

ARM[®] DS-5

Version 5.22

Debugger User Guide



ARM® DS-5**Debugger User Guide**

Copyright © 2010-2015 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
R	14 March 2014	Non-Confidential	Update for DS-5 version 5.18
S	27 June 2014	Non-Confidential	Update for DS-5 version 5.19
T	17 October 2014	Non-Confidential	Update for DS-5 version 5.20
U	20 March 2015	Non-Confidential	Update for DS-5 version 5.21
V	15 July 2015	Non-Confidential	Update for DS-5 version 5.22

Non-Confidential Proprietary Notice

This document is protected by copyright and other related rights and the practice or implementation of the information contained in this document may be protected by one or more patents or pending patent applications. No part of this document may be reproduced in any form by any means without the express prior written permission of ARM. **No license, express or implied, by estoppel or otherwise to any intellectual property rights is granted by this document unless specifically stated.**

Your access to the information in this document is conditional upon your acceptance that you will not use or permit others to use the information for the purposes of determining whether implementations infringe any third party patents.

THIS DOCUMENT IS PROVIDED “AS IS”. ARM PROVIDES NO REPRESENTATIONS AND NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY, SATISFACTORY QUALITY, NON-INFRINGEMENT OR FITNESS FOR A PARTICULAR PURPOSE WITH RESPECT TO THE DOCUMENT. For the avoidance of doubt, ARM makes no representation with respect to, and has undertaken no analysis to identify or understand the scope and content of, third party patents, copyrights, trade secrets, or other rights.

This document may include technical inaccuracies or typographical errors.

TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL ARM BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF ARM HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

This document consists solely of commercial items. You shall be responsible for ensuring that any use, duplication or disclosure of this document complies fully with any relevant export laws and regulations to assure that this document or any portion thereof is not exported, directly or indirectly, in violation of such export laws. Use of the word “partner” in reference to ARM’s customers is not intended to create or refer to any partnership relationship with any other company. ARM may make changes to this document at any time and without notice.

If any of the provisions contained in these terms conflict with any of the provisions of any signed written agreement covering this document with ARM, then the signed written agreement prevails over and supersedes the conflicting provisions of these terms. This document may be translated into other languages for convenience, and you agree that if there is any conflict between the English version of this document and any translation, the terms of the English version of the Agreement shall prevail.

Words and logos marked with ® or ™ are registered trademarks or trademarks of ARM Limited or its affiliates in the EU and/or elsewhere. All rights reserved. Other brands and names mentioned in this document may be the trademarks of their respective owners. Please follow ARM’s trademark usage guidelines at <http://www.arm.com/about/trademark-usage-guidelines.php>

Copyright © [2010-2015], ARM Limited or its affiliates. All rights reserved.

ARM Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

LES-PRE-20349

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 Debugger User Guide

Preface

<i>About this book</i>	15
------------------------------	----

Chapter 1

Getting Started with DS-5 Debugger

1.1	<i>About DS-5 Debugger</i>	1-18
1.2	<i>Debugger concepts</i>	1-19
1.3	<i>Terminology</i>	1-21
1.4	<i>Launching the debugger from Eclipse</i>	1-22
1.5	<i>About DS-5 headless command-line debugger</i>	1-23
1.6	<i>Headless command-line debugger options</i>	1-24
1.7	<i>Specifying a custom configuration database using the headless command-line debugger</i>	1-29
1.8	<i>Capturing trace using the headless debugger</i>	1-31
1.9	<i>DS-5 Debug perspective keyboard shortcuts</i>	1-33
1.10	<i>DS-5 Debugger command-line console keyboard shortcuts</i>	1-34
1.11	<i>Standards compliance in DS-5 Debugger</i>	1-35

Chapter 2

Configuring and Connecting to a Target

2.1	<i>Types of target connections</i>	2-37
2.2	<i>About Fixed Virtual Platform (FVP) entries in the configuration database</i>	2-38
2.3	<i>Configuring a connection to a Linux target using gdbserver</i>	2-39
2.4	<i>Configuring a connection to a Fixed Virtual Platform (FVP) for Linux application debug</i>	2-41
2.5	<i>Adding a custom model</i>	2-43

