registers* (PSRs) also enable you to set the format using individual bits.
- Freeze the selected view to prevent the values being updated by a running target.
- Drag and drop an address held in a register, such as R3, from the **Registers** view either into the **Memory** view to see the memory at that address, or into the **Disassembly** view to disassemble from that address.



Name	Display Name	Value	Size	Access
Core				
R0	R0	0xFFFFFE00	32	R/W
R1	R1	0x00000000	32	R/W
R2	R2	0x00000442	32	R/W
R3	R3	0x00000000	32	R/W
R4	R4	0x00000000	32	R/W
R5	R5	0x409A0470	32	R/W
R6	R6	0xBEAB1C68	32	R/W
R7	R7	0x000000F0	32	R/W
R8	R8	0x00000000	32	R/W
R9	R9	0x00000000	32	R/W
R10	R10	0x4019F980	32	R/W
R11	R11	0xBEAB1CEC	32	R/W
R12	R12	0x409A04D8	32	R/W
SP	SP	0xBEAB1C68	32	R/W
LR	LR	0x40040D20	32	R/W

Figure 10-17 Registers view

Right-click on the column headers to select the columns that you want displayed:

Name

The name of the register.

Use `$register_name` to reference a register. To refer to a register that has bitfields, such as a PSR, specify `$register_name.bitfield_name`. For example, to print the value of the M bitfield of \$CPSR, enter the following command in the **Commands** view:

```
print $CPSR.M
$1 = USR
```

Value

The value of the register. A shaded background indicates the value has changed.

If you freeze the view, then you cannot change a register value.

Type

The type of the register value.

Count

The number of array or pointer elements.

Size

The size of the register in bits.

Location

The name of the register or the bitmap of the bitfield of a PSR. For example, bitfield M of the CPSR is displayed as `$CPSR[0..4]`.

The Name, Value, and Size columns are displayed by default.

Toolbar and context menu options

The following options are available from the toolbar or context menu:

Linked: *context*

Links this view to the selected connection in the **Debug Control** view. This is the default. Alternatively you can link the view to a specific connection. If the connection you want is not shown in the drop-down list you might have to select it first in the **Debug Control** view.

Search

Searches the data in the current view for a register.

Copy

Copies the selected registers. To copy the bitfields of a PSR, you must first expand the PSR.

This is useful if you want to copy the selected registers to a text editor and compare the values when execution stops at another location.

Select All

Selects all registers currently expanded in the view.

Show Memory Pointed to By *register_name*

Where enabled, displays the Memory view starting at the address held in the register.

Show Disassembly Pointed to By *register_name*

Where enabled, displays the **Disassembly** view starting at the address held in the register.

Send to Selection

Enables you to add register filters to an **Expression** view. Displays a sub menu that enables you to add to a specific **Expressions** view.

format List

A list of formats you can use for the register values. The default is **Hexadecimal**.

View Menu

The following **View Menu** options are available:

New Register View

Creates a new instance of the **Registers** view.

Refresh

Refreshes the view.

Freeze Data

Toggles the freezing of data in the current view. This also disables or enables the **Refresh** option.

Editing context menu options

The following options are available on the context menu when you select a register value for editing:

Cut

Copies and deletes the selected value.

Copy

Copies the selected value.

Paste

Pastes a value that you have previously cut or copied into the selected register value.

Delete

Deletes the selected value.

Undo

Reverts the last change you made to the selected value.

Related concepts

[6.7 About debugging multi-threaded applications on page 6-145.](#)

[6.8 About debugging shared libraries on page 6-146.](#)

[6.9 About debugging a Linux kernel on page 6-148.](#)

[6.10 About debugging Linux kernel modules on page 6-150.](#)

[6.12 About debugging TrustZone enabled targets on page 6-153.](#)

Related references

[Working with data watchpoints.](#)

[4.8 Setting a tracepoint on page 4-119.](#)

[4.6 Working with conditional breakpoints on page 4-114.](#)

[4.6.1 Assigning conditions to an existing breakpoint on page 4-114.](#)

[4.7 About pending breakpoints and watchpoints on page 4-118.](#)

[4.5.5 Exporting DS-5™ breakpoint settings to a file on page 4-112.](#)

[4.5.4 Importing DS-5™ breakpoint settings from a file on page 4-111.](#)

[10 DS-5 Debug perspectives and views on page 10-194.](#)

10.16 RTOS Data view

Describes the view content.

This view enables you to display system information about the *Operating System* (OS).

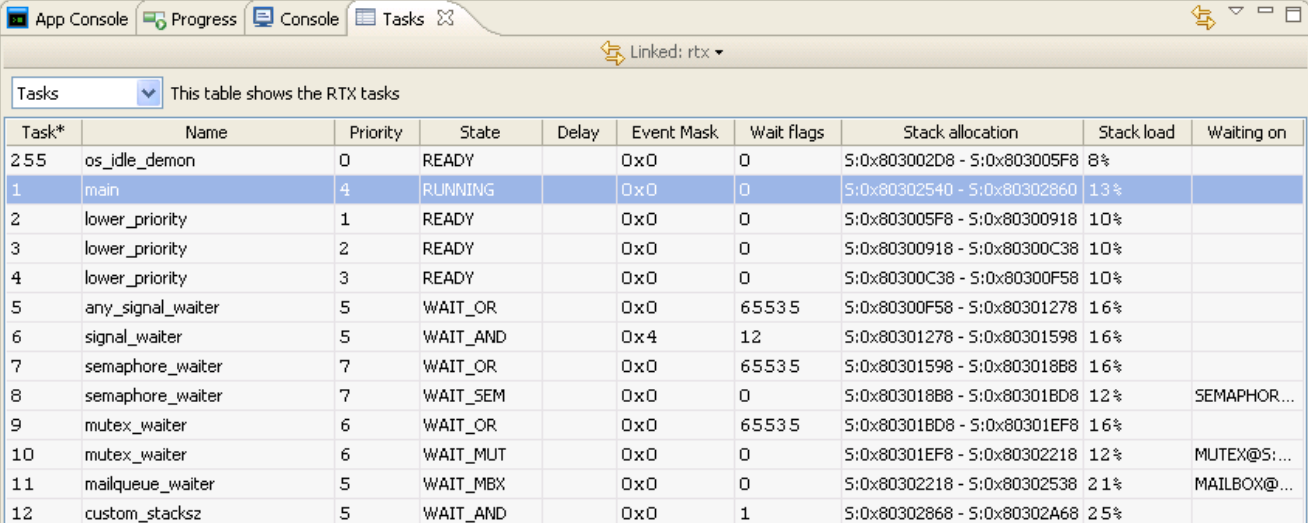
———— Note —————

Options and rows in the **RTOS Data** view are dependent on the type of table that you select.

Multiple tables are available in the drop-down box and its content is controlled by the OS support currently active.

Examples

The following is an example showing tasks specific to the current OS.



Task*	Name	Priority	State	Delay	Event Mask	Wait flags	Stack allocation	Stack load	Waiting on
255	os_idle_demon	0	READY		0x0	0	S:0x803002D8 - S:0x803005F8	8%	
1	main	4	RUNNING		0x0	0	S:0x80302540 - S:0x80302860	13%	
2	lower_priority	1	READY		0x0	0	S:0x803005F8 - S:0x80300918	10%	
3	lower_priority	2	READY		0x0	0	S:0x80300918 - S:0x80300C38	10%	
4	lower_priority	3	READY		0x0	0	S:0x80300C38 - S:0x80300F58	10%	
5	any_signal_waiter	5	WAIT_OR		0x0	65535	S:0x80300F58 - S:0x80301278	16%	
6	signal_waiter	5	WAIT_AND		0x4	12	S:0x80301278 - S:0x80301598	16%	
7	semaphore_waiter	7	WAIT_OR		0x0	65535	S:0x80301598 - S:0x803018B8	16%	
8	semaphore_waiter	7	WAIT_SEM		0x0	0	S:0x803018B8 - S:0x80301BD8	12%	SEMAPHOR...
9	mutex_waiter	6	WAIT_OR		0x0	65535	S:0x80301BD8 - S:0x80301EF8	16%	
10	mutex_waiter	6	WAIT_MUT		0x0	0	S:0x80301EF8 - S:0x80302218	12%	MUTEX@S:...
11	mailqueue_waiter	5	WAIT_MBX		0x0	0	S:0x80302218 - S:0x80302538	21%	MAILBOX@...
12	custom_stacksz	5	WAIT_AND		0x0	1	S:0x80302868 - S:0x80302A68	25%	

Figure 10-18 Typical RTOS view for a RTX table

Toolbar and context menu options

The following options are available from the toolbar or context menu:

Linked: **context**

Links this view to the selected connection in the **Debug Control** view. This is the default. Alternatively you can link the view to a specific connection. If the connection you want is not shown in the drop-down list you might have to select it first in the **Debug Control** view.

Show linked data in other Data views

Shows selected data in a view that is linked to another view.

View Menu

This menu contains the following option:

New RTOS Data View

Displays a new instance of the **RTOS Data** view.

Refresh

Refreshes the view.

Freeze Data

Toggles the freezing of data in the current view. Also, the value of a variable cannot change if the data is frozen.

Editing context menu options

The following options are available on the context menu when you select a variable value for editing:

Copy

Copies the selected value.

Select All

Selects all text.

10.17 Screen view

Describes the view content.

This view enables you to:

- See the contents of the screen buffer on the target. This view only updates when the target stops.
- Set the screen buffer parameters appropriate for the target. :

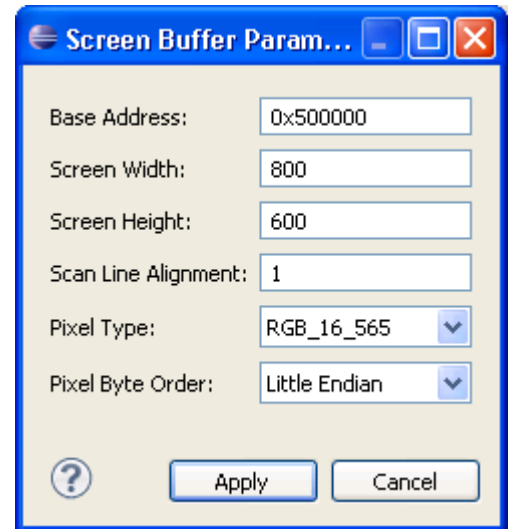


Figure 10-19 Screen buffer parameters for the Fireworks example running on a BeagleBoard

- Freeze the selected view to prevent the screen display being updated by the running target when it next stops.

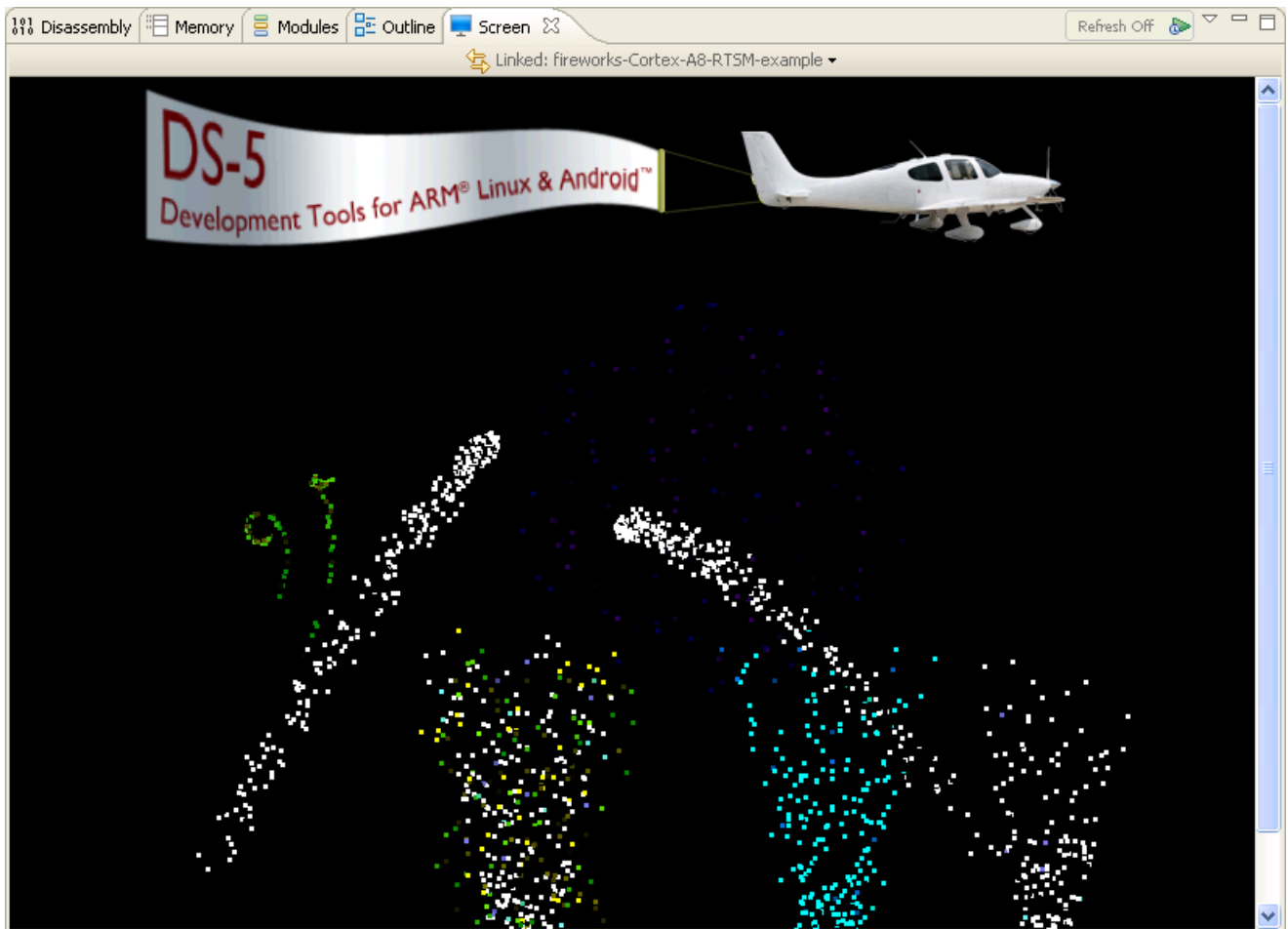


Figure 10-20 Screen view

Toolbar options

The following toolbar options are available:

Linked: context

Links this view to the selected connection in the **Debug Control** view. This is the default. Alternatively you can link the view to a specific connection. If the connection you want is not shown in the drop-down list you might have to select it first in the **Debug Control** view.

Timed auto refresh is off Cannot update

This option opens a dialog box where you can specify refresh intervals:

- If timed auto refresh is off mode is selected, the auto refresh is off.
- If the cannot update mode is selected, the auto refresh is blocked.

Refresh

Refreshes the view.

Set Screen Buffer Parameters

Displays the Screen Buffer Parameters dialog box. The dialog box contains the following parameters:

Base Address

Sets the base address of the screen buffer.

Screen Width

Sets the width of the screen in pixels.

Screen Height

Sets the height of the screen in pixels.

Scan Line Alignment

Sets the byte alignment required for each scan line.

Pixel Type

Selects the pixel type.

Pixel Byte Order

Selects the byte order of the pixels within the data.

Click **Apply** to save the settings and close the dialog box.

Click **Cancel** to close the dialog box without saving.

Freeze Data

Toggles the freezing of data in the current view. This also disables or enables the **Refresh** option.

New Screen Buffer View

Creates a new instance of the **Screen** view.

The Screen view is not visible by default. To add this view:

1. Ensure that you are in the DS-5 Debug perspective.
2. Select **Window > Show View** to open the Show View dialog box.
3. Select **Screen** view.

Related references

[10 DS-5 Debug perspectives and views on page 10-194.](#)

10.18 Scripts view

Describes the view content.

This view contains your favorite scripts. You can run, edit, or remove one or more of your favorite scripts. Scripts can be added to this view when you save commands in the **History** view.

Multiple selections are executed in the order listed in the view. To change the order, remove the scripts from the view and import them in the required order.

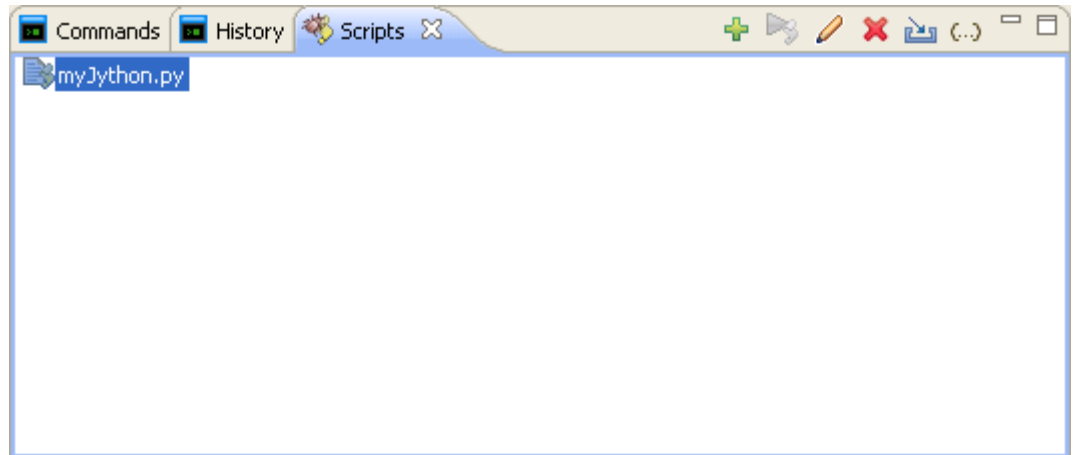


Figure 10-21 Scripts view

Note

Default settings for this view are controlled by a DS-5 Debugger setting in the Preferences dialog box. For example, default locations for script files. You can access these settings by selecting **Preferences** from the **Window** menu.

Toolbar and context menu options

The following options are available from the toolbar or context menu:

Creates a new script

Creates a new empty script. To specify the script contents after it is created, select the script and click **Edit Selected Script**.

Execute Selected Scripts

Runs the selected scripts. If you select multiple scripts, the debugger runs them in the order listed in the **Scripts** view.

Edit Selected Scripts

Enables you to edit the selected scripts. The scripts are opened in the C/C++ editor view.

Delete Selected Scripts

Deletes the selected scripts from the favorites list. You are also prompted to delete the script from the file system.

Import Script...

Imports a script file and add it to the favorites list.

Script Parameters

Enables you to pass parameters to scripts. The parameters can be any combination of fixed strings and variables including file and folder prompts. Using file and folder prompts, you can select items within standard file and folder dialogs.

———— **Note** —————

Select a script to enable the **Script Parameters** button.

Cut

Copies and removes the selected script filename. You are also prompted to delete the script from file system.

Copy

Copies the selected script.

Paste

Pastes a script filename that you have previously cut or copied.

If you deleted the file from the file system as part of a cut operation, the file contents are not restored. You must edit the file to add new commands.

If you did not delete the file as part of a cut operation, the debugger links the filename to the file in the file system.

Delete

Deletes the selected script from the favorites list. You are also prompted to delete the script from file system.

Select All

Selects all script files.

Related references

[10 DS-5 Debug perspectives and views on page 10-194.](#)

10.19 Target Console view

Describes the view content.

This view enables you to receive messages from the target setup scripts.

———— **Note** ————

Default settings for this view are controlled by a DS-5 Debugger setting in the Preferences dialog box. For example, default locations for specific files or the maximum number of lines to display. You can access these settings by selecting **Preferences** from the **Window** menu.

Toolbar and context menu options

The following options are available from the toolbar or context menu:

Linked: context

Links this view to the selected connection in the **Debug Control** view. This is the default. Alternatively you can link the view to a specific connection. If the connection you want is not shown in the drop-down list you might have to select it first in the **Debug Control** view.

Save Console Buffer

Saves the contents of the **Target Console** view to a text file.

Clear Console

Clears the contents of the **Target Console** view.

Scroll Lock

Enables or disables the automatic scrolling of messages in the **Target Console** view.

View Menu

This menu contains the following option:

New Target Console

Displays a new instance of the **Target Console** view.

Bring to Front for Write

If enabled, the debugger automatically changes the focus to this view when a target script prompts for input.

Copy

Copies the selected text.

Paste

Pastes text that you have previously copied.

Select All

Selects all text.

Related references

[10 DS-5 Debug perspectives and views on page 10-194.](#)

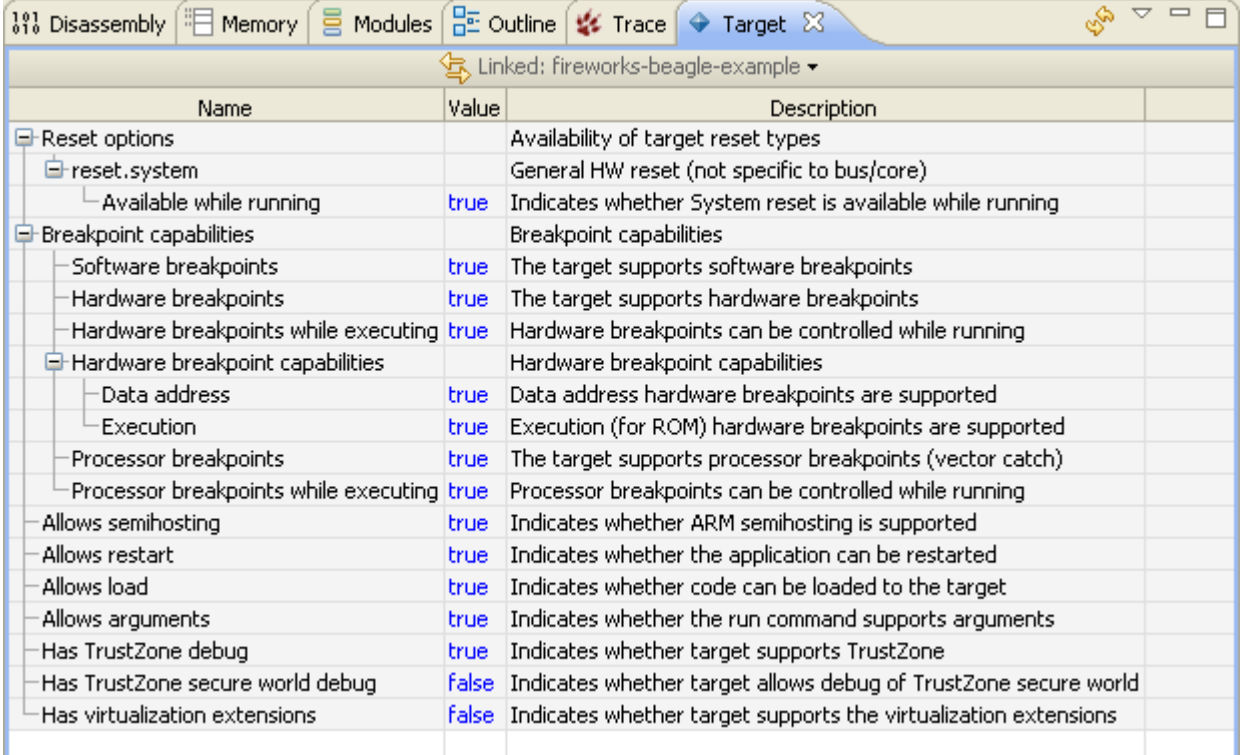
10.20 Target view

Describes the view content.

This view enables you to examine the debug capabilities supported by the target, such as:

- breakpoint types supported
- reset types supported
- memory access types supported.

All capabilities are read-only.



The screenshot shows the DS-5 Target view window. The title bar includes tabs for Disassembly, Memory, Modules, Outline, Trace, and Target (which is active). Below the tabs is a toolbar with icons for linked targets, zoom, and window management. The main area displays a table with the following data:

Name	Value	Description
Reset options		Availability of target reset types
reset.system		General HW reset (not specific to bus/core)
Available while running	true	Indicates whether System reset is available while running
Breakpoint capabilities		Breakpoint capabilities
Software breakpoints	true	The target supports software breakpoints
Hardware breakpoints	true	The target supports hardware breakpoints
Hardware breakpoints while executing	true	Hardware breakpoints can be controlled while running
Hardware breakpoint capabilities		Hardware breakpoint capabilities
Data address	true	Data address hardware breakpoints are supported
Execution	true	Execution (for ROM) hardware breakpoints are supported
Processor breakpoints	true	The target supports processor breakpoints (vector catch)
Processor breakpoints while executing	true	Processor breakpoints can be controlled while running
Allows semihosting	true	Indicates whether ARM semihosting is supported
Allows restart	true	Indicates whether the application can be restarted
Allows load	true	Indicates whether code can be loaded to the target
Allows arguments	true	Indicates whether the run command supports arguments
Has TrustZone debug	true	Indicates whether target supports TrustZone
Has TrustZone secure world debug	false	Indicates whether target allows debug of TrustZone secure world
Has virtualization extensions	false	Indicates whether target supports the virtualization extensions

Figure 10-22 Target view

Right-click on the column headers to select the columns that you want displayed:

Name

The name of the target capability.

Value

The value of the target capability.

Key

The name of the target capability. This is used by some commands in the **Commands** view.

Description

A brief description of the target capability.

The **Name**, **Value**, and **Description** columns are displayed by default.

The **Target** view is not visible by default. To add this view:

1. Ensure that you are in the DS-5 Debug perspective.
2. Select **Window > Show View** to open the Show View dialog box.
3. Select **Target** view.

Toolbar and context menu options

The following options are available from the toolbar or context menu:

Linked: *context*

Links this view to the selected connection in the **Debug Control** view. This is the default. Alternatively you can link the view to a specific connection. If the connection you want is not shown in the drop-down list you might have to select it first in the **Debug Control** view.

Refresh the Target Capabilities

Refreshes the view.

Copy

Copies the selected capabilities. To copy the capabilities in a group such as **Memory capabilities**, you must first expand that group.

This is useful if you want to copy the selected capabilities to a text editor and save them for future reference.

Select All

Selects all capabilities currently expanded in the view.

Related references

[*10 DS-5 Debug perspectives and views on page 10-194.*](#)

10.21 Trace view

Describes the view content.

When the trace has been captured the debugger extracts the information from the trace stream and decompresses it to provide a full disassembly, with symbols, of the executed code.

This view shows a graphical navigation chart that displays function executions with a navigational timeline. In addition, the disassembly trace shows function calls with associated addresses and if selected, instructions. Clicking on a specific time in the chart synchronizes the **Disassembly** view.

In the left-hand column of the chart, percentages are shown for each function of the total trace. For example, if a total of 1000 instructions are executed and 300 of these instructions are associated with `myFunction()` then this function is displayed with 30%.

In the navigational timeline, the color coding is a "heat" map showing the executed instructions and the amount of instructions each function executes in each timeline. The darker red color showing more instructions and the lighter yellow color showing less instructions. At a scale of 1:1 however, the color scheme changes to display memory access instructions as a darker red color, branch instructions as a medium orange color, and all the other instructions as a lighter green color.

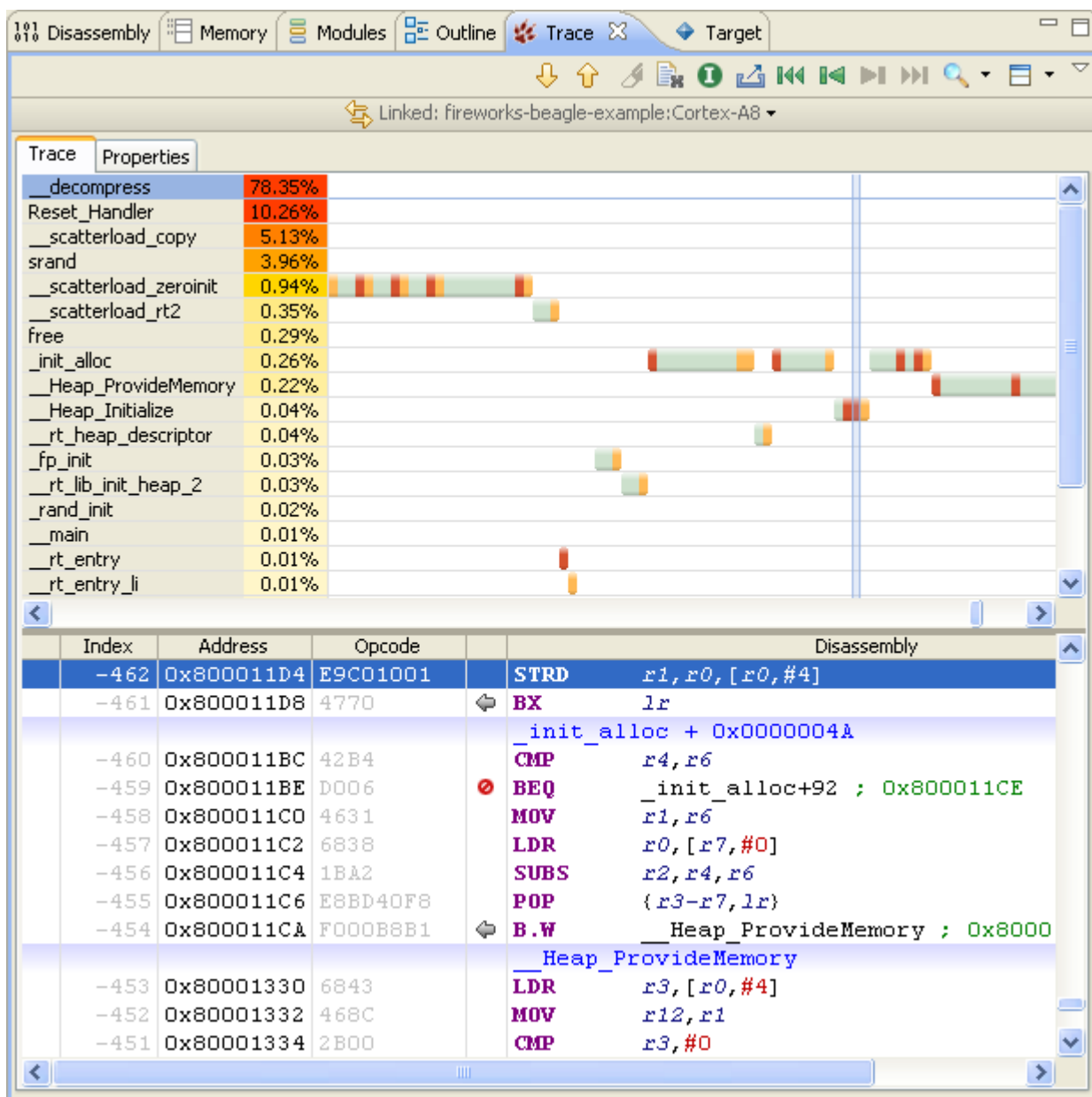


Figure 10-23 Trace view with a scale of 1:1

The **Trace** view might not be visible by default. To add this view:

1. Ensure that you are in the DS-5 Debug perspective.
2. Select **Window > Show View** to open the Show View dialog box.
3. Select **Trace** view.

The **Trace** view contains several tabs:

- Trace tab showing the graphical timeline and disassembly.

Right-click on the column headers to select the columns that you want displayed.

The Breakpoint, Index, Address, Opcode, Info and Disassembly columns are displayed by default.

Toolbar and context menu options

The following options are available from the toolbar or context menu:

Linked: *context*

Links this view to the selected connection in the **Debug Control** view. This is the default. Alternatively you can link the view to a specific connection or processor in an *Symmetric MultiProcessing* (SMP) connection. If the connection you want is not shown in the drop-down list you might have to select it first in the **Debug Control** view.

Show Next Match

Moves the focus of the navigation chart and disassembly trace to the next matching occurrence for the selected function or instruction.

Show Previous Match

Moves the focus of the navigation chart and disassembly trace to the previous matching occurrence for the selected function or instruction.

Don't mark other occurrences - click to start marking, Mark other occurrences - click to stop marking

When function trace is selected, marks all occurrences of the selected function with a shaded highlight. This is disabled when instruction trace is selected.

Clear Trace

Clears the raw trace data that is currently contained in the trace buffer and the trace view.

Showing instruction trace - click to switch to functions, Showing function trace - click to switch to instructions

Toggles the disassembly trace between instruction trace or function trace.

Export Trace Report

Displays the Export Trace Report dialog box to save the trace data to a file.

Home

Where enabled, moves the trace view to the beginning of the trace buffer. Changes might not be visible if the trace buffer is too small.

Page Back

Where enabled, moves the trace view back one page. You can change the page size by modifying the **Set Maximum Instruction Depth** setting.

Page Forward

Where enabled, moves the trace view forward one page. You can change the page size by modifying the **Set Maximum Instruction Depth** setting.

End

Where enabled, moves the trace view to the end of the trace buffer. Changes might not be visible if the trace buffer is too small.

Switch between navigation resolutions

Changes the timeline resolution in the navigation chart.

Switch between alternate views

Changes the view to display the navigation chart, disassembly trace or both.

Focus Here

At the top of the list, displays the function being executed in the selected time slot. The remaining functions are listed in the order that they are executed after the selected point in time. Any functions that do not appear after that point in time are placed at the bottom and ordered by total time.

Order By Total Time

Displays the functions ordered by the total time spent within the function. This is the default ordering.

View Menu

The following **View Menu** options are available:

New Trace View

Displays a new instance of the **Trace** view.

Set Trace Page Size...

Displays a dialog box where you can enter the maximum number of instructions to display in the disassembly trace. The number must be within the range of one thousand to one million instructions.

Find Trace Trigger Event

Enables you to search for trigger events in the trace capture buffer.

Find Timestamp...

Displays a dialog box where you can enter either a numeric timestamp as a 64 bit value or in the h:m:s format.

DTSL Options...

Displays a dialog box where you can add, edit or choose a DTSL configuration.

———— **Note** —————

This will clear the trace buffer.

—————

Refresh

Discards all the data in the view and rereads it from the current trace buffer.

Freeze Data

Toggles the freezing of data in the current view.

Trace Filter Settings...

Displays a dialog box where you can select the trace record types that you want to see in the **Trace** view.

Related references

10 DS-5 Debug perspectives and views on page 10-194.

10.22 Trace Control view

Using the **Trace Control** view, you can start or stop trace capture, and clear the trace buffer on a specified trace capture device.

The **Trace Control** view additionally displays information about the trace capture device, the trace source used, the status of the trace, and the size of the trace buffer.

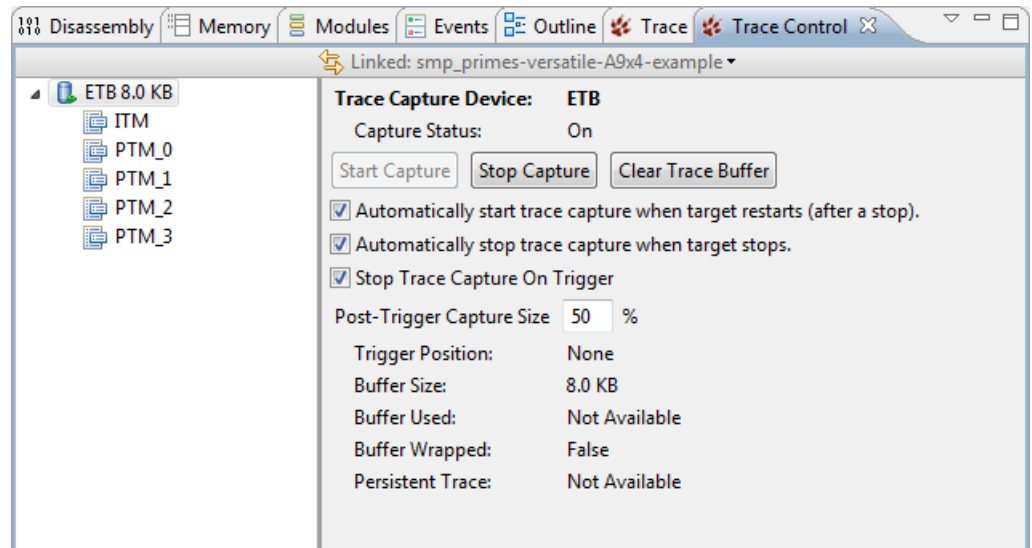


Figure 10-24 Trace Control view

The trace capture device and trace sources available in the trace capture device is displayed on the left hand side of the view. Select any trace source to view additional information. The following information is displayed in the view:

Trace Capture Device information

Trace Capture Device - The name of the trace capture device.

Capture Status - The trace capture status. **On** when capturing trace data, **Off** when not capturing trace data.

Trigger Position - The location of the trigger within the buffer.

Buffer Size - The capacity of the trace buffer.

Buffer Used - The status and amount of trace data currently in the buffer.

Buffer Wrapped - The trace buffer capacity and data wraparound status.

Persistent Trace - The persistent trace data status.

Trace Source information

Trace Source - The name of the selected trace source.

Source ID - The unique ID of the selected trace source.

Source Encoding - The trace encoding format.

Core - The core associated with the trace source.

Context IDs - The tracing context IDs availability status.

Cycle Accurate Trace - The cycle accurate trace support status.

Virtualization Extensions - The virtualization extensions availability status.

Timestamps - Timestamp availability status for the trace.

Trace Triggers - Trace triggers support status.

Trace Start Points - Trace start points support status.

Trace Stop Points - Trace stop points support status.

Trace Ranges - Trace ranges support status.

———— **Note** ————

The information displayed varies depending on the trace source.

Trace Control view options

Start Capture

Click **Start Capture** to start trace capture on the trace capture device. This is the same as the **trace start** command.

Stop Capture

Click **Stop Capture** to stop trace capture on the trace capture device. This is the same as the **trace stop** command.

Clear Trace Buffer

Click **Clear Trace Buffer** to empty the trace buffer on the trace capture device. This is the same as the **trace clear** command.

Automatically start trace capture when target restarts (after a stop)

Select this option to automatically start trace capture after a target restarts after a stop.

Automatically stop trace capture when target stops

Select this option to automatically stop trace capture when a target stops.

Stop trace capture on trigger

Select this option to stop trace capturing after a trace capture trigger has been hit.

Post-trigger capture size

Use this option to control the percentage of trace buffer that should be reserved for after a trigger point is hit. The range is from **0** to **99**.

———— **Note** ————

The **trace start** and **trace stop** commands, and the automatic start and stop trace options act as master switches. Trace triggers cannot override them.

Toolbar and context menu options

The following options are available from the toolbar or context menu:

Linked: *context*

Links this view to the selected connection in the **Debug Control** view. This is the default. Alternatively you can link the view to a specific connection or processor in an *Symmetric MultiProcessing* (SMP) connection. If the connection you want is not shown in the drop-down list, you might have to select it first in the **Debug Control** view.

10.23 Variables view

Describes the view content.

This view enables you to:

- see the contents of variables that are currently in scope
- change the values for variables that are currently in scope
- freeze the selected view to prevent the values being updated by a running target.

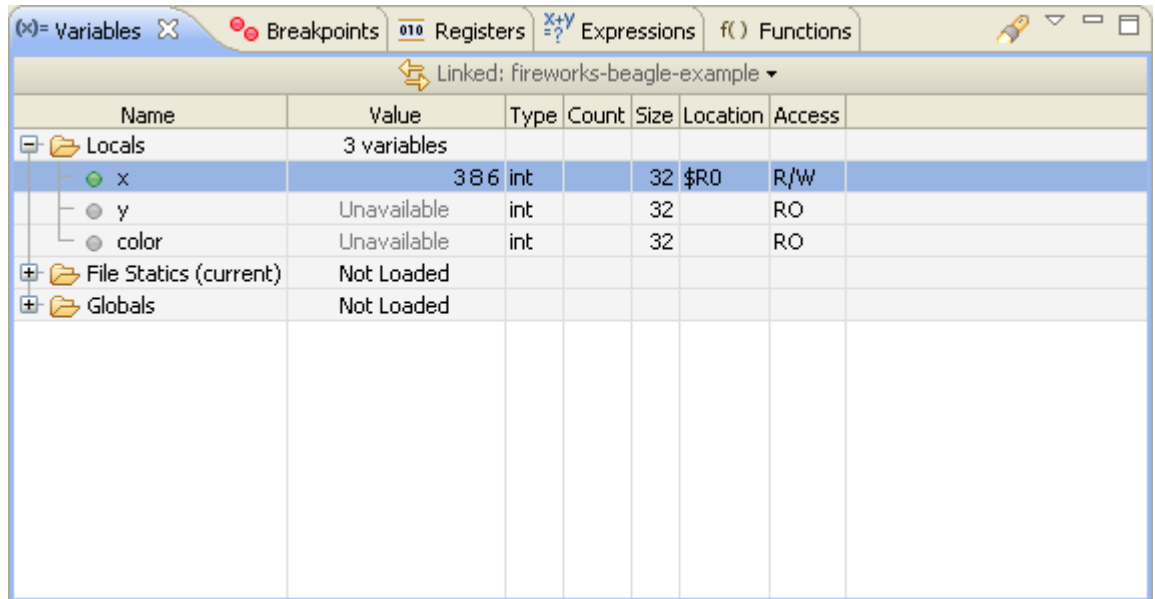


Figure 10-25 Variables view

Right-click on the column headers to select the columns that you want displayed:

Name

The name of the variable.

Value

The value of the variable.

Read-only values are displayed with a grey background. Any other color means that you can edit the value.

A value that you can edit is initially shown with a white background. If the value changes, either by performing a debug action such as stepping or by you editing the value directly, the background changes to yellow.

If you freeze the view, then you cannot change a value.

Type

The type of the variable.

Count

The number of array or pointer elements.

Size

The size of the variable in bits.

Location

The address of the variable.

All columns are displayed by default.

Toolbar and context menu options

The following options are available from the toolbar or context menu:

Linked: **context**

Links this view to the selected connection in the **Debug Control** view. This is the default. Alternatively you can link the view to a specific connection. If the connection you want is not shown in the drop-down list you might have to select it first in the **Debug Control** view.

Search

Searches the data in the current view for a variable.

Copy

Copies the selected variables. To copy the contents of an item such as a structure or an array, you must first expand that item.

This is useful if you want to copy the selected variables to a text editor and compare the values when execution stops at another location.

Select All

Selects all capabilities currently expanded in the view.

Show in Memory

Where enabled, displays the **Memory** view with the address set to either:

- the value of the selected variable, if the variable translates to an address, the address of an array, **&name**
- the location of the variable, the name of an array, **name**.

The memory size is set to the size of the variable, using the **sizeof** keyword.

Show in Registers

If the selected variable is currently held in a register, then displays the **Registers** view with that register selected. For example, the variable **t** might be held in register R5.

Show Dereference in Memory

If the selected variable is a pointer, then displays the **Memory** view with the address where the variable is pointing to in memory, selected.

Send to Selection

Enables you to add variable filters to an **Expressions** view. Displays a sub menu that enables you to add to a specific **Expressions** view.

format list

A list of formats you can use for the variable value. The default is **Unsigned Decimal**.

View Menu

The following **View Menu** options are available:

New Variable View

Displays a new instance of the **Variables** view.

Refresh

Refreshes the view.

Freeze Data

Toggles the freezing of data in the current view. This also disables or enables the **Refresh Variable View** option. Also, you cannot modify the value of a variable if the data is frozen.

If you freeze the data before you expand an item, such as an array, for the first time, the view might show **Pending...** items. Unfreeze the data to see the items.

Editing context menu options

The following options are available on the context menu when you select a variable value for editing:

Cut

Copies and deletes the selected value.

Copy

Copies the selected value.

Paste

Pastes a value that you have previously cut or copied into the selected variable value.

Delete

Deletes the selected value.

Undo

Reverts the last change you made to the selected value.

Right to left reading order

Sets the reading order for the selected variable value to be left or right justified.

Show unicode control characters

Shows any unicode control characters in the selected variable value.

Insert unicode control character

Selects the unicode control character to insert into the selected variable value.

Related concepts

[6.7 About debugging multi-threaded applications on page 6-145.](#)

[6.8 About debugging shared libraries on page 6-146.](#)

[6.9 About debugging a Linux kernel on page 6-148.](#)

[6.10 About debugging Linux kernel modules on page 6-150.](#)

[6.12 About debugging TrustZone enabled targets on page 6-153.](#)

Related references

[Working with data watchpoints.](#)

[4.8 Setting a tracepoint on page 4-119.](#)

[4.6 Working with conditional breakpoints on page 4-114.](#)

[4.6.1 Assigning conditions to an existing breakpoint on page 4-114.](#)

[4.7 About pending breakpoints and watchpoints on page 4-118.](#)

[4.5.5 Exporting DS-5™ breakpoint settings to a file on page 4-112.](#)

[4.5.4 Importing DS-5™ breakpoint settings from a file on page 4-111.](#)

[10 DS-5 Debug perspectives and views on page 10-194.](#)

10.24 Auto Refresh Properties dialog box

Describes the dialog box content.

The dialog box enables you to modify the update intervals settings.

Refresh Off/Blocked

Enables you to modify the stop/block auto refresh.

Update Interval

Specifies the auto refresh interval in seconds.

Update When

Enables you to specify when you want auto refresh to update:

Running

Refreshes the **Memory** view while the target is running.

Stopped

Refreshes the **Memory** view while the target is stopped.

Always

Always refreshes the **Memory** view.

Note

When using the **Running** or **Always** selections, the **Memory** and **Screen** views are only updated if the target supports access to that memory when running. For example, some CoreSight targets support access to physical memory at any time via the *Debug Access Port (DAP)* to the *Advanced High-performance Bus Access Port (AHB-AP)* bridge. In those cases, add the **AHB:** prefix to the address selected in the **Memory** or **Screen** views. This type of access bypasses any cache on the CPU core, so the memory content returned might be different to the value that the core reads.

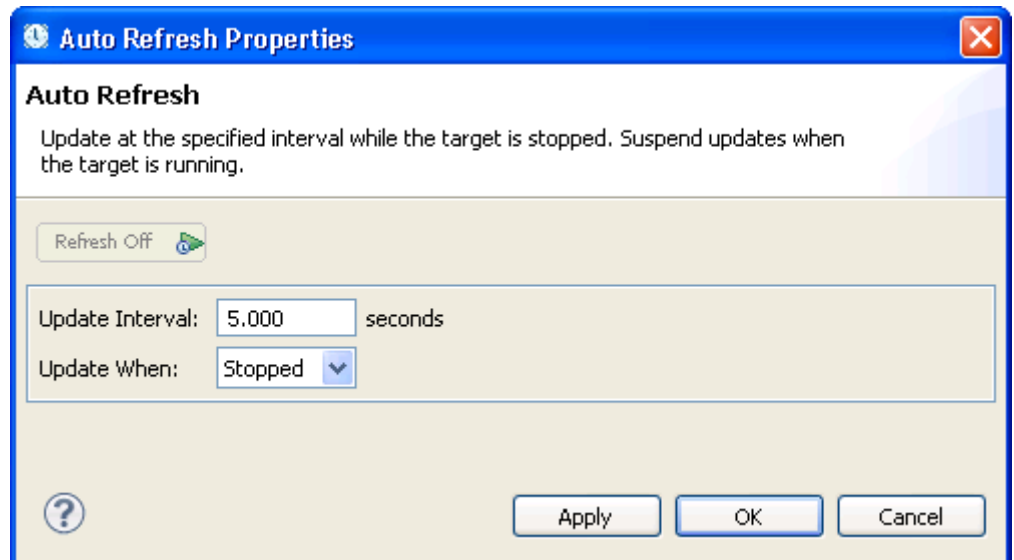


Figure 10-26 Auto Refresh properties dialog box

10.25 Memory Exporter dialog box

Describes the dialog box content.

This dialog box enables you to generate a text file containing the data from a specific region of memory.

Memory Bounds

Specifies the memory region:

Start Address

Specifies the start address for the memory.

End Address

Specifies the inclusive end address for the memory.

Length in Bytes

Specifies the number of bytes.

Output Format

Specifies the output format:

- Binary. This is the default.
- Intel Hex-32.
- Motorola 32-bit (S-records).
- Byte oriented hexadecimal (Verilog Memory Model).

Export Filename

Enter the current location of the output file in the field provided or click on:

- **File System...** to locate the output file in an external folder
- **Workspace...** to locate the output file in a workspace project.

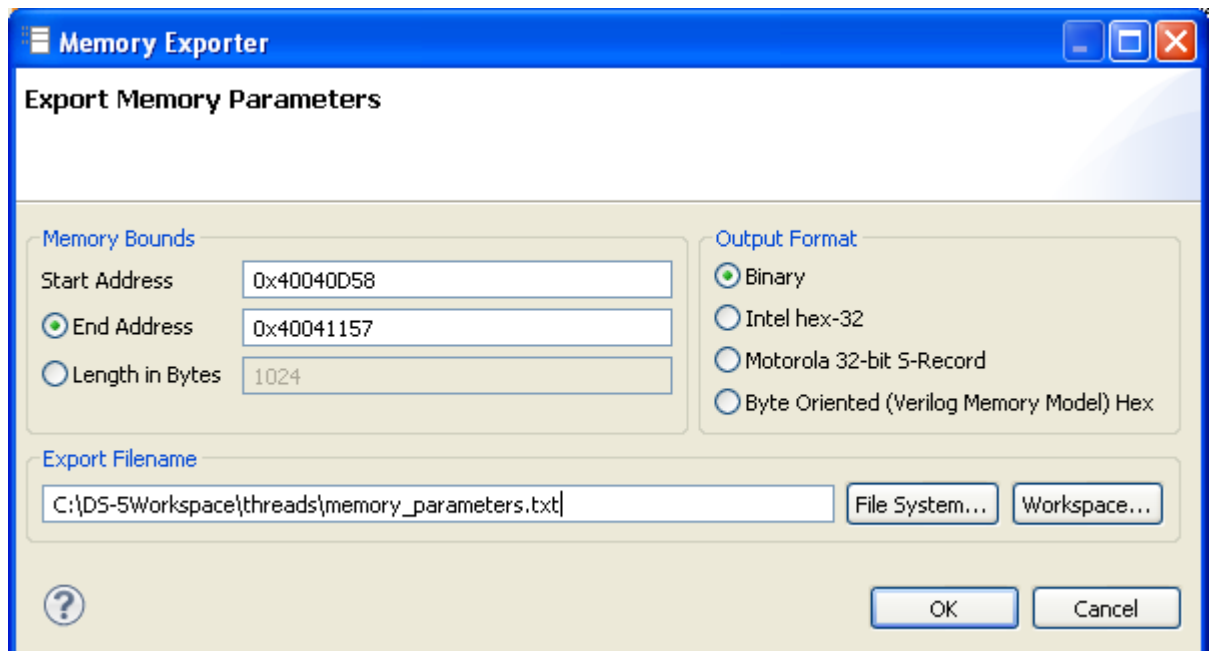


Figure 10-27 Memory Exporter dialog box

10.26 Memory Importer dialog box

Describes the dialog box content.

This dialog box enables you to generate a text file containing the data from a specific region of memory.

Offset to Embedded Address

Specifies an offset that is added to all addresses in the image prior to writing to memory. Some image formats do not contain embedded addresses and in this case the offset is the absolute address where the image is restored

Memory Limit

Enables you to define a specific region of memory that you want to import:

Limit to memory range

Select to specify an address range.

Start

Specifies the minimum address that can be written to. Any data prior to this address is not written. If no address is given then the default is address zero.

End

Specifies the maximum address that can be written to. Any data after this address is not written. If no address is given then the default is the end of the address space.

Import Filename

Select **Import file as binary image** if the file format is a binary file.

Enter the current location of the output file in the field provided or click on:

- **File System...** to locate the output file in an external folder
- **Workspace...** to locate the output file in a workspace project.

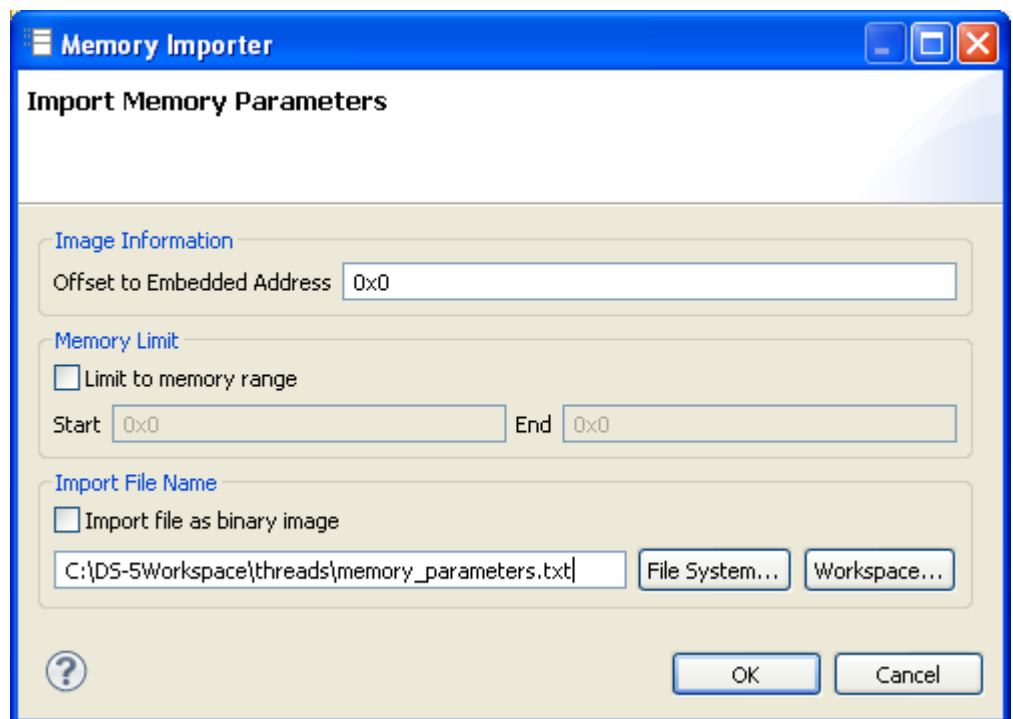


Figure 10-28 Memory Importer dialog box

10.27 Fill Memory dialog box

Describes the dialog box content.

This dialog box enables you to write a specific pattern of bytes to memory.

Memory Bounds

Specifies the memory region:

Start Address

Specifies the start address for the memory.

End Address

Specifies the inclusive end address for the memory.

Length in Bytes

Specifies the number of bytes.

Data Pattern

Specifies the fill size and pattern of bytes.

Fill size

Specifies the fill size in bytes.

Pattern

Enter the specific pattern of bytes.

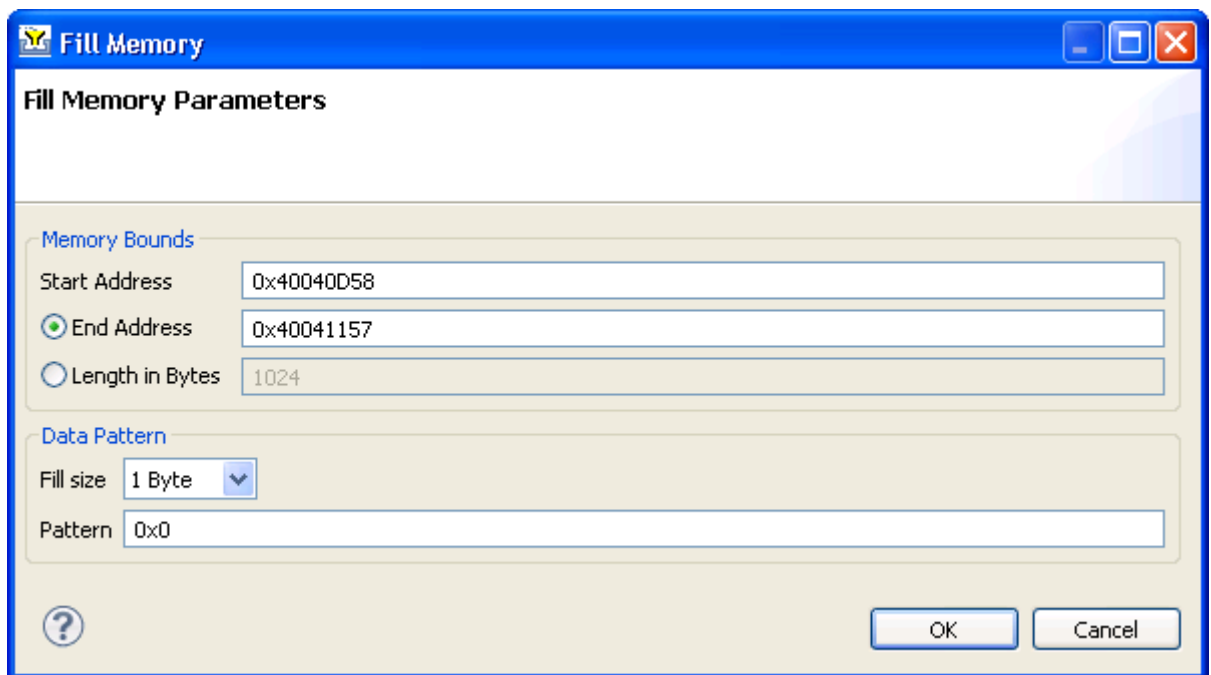


Figure 10-29 Memory Fill dialog box

10.28 Export trace report dialog box

Describes the dialog box content.

This dialog box enables you to generate a trace report.

Report Name

Enter the report location and name.

Base Filename

Enter the report name.

Output Folder

Enter the report folder location.

Browse

Selects the report location in the file system.

Include core

Enables you to add the core name in the report filename.

Include date time stamp

Enables you to add the date time stamp to the report filename.

Select source for trace report

Selects the required trace data:

Use trace view as report source

Instructions that are currently visible in the trace view.

Use trace buffer as report source

Trace data that is currently contained in the trace buffer.

———— Note ————

When specifying a range, ensure that the range is large enough otherwise you might not get any trace output. This is due to the trace packing format used in the buffer.

Report Format

Configures the report:

Output Format

Selects the output format.

Include column headers

Enables you to add column headers in the first line of the report.

Select columns to export

Enables you to filter the trace data in the report.

Record Filters

Enables or disables trace filters.

Check All

Enables you to select all the trace filters.

Uncheck All

Enables you to unselect all the trace filters.