2.6	Configuring a connection to a Linux Kernel	2-45
2.7	About configuring connections to a Linux target using Application Debug with Rewind Support	2-47
2.8	Configuring a connection to a bare-metal target	2-52
2.9	Configuring an Event Viewer connection to a bare-metal target	2-54
2.10	About the target configuration import utility	2-56
2.11	Adding a new platform	2-58
2.12	Adding a new configuration database to DS-5	2-60
2.13	Exporting an existing launch configuration	2-62
2.14	Importing an existing launch configuration	2-65
2.15	Disconnecting from a target	2-67

Chapter 3

Working with the Target Configuration Editor

3.1	About the target configuration editor	3-69
3.2	Target configuration editor - Overview tab	3-70
3.3	Target configuration editor - Memory tab	3-72
3.4	Target configuration editor - Peripherals tab	3-74
3.5	Target configuration editor - Registers tab	3-76
3.6	Target configuration editor - Group View tab	3-78
3.7	Target configuration editor - Enumerations tab	3-80
3.8	Target configuration editor - Configurations tab	3-82
3.9	Scenario demonstrating how to create a new target configuration file	3-84
3.10	Creating a power domain for a target	3-95
3.11	Creating a Group list	3-96
3.12	Importing an existing target configuration file	3-98
3.13	Exporting a target configuration file	3-100

Chapter 4

Controlling Execution

4.1	About loading an image on to the target	4-103
4.2	About loading debug information into the debugger	4-105
4.3	About passing arguments to main()	4-107
4.4	Running an image	4-108
4.5	Working with breakpoints and watchpoints	4-109
4.6	Working with conditional breakpoints	4-116
4.7	About pending breakpoints and watchpoints	4-120
4.8	Setting a tracepoint	4-121
4.9	Setting Streamline start and stop points	4-122
4.10	Stepping through an application	4-123
4.11	Handling Unix signals	4-125
4.12	Handling processor exceptions	4-127
4.13	Configuring the debugger path substitution rules	4-129

Chapter 5

Examining the Target

5.1	Examining the target execution environment	5-132
5.2	Examining the call stack	5-134
5.3	About trace support	5-135
5.4	About post-mortem debugging of trace data	5-138

Chapter 6

Debugging Embedded Systems

6.1	About endianness	6-140
6.2	About accessing AHB, APB, and AXI buses	6-141

6.3	About virtual and physical memory	6-142
6.4	About address spaces	6-143
6.5	About debugging hypervisors	6-145
6.6	About debugging big.LITTLE systems	6-146
6.7	About debugging bare-metal symmetric multiprocessing systems	6-147
6.8	About debugging multi-threaded applications	6-149
6.9	About debugging shared libraries	6-150
6.10	About OS awareness	6-152
6.11	About debugging TrustZone enabled targets	6-157
6.12	About debugging a Unified Extensible Firmware Interface (UEFI)	6-159
6.13	About application rewind	6-160
6.14	About debugging MMUs	6-161
6.15	About Debug and Trace Services Layer (DTSL)	6-163
6.16	About CoreSight™ Target Access Library	6-164
6.17	About debugging caches	6-165

Chapter 7

Controlling Runtime Messages

7.1	About semihosting and top of memory	7-169
7.2	Working with semihosting	7-171
7.3	Enabling automatic semihosting support in the debugger	7-172
7.4	Controlling semihosting messages using the command-line console	7-173
7.5	Controlling the output of logging messages	7-174
7.6	About Log4j configuration files	7-175
7.7	Customizing the output of logging messages from the debugger	7-176

Chapter 8

Debugging with Scripts

8.1	Exporting DS-5 Debugger commands generated during a debug session	8-180
8.2	Creating a DS-5 Debugger script	8-181
8.3	Creating a CMM-style script	8-182
8.4	About Jython scripts	8-183
8.5	Jython script concepts and interfaces	8-185
8.6	Creating Jython projects in Eclipse for DS-5	8-186
8.7	Creating a Jython script	8-189
8.8	Running a script	8-191

Chapter 9

Working with the Snapshot Viewer

9.1	About the Snapshot Viewer	9-194
9.2	Components of a Snapshot Viewer initialization file	9-196
9.3	Connecting to the Snapshot Viewer	9-198
9.4	Considerations when creating debugger scripts for the Snapshot Viewer	9-199