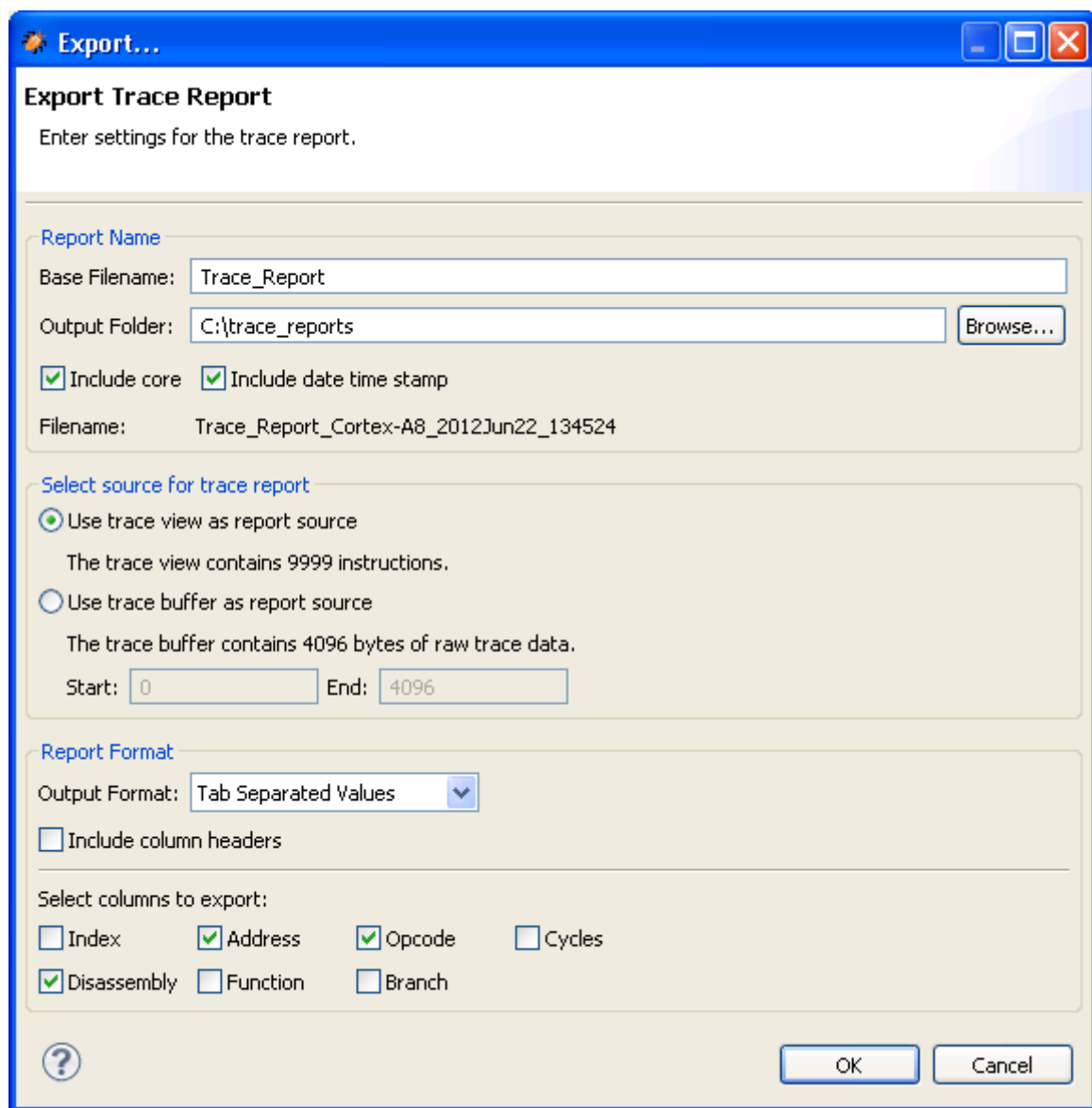


Figure 10-30 Export trace report dialog box

10.29 Breakpoint properties dialog box

Describes the dialog box content.

This dialog box enables you to:

- display the properties of a selected breakpoint
- set a conditional expression for a specific breakpoint
- set an ignore counter for a specific breakpoint
- specify a script file to run when the selected breakpoint is hit
- enable the debugger to automatically continue running on completion of all the breakpoint actions
- assign a breakpoint action to a specific thread or processor, if available.

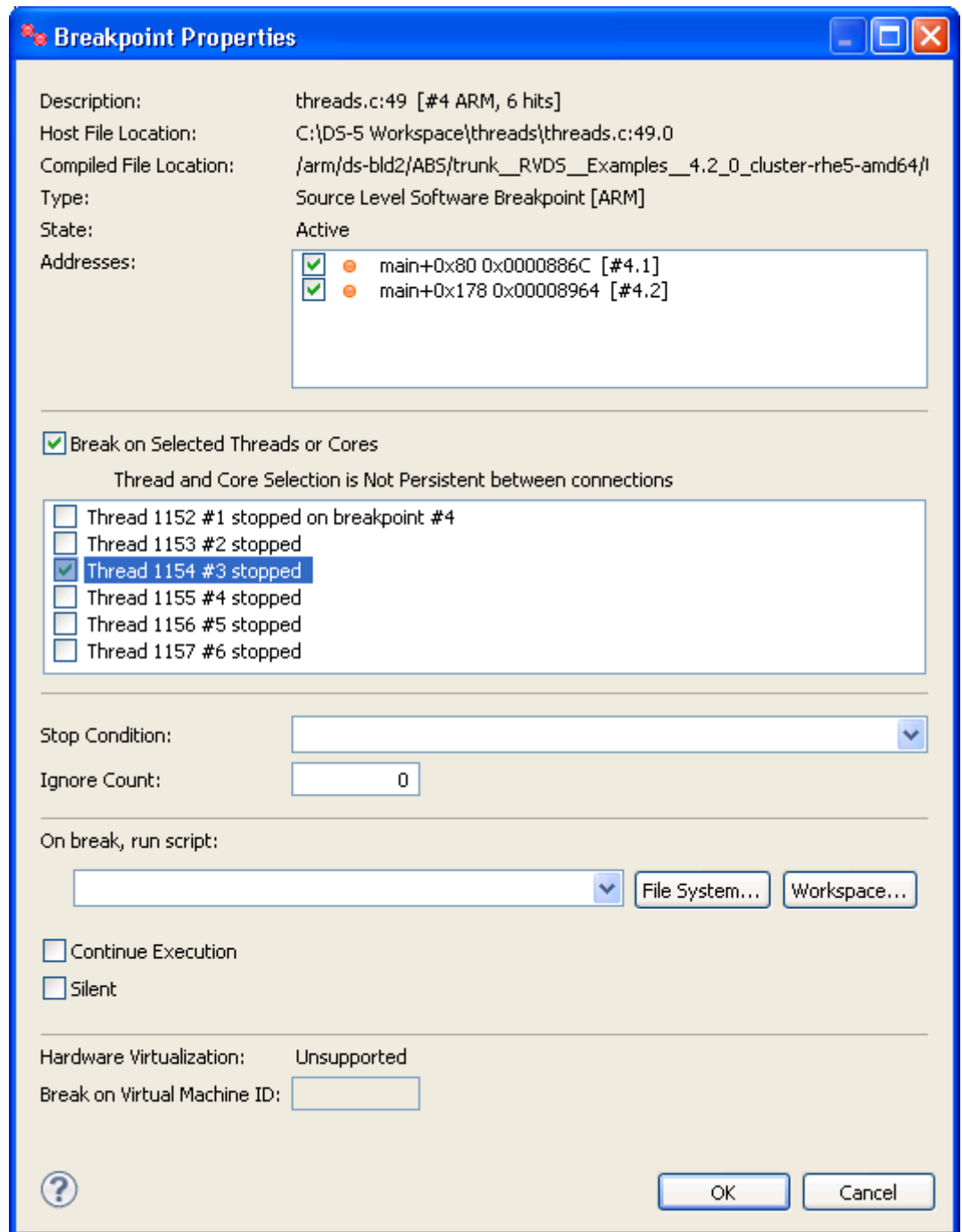


Figure 10-31 Breakpoint properties dialog box

Breakpoint information

The breakpoint information shows the basic properties of a breakpoint:

Description

A description of the breakpoint as displayed in the **Breakpoints** view. This comprises:

- The name of the function in which the breakpoint is set and the number of bytes from the start of the function. For example, `accumulate()+52` shows that the breakpoint is 52 bytes from the start of the `accumulate()` function.
- If the source file is available, the file name and line number in the file where the breakpoint is set, `threads.c:115`.
- A breakpoint ID number, `#N`. In some cases, such as in a **for** loop, a breakpoint might comprise a number of sub-breakpoints. These are identified as `N.n`, where `N` is the number of the parent. The description of a sub-breakpoint in this dialog box is shown as

```
main()+132sub-breakpoint ofmain()+132 @ threads.c:56 [#14 ARM]
(threads)
```

- The type of instruction at the address of the breakpoint, ARM or Thumb.
- An **ignore** count, if set. The display format is:

`ignore = num/count`

`num` equals `count` initially, and decrements on each pass until it reaches zero.

`count` is the value you have specified for the **ignore** count.
- A **hits** count that increments each time the breakpoint is hit. This is not displayed until the first hit. If you set an **ignore** count, **hits** count does not start incrementing until the **ignore** count reaches zero.
- The stop condition you have specified, `(i==3)`.
- The name of the image.

:

```
accumulate()+52 @ threads.c:115 [#1 ARM, ignore = 3/3, 3 hits, (i==3)]
(threads)
```

Location

The location of the source file containing the address where the breakpoint is set, for example:

```
C:/Myprojects/Eclipse/workspace_ds5/threads/threads.c:115.0
```

If no source file is available, then **Unknown** is displayed.

Type

This shows:

- whether or not the source file is available for the code at the breakpoint address, **Source Level** if available or **Address Level** if not available
- if the breakpoint is on code in a shared object, **Auto** indicates that the breakpoint is automatically set when that shared object is loaded
- if the breakpoint is **Active**, the type of breakpoint, either **Software Breakpoint** or **Hardware Breakpoint**
- the type of instruction at the address of the breakpoint, ARM or Thumb.

:

```
Source Level Software Breakpoint [ARM]
```

State

Indicates one of the following:

Active

The image or shared object containing the address of the breakpoint is loaded, and the breakpoint is set.

No Connection

The breakpoint is in an application on a target that is not connected.

Pending

The image or shared object containing the address of the breakpoint has not yet been loaded. The breakpoint becomes active when the image or shared object is loaded.

Address

A dialog box that displays one or more breakpoint or sub-breakpoint addresses with check boxes where you can enable or disable them.

Temporary

Shows `true` if this is a temporary breakpoint. Otherwise, shows `false`.

Breakpoint options

The following options are available for you to set:

Stop Condition

Specify a C-style conditional expression for the selected breakpoint. For example, to activate the breakpoint when the value of `x` equals `10`, specify `x==10`.

Ignore Count

Specify the number of times the selected breakpoint is ignored before it is activated.

The debugger decrements the counter on each pass until it reaches zero, for example:

```
main()+140 @ threads.c:51 [#1 ARM, ignore = 2/3] (threads)
```

When the value reaches zero the breakpoint activates. Each subsequent pass causes the breakpoint to activate.

Select the **Reset Ignore Count** option from the context menu to reset the counter to the value you have set and delay activation again.

On break, run script

Specify a script file to run when the selected breakpoint is activated.

———— Note —————

Take care with the commands you use in a script that is attached to a breakpoint. For example, if you use the `quit` command in a script, the debugger disconnects from the target when the breakpoint is hit.

Continue Execution

Select this option if you want to continue running the target after the breakpoint is activated.

Silent

Controls the printing of stop messages for the selected breakpoint.

Break on Selected Threads or Cores

Select this option if you want to set a breakpoint for a specific thread or processor. This option is disabled if none are available.

When a breakpoint activates, the debugger does the following:

- displays a message in the **Commands** view, for example:

```
Execution stopped at breakpoint 1: 0x00008850
In thread 1 (OS thread id 1078)
0x00008850 51,0 thread_app_data[t].thread = t;
```

- increments a hit count for the breakpoint, for example:
`main()+140 @ threads.c:51 [#1 ARM, ignore = 0/3, 2 hits] (threads)`

10.30 Watchpoint properties dialog box

Describes the dialog box content.

This dialog box enables you to:

- display the properties of a selected watchpoint
- change the watchpoint type.

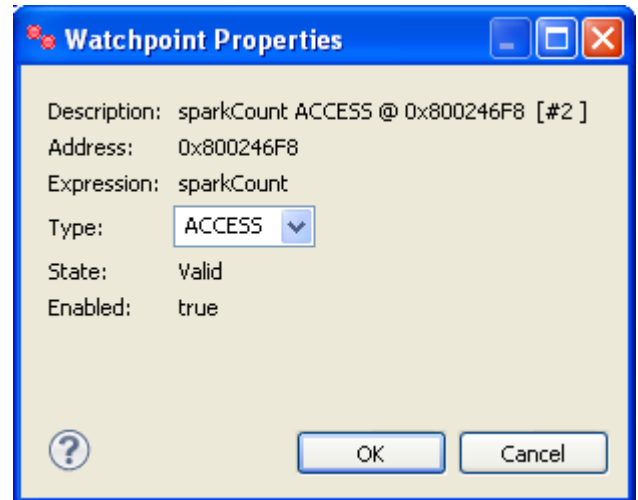


Figure 10-32 Watchpoint properties dialog box

The following types are available:

READ

The debugger stops the target when the memory is read

WRITE

The debugger stops the target when the memory is written

ACCESS

The debugger stops the target when the memory is read or written.

10.31 Tracepoint properties dialog box

Describes the dialog box content.

This dialog box enables you to display the properties of a selected tracepoint.

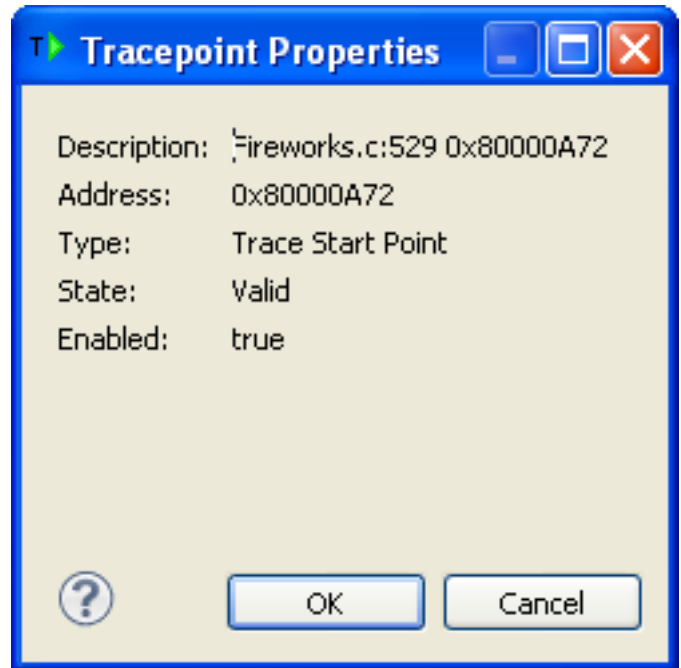


Figure 10-33 Tracepoint properties dialog box

The following types are available:

Trace Start Point

Enables trace capture when it is hit.

Trace Stop Point

Disables trace capture when it is hit.

Trace Trigger Point

Starts trace capture when it is hit.

Note

Tracepoint behavior might vary depending on the selected target.

10.32 Manage Signals dialog box

Describes the dialog box content.

This dialog box enables you to control the handler (vector catch) settings for one or more signals or processor exceptions. When a signal or processor exception occurs you can choose to stop execution, print a message, or both. **Stop** and **Print** are selected for all signals by default.

———— **Note** —————

When connected to an application running on a remote target using gdbserver, the debugger handles Unix signals but on bare-metal targets with no operating system it handles processor exceptions.

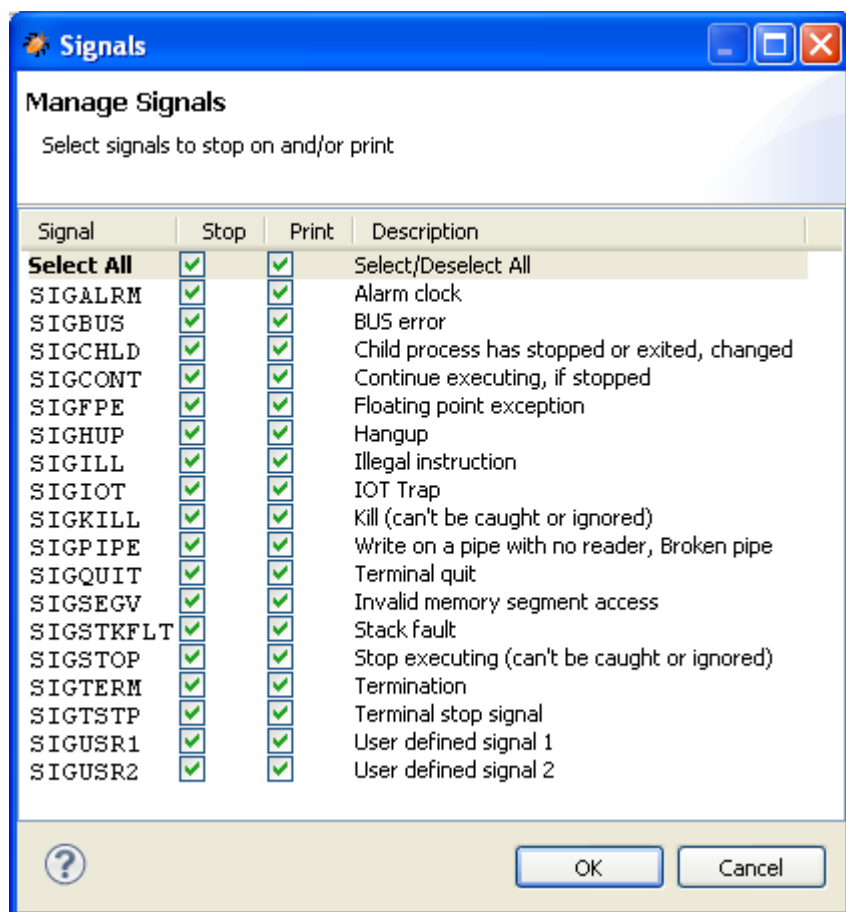


Figure 10-34 Managing signal handler settings

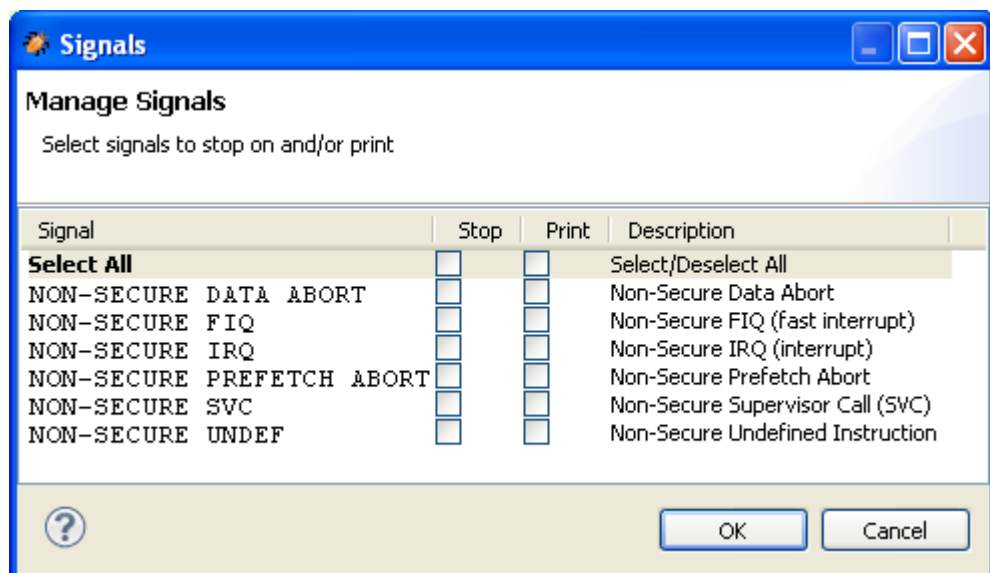


Figure 10-35 Manage exception handler settings

10.33 Event Viewer dialog box

Describes the dialog box content.

This dialog box enables you to select a trace source in addition to masters and channels to display in the view.

Select a Trace Source

Enables you to select a trace source.

Masters

Enables you to select the Masters that you want to display in the **Events** view. Masters are only available for STM trace.

All Masters

Selects all the masters.

Clear Masters

Discards all the masters.

———— Note ————

These options do not modify the data in the channels field.

Channels

Enables you to provide a list or group of channels.

Add All

Sets the channels of the selected master to the default value.

Clear

Clears the channels of the selected master.

———— Note ————

A master with zero channels displays no data.

Apply

Reorganizes the current channels into a canonical form.

Import

Enables you to import an existing configuration.

Export

Enables you to save an existing configuration for use with a different launch configuration.

Reset to Defaults

Resets the configuration to the default configuration.

OK

Reorganizes the current channels into a canonical form, saves the settings, and closes the dialog box.

Cancel

Enables you to cancel unsaved changes.

10.34 Functions Filter dialog box

Describes the dialog box content.

This dialog box enables you to filter the list of symbols that are displayed in the **Functions** view.

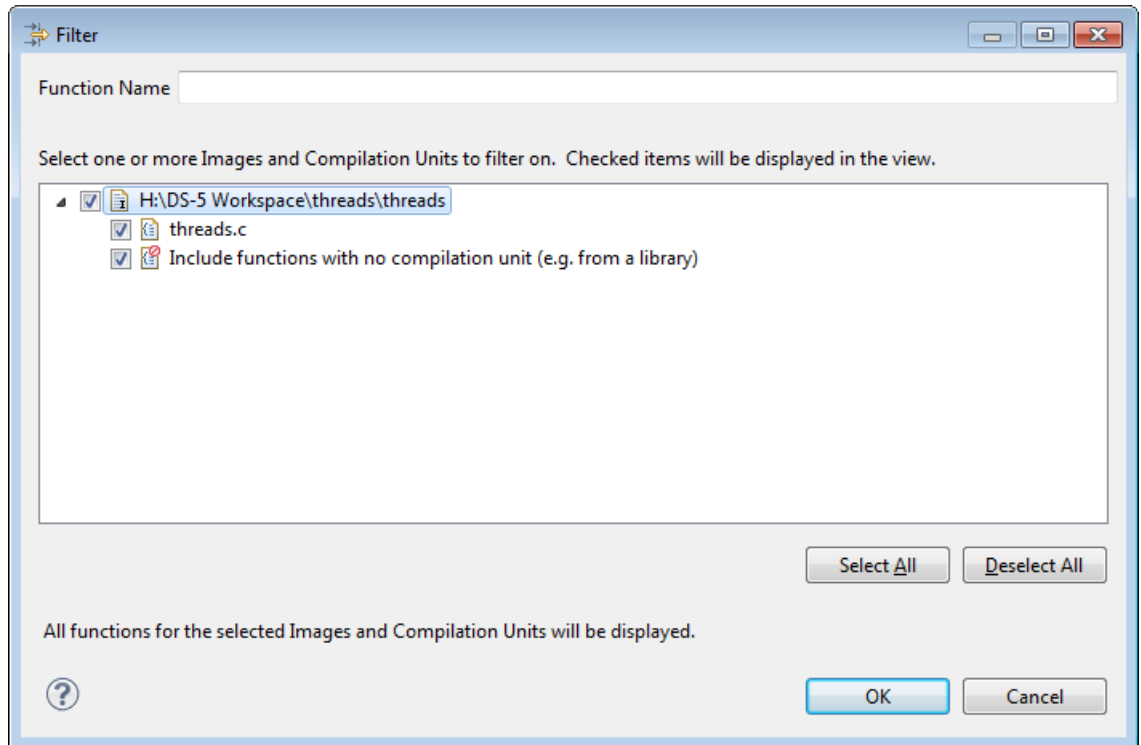


Figure 10-36 Function filter dialog box

10.35 Jython Script Parameters dialog box

Describes the dialog box content.

This dialog box enables you to specify script parameters.

Script Parameters

Enter parameters for the selected script in the text field. Parameters must be space delimited.

Variables...

This button opens the Select Variable dialog box where you can select variables that are passed to the application when the debug session starts. For more information on Eclipse variables, use the dynamic help.

OK

Saves the current parameters and closes the Jython Script Parameters dialog box.

Cancel

Closes the Jython Script Parameters dialog box without saving the parameters.

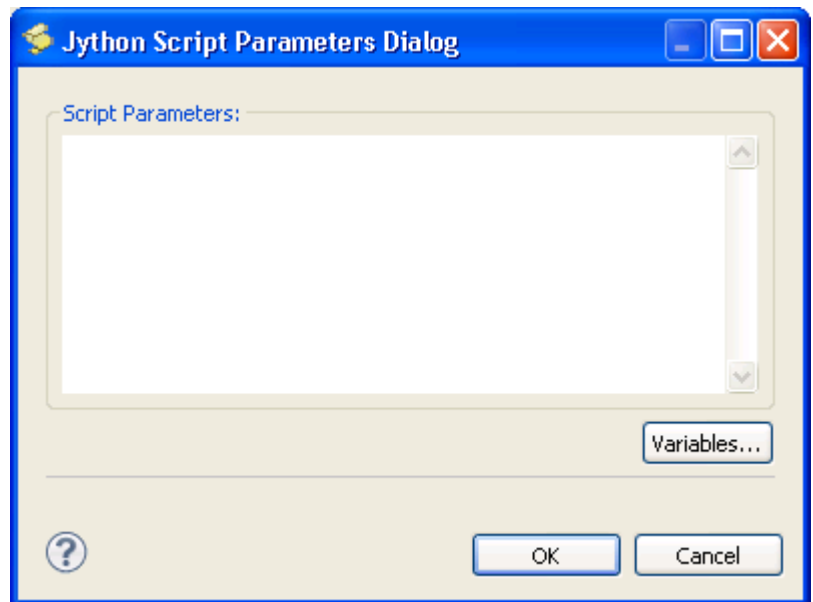


Figure 10-37 Jython Script Parameters dialog box

10.36 Debug Configurations - Connection tab

Describes the tab content.

The **Connection** tab in the Debug Configurations dialog box enables you to configure DS-5 Debugger target connections. Each configuration you create is associated with a single target processor.

If the development platform has multiple processors, then you must create a separate configuration for each processor. Be aware that when connecting to multiple targets you cannot perform synchronization or cross-triggering operations.

———— **Note** —————

Options in the **Connection** tab are dependent on the type of platform that you select.

Select target

These options enable you to select the target manufacturer, board, project type, and debug operation.

DTSL Options

Select **Edit...** to open a dialog box to configure additional debug and trace settings.

Connections

These options enable you to configure the connection between the debugger and the target:

RSE connection

A list of *Remote Systems Explorer* (RSE) configurations that you have previously set up. Select the required RSE configuration that you want to use for this debug configuration.

Android devices

A list of Android devices that you have previously configured. Select the required device that you want to use for this debug configuration.

Connect as root

Select to give **root** access when starting **gdbserver**. This option is dependent on the selected debug operation and might not be available.

gdbserver (TCP)

Specify the target IP address or name and the associated port number for the connection between the debugger and **gdbserver**.

The following options might also be available, depending on the debug operation you selected:

- Select the **Use Extended Mode** checkbox if you want to restart an application under debug. Be aware that this might not be fully implemented by **gdbserver** on all targets.
- Select the **Terminate gdbserver on disconnect** checkbox to terminate **gdbserver** when you disconnect from the target.
- Select the **Use RSE Host** checkbox to connect to **gdbserver** using the RSE configured host.

gdbserver (serial)

Specify the local serial port and connection speed for the serial connection between the debugger and **gdbserver**.

For model connections, details for **gdbserver** are obtained automatically from the target.

Select the **Use Extended Mode** checkbox if you want to restart an application under debug. Be aware that this might not be fully implemented by **gdbserver** on all targets.

Bare Metal Debug

Specify the target IP address or name of the debug hardware adapter. You can also click on **Browse...** to display all the available debug hardware adapters on your local subnet or USB connections.

Model parameters

Specify the parameter for launching the model.

Model parameters (pre-configured to boot ARM Embedded application)

These options are only enabled for the pre-configured option that boots an ARM Embedded *Fixed Virtual Platform*.

You can configure a *Virtual File System* (VFS) that enables a model to run an application and related shared library files from a directory on the local host. Alternatively, you can disable VFS and manually transfer the files to a directory on the model.

Enable Virtual File System support

Enable or disable the use of VFS.

Host mount point

Specify the location of the file system on the local host; you can select the workspace root directory.

- enter the location in the field provided
- click on **File System...** to locate the directory in an external location from the workspace
- click on **Workspace...** to locate the directory within the workspace.

Remote target mount point

Displays the default location of the file system on the model. The default is the **/writeable** directory.

Apply

Save the current configuration. This does not connect to the target.

Revert

Undo any changes and revert to the last saved configuration.

Debug

Connect to the target and close the Debug Configurations dialog box.

Close

Close the Debug Configurations dialog box.

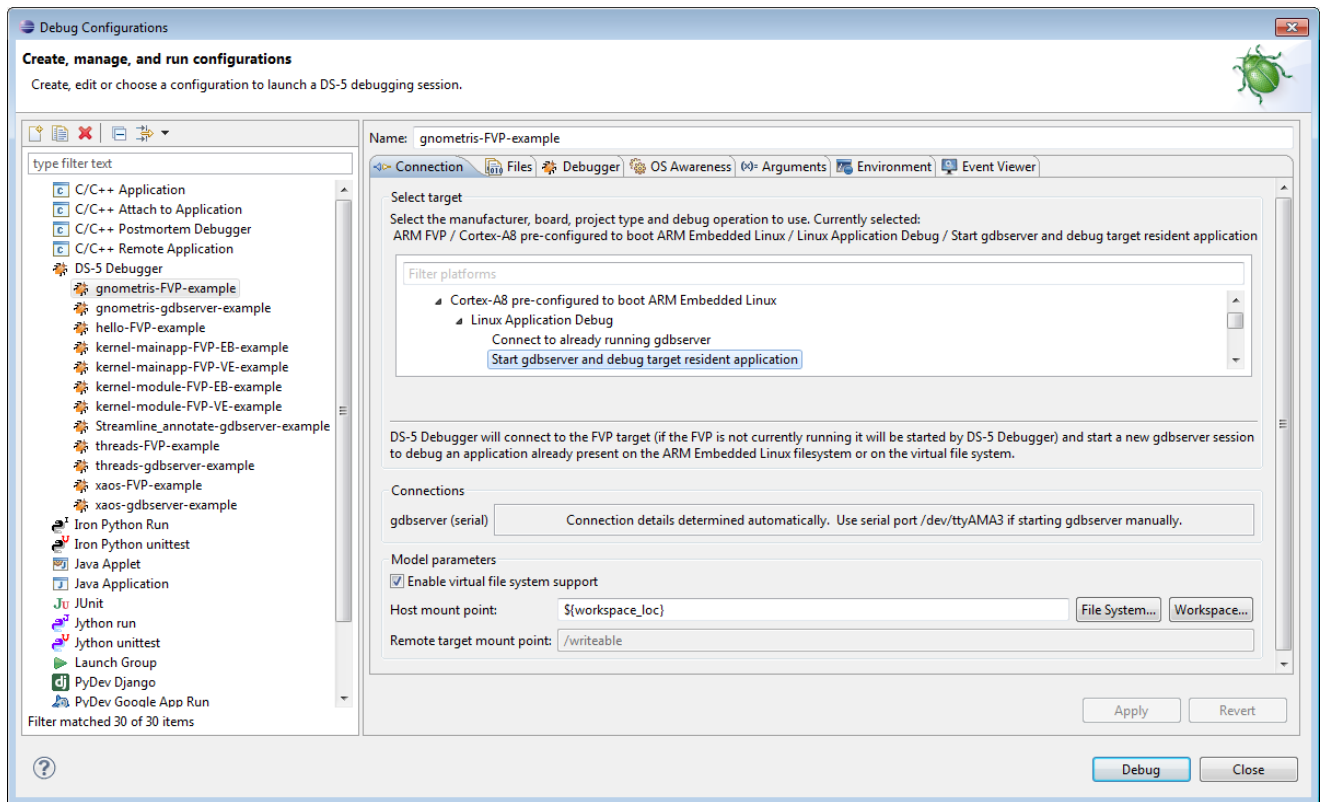


Figure 10-38 Connection configuration for a model using VFS

10.37 Debug Configurations - Files tab

Describes the tab content.