Chapter 10

DS-5 Debug Perspectives and Views

10.1	App Console view	10-202
10.2	ARM Asm Info view	10-204
10.3	ARM assembler editor	10-205
10.4	Breakpoints view	10-207
10.5	C/C++ editor	10-211
10.6	Commands view	10-214
10.7	Debug Control view	10-217
10.8	Disassembly view	10-221
10.9	Events view	10-225

10.10	Event Viewer Settings dialog box	10-227
10.11	Expressions view	10-229
10.12	Functions view	10-233
10.13	History view	10-235
10.14	Memory view	10-237
10.15	MMU view	10-242
10.16	Modules view	10-246
10.17	Registers view	10-248
10.18	OS Data view	10-254
10.19	Cache Data view	10-255
10.20	Screen view	10-257
10.21	Scripts view	10-260
10.22	Target Console view	10-262
10.23	Target view	10-263
10.24	Trace view	10-265
10.25	Trace Control view	10-268
10.26	Variables view	10-271
10.27	Timed Auto-Refresh Properties dialog box	10-278
10.28	Memory Exporter dialog box	10-279
10.29	Memory Importer dialog box	10-280
10.30	Fill Memory dialog box	10-281
10.31	Export Trace Report dialog box	10-282
10.32	Breakpoint Properties dialog box	10-284
10.33	Watchpoint Properties dialog box	10-287
10.34	Tracepoint Properties dialog box	10-288
10.35	Manage Signals dialog box	10-289
10.36	Functions Filter dialog box	10-291
10.37	Script Parameters dialog box	10-292
10.38	Debug Configurations - Connection tab	10-293
10.39	Debug Configurations - Files tab	10-296
10.40	Debug Configurations - Debugger tab	10-300
10.41	Debug Configurations - OS Awareness tab	10-303
10.42	Debug Configurations - Arguments tab	10-304
10.43	Debug Configurations - Environment tab	10-306
10.44	DTSL Configuration Editor dialog box	10-308
10.45	Configuration Database panel	10-310
10.46	Configuration Perspective	10-312
10.47	About the Remote System Explorer	10-329
10.48	Remote Systems view	10-330
10.49	Remote System Details view	10-331
10.50	Target management terminal for serial and SSH connections	10-332
10.51	Remote Scratchpad view	10-333
10.52	Remote Systems terminal for SSH connections	10-334
10.53	Terminal Settings dialog box	10-335
10.54	Debug Hardware Configure IP view	10-337
10.55	Debug Hardware Firmware Installer view	10-339
10.56	Connection Browser dialog box	10-341
10.57	DS-5 Debugger menu and toolbar icons	10-342

Chapter 11

Troubleshooting

11.1	ARM Linux problems and solutions	11-346
11.2	Enabling internal logging from the debugger	11-347
11.3	Target connection problems and solutions	11-348

Chapter 12

File-based Flash Programming in ARM DS-5

12.1	About file-based flash programming in ARM® DS-5	12-350
12.2	Flash programming configuration	12-352
12.3	Creating an extension database for flash programming	12-354
12.4	About using or extending the supplied ARM® Keil® flash method	12-355
12.5	About creating a new flash method	12-357
12.6	About testing the flash configuration	12-361
12.7	About flash method parameters	12-362
12.8	About getting data to the flash algorithm	12-363
12.9	About interacting with the target	12-364

Chapter 13

Writing OS Awareness for DS-5 Debugger

13.1	About Writing operating system awareness for DS-5 Debugger	13-372
13.2	Creating an OS awareness extension	13-373
13.3	Implementing the OS awareness API	13-377
13.4	Enabling the OS awareness	13-379
13.5	Implementing thread awareness	13-382
13.6	Implementing data views	13-385
13.7	Programming advice and noteworthy information	13-388

Chapter 14

Debug and Trace Services Layer (DTSL)

14.1	Additional DTSL documentation and files	14-390
14.2	Need for DTSL	14-391
14.3	DS-5 configuration database	14-396
14.4	DTSL as used by DS-5 Debugger	14-402
14.5	Main DTSL classes and hierarchy	14-404
14.6	DTSL options	14-412
14.7	DTSL support for SMP and AMP configurations	14-418
14.8	DTSL Trace	14-422
14.9	Extending the DTSL object model	14-429
14.10	Debugging DTSL Jython code within DS-5 Debugger	14-434
14.11	DTSL in stand-alone mode	14-438