The **Files** tab in the Debug Configurations dialog box enables you to select debug versions of the application file and libraries on the host that you want the debugger to use. You can also specify the target file system folder to which files can be transferred if required.

———— **Note** ————

Options in the **Files** tab depend on the type of platform and debug operation that you select.

Files

These options enable you to configure the target file system and select files on the host that you want to download to the target or use by the debugger. The **Files** tab options available for each **Debug operation** are:

Table 10-1 Files tab options available for each Debug operation

	Download and debug application	Debug target resident application	Connect to already running gdbserver	Debug via DSTREAM\RV1	Debug and ETB Trace via DSTREAM\RV1
Application on host to download	Yes	-	-	Yes	Yes
Application on target	-	Yes	-	-	-
Target download directory	Yes	-	-	-	-
Target working directory	Yes	Yes	-	-	-
Load symbols from file	Yes	Yes	Yes	Yes	Yes
Other file on host to download	Yes	-	-	-	-
Path to target system root directory	Yes	Yes	Yes	-	-

Apply

Save the current configuration. This does not connect to the target.

Revert

Undo any changes and revert to the last saved configuration.

Debug

Connect to the target and close the Debug Configurations dialog box.

Close

Close the Debug Configurations dialog box.

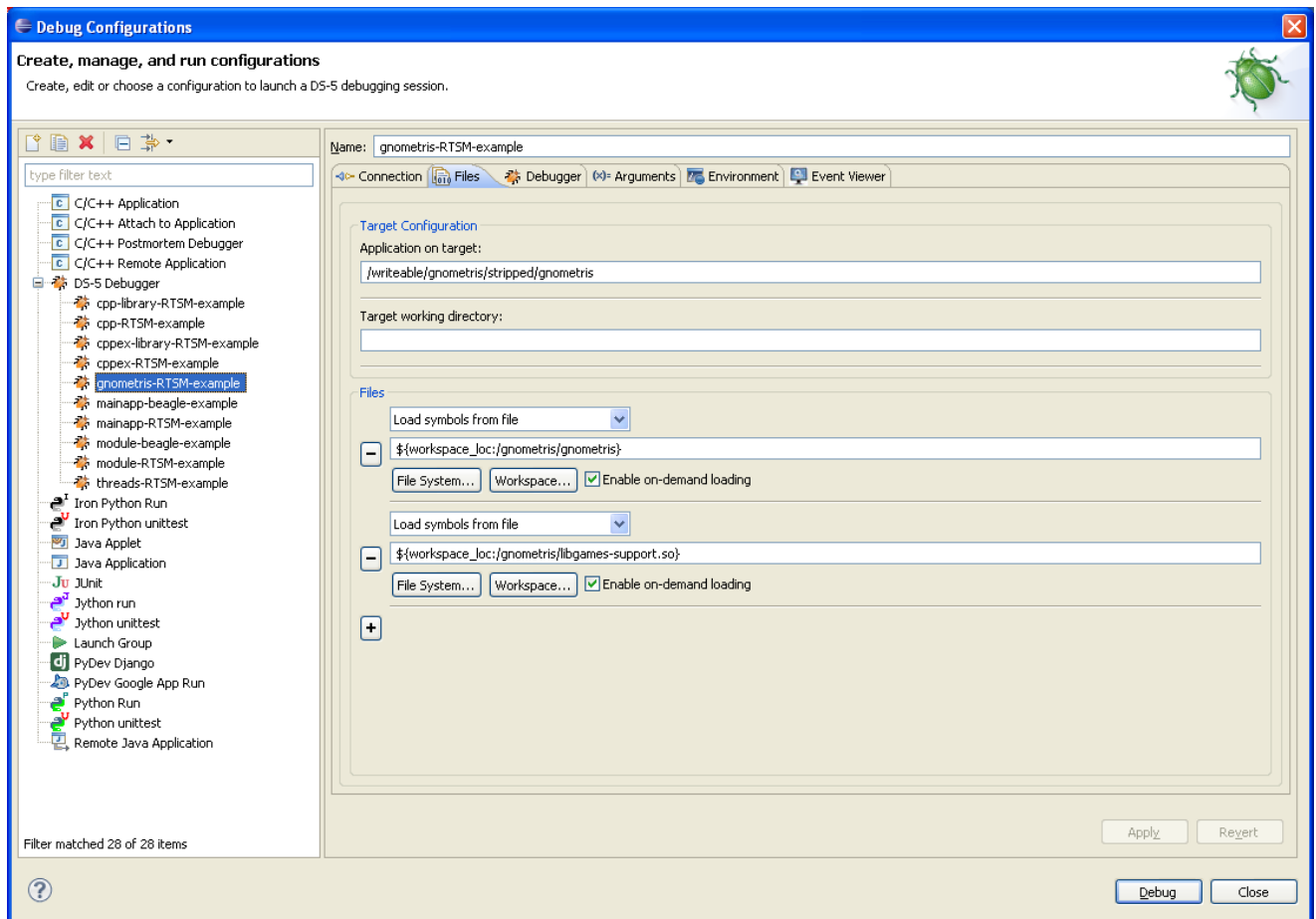


Figure 10-39 File system configuration for an application on a model

Files options summary

The Files options available depend on the debug operation you selected on the **Connection** tab. The possible Files options are:

Application on host to download

Specify the application image file on the host that you want to download to the target:

- enter the host location and file name in the field provided
- click on **File System...** to locate the file in an external directory from the Eclipse workspace
- click on **Workspace...** to locate the file in an project directory or sub-directory within the Eclipse workspace.

For example, to download the stripped (no debug) Gnometris application image select the `gnometris/stripped/gnometris` file.

Select **Load symbols** to load the debug symbols from the specified image.

Select **Enable on-demand loading** to specify how you want the debugger to load debug information. Enabling this option can provide a faster load and use less memory but debugging might be slower.

Project directory

Specify the Android project directory on the host:

- enter the host location in the field provided
- click on **File System...** to locate the project directory in an external location from the Eclipse workspace
- click on **Workspace...** to locate the project directory from within the Eclipse workspace.

APK file

Specify the Android APK file on the host that you want to download to the target:

- enter the host location and file name in the field provided
- click on **File System...** to locate the file in an external directory from the Eclipse workspace
- click on **Workspace...** to locate the file in an project directory or sub-directory within the Eclipse workspace.

Process

This field is automatically populated from the `AndroidManifest.xml` file.

Activity

This field is automatically populated from the `AndroidManifest.xml` file.

Application on target

Specify the location of the application on the target. `gdbserver` uses this to launch the application.

For example, to use the stripped (no debug) Gnometriz application image when using a model and VFS is configured to mount the host workspace as `/writeable` on the target, specify the following in the field provided:

`/writeable/gnometriz/stripped/gnometriz`.

Target download directory

If the target has a preloaded image, then you might have to specify the location of the corresponding image on your host.

The debugger uses the location of the application image on the target as the default current working directory. To change the default setting for the application that you are debugging, enter the location in the field provided. The current working directory is used whenever the application references a file using a relative path.

Load symbols

Specify the application image containing the debug information to load:

- enter the host location and file name in the field provided
- click on **File System...** to locate the file in an external directory from the workspace
- click on **Workspace...** to locate the file in a project directory or sub-directory within the workspace.

For example, to load the debug version of Gnometriz you must select the `gnometriz` application image that is available in the `gnometriz` project root directory.

Although you can specify shared library files here, the usual method is to specify a path to your shared libraries with the **Shared library search directory** option on the **Debugger** tab.

———— **Note** —————

Load symbols is ticked by default.

Add peripheral description files from directory

A directory with configuration files defining peripherals that must be added before connecting to the target.

Other file on host to download

Specify other files that you want to download to the target:

- Enter the host location and file name in the field provided.
- Click on **File System...** to locate the file in an external directory from the workspace.
- Click on **Workspace...** to locate the file in a project directory or sub-directory within the workspace.

For example, to download the stripped (no debug) Gnometriz shared library to the target you can select the `gnometris/stripped/libgames-support.so` file.

Path to target system root directory

Specifies the system root directory to search for shared library symbols.

The debugger uses this directory to search for a copy of the debug versions of target shared libraries. The system root on the host workstation must contain an exact representation of the libraries on the target root filesystem.

Target working directory

If this field is not specified, the debugger uses the location of the application image on the target as the default current working directory. To change the default setting for the application that you are debugging, enter the location in the field provided. The current working directory is used whenever the application refers to a file using a relative path.

Remove this resource from the list

To remove a resource from the configuration settings, click this button next to the resource that you want to remove.

Add a new resource to the list

To add a new resource to the file settings, click this button and then configure the options as required.

Related concepts

[6.9 About debugging a Linux kernel on page 6-148.](#)

10.38 Debug Configurations - Debugger tab

Describes the tab content.

The **Debugger** tab in the Debug Configurations dialog box enables you to specify the actions that you want the debugger to do after connection to the target.

Run Control

These options enable you to define the running state of the target when you connect:

Connect only

Connect to the target, but do not run the application.

———— Note ————

The PC register is not set and pending breakpoints or watchpoints are subsequently disabled when a connection is established.

Debug from entry point

Run the application when a connection is established, then stop at the image entry point.

Debug from symbol

Run the application when a connection is established, then stop at the address of the specified symbol. The debugger must be able to resolve the symbol. If you specify a C or C++ function name, then do not use the () suffix.

If the symbol can be resolved, execution stops at the address of that symbol.

If the symbol cannot be resolved, a message is displayed in the **Commands** view warning that the symbol cannot be found. The debugger then attempts to stop at the image entry point.

Run target initialization debugger script (.ds)

Select this option to execute target initialization scripts (a file containing debugger commands) immediately after connection. To select a file:

- enter the location and file name in the field provided
- click on **File System...** to locate the file in an external directory from the workspace
- click on **Workspace...** to locate the file in a project directory or sub-directory within the workspace.

Run debug initialization debugger script (.ds)

Select this option to execute debug initialization scripts (a file containing debugger commands) after execution of any target initialization scripts and also running to an image entry point or symbol, if selected. To select a file:

- enter the location and file name in the field provided
- click on **File System...** to locate the file in an external directory from the workspace
- click on **Workspace...** to locate the file in a project directory or sub-directory within the workspace.

———— Note ————

You might have to insert a **wait** command before a **run** or **continue** command to enable the debugger to connect and run the application to the specified function.

Execute debugger commands

Enter debugger commands in the field provided if you want to automatically execute specific debugger commands that run on completion of any initialization scripts. Each line must contain only one debugger command.

Host working directory

The debugger uses the Eclipse workspace as the default working directory on the host. To change the default setting for the application that you are debugging, deselect the **Use default** check box and then:

- enter the location in the field provided
- click on **File System...** to locate the external directory
- click on **Workspace...** to locate the project directory.

Paths

You can modify the search paths on the host used by the debugger when it displays source code.

Source search directory

Specify a directory to search for source files:

- enter the location and file name in the field provided
- click on **File System...** to locate the directory in an external location from the workspace
- click on **Workspace...** to locate the directory within the workspace.

Shared library search directory

Specify a directory to search for shared libraries:

- enter the location in the field provided
- click on **File System...** to locate the directory in an external location from the workspace
- click on **Workspace...** to locate the directory within the workspace.

Remove this resource from the list

To remove a search path from the configuration settings, click this button next to the resource that you want to remove.

Add a new resource to the list

To add a new search path to the configuration settings, click this button and then configure the options as required.

Apply

Save the current configuration. This does not connect to the target.

Revert

Undo any changes and revert to the last saved configuration.

Debug

Connect to the target and close the Debug Configurations dialog box.

Close

Close the Debug Configurations dialog box.

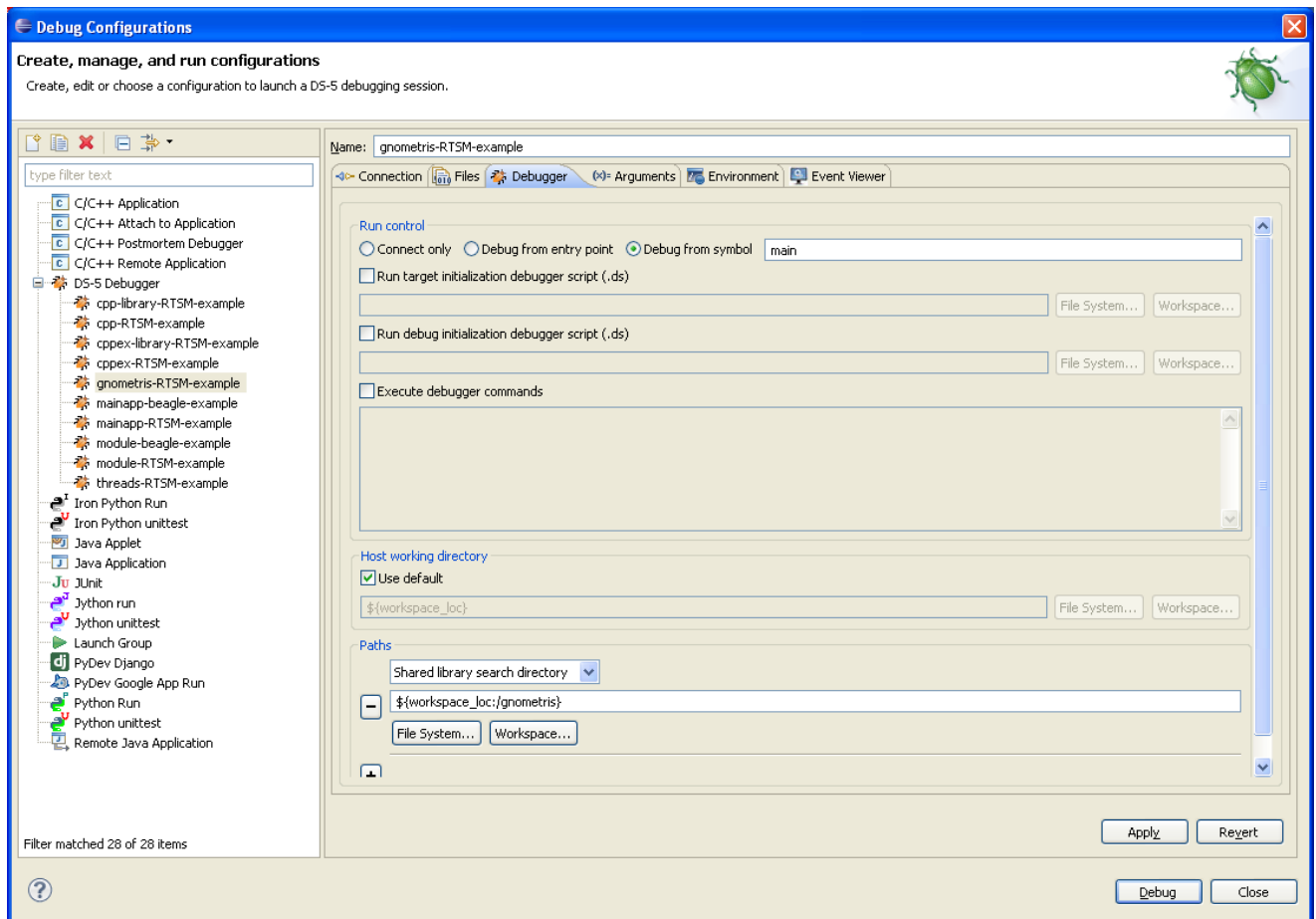


Figure 10-40 Debugger configuration to set application starting point and search paths

Related concepts

[6.9 About debugging a Linux kernel on page 6-148.](#)

10.39 Debug Configurations - OS Awareness tab

Describes the OS Awareness tab content.

The **OS Awareness** tab in the Debug Configurations dialog box enables you to inform the debugger of the *Operating system* (OS) the target is running. This enables the debugger to provide additional functionality specific to the selected OS.

Multiple options are available in the drop-down box and its content is controlled by the selected platform and connection type in the **Connection** tab. OS awareness depends on having debug symbols for the OS loaded within the debugger.

———— Note ————

Linux OS awareness is not currently available in this tab, and remains in the **Connection** tab as a separate debug operation.

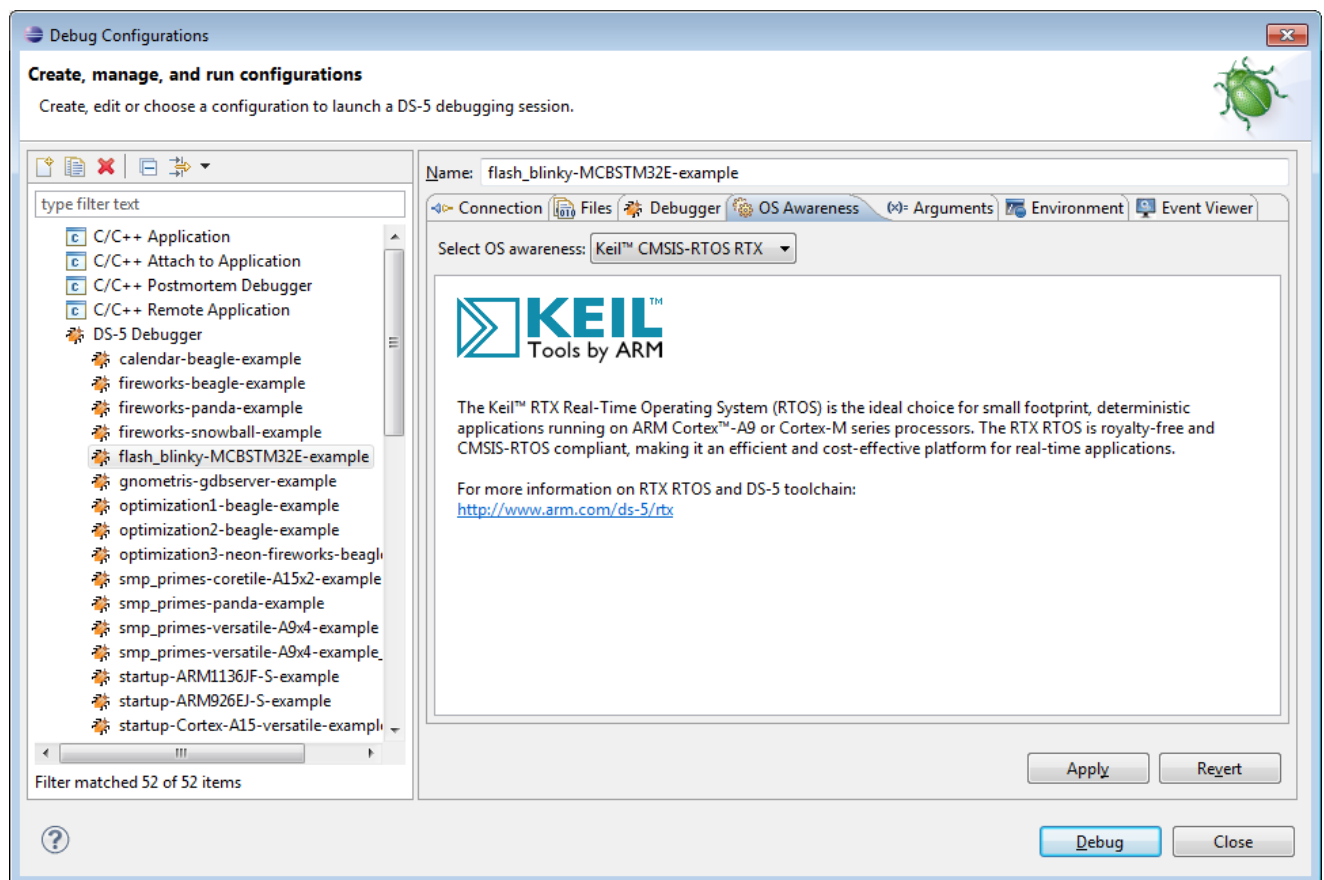


Figure 10-41 OS Awareness tab

10.40 Debug Configurations - Arguments tab

Describes the tab content.

If your application accepts command-line arguments to `main()`, you can specify the values to pass to the application when execution starts.

The **Arguments** tab in the Debug Configurations dialog box enables you to enter arguments that are passed to the application.

Note

These settings only apply if the target supports semihosting and they cannot be changed while the target is running.

The **Arguments** tab contains the following elements:

Program Arguments

This panel enables you to enter the arguments. Arguments are passed to the target application unmodified except when the text is an eclipse argument variable of the form `${var_name}` where Eclipse replaces it with the related value.

For a Linux target you might have to escape some characters using a backslash (`\`) character. For example, the `@`, `(`, `)`, `"`, and `#` characters must be escaped.

Variables...

This button opens the Select Variable dialog box where you can select variables that are passed to the application when the debug session starts. For more information on variables, use the dynamic help.

Apply

Save the current configuration. This does not connect to the target.

Revert

Undo any changes and revert to the last saved configuration.

Debug

Connect to the target and close the Debug Configurations dialog box.

Close

Close the Debug Configurations dialog box.

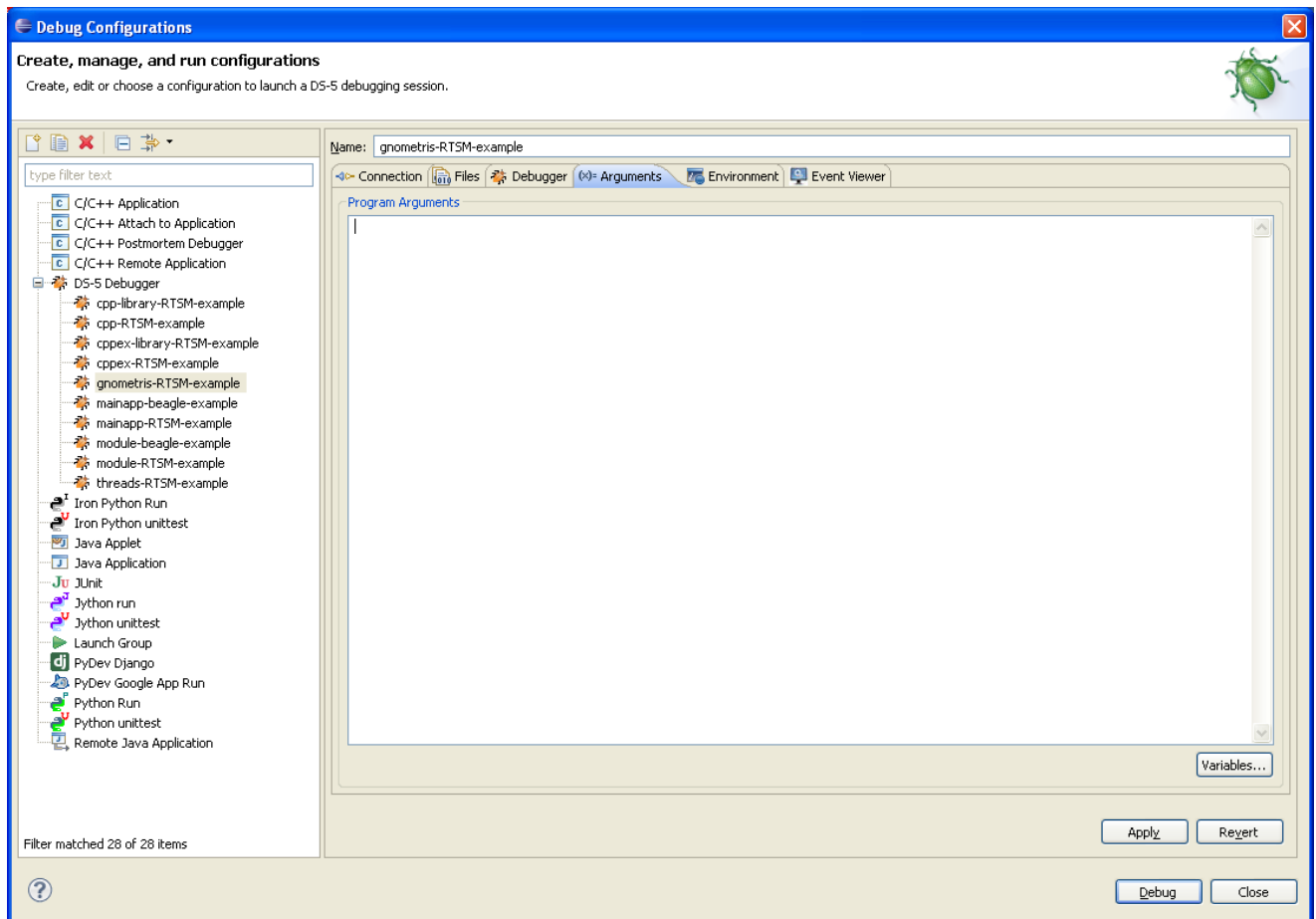


Figure 10-42 Application arguments configuration

Related references

- [4.3 About passing arguments to main\(\) on page 4-105.](#)
- [7.1 About semihosting and top of memory on page 7-162.](#)
- [7.2 Working with semihosting on page 7-163.](#)
- [7.3 Enabling automatic semihosting support in the debugger on page 7-164.](#)
- [7.4 Controlling semihosting messages using the command-line console on page 7-165.](#)

Related information

[DS-5 Debugger commands.](#)

10.41 Debug Configurations - Environment tab

Describes the tab content.

The **Environment** tab in the Debug Configurations dialog box enables you to create and configure the target environment variables that are passed to the application when the debug session starts.

———— **Note** ————

The settings in this tab are not used for connections that use the **Connect to already running gdbserver** debug operation.

The **Environment** tab contains the following elements:

Target environment variables to set

This panel displays the current target environment variables in use by the debugger.

New...

This button opens the New Environment Variable dialog box where you can create a new target environment variable.

For example, to debug the Gnetris application on a model you must create a target environment variable for the DISPLAY setting.

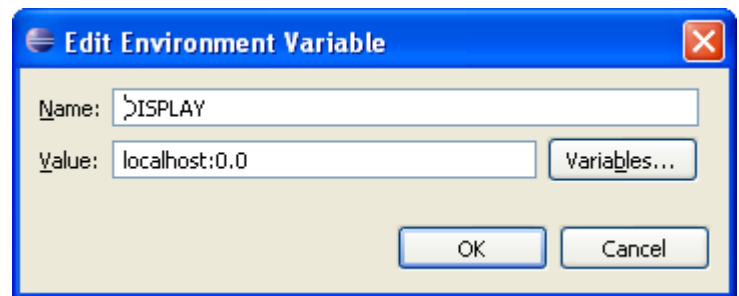


Figure 10-43 Setting up target environment variables

Edit...

This button opens the Edit Environment Variable dialog box where you can edit the properties for the selected target environment variable.

Remove

This button removes the select target environment variables from the list.

Apply

Save the current configuration. This does not connect to the target.

Revert

Undo any changes and revert to the last saved configuration.

Debug

Connect to the target and close the Debug Configurations dialog box.

Close

Close the Debug Configurations dialog box.