List of Figures

ARM® DS-5 Debugger User Guide

Figure 1-1	Enable trace in the DTSL options	1-31
Figure 1-2	Headless debugger connection with DTSL option	1-32
Figure 2-1	Adding a new configuration database	2-61
Figure 2-2	Export launch configuration dialog box	2-62
Figure 2-3	Launch configuration selection panels	2-63
Figure 2-4	Import launch configuration dialog box	2-65
Figure 2-5	Launch configuration file selection panels	2-66
Figure 3-1	Specifying TCF files in the Debug Configurations window	3-69
Figure 3-2	Target configuration editor - Overview tab	3-71
Figure 3-3	Target configuration editor - Memory tab	3-73
Figure 3-4	Target configuration editor - Peripherals tab	3-75
Figure 3-5	Target configuration editor - Registers tab	3-77
Figure 3-6	Target configuration editor - Group View tab	3-78
Figure 3-7	Target configuration editor - Enumerations tab	3-80
Figure 3-8	Target configuration editor - Configuration tab	3-83
Figure 3-9	LED register and bitfields	3-84
Figure 3-10	Core module and LCD control register	3-84
Figure 3-11	Creating a Memory map	3-85
Figure 3-12	Creating a peripheral	3-86
Figure 3-13	Creating a standalone register	3-87
Figure 3-14	Creating a peripheral register	3-88
Figure 3-15	Creating enumerations	3-89
Figure 3-16	Assigning enumerations	3-90

Figure 3-17	Creating remapping rules	3-91
Figure 3-18	Creating a memory region for remapping by a control register	3-92
Figure 3-19	Applying the Remap_RAM_block1 map rule	3-93
Figure 3-20	Applying the Remap_ROM map rule	3-94
Figure 3-21	Power Domain Configurations	3-95
Figure 3-22	Creating a group list	3-96
Figure 3-23	Selecting an existing target configuration file	3-98
Figure 3-24	Importing the target configuration file	3-99
Figure 3-25	Exporting to C header file	3-100
Figure 3-26	Selecting the files	3-101
Figure 4-1	Load File dialog box	4-103
Figure 4-2	Load additional debug information dialog box	4-106
Figure 4-3	Setting an execution breakpoint	4-111
Figure 4-4	Setting a data watchpoint	4-112
Figure 4-5	Viewing the properties of a data watchpoint	4-112
Figure 4-6	Breakpoint Properties dialog	4-117
Figure 4-7	Debug Control view	4-123
Figure 4-8	Manage signals dialog (Unix signals)	4-125
Figure 4-9	Manage signals dialog	4-127
Figure 4-10	Path Substitution dialog box	4-129
Figure 4-11	Edit Substitute Path dialog box	4-130
Figure 5-1	Target execution environment	5-132
Figure 5-2	Debug Control view	5-134
Figure 6-1	Threading call stacks in the Debug Control view	6-149
Figure 6-2	Adding individual shared library files	6-150
Figure 6-3	Modifying the shared library search paths	6-151
Figure 6-4	Cache Data view (showing L1 TLB cache)	6-165
Figure 6-5	DTSL Configuration Editor (Cache RAMs configuration tab)	6-166
Figure 7-1	Typical layout between top of memory, stack, and heap	7-169
Figure 8-1	Commands generated during a debug session	8-180
Figure 8-2	PyDev project wizard	8-186
Figure 8-3	PyDev project settings	8-187
Figure 8-4	Jython auto-completion and help	8-189
Figure 8-5	Scripts view	8-191
Figure 10-1	App Console view	10-202
Figure 10-2	ARM Asm Info view	10-204
Figure 10-3	ARM assembler editor	10-205
Figure 10-4	Breakpoints view	10-207
Figure 10-5	C/C++ editor	10-211
Figure 10-6	Show disassembly for selected source line	10-213
Figure 10-7	Commands view	10-214
Figure 10-8	Debug Control view	10-217
Figure 10-9	Disassembly view	10-221
Figure 10-10	Events view (Shown with all ports enabled for an ETB:ITM trace source)	10-225
Figure 10-11	Event Viewer Settings (Shown with all Masters and Channels enabled for an ETR:STM trace source)	10-227
Figure 10-12	Expressions view	10-229
Figure 10-13	Functions view	10-233
Figure 10-14	History view	10-235
Figure 10-15	Memory view	10-237

Figure 10-16	Memory view with Show Cache	10-239
Figure 10-17	MMU Translation tab view	10-242
Figure 10-18	MMU Tables tab view	10-243
Figure 10-19	Memory Map tab view	10-244
Figure 10-20	MMU settings	10-244
Figure 10-21	Modules view showing shared libraries	10-246
Figure 10-22	Registers view	10-248
Figure 10-23	OS Data view (showing Keil CMSIS-RTOS RTX Tasks)	10-254
Figure 10-24	Cache Data view (showing L1 TLB cache)	10-255
Figure 10-25	Screen buffer parameters	10-257
Figure 10-26	Screen view	10-258
Figure 10-27	Scripts view	10-260
Figure 10-28	Target view	10-263
Figure 10-29	Trace view with a scale of 100:1	10-265
Figure 10-30	Trace Control view	10-268
Figure 10-31	Variables view	10-271
Figure 10-32	Timed Auto-Refresh Properties dialog box	10-278
Figure 10-33	Memory Exporter dialog box	10-279
Figure 10-34	Memory Importer dialog box	10-280
Figure 10-35	Fill Memory dialog box	10-281
Figure 10-36	Export Trace Report dialog box	10-283
Figure 10-37	Breakpoint properties dialog box	10-284
Figure 10-38	Watchpoint Properties dialog box	10-287
Figure 10-39	Tracepoint Properties dialog box	10-288
Figure 10-40	Manage Signals dialog box	10-289
Figure 10-41	Manage exception handler settings	10-290
Figure 10-42	Function filter dialog box	10-291
Figure 10-43	Script Parameters dialog box	10-292
Figure 10-44	Connection tab (Shown with connection configuration for an FVP with virtual file system support enabled)	10-295
Figure 10-45	Files tab (Shown with file system configuration for an application on an FVP)	10-297
Figure 10-46	Debugger tab (Shown with settings for application starting point and search paths)	10-302
Figure 10-47	OS Awareness tab	10-303
Figure 10-48	Arguments tab	10-305
Figure 10-49	New Environment Variable dialog box	10-306
Figure 10-50	Environment tab (Shown with environment variables configured for an FVP)	10-307
Figure 10-51	DTSL Configuration Editor (Shown with Trace capture method set to DSTREAM)	10-308
Figure 10-52	Configuration Database panel	10-311
Figure 10-53	Configuration Database	10-313
Figure 10-54	Model Devices and Cluster Configuration tab	10-314
Figure 10-55	New PCE project	10-316
Figure 10-56	New configuration database	10-317
Figure 10-57	New platform information	10-317
Figure 10-58	Platform creation options	10-318
Figure 10-59	Connection address	10-319
Figure 10-60	Autodetect summary	10-319
Figure 10-61	Platform Configuration view	10-320
Figure 10-62	Component Connections	10-321
Figure 10-63	Device hierarchy	10-322
Figure 10-64	Add Core Trace	10-322