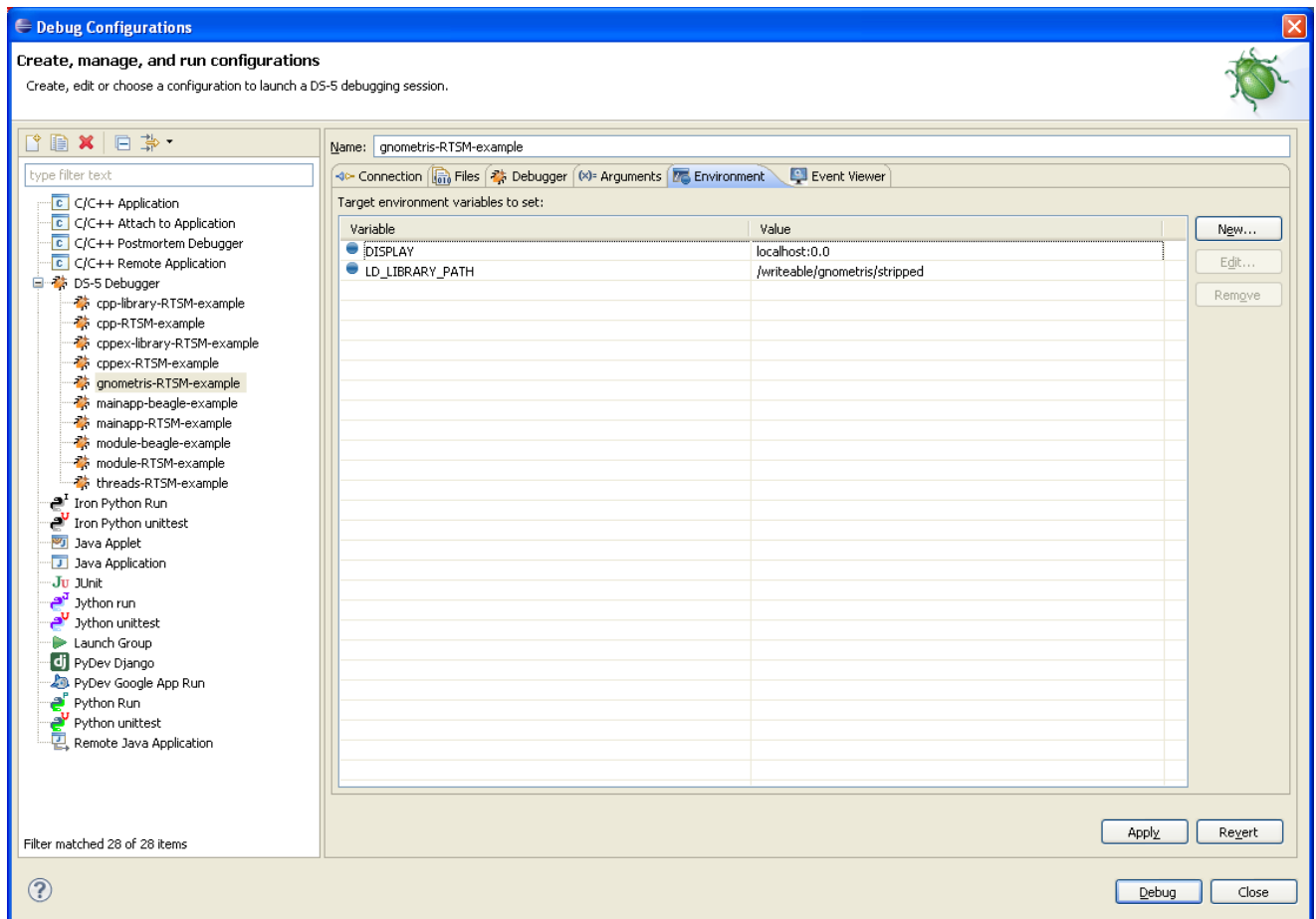


Figure 10-44 Environment configuration for a model

10.42 DTSL Configuration Editor dialog box

Describes the dialog box content.

The *Debug and Trace Services Layer* (DTSL) configuration editor enables you to configure additional debug and trace settings. The available configuration options depend on the capabilities of the target, and typically enable configuration of the trace collection method and the trace that is generated. A typical set of configuration options might include:

Trace capture method

Select the collection method that you want to use for this debug configuration. The available trace collection methods depend on the target and trace capture unit but can include *Embedded Trace Buffer* (ETB)/*Micro Trace Buffer* (MTB) (trace collected from an on-chip buffer) or *DSTREAM* (trace collected from the *DSTREAM* trace buffer). If no trace collection method is selected then no trace can be collected, even if the trace capture for processors and *Instruction Trace Macrocell* (ITM) are enabled.

Enable core trace

Enable or disable trace collection. If enabled then the following options are available:

Enable core *n* trace

Specify trace capture for specific processors.

Cycle accurate

Enable or disable cycle accurate trace.

Trace capture range

Specify an address range to limit the trace capture.

Enable ITM trace

Enable or disable trace collection from the ITM unit.

Named DTSL configuration profiles can be saved for later use.

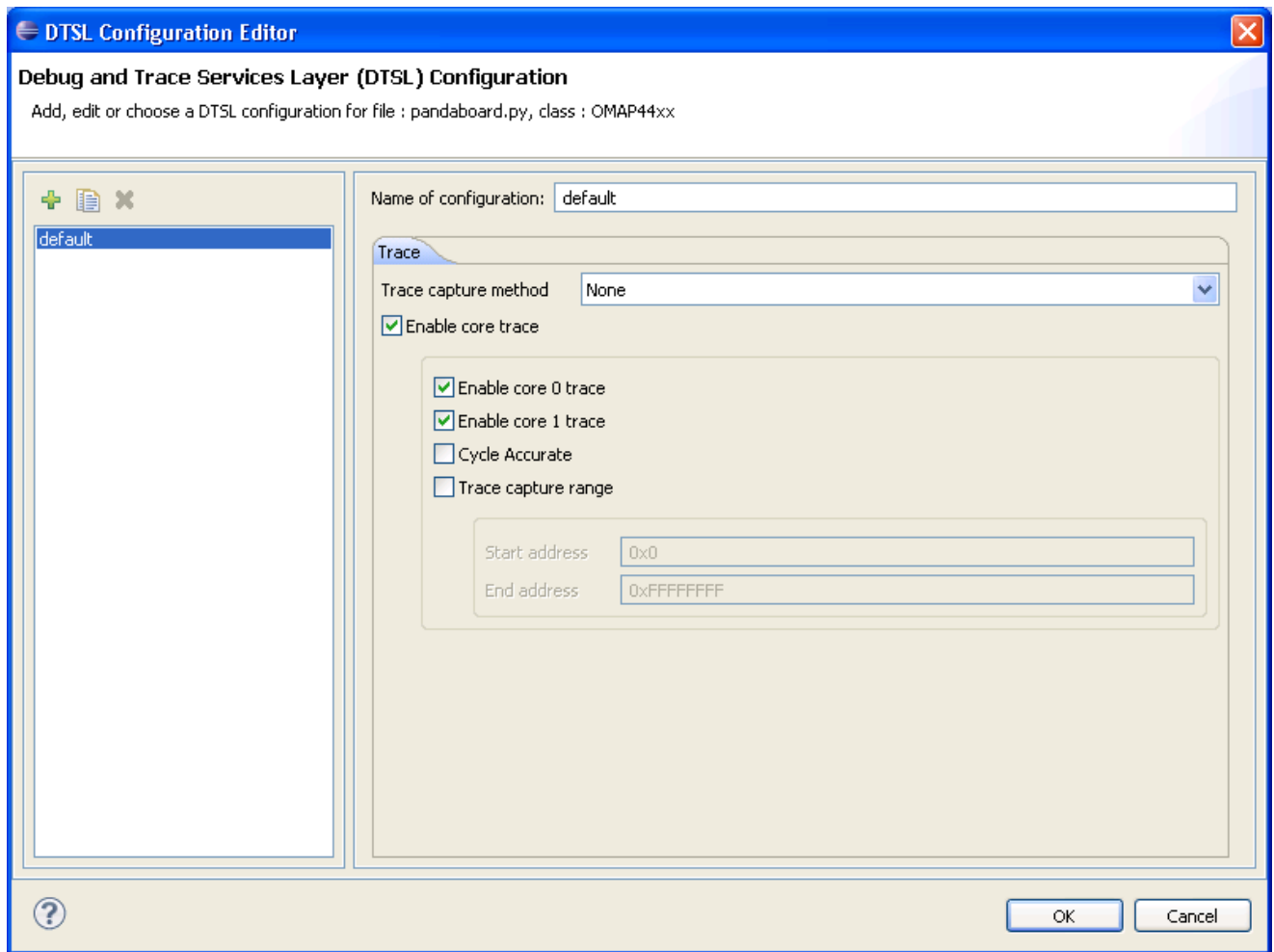


Figure 10-45 DTSL configuration editor

10.43 Configuration database panel

Describes the dialog box content.

This panel enables you to manage the configuration database settings.

Default Target Database

Provides you with the default DS-5 configuration databases.

Note

ARM recommends that you do not disable these.

User Target Database

Enables you to add your own configuration database.

Add...

Opens a dialog box where you can select the new configuration database folder.

Edit

Opens a dialog box where you can modify the existing name and location for the selected configuration database.

Remove

Removes the selected configuration database.

Up

Moves the selected configuration database up the list.

Down

Moves the selected configuration database down the list.

Note

Databases process from top to bottom with information in the lower databases replacing information in higher databases. For example, if you produced a modified core definition with different registers, you would add it to the database at the bottom of the list so that the database uses it instead of the core definitions in the shipped database.

Rebuild database...

Rebuild the configuration database.

Restore Defaults

Removes all the configuration databases from the field text that do not belong to the DS-5 default system.

Apply

Saves the current configuration database configuration settings.

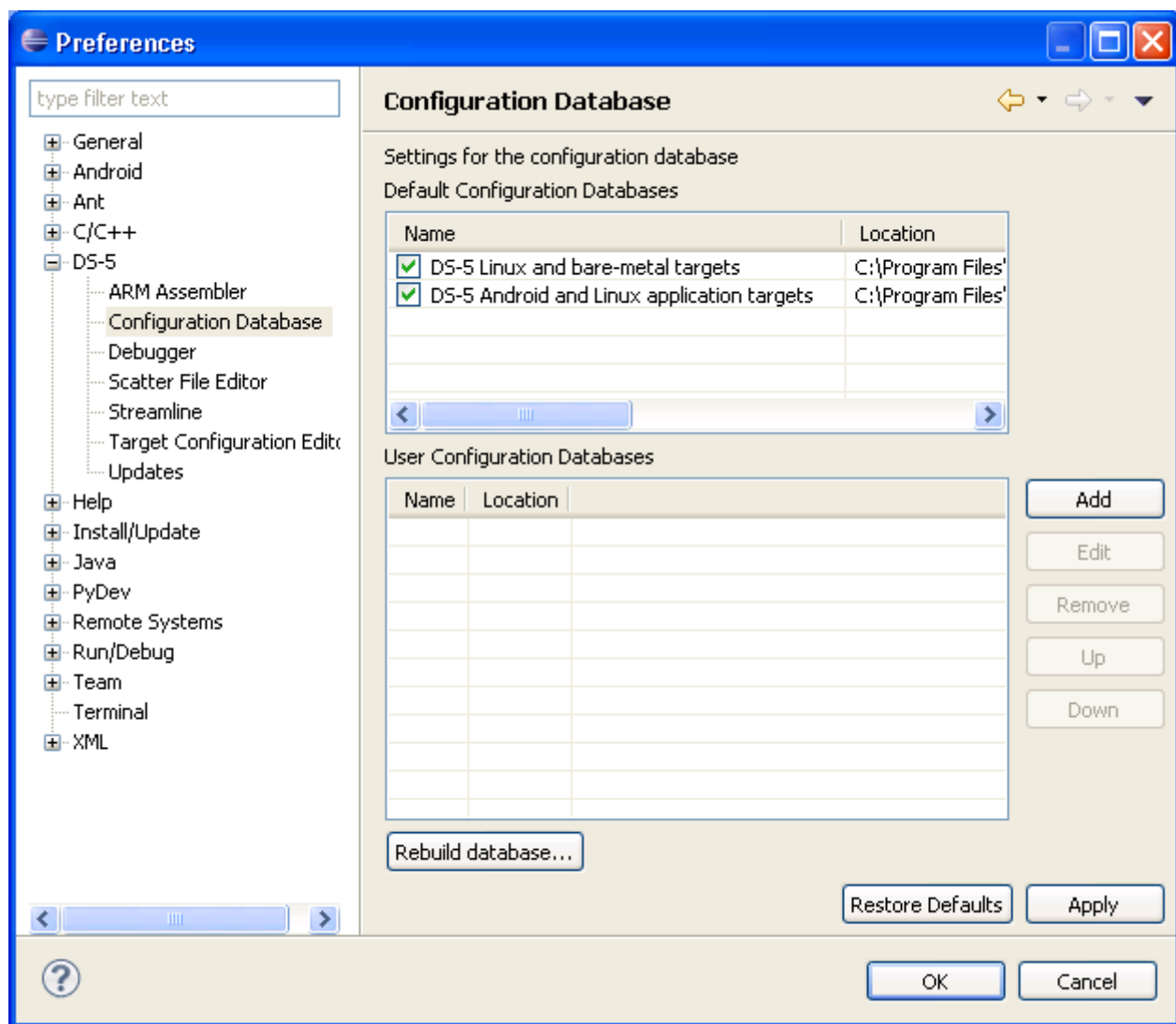


Figure 10-46 Configuration Database panel

Related concepts

[2.9 About the target configuration import utility on page 2-51.](#)

Related tasks

[2.10 Adding a new platform on page 2-53.](#)

[2.11 Adding a new configuration database to DS-5 on page 2-55.](#)

10.44 About the Remote System Explorer

Describes the tab content.

The *Remote Systems Explorer* (RSE) enables you to:

- set up Linux SSH connections to remote targets using TCP/IP
- create, copy, delete, and rename resources
- set the read, write, and execute permissions for resources
- edit files by double-clicking to open in the C/C++ editor view
- execute commands on the remote target
- view and kill running processes
- transfer files between the host workstation and remote targets
- launch terminal views.

Useful RSE views that can be added to the DS-5 Debug perspective are:

- Remote Systems
- Remote System Details
- Remote Scratchpad
- Terminals.

To add a view to the DS-5 Debug perspective:

1. Ensure that you are in the DS-5 perspective. You can change perspective by using the perspective toolbar or you can select **Window > Open perspective** from the main menu.
2. Select **Window > Show View > Other...** to open the Show View dialog box.
3. Select the required view from the **Remote Systems** group.
4. Click **OK**.

10.45 Remote Systems view

Describes the tab content.

This hierarchical tree view enables you to:

- set up a Linux connection to a remote target using the *Secure SHell* (SSH) protocol
- access resources on the host workstation and remote targets
- display a selected file in the C/C++ editor view
- open the **Remote System Details** view and show the selected connection configuration details in a table
- open the **Remote Monitor view** and show the selected connection configuration details
- import and export the selected connection configuration details
- connect to the selected target
- delete all passwords for the selected connection
- open the Properties dialog box and display the current connection details for the selected target.

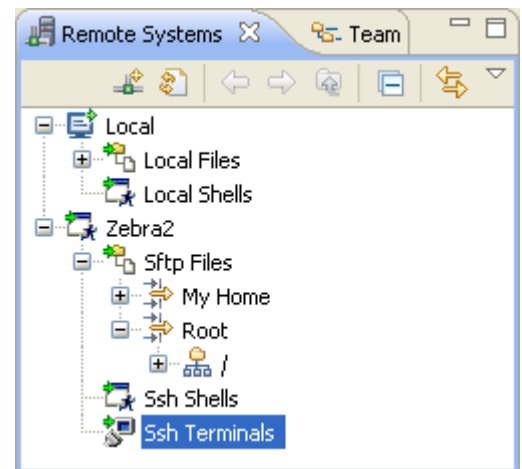


Figure 10-47 Remote Systems view

10.46 Remote System Details view

Describes the tab content.

This tabular view enables you to:

- set up a Linux connection to a remote target using the *Secure SHell* (SSH) protocol
- access resources on the host workstation and remote targets
- display a selected file in the C/C++ editor view
- open the **Remote Systems** view and show the selected connection configuration details in a hierarchical tree
- open the **Remote Monitor** view and show the selected connection configuration details
- import and export the selected connection configuration details
- connect to the selected target
- delete all passwords for the selected connection
- open the Properties dialog box and display the current connection details for the selected target.

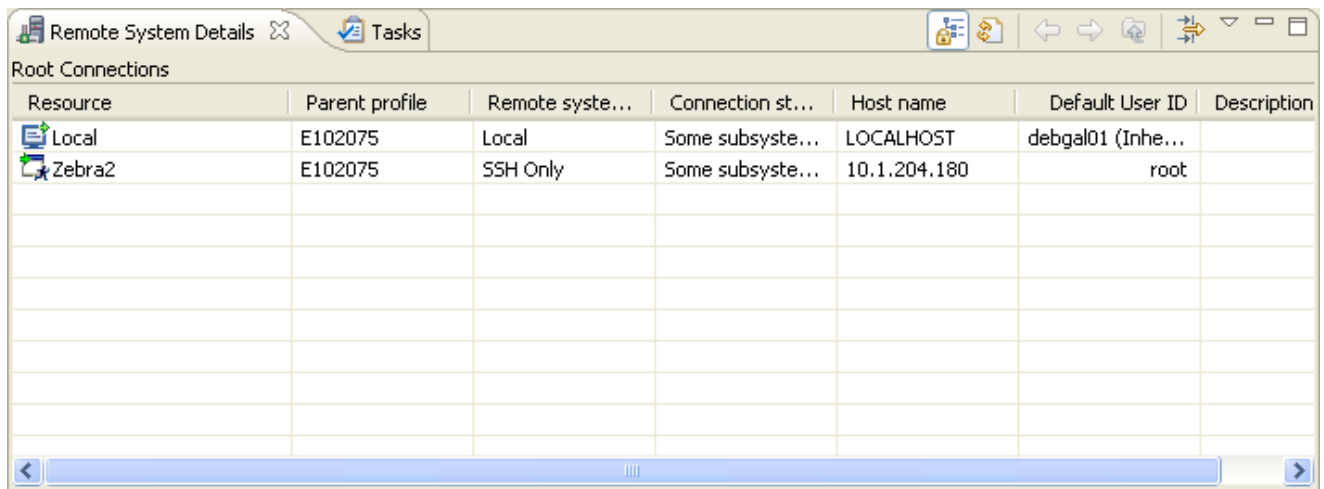


Figure 10-48 Remote System Details view

The **Remote System Details** view is not visible by default. To add this view:

1. Select **Window > Show View > Other...** to open the Show View dialog box.
2. Expand the **Remote Systems** group and select **Remote System Details**.
3. Click **OK**.

10.47 Target management terminal for serial and SSH connections

Describes the tab content.

The target management terminal enables you to enter shell commands directly on the target without launching any external application. For example you can browse remote files and folders by entering the `ls` or `pwd` commands in the same way as you would in a Linux terminal.

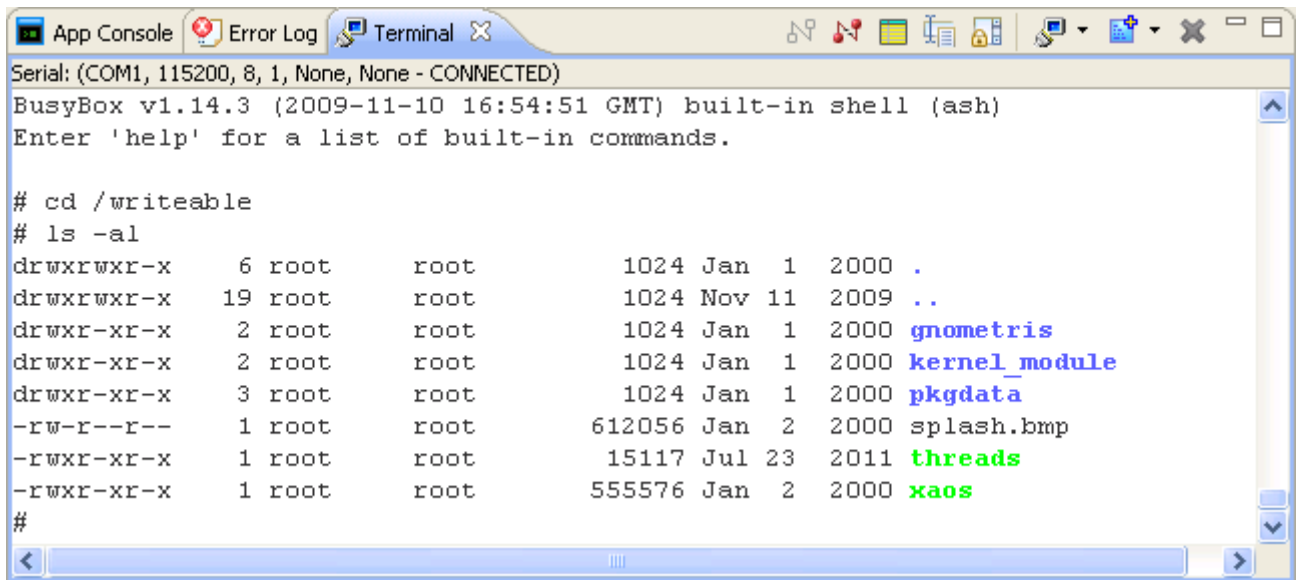


Figure 10-49 Terminal view

The **Terminal** view is not visible by default. To add this view:

1. Select **Window > Show View > Other...** to open the Show View dialog box.
2. Expand the **Terminal** group and select **Terminal**
3. Click **OK**.
4. In the **Terminal** view, click on the **Settings**
5. Select the required connection type.
6. Enter the appropriate information in the Settings dialog box
7. Click **OK**.

Related tasks

[2.3 Configuring a connection to a Linux target using gdbserver on page 2-35.](#)

[2.4 Configuring a connection to a Linux Kernel on page 2-37.](#)

10.48 Remote Scratchpad view

Describes the tab content.

The **Remote Scratchpad** view is an electronic clipboard where you can copy and paste or drag and drop useful files and folders into this view for use at a later point in time. This enables you to keep a list of resources from any connection in one place.

———— **Note** ————

Be aware that although the scratchpad only shows links, any changes made to a linked resource also changes it in the original file system.

—————

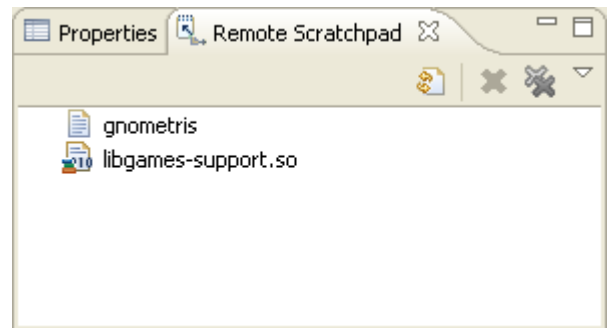


Figure 10-50 Remote Scratchpad

The **Remote Scratchpad** view is not visible by default. To add this view:

1. Select **Window > Show View > Other...** to open the Show View dialog box.
2. Expand the **Remote Systems** group and select **Remote Scratchpad**.
3. Click **OK**.

10.49 Remote Systems terminal for SSH connections

Describes the tab content.

The Remote Systems terminal enables you to enter shell commands directly on the target without launching any external application. For example you can browse remote files and folders by entering the `ls` or `pwd` commands in the same way as you would in a Linux terminal.

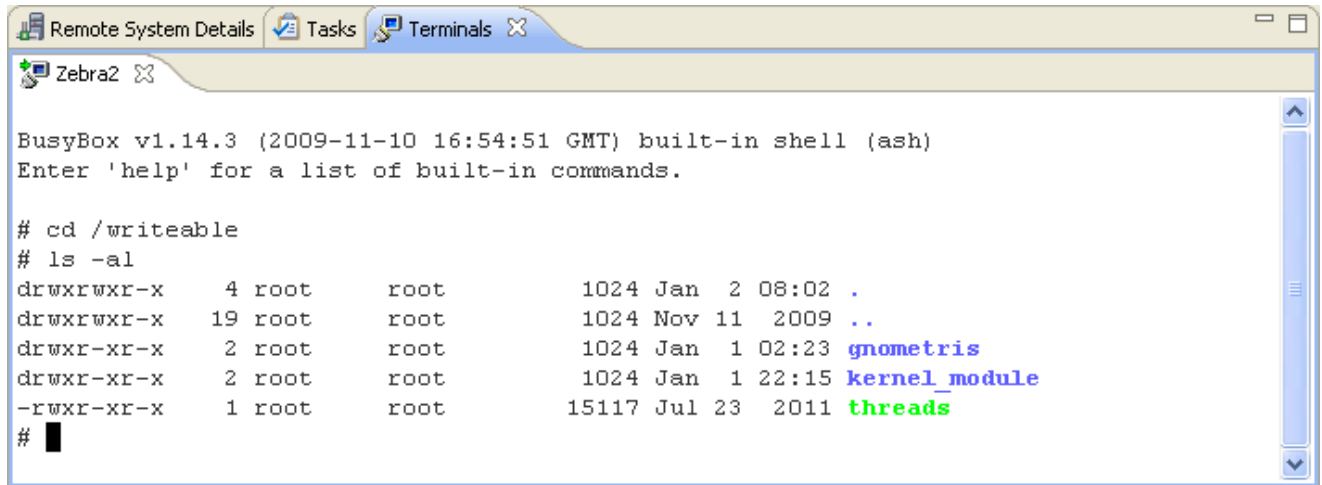


Figure 10-51 Remote Systems Terminals view

The **Terminals** view is not visible by default. To add this view:

1. Select **Window > Show View > Other...** to open the Show View dialog box.
2. Expand the **Remote Systems** group and select **Remote Systems**.
3. Click **OK**.
4. In the **Remote Systems** view:
 - a. Click on the toolbar icon **Define a connection to a remote system** and configure an *Secure SHell* (SSH) connection to the target.
 - b. Right-click on the connection and select **Connect** from the context menu.
 - c. Enter the User ID and password in the relevant fields.
 - d. Click **OK** to connect to the target.
 - e. Right-click on **Ssh Terminals**.
5. Select **Launch Terminal** to open a terminal shell that is connected to the target.

10.50 New Terminal Connection dialog box

Describes the dialog box content.

The dialog box enables you to:

- change the view title
- select the connection type
- set the connection settings.

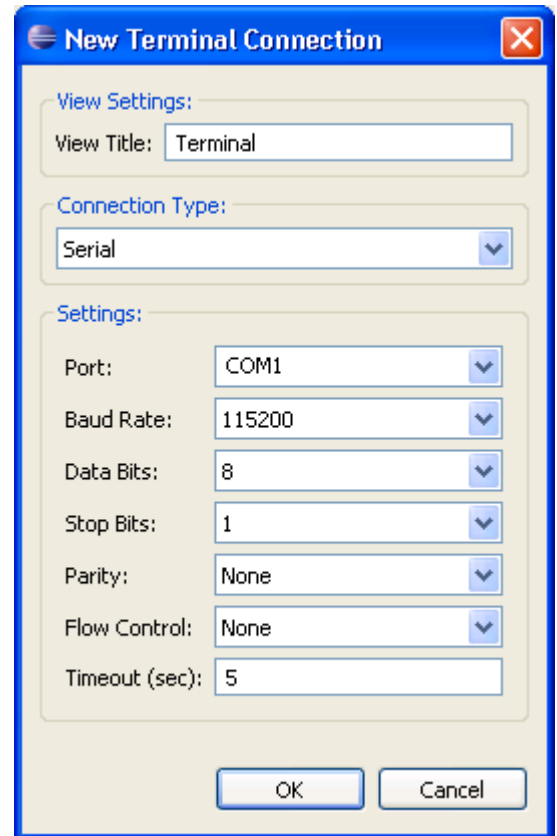


Figure 10-52 Terminal Settings dialog box

View Settings

Enables you to give a name to the **Terminal** view.

View Title

Enter a name for the **Terminal** view.

Connection Type

Specifies a connection type. Either Serial or *Secure SHell* (SSH).

Settings

Enables you to configure the connection settings.

Port

Specifies the port that the target is connected to.

Baud Rate

Specifies the connections baud rate.

Data Bits

Specifies the number of data bits.

Stop Bits

Specifies the number of stop bits for each character.

Parity

Specifies the parity type:

- None. This is the default.
- Even.
- Odd.
- Mark.
- Space.

Flow Control

Specifies the flow control of the connection:

- None. This is the default.
- RTS/CTS.
- Xon/Xoff.

Timeout (sec)

Specifies the connections timeout in seconds.

10.51 DS-5 Debugger menu and toolbar icons

Describes the menus and toolbar icons used in the DS-5 Debug perspective.

These tables list the most common menu and toolbar icons available for use with DS-5 Debugger. For information on icons, markers, and buttons not listed in the following tables, see the standard *Workbench User Guide* or the *C/C++ Development User Guide* in the **Help > Help Contents** window.

If you leave the mouse pointer positioned on a toolbar icon for a few seconds without clicking, a tooltip appears informing you of the purpose of the icon.

Table 10-2 DS-5 Debugger icons





























Icon	Description	Icon	Description
	Connect to target		Connected to target
	Disconnect from target		Delete connection
	Start application and run to main		Start application and run to entry point
	Run application from entry point		Restart the application
	Continue running application		Stop application
	Step into		Step over
	Step out		Toggle stepping mode
	Continue running application backwards		Reverse step source line
	Reverse step over source line		Reverse step out source line
	Collapse all configurations in stack trace		Call stack
	Thread		Process
	Kernel module		Define a new RSE connection
	Refresh the RSE resource tree		Save view contents to a file
	Clear view contents		Switch to History view
	Synchronize view contents		Toggle scroll lock
	Run commands from a script file		Export commands to a script file
	Remove selected breakpoint, watchpoints, or expression (view dependent)		Remove all breakpoints, watchpoints, or expressions (view dependent)
	Display breakpoint location in source file		Deactivate all breakpoints and watchpoints
	Import from a file		Export to a file
	Create new script file or add new expression (view dependent)		Run select script file
	Open selected file for editing		Delete the selected files
	Set display width		Set display format

Table 10-2 DS-5 Debugger icons (continued)

Icon	Description	Icon	Description
	Toggle the display of ASCII characters		Toggle freeze mode
	Edit Screen view parameters		Add new Screen view
	Add new Disassembly view		Add new Variables view
	Add new Registers view		Add new Memory view
	Add new Expression view		Add new Trace view
	Add Functions view		View update in progress
	Toggle trace marker		Show next match
	Show previous match		Show instruction trace
	Show function trace		Toggle navigation resolution
	Toggle the views		Application rewind information displayed in view







Perspective icons

Table 10-3 Perspective icons

Icon	Description	Icon	Description
	Open new perspective		C/C++ perspective
	DS-5 Debug perspective		Fast view bar

View icons

Table 10-4 View icons

Button	Description	Button	Description
	Display drop-down menu		Synchronize view contents
	Minimize		Maximize
	Restore		Close

View markers

Table 10-5 View markers






































Icon	Description	Icon	Description
	Software breakpoint enabled		Hardware breakpoint enabled
	Access watchpoint enabled		Read watchpoint enabled
	Write watchpoint enabled		Software breakpoint disabled

Table 10-5 View markers (continued)

Icon	Description	Icon	Description
	Hardware breakpoint disabled		Access watchpoint disabled
	Read watchpoint disabled		Write watchpoint disabled
	Software breakpoint pending		Hardware breakpoint pending
	Access watchpoint pending		Read watchpoint pending
	Write watchpoint pending		Software breakpoint disconnected
	Hardware breakpoint disconnected		Access watchpoint disconnected
	Read watchpoint disconnected		Write watchpoint disconnected
	Multiple-statement software breakpoint enabled		Multiple-statement software breakpoint disabled
	Error		Current location
	Warning		Bookmark
	Information		Task
	Search result		

Miscellaneous icons

Table 10-6 Miscellaneous icons

Icon	Description	Icon	Description
	Open a new resource wizard		Open new project wizard
	Open new folder wizard		Open new file wizard
	Open search dialog box		Display context-sensitive help
	Open import wizard		Open export wizard

Chapter 11

Troubleshooting

Describes how to diagnose problems when debugging applications using DS-5 Debugger. It contains the following:

- *11.1 ARM Linux problems and solutions on page 11-307.*
- *11.2 Enabling internal logging from the debugger on page 11-308.*
- *11.3 Target connection problems and solutions on page 11-309.*

11.1 ARM Linux problems and solutions

Lists possible problems when debugging a Linux application.

You might encounter the following problems when debugging a Linux application.

ARM Linux permission problem

If you receive a permission denied error message when starting an application on the target then you might have to change the execute permissions on the application. :

```
chmod +x myImage
```

A breakpoint is not being hit

You must ensure that the application and shared libraries on your target are the same as those on your host. The code layout must be identical, but the application and shared libraries on your target do not require debug information.

Operating system support is not active

When Operating System (OS) support is required, the debugger activates it automatically where possible. If OS support is required but cannot be activated, the debugger produces an error. :

```
ERROR(CMD16-LKN36):
! Failed to load image "gator.ko"
! Unable to parse module because the operating system support is not active
```

OS support cannot be activated if:

- debug information in the `vmlinux` file does not correctly match the data structures in the kernel running on the target
- it is manually disabled by using the `set os enabled off` command.

To determine whether the kernel versions match:

- stop the target after loading the `vmlinux` image
- enter the `print init_nsproxy.uts_ns->name` command
- verify that the `$1` output is correct. :

```
$1 = {sysname = "Linux", nodename = "(none)", release = "3.4.0-rc3",
version = "#1 SMP Thu Jan 24 00:46:06 GMT 2013", machine = "arm",
domainname = "(none)"}
```

Related tasks

[2.3 Configuring a connection to a Linux target using gdbserver on page 2-35.](#)

[2.4 Configuring a connection to a Linux Kernel on page 2-37.](#)

11.2 Enabling internal logging from the debugger

Describes how to enable internal logging to help diagnose error messages.

On rare occasions an internal error might occur causing the debugger to generate an error message suggesting that you report it to your local support representatives. You can help to improve the debugger by giving feedback with an internal log that captures the stacktrace and shows where in the debugger the error occurs. To obtain the current version of DS-5, you can select **About ARM DS-5** from the **Help** menu in Eclipse or open the product release notes.

To enable internal logging within Eclipse, enter the following in the Commands view of the **DS-5 Debug** perspective:

1. To enable the output of logging messages from the debugger using the predefined DEBUG level configuration:

```
log config debug
```

2. To redirect all logging messages from the debugger to a file:

```
log file debug.Log
```

Note

Enabling internal logging can produce very large files and slow down the debugger significantly. Only enable internal logging when there is a problem.

Related references

[10.6 Commands view on page 10-209.](#)

[7.5 Controlling the output of logging messages on page 7-166.](#)

11.3 Target connection problems and solutions

Lists possible problems when connecting to a target.

Failing to make a connection

The debugger might fail to connect to the selected debug target because of the following reasons:

- you do not have a valid license to use the debug target
- the debug target is not installed or the connection is disabled
- the target hardware is in use by another user
- the connection has been left open by software that exited incorrectly
- the target has not been configured, or a configuration file cannot be located
- the target hardware is not powered up ready for use
- the target is on a scan chain that has been claimed for use by something else
- the target hardware is not connected
- you want to connect through gdbserver but the target is not running **gdbserver**
- there is no ethernet connection from the host to the target
- the port number in use by the host and the target are incorrect

Check the target connections and power up state, then try and reconnect to the target.

Debugger connection settings

When debugging a bare-metal target the debugger might fail to connect because of the following reasons:

- **Heap Base** address is incorrect
- **Stack Base** (top of memory) address is incorrect
- **Heap Limit** address is incorrect
- Incorrect vector catch settings.

Check that the memory map settings are correct for the selected target. If set incorrectly, the application might crash because of stack corruption or because the application overwrites its own code.

Related tasks

[2.3 Configuring a connection to a Linux target using gdbserver on page 2-35.](#)

[2.4 Configuring a connection to a Linux Kernel on page 2-37.](#)

Chapter 12

File-based flash programming in DS-5™

This chapter describes the DS-5 file-based flash programming architecture, how to add support for new boards, and how to add support for new types of flash devices. It contains the following:

- *12.1 About file-based flash programming in DS-5™ on page 12-311.*
- *12.2 Flash programming configuration on page 12-314.*
- *12.3 Creating an extension database for flash programming on page 12-316.*
- *12.4 About using or extending the supplied Keil flash method on page 12-317.*
- *12.5 About creating a new flash method on page 12-319.*
- *12.6 About testing the flash configuration on page 12-323.*
- *12.7 About flash method parameters on page 12-324.*
- *12.8 About getting data to the flash algorithm on page 12-325.*
- *12.9 About interacting with the target on page 12-326.*

12.1 About file-based flash programming in DS-5™

The DS-5 configdb platform entry for a board can contain a flash definition section. This section can define one or more areas of flash, each with its own flash method and configuration parameters.

Flash methods are implemented in Jython and are typically located within the configdb. Each flash method is implemented with a specific technique of programming flash.

These techniques might involve:

- Running an external program supplied by a third party to program a file into flash.
- Copying a file to a file system mount point. For example, as implemented in the Versatile Express designs.
- Download a code algorithm into the target system and to keep running that algorithm on a data set (typically a flash sector) until the entire flash device has been programmed.

Note

You can use the DS-5 Debugger **info flash** command to view the flash configuration for your board.

Examples of downloading a code algorithm into the target system are the Keil flash programming algorithms which are fully supported by DS-5 Debugger. For the Keil flash method, one of the method configuration items is the algorithm to use to perform the flash programming. These algorithms all follow the same top level software interface and so the same DS-5 Keil flash method can be used to program different types of flash. This means that DS-5 Debugger should be able to make direct use of any existing Keil flash algorithm.

Note

All flash methods which directly interact with the target should do so using the DS-5 Debugger's DTSL connection.

Flash programming supported features

The following features are available in file flash programming operations:

- Supports ELF files (.axf) programming into flash.
- Supports ELF files containing multiple flash areas which can each be programmed into a flash device or possibly several different flash devices.
- Supports many and varied flash programming methods.
- Supports all Keil flash programming algorithms.
- Supports target board setup and teardown to prepare it for flash programming.
- Supports DS-5 configuration database to learn about target flash devices and the options required for flash programming on a specific board or system on chip.
- Supports default flash options modification.
- Supports graphical progress reporting within Eclipse and on a text only type console when used with the debugger outside Eclipse, along with the ability to cancel the programming operation.
- Supports a simple flash programming user interface where you can specify minimal configurations or options.
- Supports displaying warning and error messages to the user.

Note

An example, `flash_example-FVP-A9x4`, is provided with DS-5. This example shows two ways of programming flash devices using DS-5, one using a Keil Flash Method and the other using a Custom Flash Method written in Jython. For convenience, the Cortex-A9x4 FVP model supplied with DS-5 is used as the target device. This example can be used as a template for creating new flash algorithms. The `readme.html` provided with the example contains basic information on how to use the example.

DS-5™ File Flash Architecture

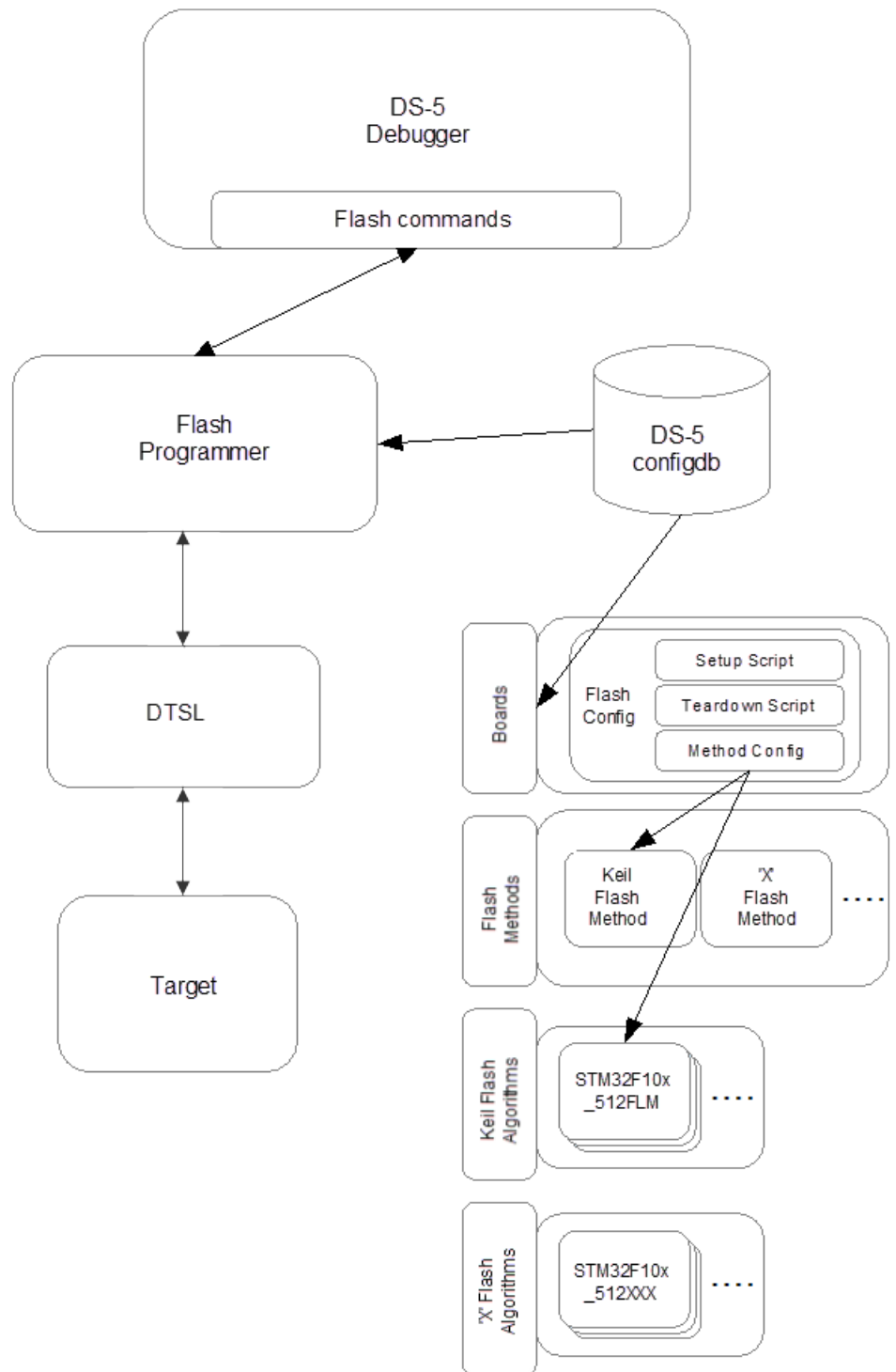


Figure 12-1 DS-5 File Flash Architecture

Related information

[Flash commands.](#)

12.2 Flash programming configuration

Each target platform supported by DS-5 has an entry in the DS-5 configuration database. To add support for flash programming, a target's platform entry in the database must define both the flash programming method and any required parameters.

Configuration files

The target's platform entry information is stored across two files in the configuration database:

- **project_types.xml** - This file describes the debug operations supported for the platform and may contain a reference to a flash configuration file. This is indicated by a tag such as `<flash_config>CDB://flash.xml</flash_config>`.

The `CDB://` tag indicates a path relative to the target's platform directory which is usually the one that contains the **project_types.xml** file. You can define a relative path above the target platform directory using `../`. For example, a typical entry would be similar to `<flash_config>CDB://../../Flash/STM32/flash.xml</flash_config>`.

Using relative paths allows the flash configuration file to be shared between a number of targets with the same chip and same flash configuration.

- The `FDB://` tag indicates a path relative to where the Jython flash files (such as the **stm32_setup.py** and **keil_flash.py** used in the examples) are located. For DS-5 installations, this is usually `<DS-5 Install folder>\sw\debugger\configdb\Flash`.
- A flash configuration **.xml** file. For example, **flash.xml**. This **.xml** file describes flash devices on a target, including which memory regions they are mapped to and what parameters need to be passed to the flash programming method.

A flash configuration must always specify the flash programming method to use, but can also optionally specify a setup script and a teardown script. Setup and teardown scripts are used to prepare the target platform for flash programming and to re-initialize it when flash programming is complete. These scripts might be very specific to the target platform, whereas the flash programming method might be generic.

Configuration file example

This example **flash.xml** is taken from the Keil MCBSTM32E platform. It defines two flash devices even though there is only one built-in flash device in the MCBSTM32E. This is because the two flash sections, the main flash for program code and the option flash for device configuration, are viewed as separate devices when programming.

Note how the flash method is set to the **keil_flash.py** script and how the parameters for that method are subsequently defined.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!--Copyright (C) 2012 ARM Limited. All rights reserved.-->
<flash_config
  xmlns:xi="http://www.w3.org/2001/XInclude"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://www.arm.com/flash_config"
  xsi:schemaLocation="http://www.arm.com/flash_config flash_config.xsd">
  <devices>
    <!-- STM32F1xx has 2 flash sections: main flash for program code and option
    flash for device configuration. These are viewed as separate devices
    when programming -->
    <!-- The main flash device -->
    <device name="MainFlash">
      <programming_type type="FILE">
        <!-- Use the standard method for running Keil algorithms -->
        <method language="JYTHON" script="FDB://keil_flash.py"
class="KeilFlash" method_config="Main"/>
        <!-- Target specific script to get target in a suitable state for
programming -->
        <setup_script="FDB://stm32_setup.py" method="setup"/>
      </programming_type>
    </device>
  </devices>
</flash_config>
```

```

</device>
<!-- The option flash device -->
<device name="OptionFlash">
  <programming_type type="FILE">
    <method language="JYTHON" script="FDB://keil_flash.py"
class="KeilFlash" method_config="Option"/>
    <setup script="FDB://stm32_setup.py" method="setup"/>
  </programming_type>
</device>
</devices>
<method_configs>
  <!-- Parameters for programming the main flash -->
  <method_config id="Main">
    <params>
      <!-- Programming algorithm binary to load to target -->
      <param name="algorithm" type="string" value="FDB://algorithms/
STM32F10x_512.FLM"/>
      <!-- The core in the target to run the algorithm -->
      <param name="coreName" type="string" value="Cortex-M3"/>
      <!-- RAM location & size for algorithm code and write buffers -->
      <param name="ramAddress" type="integer" value="0x20000000"/>
      <param name="ramSize" type="integer" value="0x10000"/>
      <!-- Allow timeouts to be disabled -->
      <param name="disableTimeouts" type="string" value="false"/>
      <!-- Set to false to skip the verification stage -->
      <param name="verify" type="string" value="true"/>
    </params>
  </method_config>
  <!-- Parameters for programming the option flash -->
  <method_config id="Option">
    <params>
      <!-- Programming algorithm binary to load to target -->
      <param name="algorithm" type="string" value="FDB://algorithms/
STM32F10x_OPT.FLM"/>
      <!-- The core in the target to run the algorithm -->
      <param name="coreName" type="string" value="Cortex-M3"/>
      <!-- RAM location & size for algorithm code and write buffers -->
      <param name="ramAddress" type="integer" value="0x20000000"/>
      <param name="ramSize" type="integer" value="0x10000"/>
      <!-- Allow timeouts to be disabled -->
      <param name="disableTimeouts" type="string" value="false"/>
      <!-- Set to false to skip the verification stage -->
      <param name="verify" type="string" value="true"/>
    </params>
  </method_config>
</method_configs>
</flash_config>

```

Related tasks

[12.3 Creating an extension database for flash programming on page 12-316.](#)

12.3 Creating an extension database for flash programming

In certain scenarios, it might not be desirable or possible to modify the default DS-5 configuration database. In this case, you can create your own configuration databases and use them to extend the default installed database.

To create an extension configuration database:

Procedure

1. At your preferred location, create a new directory with the name of your choice for the extension database.
2. In your new directory, create two subdirectories and name them **Boards** and **Flash** respectively.
 - a) In the **Boards** directory, create a subdirectory for the board manufacturer.
 - b) In the board manufacturer subdirectory, create another directory for the board.
 - c) In the **Flash** directory, create a subdirectory and name it **Algorithms**.

For example, for a manufacturer **MegaSoc-Co** who makes **Acme-Board-2000**, the directory structure would look similar to this:

```
Boards
 \---> MegaSoc-Co
       \---> Acme-Board-2000
              project_types.xml

Flash
 \---> Algorithms
       Acme-Board-2000.flm
       Acme-Board-2000-Flash.py
```

3. From the main menu in DS-5, select **Window > Preferences > DS-5 > Configuration Database**.
 - a) In the **User Configuration Databases** area, click **Add**.
 - b) In the Add configuration database location dialog, enter the **Name** and **Location** of the your configuration database and click **OK**.
4. In the Preferences dialog, click **OK** to confirm your changes.

Within the `project_types.xml` file for your platform, any reference to a `CDB://` location will now resolve to the `Boards/<manufacturer>/<board>` directory and any reference to a `FDB://` location will resolve to the `Flash` directory.

12.4 About using or extending the supplied Keil flash method

DS-5 Debugger contains a full implementation of the Keil flash programming method. This may be used to program any flash device supported by Keil's MDK product. It may also be used to support any future device for which a Keil flash programming algorithm can be created.

For details on creating new Keil Flash Programming algorithms (these links apply to the Keil uVision product), see:

[Algorithm Functions](#)

[Creating New Algorithms](#)

To aid in the creation of new Keil flash programming algorithms within DS-5, DS-5 Debugger contains a full platform flash example for the Keil MCBSTM32E board. This can be used as a template for new flash support.

———— Note ————

An example, `flash_example-FVP-A9x4`, is provided with DS-5. This example shows two ways of programming flash devices using DS-5, one using a Keil Flash Method and the other using a Custom Flash Method written in Jython. For convenience, the `Cortex-A9x4 FVP` model supplied with DS-5 is used as the target device. This example can be used as a template for creating new flash algorithms. The `readme.html` provided with the example contains basic information on how to use the example.

This section describes how to add flash support to an existing platform using an existing Keil flash program, and how to add flash support to an existing platform using a new Keil flash algorithm.

It contains the following:

- [12.4.1 Adding flash support to an existing platform using an existing Keil flash algorithm on page 12-317.](#)
- [12.4.2 Adding flash support to an existing target platform using a new Keil flash algorithm on page 12-318.](#)

12.4.1 Adding flash support to an existing platform using an existing Keil flash algorithm

To use the Keil MDK flash algorithms within DS-5, the algorithm binary needs to be imported into the target configuration database and the flash configuration files created to reference the `keil_flash.py` script.

This example uses the flash configuration for the Keil MCBSTM32E board example in Flash programming configuration as a template to add support to a board called the Acme-Board-2000 made by MegaSoc-Co.

Procedure

1. Copy the algorithm binary `.FLM` into your configuration database `Flash/Algorithms` directory.
2. Copy the flash configuration file from `Boards/Keil/MCBSTM32E/keil-mcbstm32e_flash.xml` to `Boards/MegaSoc-Co/Acme-Board-2000/flash.xml`.
3. Edit the platform's `project_types.xml` to reference the `flash.xml` file by inserting `<flash_config>CDB://flash.xml</flash_config>` below `platform_data` entry, for example:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!--Copyright (C) 2009-2012 ARM Limited. All rights reserved.-->
<platform_data xmlns="http://www.arm.com/project_type"
  xmlns:peripheral="http://com.arm.targetconfigurationeditor"
  xmlns:xi="http://www.w3.org/2001/XInclude"
```

```
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" type="HARDWARE"  
xsi:schemaLocation="http://www.arm.com/project_type../../Schemas/  
platform_data-1.xsd">  
  <flash_config>CDB://flash.xml</flash_config>
```

4. Edit the devices section, and create a <device> block for each flash device on the target.

———— **Note** ————

The `method_config` attribute should refer to a unique <method_config> block for that device in the <method_configs> section.

5. Optionally, create and then reference any setup or teardown script required for your board. If your board does not need these, then do not add these lines to your configuration.

```
<setup script="FDB://Acme-Board-2000-Flash.py" method="setup"/>  
<teardown script="FDB://Acme-Board-2000-Flash.py" method="teardown"/>
```

6. Edit the `method_configs` section, creating a <method_config> block for each device.

———— **Note** ————

- The value for the `algorithm` parameter should be changed to the path to the algorithm copied in *Step 1*. The `FDB://` prefix is used to indicate the file can be found in the configuration database **Flash** directory.
- The `coreName` parameter should be the name of the core on the target that will run the algorithm. This must be the same name as used in the <core> definition within `project_types.xml`. For example, <core connection_id = "Cortex-M3" core_definition = "Cortex-M3"/>.
- The `ramAddress` and `ramSize` parameters should be set to an area of RAM that the algorithm can be downloaded in to and used as working RAM. It should be big enough to hold the algorithm, stack plus scratch areas required to run the algorithm, and a sufficiently big area to download image data.
- The other parameters do not normally need to be changed.

12.4.2 Adding flash support to an existing target platform using a new Keil flash algorithm

DS-5 ships with a complete Keil flash algorithm example for the STM32 device family. You can use this as a template for creating and building your new flash algorithm.

Locate the `Bare-metal_examples.zip` file within the `DS-5/Examples` directory. Extract the `DS-5Examples\flash_algo-STM32F10x` directory to your file system and then import this project into your DS-5 Eclipse Workspace. Using this as your template, create a new project, copy the content from the example into your new project and modify as needed.

Once you have successfully built your .FLM file(s), proceed as explained in the *Adding flash support to an existing platform using an existing Keil flash algorithm* topic.

Related tasks

[12.4.1 Adding flash support to an existing platform using an existing Keil flash algorithm on page 12-317.](#)

12.5 About creating a new flash method

If the Keil flash method is inappropriate for your requirements, it is necessary to create a new custom flash method for your use.

Programming methods are implemented in Jython (Python, utilizing the Jython runtime). The use of Jython allows access to the DTSL APIs used by the DS-5 Debugger. DS-5 includes the PyDev tools to assist in writing Python scripts.

In a DS-5 install, the `configdb\Flash\flashprogrammer` directory holds a number of Python files which contain utility methods used in the examples.

This section describes a default implementation of `com.arm.debug.flashprogrammer.FlashMethodv1` and creating a flash method using a Python script.

It contains the following:

- [12.5.1 About using the default implementation FlashMethodv1 on page 12-319.](#)
- [12.5.2 About creating the flash method Python script on page 12-320.](#)

12.5.1 About using the default implementation FlashMethodv1

Flash programming methods are written as Python classes that are required to implement the `com.arm.debug.flashprogrammer.IFlashMethod` interface. This interface defines the methods the flash programming layer of DS-5 Debugger might invoke.

See the `flash_method_v1.py` file in the `<Install folder>\DS-5\sw\debugger\configdb\Flash\flashprogrammer` for a default implementation of `com.arm.debug.flashprogrammer.FlashMethodv1`. This has empty implementations of all functions - this allows a Python class derived from this object to only implement the required functions.

Running a flash programming method is split into three phases:

1. Setup - the `setup()` function prepares the target for performing flash programming. This might involve:
 - Reading and validating parameters passed from the configuration file.
 - Opening a connection to the target.
 - Preparing the target state, for example, to initialize the flash controller.
 - Loading any flash programming algorithms to the target.
2. Programming - the `program()` function is called for each section of data to be written. Images might have multiple load regions, so the `program()` function might be called several times. The data to write is passed to this function and the method writes the data into flash at this stage.
3. Teardown - the `teardown()` function is called after all sections have been programmed. At this stage, the target state can be restored (for example, take the flash controller out of write mode or reset the target) and any debug connection closed.

Note

The `setup()` and `teardown()` functions are not to be confused with the target platform optional `setup()` and `teardown()` scripts. The `setup()` and `teardown()` functions defined in the flash method class are for the method itself and not the board.

12.5.2 About creating the flash method Python script

For the purposes of this example the Python script is called `example_flash.py`.

- Start by importing the objects required in the script:

```
from flashprogrammer.flash_method_v1 import FlashMethodv1
from com.arm.debug.flashprogrammer import TargetStatus
```

- Then, define the class implementing the method:

```
class ExampleFlashWriter(FlashMethodv1):
    def __init__(self, methodServices):
        FlashMethodv1.__init__(self, methodServices)

    def setup(self):
        # perform any setup for the method here
        pass

    def teardown(self):
        # perform any clean up for the method here
        # return the target status
        return TargetStatus.STATE_RETAINED

    def program(self, regionID, offset, data):
        # program a block of data to the flash
        # regionID indicates the region within the device (as defined in the flash
        # configuration file)
        # offset is the byte offset within the region

        # perform programming here

        # return the target status
        return TargetStatus.STATE_RETAINED
```

———— Note ————

- The `__init__` function is the constructor and is called when the class instance is created.
- `methodServices` allows the method to make calls into the flash programmer - it should not be accessed directly.
- `FlashMethodv1` provides functions that the method can call while programming.
- The `program()` and `teardown()` methods should return a value that describes the state the target has been left in. This can be one of:
 - `STATE_RETAINED` - The target state has not been altered from the state when programming started. In this state, the register and memory contents have been preserved or restored.
 - `STATE_LOST` - Register and memory contents have been altered, but a system reset is not required.
 - `RESET_REQUIRED` - It is recommended or required that the target be reset.
 - `POWER_CYCLE_REQUIRED` - It is required that the target be manually power cycled. For example, when a debugger-driven reset is not possible or not sufficient to reinitialize the target.

Creating the target platform setup and teardown scripts

If the hardware platform requires some setup (operations to be performed before flash programming) and/or teardown (operations to be performed after flash programming) functionality, you must create one or more scripts which contain `setup()` and `teardown()` functions. These can be in separate script files or you can combine them into a single file. This file should be placed into the configdb `Flash` directory so that it can be referenced using a `FDB://` prefix in the flash configuration file.

For example, the contents of a single file which contains both the `setup()` and `teardown()` functions would be similar to:

```
from com.arm.debug.flashprogrammer.IFlashClient import MessageLevel
from flashprogrammer.device import ensureDeviceOpen
from flashprogrammer.execution import ensureDeviceStopped
from flashprogrammer.device_memory import writeToTarget

def setup(client, services):
    # get a connection to the core
    conn = services.getConnection()
    dev = conn.getDeviceInterfaces().get("Cortex-M3")
    ensureDeviceOpen(dev)
    ensureDeviceStopped(dev)
    # Perform some target writes to enable flash programming
    writeToTarget(dev, FLASH_EN, intToBytes(0x81))

def teardown(client, services):
    # get a connection to the core
    conn = services.getConnection()
    dev = conn.getDeviceInterfaces().get("Cortex-M3")
    ensureDeviceOpen(dev)
    ensureDeviceStopped(dev)
    # Perform some target writes to disable flash programming
    writeToTarget(dev, FLASH_EN, intToBytes(0))
```

Creating the flash configuration file

To use the method to program flash, a configuration file must be created that describes the flash device, the method to use and any parameters or other information required. This is an `.xml` file and is typically stored in the same directory as the target's other configuration files (`Boards/` `<Manufacturer>/<Board name>`) as it contains target-specific information.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<flash_config
  xmlns:xi="http://www.w3.org/2001/XInclude"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://www.arm.com/flash_config"
  xsi:schemaLocation="http://www.arm.com/flash_config flash_config.xsd">
  <devices>
    <device name="Example">
      <regions>
        <region address="0x8000" size="0x10000"/>
      </regions>
      <programming_type type="FILE">
        <method language="JYTHON" script="FDB://example_flash.py"
class="ExampleFlashWriter" method_config="Default"/>
        <setup script="FDB://file_target.py" method="setup"/>
        <teardown script="FDB://file_target.py" method="teardown"/>
      </programming_type>
    </device>
  </devices>
  <method_configs>
    <method_config id="Default">
      <params>
        <!-- Use last 2K of RAM -->
        <param name="ramAddress" type="integer" value="0x00100000"/>
        <param name="ramSize" type="integer" value="0x800"/>
      </params>
    </method_config>
  </method_configs>
</flash_config>
```

- The `flash_config` tag defines used XML spaces and schemas. This does not usually need to be changed. Under the `flash_config` tag, a `devices` tag is required. This contains a number of device tags, each representing one flash device on the target. The device tag defines the name of the device - this is the name reported by the **info flash** command and is used only when programming to a specific device. It also defines a number of regions where the flash device appears in the target's memory - the addresses of each region are matched against the address of each load region of the image being programmed.
- The `programming_type` tag defines the programming method and setup/teardown scripts to be used for a flash programming operation. Currently, only `FILE` is supported.

- The **method** tag defines the script which implements the programming method. Currently, only JYTHON is supported for the language attribute. The script and class attributes define which script file to load and the name of the class that implements the programming method within the script. The **method_config** attributes define which set of parameters are used by the device. This allows multiple devices to share a set of parameters.
- The **programming_type** may also have optional setup and teardown tags. These define a script and a method within that script to call before or after flash programming.
- Within the **method_configs** tag, the parameters for each device are contained within **method_config** tags.
- Parameters must have a unique name and a default value. You can override the value passed to the method. See the help for the **flash load** command within the DS-5 Debugger.
- Where the configuration file references another file, for example, the script files, the **FDB://** prefix indicates that the file is located in the **Flash** subdirectory of the configuration database. If there are multiple databases, then the **Flash** subdirectory of each database is searched until the file is found.
- The last file that needs to be changed is the **project_types.xml** file in the target's directory to tell DS-5 that the flash configuration can be found in the file created above. The following line should be added under the top-level **platform_data** tag:

```
<flash_config>CDB://flash.xml</flash_config>
```

The **CDB://** prefix tells DS-5 that the **flash.xml** file is located in the same directory as the **project_types.xml** file.

12.6 About testing the flash configuration

With the files described in the previous sections in place, it should be possible to make a connection to the target in DS-5 and inspect the flash devices available and program an image. Although, with the files in their current form, no data will actually be written to flash.

Note

If DS-5 is already open and `project_types.xml` is changed, it will be necessary to rebuild the configuration database.

Within DS-5 Debugger, connect to your target system and enter **info flash** into the Commands view. You should get an output similar to:

```
info flash
MainFlash
regions: 0x8000000-0x807FFFF
parameters: programPageTimeout: 100
             driverVersion: 257
             programPageSize: 0x400
             eraseSectorTimeout: 500
             sectorSizes: ((0x800, 0x00000000))
             valEmpty: 0xff
             type: 1
             size: 0x00080000
             name: STM32F10x High-density Flash
             address: 0x08000000
             algorithm: FDB://algorithms/STM32F10x_512.FLM
             coreName: Cortex-M3
             ramAddress: 0x20000000
             ramSize: 0x10000
             disableTimeouts: false
             verify: true
```

You can test the flash programming operation by attempting to program with a test ELF file.