Figure 10-65	Add CTI Trigger	10-323
Figure 10-66	User added component connections	10-324
Figure 10-67	Project Explorer	10-324
Figure 10-68	Full debug and trace	10-325
Figure 10-69	Debug Activities	10-325
Figure 10-70	DTSL Options	10-326
Figure 10-71	Device hierarchy	10-326
Figure 10-72	Devices Panel	10-327
Figure 10-73	Autodetect settings	10-328
Figure 10-74	Remote Systems view	10-330
Figure 10-75	Remote System Details view	10-331
Figure 10-76	Terminal view	10-332
Figure 10-77	Remote Scratchpad	10-333
Figure 10-78	Remote Systems terminal	10-334
Figure 10-79	Terminal Settings dialog box	10-335
Figure 10-80	Debug Hardware Configure IP view	10-337
Figure 10-81	Debug Hardware Firmware Installer	10-339
Figure 10-82	Connection Browser (Showing a USB connected DSTREAM)	10-341
Figure 12-1	DS-5 File Flash Architecture	12-351
Figure 13-1	Eclipse preferences for mydb	13-374
Figure 13-2	Custom OS awareness displayed in Eclipse Debug Configurations dialog	13-376
Figure 13-3	myos No OS Support	13-379
Figure 13-4	myos waiting for target to stop	13-380
Figure 13-5	myos Enabled	13-380
Figure 13-6	myos waiting for OS initialization	13-381
Figure 13-7	myos Debug Control view data	13-383
Figure 13-8	myos Empty Tasks table	13-386
Figure 13-9	myos populated Tasks table	13-387
Figure 14-1	A simple CoreSight Design	14-391
Figure 14-2	Initial DS-5 Debugger SW Stack	14-393
Figure 14-3	Post DTSL DS-5 Debugger SW Stack	14-394
Figure 14-4	DTSL Configuration class hierarchy	14-404
Figure 14-5	DTSL Device object hierarchy	14-405
Figure 14-6	DTSL Trace Source class hierarchy	14-407
Figure 14-7	DTSL Trace Capture Objects	14-408
Figure 14-8	MEM-AP Access Ports	14-409
Figure 14-9	MEM-AP Class Hierarchy	14-410
Figure 14-10	DTSL Option Classes	14-412
Figure 14-11	DSTREAM Trace Options	14-414
Figure 14-12	Example use of CTI for H/W execution synchronization	14-421
Figure 14-13	Trace Generation	14-422
Figure 14-14	DTSL Trace Decoding Stages for DSTREAM	14-423
Figure 14-15	DTSL Trace Pipeline Hierarchy	14-424
Figure 14-16	ETB Trace Decode Pipeline Stages	14-424
Figure 14-17	Example of Multiple Trace Capture Devices	14-425
Figure 14-18	STM Object Model	14-428
Figure 14-19	Launcher panel reporting DTSL Jython script error	14-434
Figure 14-20	Connection Error Dialog	14-435

List of Tables

ARM® DS-5 Debugger User Guide

Table 2-1	ETM/PTM versions for each type of processor	2-57
Table 3-1	DMA map register SYS_DMAPSR0	3-84
Table 3-2	Control bit that remaps an area of memory	3-85
Table 7-1	Log4j Components	7-175
Table 10-1	Files tab options available for each Debug operation	10-296
Table 10-2	DS-5 Debugger icons	10-342
Table 10-3	Perspective icons	10-343
Table 10-4	View icons	10-343
Table 10-5	View markers	10-343
Table 10-6	Miscellaneous icons	10-344
Table 14-1	CTI Signal Connections	14-420

Preface

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

It contains the following:

- [About this book on page 15.](#)

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 ARM DS-5

Describes the file-based flash programming options available in DS-5.

Chapter 13 Writing OS Awareness for DS-5 Debugger

Describes the OS awareness feature available in DS-5.

Chapter 14 Debug and Trace Services Layer (DTSL)

Describes the DS-5 Debugger *Debug and Trace Services Layer* (DTSL).

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 DUI0446V.
- The page number(s) to which your comments refer.
- A concise explanation of your comments.

ARM also welcomes general suggestions for additions and improvements.

————— Note —————

ARM tests the PDF only in Adobe Acrobat and Acrobat Reader, and cannot guarantee the quality of the represented document when used with any other PDF reader.

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 sections:

- [1.1 About DS-5 Debugger on page 1-18.](#)
- [1.2 Debugger concepts on page 1-19.](#)
- [1.3 Terminology on page 1-21.](#)
- [1.4 Launching the debugger from Eclipse on page 1-22.](#)
- [1.5 About DS-5 headless command-line debugger on page 1-23.](#)
- [1.6 Headless command-line debugger options on page 1-24.](#)
- [1.7 Specifying a custom configuration database using the headless command-line debugger on page 1-29.](#)
- [1.8 Capturing trace using the headless debugger on page 1-31.](#)
- [1.9 DS-5 Debug perspective keyboard shortcuts on page 1-33.](#)
- [1.10 DS-5 Debugger command-line console keyboard shortcuts on page 1-34.](#)
- [1.11 Standards compliance in DS-5 Debugger on page 1-35.](#)

1.1 About DS-5 Debugger

DS-5 Debugger is a graphical debugger supporting end-to-end software development on ARM® processor-based targets and *Fixed Virtual Platform* (FVP) targets.

Using DS-5 Debugger, you can have complete control over the flow of the execution to get to the root of software bugs throughout your development process. It makes it easy to debug bare-metal, Linux, and Android applications with comprehensive and intuitive views, including synchronized source and disassembly, call stack, memory, registers, expressions, variables, threads, breakpoints, and trace.