```
flash load flashyprogram.axf
Writing segment 0x00008000 ~ 0x0000810C (size 0x10C)
Flash programming completed OK (target state has been preserved)
```

Note

You can use any ELF (`.axf`) file which contains data within the configured address range.

12.7 About flash method parameters

Programming methods can take parameters that serve to change the behavior of the flash programming operation.

Example parameters could be:

- The programming algorithm image to load, for example, the Keil Flash Algorithm file.
- The location & size of RAM the method can use for running code, buffers, and similar items.
- Clock speeds.
- Timeouts.
- Programming and erase page sizes.

The default values of the parameters are taken from the flash configuration file.

Note

You can override the parameters from the DS-5 command line.

The programming method can obtain the value of the parameters with:

- `getParameter(name)` returns the value of a parameter as a string. The method can convert this to another type, such as integers, as required. `None` is returned if no value is set for this parameter.
- `getParameters()` returns a map of all parameters to values. Values can then be obtained with the `[]` operator.

For example:

```
def setup(self):
    # get the name of the core to connect to
    coreName = self.getParameter("coreName")

    # get parameters for working RAM
    self.ramAddr = int(self.getParameter("ramAddress"), 0)
    self.ramSize = int(self.getParameter("ramSize"), 0)
```

12.8 About getting data to the flash algorithm

Data is passed to the `program()` function by the data parameter.

A data parameter is an object that provides the following functions:

- `getSize()` returns the amount of data available in bytes.
- `getData(sz)` returns a buffer of up to `sz` data bytes. This may be less, for example, at the end of the data. The read position is advanced.
- `seek(pos)` move the read position.
- `getUnderlyingFile()` gets the file containing the data. (None, if not backed by a file). This allows the method to pass the file to an external tool.

The method can process the data with:

```
def program(self, regionID, offset, data):
    data.seek(0)
    bytesWritten = 0
    while bytesWritten < data.getSize():
        # get next block of data
        buf = data.getData(self.pageSize)

        # write buf to flash
        bytesWritten += len(buf)
```

12.9 About interacting with the target

To perform flash programming, the programming method might need to access the target.

The flash programmer provides access to the DTSL APIs for this and the programming method can then get a connection with the `getConnection()` function of class `FlashMethodv1`.

This is called from the `setup()` function of the programming method. If there is already an open connection, for example, from the DS-5 Debugger, this will be re-used.

```
def setup(self):
    # connect to core
    self.conn = self.getConnection()
```

———— Note —————

An example, `flash_example-FVP-A9x4`, is provided with DS-5. This example shows two ways of programming flash devices using DS-5, one using a Keil Flash Method and the other using a Custom Flash Method written in Jython. For convenience, the `Cortex-A9x4 FVP` model supplied with DS-5 is used as the target device. This example can be used as a template for creating new flash algorithms. The `readme.html` provided with the example contains basic information on how to use the example.

Accessing the core

When interacting with the target, it might be necessary to open a connection to the core. If the debugger already has an open connection, a new connection might not be always possible. A utility function, `ensureDeviceOpen()`, is provided that will open the connection only if required. It will return `true` if the connection is open and so should be closed after programming in the `teardown()` function.

To access the core's registers and memory, the core has to be stopped. Use the `ensureDeviceStopped()` function to assist with this.

```
def setup(self):
    # connect to core & stop
    self.conn = self.getConnection()
    coreName = self.getParameter("coreName")
    self.dev = self.conn.getDeviceInterfaces().get(coreName)
    self.deviceOpened = ensureDeviceOpen(self.dev)
    ensureDeviceStopped(self.dev)

def teardown(self):
    if self.deviceOpened:
        # close device connection if opened by this script
        self.dev.closeConn()
```

Reading/writing memory

The core's memory can be accessed using the `memWrite()`, `memFill()`, and `memRead()` functions of the `dev` object (`IDevice`).

```
from com.arm.rddi import RDDI
from com.arm.rddi import RDDI_ACC_SIZE
from jarray import zeros

...

def program(self):
    ...
    self.dev.memFill(0, addr, RDDI_ACC_SIZE.RDDI_ACC_WORD,
                    RDDI.RDDI_MRUL_NORMAL, False, words, 0)
    self.dev.memWrite(0, addr, RDDI_ACC_SIZE.RDDI_ACC_WORD,
                    RDDI.RDDI_MRUL_NORMAL, False, len(buf), buf)
    ...
```

```
def verify(self):
    ...
    readBuf = zeros(len(buf), 'b')
    self.dev.memRead(0, addr, RDDI_ACC_SIZE.RDDI_ACC_WORD,
                    RDDI.RDDI_MRUL_NORMAL, len(readBuf), readBuf)
    ...
```

Utility routines to make the method code clearer are provided in `device_memory`:

```
from flashprogrammer.device_memory import writeToTarget, readFromTarget
...
def program(self):
    ...
    writeToTarget(self.dev, address, buf)
    ...
def verify(self):
    ...
    readBuf = readFromTarget(self.dev, addr, count)
    ...
```

Reading and writing registers

The core's registers can be read using the `regReadList()` and written using the `regWriteList()` functions of `Idevice`.

———— Note ————

You must be careful to only pass integer values and not long values.

These registers are accessed by using numeric IDs. These IDs are target specific. For example, `R0` is register 1 on a Cortex-A device, but register 0 on a Cortex-M device.

`execution.py` provides functions that map register names to numbers and allow reading or writing by name.

- `writeRegs(device, regs)` writes a number of registers to a device. `regs` is a list of (name, value) pairs. For example:

```
writeRegs (self.dev, [ ("R0", 0), ("R1", 1234), ("PC", 0x8000) ]
```

will set `R0`, `R1`, and `PC` (`R15`).

- `readReg(device, reg)` reads a named register. For example:

```
value = readReg ("R0")
```

will read `R0` and return its value.

Running code on the core

The core can be started and stopped via the `go()` and `stop()` functions. Breakpoints can be set with the `setSWBreak()` or `setHWBreak()` functions and cleared with the `clearSWBreak()` or `clearHWBreak()` functions. As it may take some time to reach the breakpoint, before accessing the target further, the script should wait for the breakpoint to be hit and the core stopped.

`execution.py` provides utility methods to assist with running code on the target.

- To request the core to stop and wait for the stop status event to be received, and raise an error if no event is received before timeout elapses.

```
stopDevice(device, timeout=1.0):
```

- To check the device's status and calls `stopDevice()` if it is not stopped.

```
ensureDeviceStopped(device, timeout=1.0):
```

- To start the core and wait for it to stop, forces the core to stop and raise an error if it doesn't stop before timeout elapses. The caller must set the registers appropriately and have set a breakpoint or vector catch to cause the core to stop at the desired address.

```
runAndWaitForStop(device, timeout=1.0):
```

- To set a software breakpoint at `addr`, start the core and wait for it to stop by calling `runAndWaitForStop()`. The caller must set the registers appropriately.

```
runToBreakpoint(device, addr, bpFlags = RDDI.RDDI_BRUL_STD, timeout=1.0):
```

Flash programming algorithms are often implemented as functions that are run on the target itself. These functions may take parameters where the parameters are passed through registers.

`funcCall()` allows methods to call functions that follow AAPCS (with some restrictions):

- Up to the first four parameters are passed in registers `R0-R3`.
- Any parameters above this are passed via the stack.
- Only integers up to 32-bit or pointer parameters are supported. Floating point or 64-bit integers are not supported.
- The result is returned in `R0`.

We can use the above to simulate flash programming by writing the data to RAM. See `example_method_1.py`. This:

- Connects to the target on `setup()`.
- Fills the destination RAM with 0s to simulate erase.
- Writes data to a write buffer in working RAM.
- Runs a routine that copies the data from the write buffer to the destination RAM.
- Verifies the write by reading from the destination RAM.

Loading programming algorithm images onto the target

Programming algorithms are often compiled into `.elf` images.

`FlashMethodv1.locateFile()` locates a file for example, from a parameter, resolving any `FDB://` prefix to absolute paths.

`symfile.py` provides a class, `SymbolFileReader`, that allows the programming method to load an image file and get the locations of symbols. For example, to get the location of a function:

```
# load the algorithm image
algorithmFile = self.locateFile(self.getParameter('algorithm'))
algoReader = SymbolFileReader(algorithmFile)

# Find the address of the Program() function
funcInfo = algoReader.getFunctionInfo()['Program']
programAddr = funcInfo['address']
if funcInfo['thumb']:
    # set bit 0 if symbol is thumb
    programAddr |= 1
```

`image_loader.py` provides routines to load the image to the target:

```
# load algorithm into working RAM
algoAddr = self.ramAddr + 0x1000 # allow space for stack, buffers etc
loadAllCodeSegmentsToTarget(self.dev, algoReader, algoAddr)
```


If the algorithm binary was linked as position independent, the addresses of the symbols are relative to the load address and this offset should be applied when running the code on the target:

```
programAddr += algoAddr
args = [ writeBuffer, destAddr, pageSize ]
funcCall(self.dev, programAddr, args, self.stackTop)
```

Progress reporting

Flash programming can be a slow process, so it is desirable to have progress reporting features. The method can do this by calling `operationStarted()`. This returns an object with functions:

- `progress()` - update the reported progress.
- `complete()` - report the operation as completed, with a success or failure.

Progress reporting can be added to the `program()` function in the previous example:

```
def program(self, regionID, offset, data):
    # calculate the address to write to
    region = self.getRegion(regionID)
    addr = region.getAddress() + offset

    # Report progress, assuming erase takes 20% of the time, program 50%
    # and verify 30%
    progress = self.operationStarted(
        'Programming 0x%x bytes to 0x%08x' % (data.getSize(), addr),
        100)

    self.doErase(addr, data.getSize())
    progress.progress('Erasing completed', 20)

    self.doWrite(addr, data)
    progress.progress('Writing completed', 20+50)

    self.doVerify(addr, data)
    progress.progress('Verifying completed', 20+50+30)

    progress.completed(OperationResult.SUCCESS, 'All done')

    # register values have been changed
    return TargetStatus.STATE_LOST
```

The above example only has coarse progress reporting, only reporting at the end of each phase. Better resolution can be achieved by allowing each sub-task to have a progress monitor. `subOperation()` creates a child progress monitor.

Care should be taken to ensure `completed()` is called on the progress monitor when an error occurs. It is recommended that a `try: except:` block is placed around the code after a progress monitor is created.

```
import java.lang.Exception
def program(self, regionID, offset, data):
    progress = self.operationStarted(
        'Programming 0x%x bytes to 0x%08x' % (data.getSize(), addr),
        100)
    try:
        # Do programming
    except (Exception, java.lang.Exception), e:
        # exceptions may be derived from Java Exception or Python Exception
        # report failure to progress monitor & rethrow
        progress.completed(OperationResult.FAILURE, 'Failed')
        raise
```

———— Note ————

`import java.lang.Exception` - If you omit import and a Java exception is thrown, you may get a confusing error report from Jython indicating that it cannot find the Java namespace. Further, the python line location indicated as the source of the error will not be accurate.

Cancellation

If you wish to abort a long-running flash operation, programming methods can call `isCancelled()` to check if the operation is cancelled. If this returns true, the method stops programming.

———— **Note** ————

The `teardown()` functions are still called.

Messages

The programming method can report messages by calling the following:

- `warning()` - reports a warning message.
- `info()` - reports an informational message.
- `debug()` - reports a debug message - not normally displayed.

Locating and resolving files

`FlashMethodv1.locateFile()` locates a file for example, from a parameter, resolving any `FDB://` prefix to absolute paths.

This searches paths of all flash subdirectories of every configuration database configured in DS-5.

For example:

```
<DS5_INSTALL_DIR>/sw/debugger/configdb/Flash/
c:\MyDB\Flash
```

Error handling

Exceptions are thrown when errors occur. Errors from the API calls made by the programming method will be `com.arm.debug.flashprogrammer.FlashProgrammerException` (or derived from this). Methods may also report errors using Python's `raise` keyword. For example, if verification fails:

```
# compare contents
res = compareBuffers(buf, readBuf)
if res != len(buf):
    raise FlashProgrammerRuntimeException, "Verify failed at address: %08x" %
(addr + res)
```

If a programming method needs to ensure that a cleanup occurs when an exception is thrown, the following code forms a template:

```
import java.lang.Exception
...
try:
    # Do programming
except (Exception, java.lang.Exception), e:
    # exceptions may be derived from Java Exception or Python Exception
    # report failure to progress monitor & rethrow

    # Handle errors here

    # Rethrow original exception
    raise
finally:
    # This is always executed on success or failure
    # Close resources here
```

See the Progress handler section for example usage.

Note

`import java.lang.Exception` - If you omit import and a Java exception is thrown, you may get a confusing error report from Jython indicating that it cannot find the Java namespace. Further, the python line location indicated as the source of the error will not be accurate.

Running an external tool

Some targets may already have a standalone flash programming tool. It is possible to create a DS-5 Debugger programming method to call this tool, passing it to the path of the image to load. The following example shows how to do this, using the `fromelf` tool in place of a real flash programming tool.

```
from flashprogrammer.flash_method_v1 import FlashMethodv1
from com.arm.debug.flashprogrammer.IProgress import OperationResult
from com.arm.debug.flashprogrammer import TargetStatus
import java.lang.Exception
import subprocess

class RunProgrammer(FlashMethodv1):
    def __init__(self, methodServices):
        FlashMethodv1.__init__(self, methodServices)

    def program(self, regionID, offset, data):
        progress = self.operationStarted(
            'Programming 0x%x bytes with command %s' % (data.getSize(),
            ' '.join(cmd)),
            100)
        try:
            # Get the path of the image file
            file = data.getUnderlyingFile().getCanonicalPath()

            cmd = [ 'fromelf', file ]
            self.info("Running %s" % ' '.join(cmd))

            # run command
            proc = subprocess.Popen(cmd, stdout=subprocess.PIPE)
            out, err = proc.communicate()

            # pass command output to user as info message
            self.info(out)

            progress.progress('Completed', 100)
            progress.completed(OperationResult.SUCCESS, 'All done')

        except (Exception, java.lang.Exception), e:
            # exceptions may be derived from Java Exception or Python Exception
            # report failure to progress monitor & rethrow
            progress.completed(OperationResult.FAILURE, 'Failed')
            raise

        return TargetStatus.STATE_RETAINED
```

`os.environ` can be used to lookup environment variables, for example, the location of a target's toolchain:

```
programmerTool = os.path.join(os.environ['TOOLCHAIN_INSTALL'], 'flashprogrammer')
```

Setup and teardown

The flash configuration file can specify scripts to be run before and after flash programming. These are termed setup and teardown scripts and are defined using setup and teardown tags. The setup script should put the target into a state ready for flash programming.

This might involve one or more of:

- Reset the target.
- Disable interrupts.

- Disable peripherals that might interfere with flash programming.
- Setup DRAM.
- Enable flash control.
- Setup clocks appropriately.

The teardown script should return the target to a usable state following flash programming.

In both cases, it may be necessary to reset the target. The following code can be used to stop the core on the reset vector.

Note

This example code assumes that the core supports the RSET vector catch feature.

```
def setup(client, services):
    # get a connection to the core
    conn = services.getConnection()
    dev = conn.getDeviceInterfaces().get("Cortex-M3")
    ensureDeviceOpen(dev)
    ensureDeviceStopped(dev)

    dev.setProcBreak("RSET")
    dev.systemReset(0)
    # TODO: wait for stop!
    dev.clearProcBreak("RSET")
```

Other ways of providing flash method parameters

The flash configuration file can provide flash region information and flash parameter information encoded into the XML. However, for some methods, this information may need to be extracted from the flash algorithm itself.

Programming methods can extend any information in the flash configuration file (if any) with address regions and parameters for the method by overriding a pair of class methods - `getDefaultRegions()` and `getDefaultParameters()`.

```
getDefaultParameters().
from com.arm.debug.flashprogrammer import FlashRegion

...
class ProgrammingMethod(FlashMethodv1):
    ...

    def getDefaultRegions(self):
        return [ FlashRegion(0x00100000, 0x10000), FlashRegion(0x00200000, 0x10000) ]

    def getDefaultParameters(self):
        params = {}
        params['param1'] = "DefaultValue1"
        params['param2'] = "DefaultValue2"
        return params
```

The above code defines two 64kB regions at 0x00100000 and 0x00200000. Regions supplied by this method are only used if no regions are specified for the device in the configuration file. The above code defines 2 extra parameters. These parameters are added to the parameters in the flash configuration. If a parameter is defined in both, the default value in the flash configuration file is used. This region and parameter information can be extracted from the algorithm binary itself (rather than being hard-coded as in the above example). The Keil algorithm images contain a data structure defining regions covered by the device and the programming parameters for the device. The Keil programming method loads the algorithm binary (specified by a parameter in the configuration file) and extracts this information to return in these calls.

Chapter 13

Writing OS awareness for DS-5™ Debugger

This chapter contains information for programmers who wish to extend the operating system awareness in DS-5 Debugger to support additional operating systems. It contains the following:

- *13.1 About Writing operating system awareness for DS-5™ Debugger on page 13-334.*
- *13.2 Creating an OS awareness extension on page 13-335.*
- *13.3 Implementing the OS awareness API on page 13-339.*
- *13.4 Enabling the OS awareness on page 13-341.*
- *13.5 Implementing thread awareness on page 13-345.*
- *13.6 Implementing data views on page 13-348.*
- *13.7 Programming advice and noteworthy information on page 13-351.*

13.1 About Writing operating system awareness for DS-5™ Debugger

DS-5 Debugger offers an Application Programming Interface (API) for third parties to contribute awareness for their operating systems (OS).

The OS awareness extends the debugger to provide a representation of the OS threads - or tasks - and other relevant data structures, typically semaphores, mutexes, or queues. Thread-awareness, in particular, enables the following features in the debugger:

- Setting breakpoints for a particular thread, or a group of threads.
- Displaying the call stack for a specific thread.
- For any given thread, inspecting local variables and register values at a selected stack frame.

To illustrate different stages of the implementation, this chapter explains how to add support for a fictional OS named `myos`.

The steps can be summarized as follows:

1. Create a new configuration database folder to host the OS awareness extension and add it to the DS-5 Debugger preferences in Eclipse.
2. Create the files `extension.xml` and `messages.properties` so that the extension appears on the **OS Awareness** tab in the Debug configuration dialog.
3. Add `provider.py` and implement the awareness enablement logic.
4. Add `contexts.py` and implement the thread awareness.
5. Add `tasks.py` to contribute a table to the **RTOS Data** view, showing detailed information about tasks.

13.2 Creating an OS awareness extension

The debugger searches for OS awareness extensions in its configuration database. All files pertaining to a particular extension must be located in a folder or at the root of a Java Archive (JAR) file in the configuration database **OS/** folder.

The actual name of the folder or JAR file is not relevant and is not shown anywhere. Within this folder or JAR file, the debugger looks for a file named **extension.xml** to discover the OS awareness extension.

The actual name of the folder or JAR file is not relevant and is not shown anywhere. Within this folder or JAR file, the debugger looks for a file named **extension.xml** to discover the OS awareness extension.

This file contains the following information:

- The OS name, description, and, optionally, a logo to display in the **OS Awareness** selection pane.
- The root Python script or Java class providing the actual implementation.
- The details of cores, architectures, or platforms this implementation applies to.

You can create a new OS awareness extension by directly modifying the configuration database in the DS-5 installation folder with appropriate access privileges, but this is not recommended.

Instead, create a new configuration database folder, containing an empty folder named **OS** (upper case):

```
<some folder>  
  /mydb  
    /OS
```

Then, add **mydb** to the known configuration databases in the Eclipse preferences panel for DS-5.

1. In Eclipse, go to menu **Windows > Preferences**, then expand the DS-5 node and select **Configuration Database**.
2. Then click **Add** and enter the path to **mydb**.

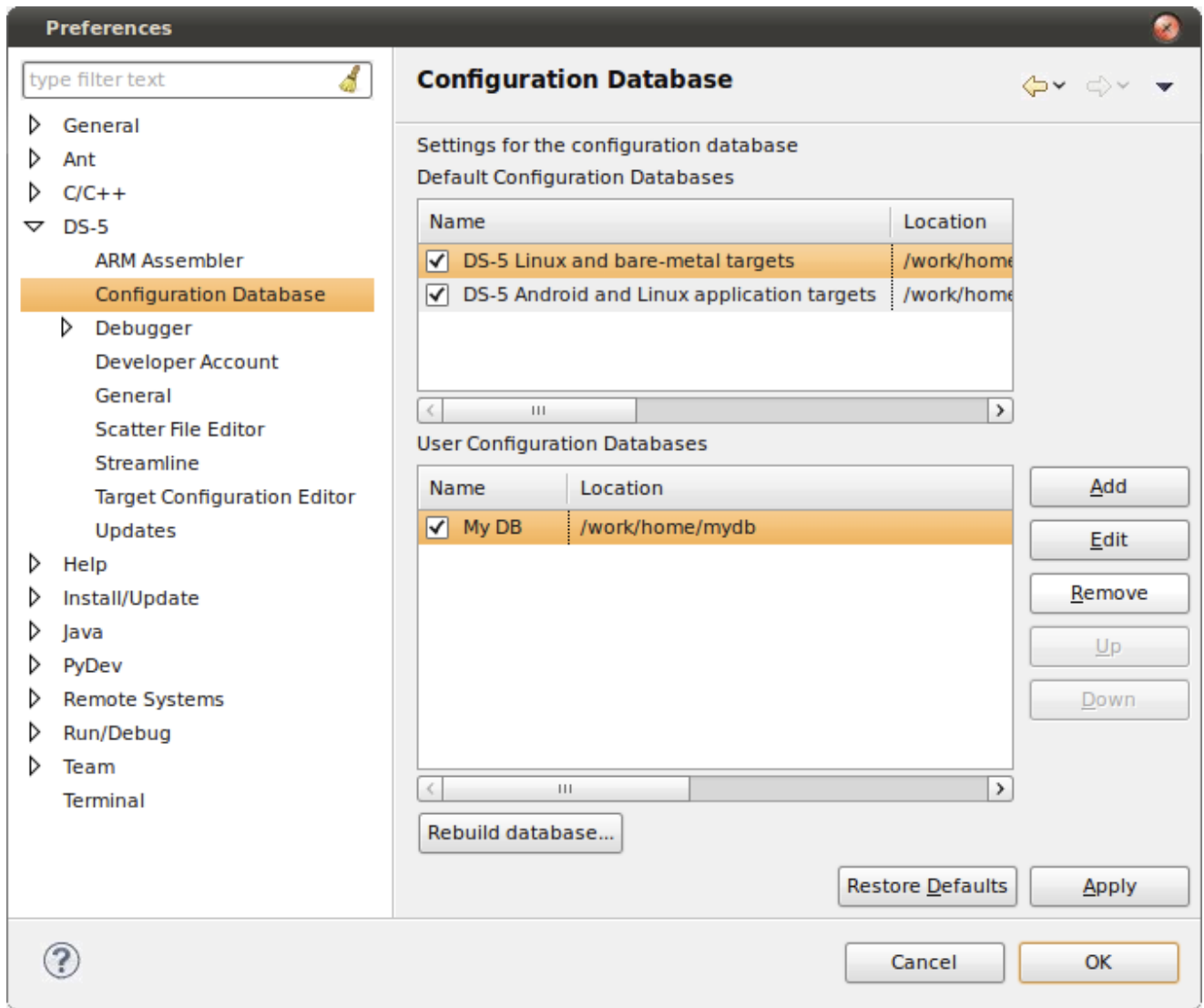


Figure 13-1 Eclipse preferences for mydb

———— **Note** ————

The some folder entry is represented by /work/home in Figure 1.

3. Now, add the OS awareness extension in mydb/OS. To do so, create a new folder named myos containing the following files:

```
<some folder>
  /mydb
    /OS
      /myos
        /extension.xml
        /messages.properties
```

As explained earlier, `extension.xml` declares the OS awareness extension. The schema describing the structure of file `extension.xml` can be found in the DS-5 installation folder at `sw/debugger/configdb/Schemas/os_extension.xsd`.

The file `messages.properties` contains all user-visible strings. The file format is documented here:

[http://docs.oracle.com/javase/7/docs/api/java/util/Properties.html#load\(java.io.Reader\)](http://docs.oracle.com/javase/7/docs/api/java/util/Properties.html#load(java.io.Reader))

Having user-visible strings in a separate file allows them to be translated. The debugger searches for translations in the following order in the named files:

- First `messages_<language code>_<country code>.properties`,
- Then `messages_<language code>.properties`,
- And finally in `messages.properties`.

Language and country codes are defined here respectively:

- <http://www.loc.gov/standards/iso639-2/englangn.html>
- <http://www.iso.ch/iso/en/prods-services/iso3166ma/02iso-3166-code-lists/list-en1.html>

4. Consider the following content to start adding support for `myos`:

- `extension.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<os id="myos" version="5.15" xmlns="http://www.arm.com/os_extension">
  <name>myos.title</name>
  <description>myos.desc</description>
  <provider><!-- todo --></provider>
</os>
```

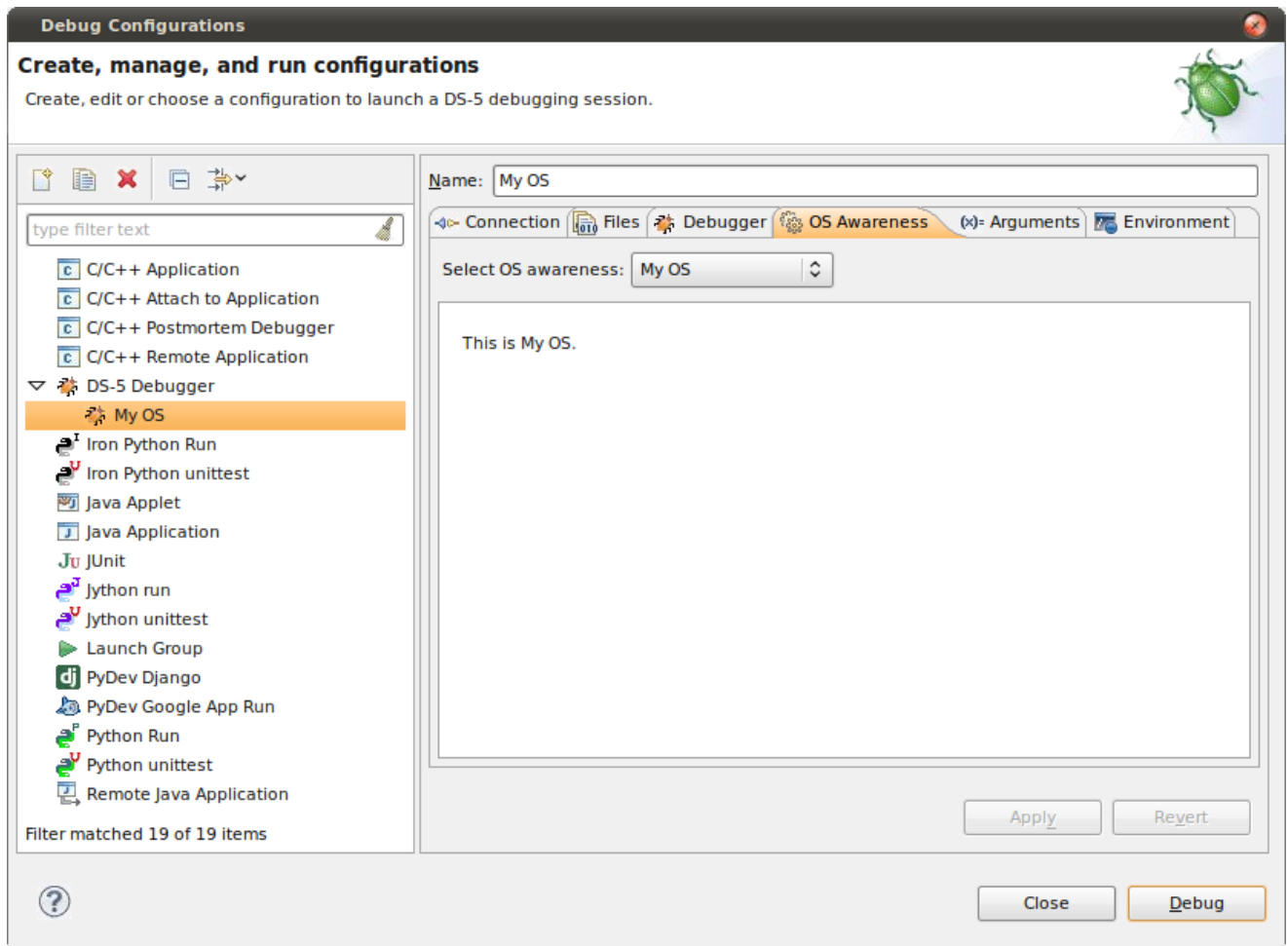
———— **Note** ————

The version attribute in the OS element refers to the API version, which is aligned with the version of DS-5 the API was made public with. There currently is a unique API version: 5.15. In the future, there may be higher versions as the API evolves. Backwards compatibility will be maintained as much as possible, until earlier versions start being unsupported. The debugger will no longer load extensions built against an unsupported version of the API.