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 2-37.](#)

[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 some of the concepts that you need to be aware of when working with DS-5 Debugger.

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

A debug target is an environment where your program can be executed and debugged. This environment can be hardware, software that simulates hardware, or a hardware emulator.

A hardware target can be anything from a mass-produced development board or electronic equipment to a prototype product, or a printed circuit board.

During the early stages of product development, if no hardware is available, a simulation or software target might be used to simulate hardware behavior. *Fixed Virtual Platform* (FVP) are software models from ARM that provide functional behavior equivalent to real hardware.

Note

Even though you might run an FVP on the same host as the debugger, it is useful to think of it as a separate piece of hardware.

Also, during the early stages of product development, hardware emulators are used to verify hardware and software designs for pre-silicon testing.

Debug adapter

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

Debug adapters vary in complexity and capability but, combined with software debug agents, they provide high-level debug functionality for the target that is being debugged, for example:

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

Note

A debug adapter or connection is not the application being debugged, nor the debugger itself.

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 `cdbimporter` tool.

Contexts

Each processor in the target can run more than one process. However, the processor only executes one process at any given time. 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).

Debug and Trace Services Layer (DTSL)

DTSL 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. ARM has made DTSL available for your own use so that you can create programs (Java or Jython) to access/control the target platform.

Related concepts

[1.5 About DS-5 headless command-line debugger on page 1-23.](#)

Related tasks

[2.7.1 Connecting to an existing application and application rewind session on page 2-47.](#)

[2.7.2 Downloading your application and application rewind server on the target system on page 2-48.](#)

[2.7.3 Starting the application rewind server and debugging the target-resident application on page 2-50.](#)

[2.4 Configuring a connection to a Fixed Virtual Platform \(FVP\) for Linux application debug on page 2-41.](#)

[2.3 Configuring a connection to a Linux target using gdbserver on page 2-39.](#)

[2.6 Configuring a connection to a Linux Kernel on page 2-45.](#)

[2.8 Configuring a connection to a bare-metal target on page 2-52.](#)

[2.9 Configuring an Event Viewer connection to a bare-metal target on page 2-54.](#)

[1.7 Specifying a custom configuration database using the headless command-line debugger on page 1-29.](#)

Related information

[Debug options supported by DS-5.](#)

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

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

1.3 Terminology

This document uses the following terminology:

Bare metal

A bare metal embedded application is one which does not run on an OS.

BBB

The old name for the MTB.

CADI

Cycle Accurate Debug Interface. This is the API used by debuggers to control models.

DS-5

The ARM Eclipse based development environment, including an IDE, a compiler, a debugger and a profiler.

DSTREAM

ARM DSTREAM combined debug and trace unit.

ETB

Embedded Trace Buffer. This is a private CoreSight memory component which stores trace data. Columbus does not have one of these by default.

ETM

Embedded Trace Macrocell. This is a CoreSight component which is paired with a core to deliver instruction flow trace and, optionally, address and data trace.

ETR

Embedded Trace Router. This is a CoreSight component which routes trace data to system memory or other trace sinks, such as HSSTP.

ITM

Instruction Trace Macrocell. This is a CoreSight component which delivers code instrumentation output and specific hardware data streams. It is a predecessor to the STM.

jRDDI

The Java API implementation of RDDI.

Jython

An implementation of the Python language which is closely integrated with Java.

MTB

Micro Trace Buffer. This is used in the Cortex-M0 and Cortex-M0+.

PTM

Program Trace Macrocell. This is a CoreSight component which is paired with a core to deliver instruction only program flow trace data.

RDDI

RealView Device Debug Interface. This is a C-level API which allows access to target debug and trace functionality, typically through an RVI, a DSTREAM box or a CADI model.

RTSM

Real Time System Model.

RVI

ARM Realview ICE unit.

STM

System Trace Macrocell. This is a CoreSight component which delivers code instrumentation output and other hardware generated data streams.

TPIU

Trace Port Interface Unit. This is a CoreSight component which delivers trace data to an external trace capture device.