- `messages.properties`

```
myos.title=My OS
myos.desc=This is My OS.
myos.help=Displays information about My OS.
```

This is sufficient for the OS awareness extension to appear in the Eclipse debug configuration, even though the implementation is obviously not complete and would cause errors if it is used for an actual debugger connection at this stage:



The `myos.help` string is only visible from the debugger's command line interface, for instance, when typing **help myos** once connected.

Using the `extension.xml`, you can include a reference to an image file to be shown above the description in the **OS Awareness** tab. Supported image formats are `.BMP`, `.GIF`, `.JPEG`, `.PNG`, and `.TIFF`.

Also, it is possible to control the targets for which the OS awareness extension is available.

The complete XML schema for `extension.xml` file is available in `[DS-5 install folder]/sw/debugger/configdb/Schemas/os_extension.xsd`.

Figure 13-2 Custom OS awareness displayed in Eclipse Debug Configurations dialog

13.3 Implementing the OS awareness API

The OS awareness API consists of callbacks that the debugger makes at specific times. For each callback, the debugger provides a means for the implementation to retrieve information about the target and resolve variables and pointers, through an expression evaluator.

The API exists primarily as a set of Java interfaces since the debugger itself is written in Java. However, the debugger provides a Python interpreter and bindings to translate calls between Python and Java, allowing the Java interfaces to be implemented by Python scripts. This section and the next ones refer to the Java interfaces but explain how to implement the extension in Python.

———— Note ————

A Python implementation does not require any particular build or compilation environment, as opposed to a Java implementation. On the other hand, investigating problems within Python code is more difficult, and you are advised to read the *Programming advice and noteworthy information* section before starting to write your own Python implementation.

The detailed Java interfaces to implement are available in the DS-5 installation folder under `sw/eclipse/dropins/plugins`, within the `com.arm.debug.extension.source_<version>.jar` file.

———— Note ————

You are encouraged to read the Javadoc documentation on Java interfaces as it contains essential information that is not presented here.

The Java interface of immediate interest at this point is `IOSProvider`, in the package `com.arm.debug.extension.os`. This interface must be implemented by the provider instance that was left out with a `todo` comment in `extension.xml`.

First, add the simplest implementation to the configuration database entry:

```
<some folder>
  /mydb
    /OS
      /myos
        /extension.xml
        /messages.properties
        /provider.py
```

- `extension.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<os id="myos" version="5.15" xmlns="http://www.arm.com/os_extension">
  <name>myos.title</name>
  <description>myos.desc</description>
  <provider>provider.py</provider>
</os>
```

- `provider.py`

```
# this script implements the Java interface IOSProvider
def areOSSymbolsLoaded(debugger):
    return False

def isOSInitialised(debugger):
    return False

def getOSContextProvider():
    return None

def getDataModel():
    return None
```

This is enough to make the OS awareness implementation valid, and a debug configuration with this OS awareness selected will work, although this would not add anything on top of a plain bare-metal connection. However, this illustrates the logical lifecycle of the OS awareness:

1. Ensure debug information for the OS is available. On loading symbols, the debugger calls `areOSSymbolsLoaded()`; the implementation returns true if it recognizes symbols as belonging to the OS, enabling the next callback.
2. Ensure the OS is initialized. Once the symbols for the OS are available, the debugger calls `isOSInitialised()`, immediately if the target is stopped or whenever the target stops next. This is an opportunity for the awareness implementation to check that the OS has reached a state where threads and other data structures are ready to be read, enabling the next two callbacks.
3. Retrieve information about threads and other data structures. Once the OS is initialized, the debugger calls out to `getOSContextProvider()` and `getDataModel()` to read information from the target. In reality, the debugger may call out to `getOSContextProvider()` and `getDataModel()` earlier on, but does not use the returned objects to read from the target until `areOSSymbolsLoaded()` and `isOSInitialised()` both returned true.

13.4 Enabling the OS awareness

The below implementation in `provider.py`, assumes `myos` has a global variable called `tasks` listing the OS tasks in an array and another global variable `scheduler_running` indicating that the OS has started scheduling tasks.

```
# this script implements the Java interface IOSProvider
from osapi import DebugSessionException

def areOSSymbolsLoaded(debugger):
    return debugger.symbolExists("tasks") \
        and debugger.symbolExists("scheduler_running")

def isOSInitialised(debugger):
    try:
        result = debugger.evaluateExpression("scheduler_running")
        return result.readAsNumber() == 1
    except DebugSessionException:
        return False

def getOSContextProvider():
    return None

def getDataModel():
    return None
```

The `osapi` module in the import statement at the top of `provider.py` is a collection of wrappers around Java objects and utility functions. The file `osapi.py` itself can be found in JAR file `com.arm.debug.extension_<version>.jar`.

Connecting to a running target and loading symbols manually for the OS shows both `areOSSymbolsLoaded()` and `isOSInitialised()` stages distinctly.

- On connecting to the target running the OS, without loading symbols, the **Debug Control** view displays **No OS Support**.

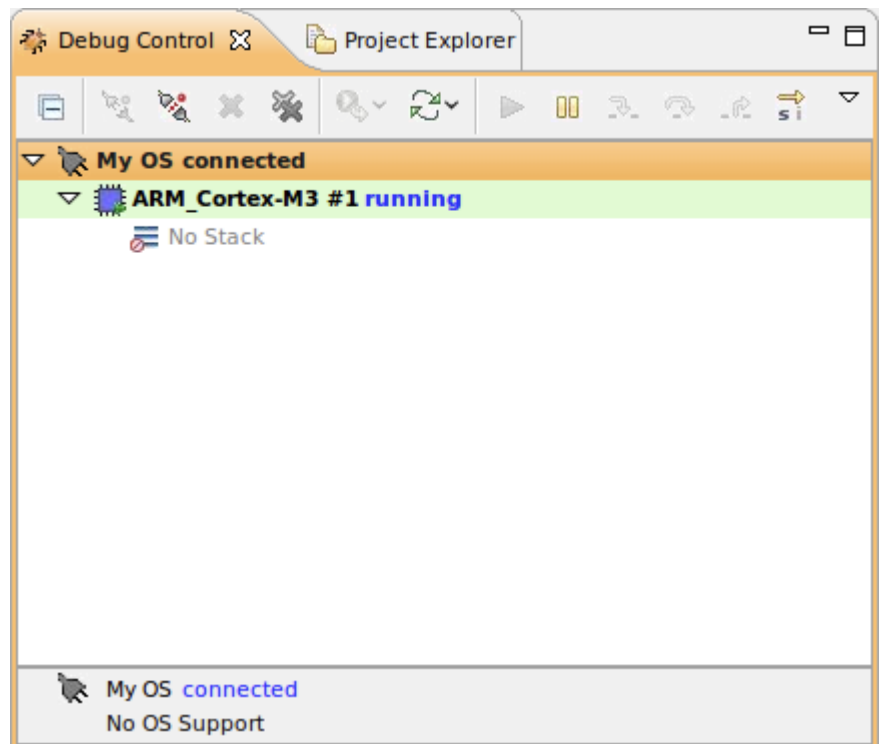


Figure 13-3 myos No OS Support

- After loading symbols for the OS, with the target still running, the **Debug Control** view now displays **Waiting for the target to stop**. At this point, `areOSSymbolsLoaded()` has been called and returned true, and the debugger is now waiting for the target to stop to call `isOSInitialised()`.

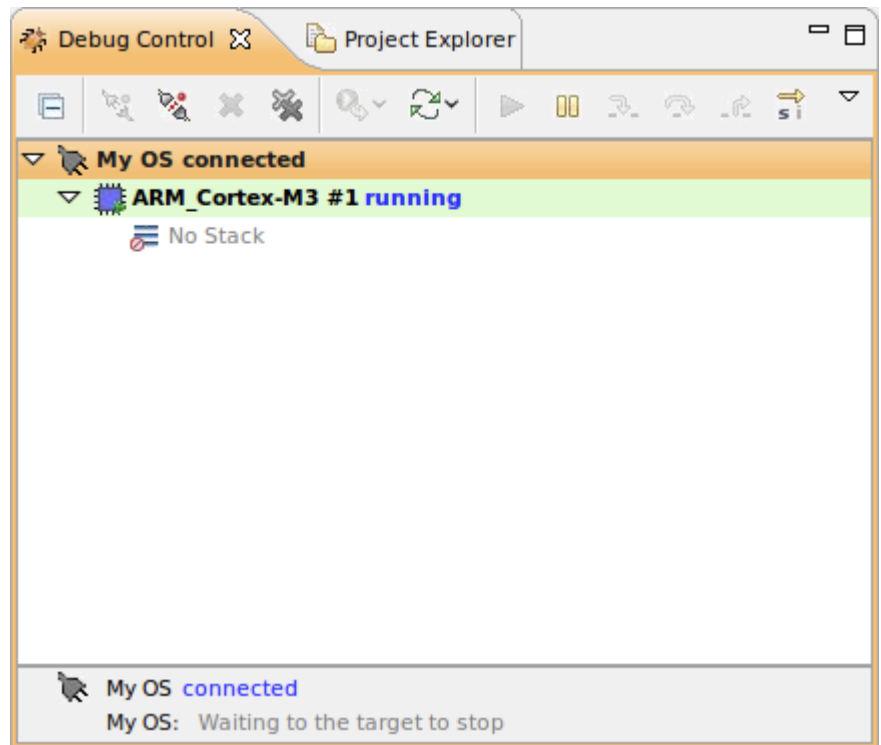


Figure 13-4 myos waiting for target to stop

- As soon as the target is stopped, the **Debug Control** view updates to show the OS awareness is enabled. At this point, `isOSInitialised()` has been called and returned true.

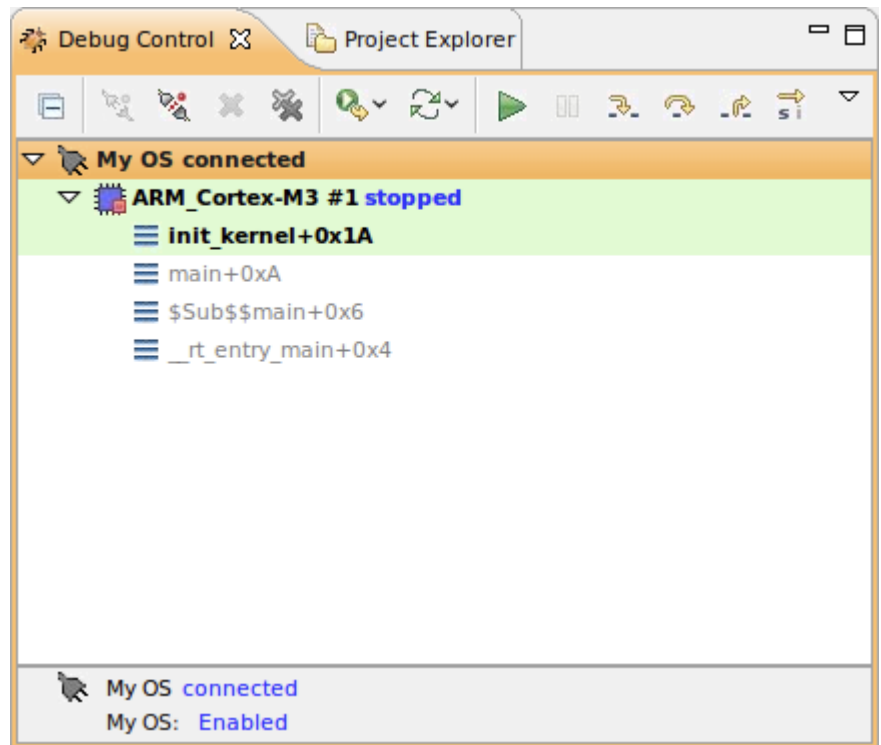


Figure 13-5 myos Enabled

- Another case worth mentioning is where `areOSSymbolsLoaded()` returns `true` but `isOSInitialised()` returns `false`. This can happen for instance when connecting to a stopped target, loading both the kernel image to the target and associated symbols in the debugger and starting debugging from a point in time earlier than the OS initialization, for example, debugging from the image entry point. In this case, the **Debug Control** view shows **Waiting for the OS** to be initialised as `scheduler_running` is not set to 1 yet, but symbols are loaded:

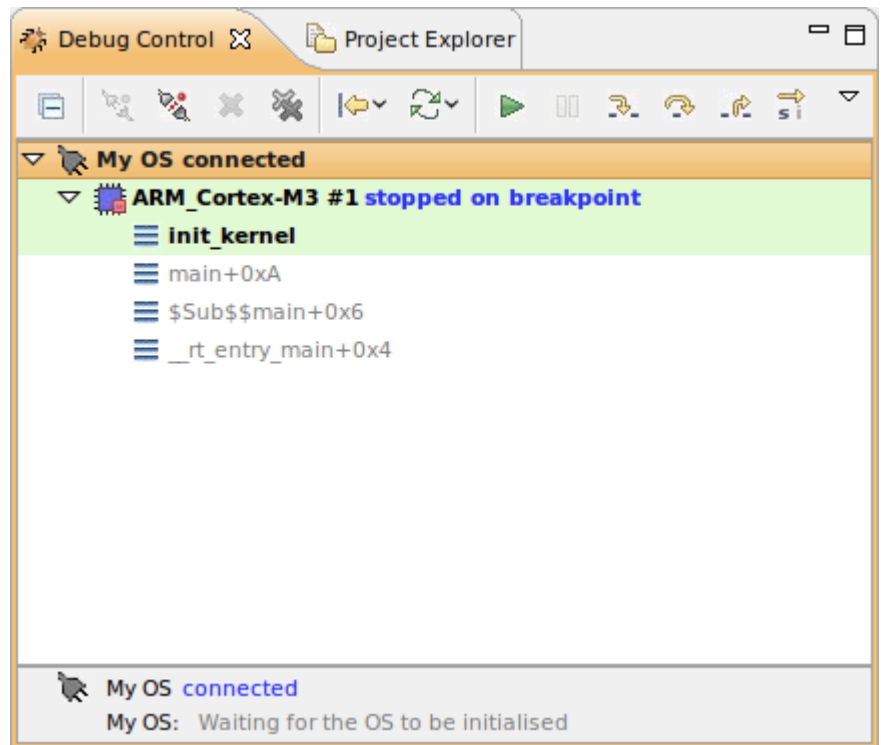


Figure 13-6 myos waiting for OS initialization

Without the call to `isOSInitialised()` the debugger lets the awareness implementation start reading potentially uninitialized memory, which is why this callback exists. The debugger keeps calling back to `isOSInitialised()` on subsequent stops until it returns true, and the OS awareness can finally be enabled.

13.5 Implementing thread awareness

Thread awareness is probably the most significant part of the implementation.

The corresponding call on the API is `getOSContextProvider()`, where `context` here means `execution context`, as in a thread or a task. The API expects an instance of the Java interface `IOSContextProvider` to be returned by `getOSContextProvider()`. This interface can be found in package `com.arm.debug.extension.os.context` within the same JAR file as `IOSProvider` mentioned earlier.

Given the following C types for myos tasks:

```
typedef enum {
    UNINITIALIZED = 0,
    READY
} tstatus_t ;

typedef struct {
    uint32_t      id;
    char          *name;
    volatile tstatus_t status;
    uint32_t      stack[STACK_SIZE];
    uint32_t      *sp;
} task_t;
```

And assuming the OS always stores the currently running task at the first element of the tasks array, further callbacks can be implemented to return the currently running (or scheduled) task and all the tasks (both scheduled and unscheduled) in a new `contexts.py` file:

```
<some folder>
  /mydb
    /OS
      /myos
        /extension.xml
        /messages.properties
        /provider.py
        /contexts.py
```

- `provider.py`

```
# this script implements the Java interface IOSProvider
from osapi import DebugSessionException
from contexts import ContextsProvider

def areOSSymbolsLoaded(debugger):
    [...]

def isOSInitialised(debugger):
    [...]

def getOSContextProvider():
    # returns an instance of the Java interface IOSContextProvider
    return ContextsProvider()

def getDataModel():
    [...]
```

- `contexts.py`

```
from osapi import ExecutionContext
from osapi import ExecutionContextsProvider

# this class implements the Java interface IOSContextProvider
class ContextsProvider(ExecutionContextsProvider):
    def getCurrentOSContext(self, debugger):
        task = debugger.evaluateExpression("tasks[0]")
        return self.createContext(debugger, task)

    def getAllIOSContexts(self, debugger):
        tasks = debugger.evaluateExpression("tasks").getArrayElements()
        contexts = []
        for task in tasks:
            if task.getStructureMembers()["status"].readAsNumber() > 0:
```

```

        contexts.append(self.createContext(debugger, task))
    return contexts

def getOSContextSavedRegister(self, debugger, context, name):
    return None

def createContext(self, debugger, task):
    members = task.getStructureMembers()
    id = members["id"].readAsNumber()
    name = members["name"].readAsNullTerminatedString()
    context = ExecutionContext(id, name, None)
    return context

```

Although `getOSContextSavedRegister()` is not yet implemented, this is enough for the debugger to now populate the **Debug Control** view with the OS tasks as soon as the OS awareness is enabled:

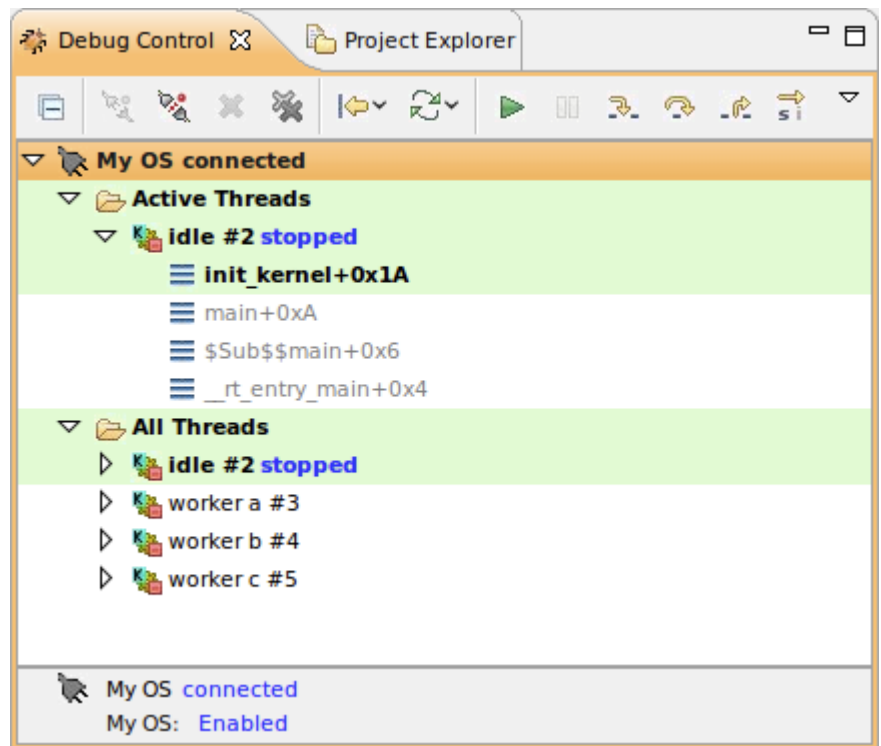


Figure 13-7 myos Debug Control view data

Decoding the call stack of the currently running task and inspecting local variables at specific stack frames for that task works without further changes since the task's registers values are read straight from the core's registers. For unscheduled tasks, however, `getOSContextSavedRegister()` must be implemented to read the registers values saved by the OS on switching contexts. How to read those values depends entirely on the OS context switching logic.

Here is the implementation for `myos`, based on a typical context switching routine for M-class ARM processors where registers are pushed onto the stack when a task is switched out by the OS scheduler:

```

from osapi import ExecutionContext
from osapi import ExecutionContextsProvider

STACK_POINTER = "stack pointer"
REGISTER_OFFSET_MAP = {
    "R4":0L, "R5":4L, "R6":8L, "R7":12L,
    "R8":16L, "R9":20L, "R10":24L, "R11":28L,
    "R0":32L, "R1":36L, "R2":40L, "R3":44L,
    "R12":48L, "LR":52L, "PC":56L, "XPSR":60L,
    "SP":64L}

```

```
# this class implements the Java interface IOSContextProvider
class ContextsProvider(ExecutionContextsProvider):

    def getCurrentOSContext(self, debugger):
        [...]

    def getAllOSContexts(self, debugger):
        [...]

    def getOSContextSavedRegister(self, debugger, context, name):
        offset = REGISTER_OFFSET_MAP.get(name)
        base = context.getAdditionalData()[STACK_POINTER]
        addr = base.addOffset(offset)

        if name == "SP":
            # SP itself isn't pushed onto the stack: return its computed value
            return debugger.evaluateExpression("(long)" + str(addr))
        else:
            # for any other register, return the value at the computed address
            return debugger.evaluateExpression("(long*)" + str(addr))

    def createContext(self, debugger, task):
        members = task.getStructureMembers()
        id = members["id"].readAsNumber()
        name = members["name"].readAsNullTerminatedString()
        context = ExecutionContext(id, name, None)
        # record the stack address for this task in the context's
        # additional data; this saves having to look it up later in
        # getOSContextSavedRegister()
        stackPointer = members["sp"].readAsAddress()
        context.getAdditionalData()[STACK_POINTER] = stackPointer

        return context
```

The debugger can now get the values of saved registers, allowing unwinding the stack of unscheduled tasks.

Note

Enter **info threads** in the **Commands** view to display similar information as displayed in the **Debug Control** view.

13.6 Implementing data views

Along with threads, OS awareness can provide arbitrary tabular data, which the debugger shows in the **RTOS Data** view.

The corresponding callback on the API is `getDataModel()`. It must return an instance of the Java interface `com.arm.debug.extension.datamodel.IDataModel`, which sources can be found in `com.arm.debug.extension.source_<version>.jar`.

This section demonstrates how to implement a view, listing the tasks, including all available information. The following additions to the implementation creates an empty **Tasks** table in the **RTOS Data** view:

```
<some folder>
  /mydb
    /OS
      /myos
        /extension.xml
        /messages.properties
        /provider.py
        /contexts.py
        /tasks.py
```

- `provider.py`

```
# this script implements the Java interface IOSProvider
from osapi import DebugSessionException
from osapi import Model
from contexts import ContextsProvider
from tasks import Tasks

def areOSSymbolsLoaded(debugger):
    [...]

def isOSInitialised(debugger):
    [...]

def getOSContextProvider():
    [...]

def getDataModel():
    # returns an instance of the Java interface IDataModel
    return Model("myos", [Tasks()])
```

- `messages.properties`

```
myos.title=My OS
myos.desc=This is My OS.
myos.help=Displays information about My OS.

tasks.title=Tasks
tasks.desc=This table shows all tasks, including uninitialized ones
tasks.help=Displays tasks defined within the OS and their current status.

tasks.id.title=Task
tasks.id.desc=The task identifier
tasks.name.title=Name
tasks.name.desc=The task name
tasks.status.title=Status
tasks.status.desc=The task status
tasks.priority.title=Priority
tasks.priority.desc=The task priority
tasks.sp.title=Stack pointer
tasks.sp.desc=This task's stack address
```

- `tasks.py`

```
from osapi import Table
from osapi import createField
from osapi import DECIMAL, TEXT, ADDRESS

# this class implements the Java interface IDataModelTable
class Tasks(Table):
    def __init__(self):
```

```
id = "tasks"
fields = [createField(id, "id", DECIMAL),
          createField(id, "name", TEXT),
          createField(id, "status", TEXT),
          createField(id, "priority", DECIMAL),
          createField(id, "sp", ADDRESS)]
Table.__init__(self, id, fields)

def getRecords(self, debugger):
    records = [] # todo
```

Functions `createField` and `Table.__init__()` automatically build up the keys to look for at run-time in the `messages.properties` file. Any key that is not found in `messages.properties` is printed as is.

The above modifications create a new empty Tasks table:

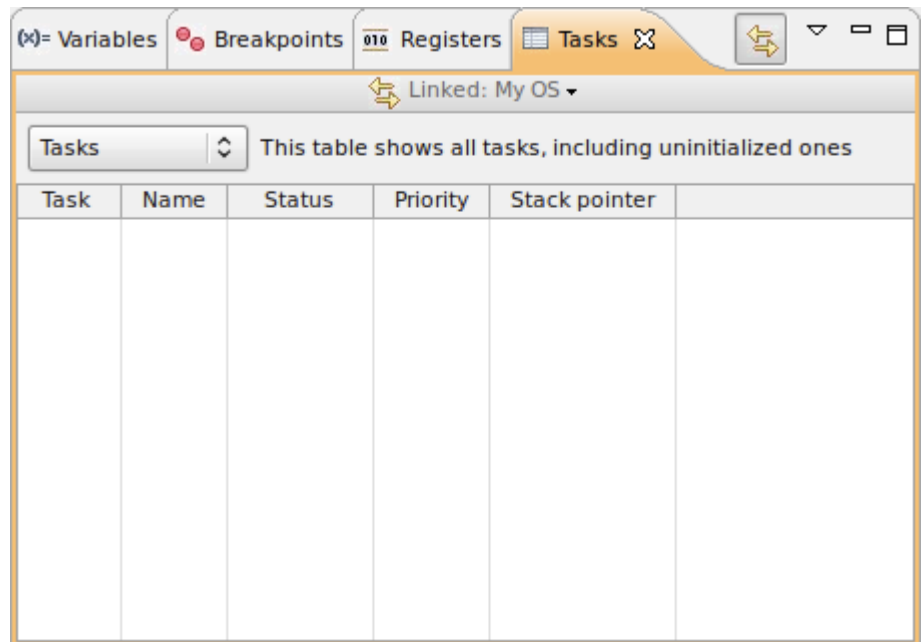


Figure 13-8 myos Empty Tasks table

To populate the table, `getRecords()` in `tasks.py` must be implemented:

```
from osapi import Table
from osapi import createField
from osapi import createNumberCell, createTextCell, createAddressCell
from osapi import DECIMAL, TEXT, ADDRESS

# this class implements the Java interface IDataModelTable
class Tasks(Table):
    def __init__(self):
        [...]

    def readTask(self, task, first):
        members = task.getStructureMembers()
        id = members["id"].readAsNumber()

        if (members["status"].readAsNumber() == 0):
            status = "Uninitialized"
            name = None
            sp = None
            priority = None
        else:
            if (first):
                status = "Running"
            else:
                status = "Ready"

        name = members["name"].readAsNullTerminatedString()
```

```

        sp = members["sp"].readAsAddress()
        priority = members["priority"].readAsNumber()

        cells = [createNumberCell(id),
                  createTextCell(name),
                  createTextCell(status),
                  createNumberCell(priority),
                  createAddressCell(sp)]

        return self.createRecord(cells)

def getRecords(self, debugger):
    records = []
    tasks = debugger.evaluateExpression("tasks").getArrayElements()
    first = True

    for task in tasks:
        records.append(self.readTask(task, first))
        first = False

    return records

```

With this implementation, the **Tasks** table now shows the following values:

The screenshot shows the 'Tasks' window in a debugger. The window has tabs for Variables, Breakpoints, Registers, and Tasks. The 'Tasks' tab is active, showing a table with columns: Task, Name, Status, Priority, Stack pointer, and an empty column. A dropdown menu is open for the 'Task' column, showing 'Tasks' and a refresh icon. The text 'This table shows all tasks, including uninitialized ones' is displayed. The window title bar shows 'Linked: My OS'.

Task	Name	Status	Priority	Stack pointer	

Figure 13-9 myos populated Tasks table

- Note

The debugger command **info myos tasks** prints the same information in the **Commands** view.

13.7 Programming advice and noteworthy information

Investigating issues in Python code for an OS awareness extension can sometimes be difficult.

Here are a few recommendations to make debugging easier:

- Start Eclipse from a console. Python `print` statements go to the Eclipse process standard output/error streams, which are not visible unless Eclipse is started from a console.

— On Linux, open a new terminal and run:

```
<DS-5 installation folder>/bin/eclipse
```

— On Windows, open command prompt and run:

```
<DS-5 installation folder>\bin\eclipsec
```

Note the trailing `c` in `eclipsec`.

- Use the **Error Log** view. Most errors that occur in the debugger are logged in details in the **Error Log** view. The full stack trace of an error is particularly useful as it often contains references to the location in the source files that generated the error.
- Turn on **verbose error logging** in the debugger

Although most errors are logged in the **Error Log** view, any error happening in the debugger event processing logic is not. One alternative is to turn on verbose error logging to print the full stack trace of errors in the **Console** view.

To turn on verbose error logging, execute the following command early in the debug session:

log config infoex

———— **Note** —————

- It is worth understanding that an OS awareness implementation interacts at the deepest level with the debugger, and some errors may cause the debugger to lose control of the target.
- Also note that semihosting is not available when OS awareness is specified.