For more help with terms and abbreviations, see the [ARM® Glossary](#)

1.4 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 `ecclipse` 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

- [1.6 Headless command-line debugger options on page 1-24.](#)
- [2.7.1 Connecting to an existing application and application rewind session on page 2-47.](#)
- [2.7.2 Downloading your application and application rewind server on the target system on page 2-48.](#)
- [2.7.3 Starting the application rewind server and debugging the target-resident application on page 2-50.](#)
- [2.4 Configuring a connection to a Fixed Virtual Platform \(FVP\) for Linux application debug on page 2-41.](#)
- [2.3 Configuring a connection to a Linux target using gdbserver on page 2-39.](#)
- [2.6 Configuring a connection to a Linux Kernel on page 2-45.](#)
- [2.8 Configuring a connection to a bare-metal target on page 2-52.](#)
- [2.9 Configuring an Event Viewer connection to a bare-metal target on page 2-54.](#)

Related references

- [2.1 Types of target connections on page 2-37.](#)
- [10.38 Debug Configurations - Connection tab on page 10-293.](#)
- [10.39 Debug Configurations - Files tab on page 10-296.](#)
- [10.40 Debug Configurations - Debugger tab on page 10-300.](#)
- [10.42 Debug Configurations - Arguments tab on page 10-304.](#)
- [10.43 Debug Configurations - Environment tab on page 10-306.](#)
- [Chapter 10 DS-5 Debug Perspectives and Views on page 10-200.](#)
- [10.7 Debug Control view on page 10-217.](#)

1.5 About DS-5 headless command-line debugger

DS-5 Debugger can operate in a headless command-line mode. This means that you can use the debugger in a command-line only mode, with no graphical user interface.

The headless command-line debugger is useful for automated testing and other repetitive tasks.

You launch DS-5 headless command-line debugger using the DS-5 Command Prompt or Unix shell with the required environment variables set.

When you launch the debugger in headless mode, you must provide command-line arguments to the debugger to select which target to connect to. The debugger's configuration database contains a list of targets that the debugger can connect to. After connecting to the target, you can load an image, run, stop, and debug the application by using the DS-5 debugger commands. You can use any DS-5 debugger command in the headless debugger mode.

Related tasks

[1.7 Specifying a custom configuration database using the headless command-line debugger on page 1-29.](#)

Related references

[1.6 Headless command-line debugger options on page 1-24.](#)

Related information

[DS-5 Debugger commands.](#)

1.6 Headless command-line debugger options

You can use the 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 or to connect to a target.

...

Additional 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

--browse

Browses for available connections and lists targets that match the connection type specified in the configuration database entry.

Note

You must specify `--cdb-entry arg` to use `--browse`.

--cdb-entry arg

Specifies a target from the configuration database that the debugger can connect to.

Use *arg* to specify the target configuration. *arg* is a string concatenated using entries in each level of the configuration database. The syntax of *arg* is:

"Manufacturer::Platform::Project type::Execution

environment::Activity::Connection type". Use `--cdb-list` to determine the entries in the configuration database that the debugger can connect to.

--cdb-entry-param arg

Specifies connection parameters for the debugger.

Use *arg* to specify the parameters and their values. The syntax for *arg* is comma separated pairs of parameters and values: *"param1=value1"*. Use `--cdb-list` to identify what parameters the debugger needs. Parameters that the debugger might need are:

Connection

Specifies the name of the target to connect to.

Address

Specifies the address for a gdbserver connection.

Port

Specifies the port for a gdbserver connection.

dtssl_options_file

Specifies a file containing the DTSL options.

Model parameters

Specifies parameters for a model connection. The model parameters depend on the specific model that the debugger connects to. See the documentation on the model for the parameters and how to configure them. The debugger uses the default model parameter values if you do not specify them.

Use `--cdb-entry-param` for each parameter. For example `--cdb-entry-param`
`"Connection=TestTarget" --cdb-entry-param`
`"dtls_options_file=my_dtls_settings.dtlsprops"`

`--cdb-list filter`

Lists the entries in the configuration database. This option does not connect to any target. The configuration database has a tree data structure, where each entry has entries within it. `--cdb-list` identifies the entries in each level of the database. The levels are:

1. Manufacturer
2. Platform
3. Project type
4. Execution environment
5. Activity
6. Connection type.

Use *filter* to specify the entries in each level, to identify the target and how to connect to it. *filter* is a string concatenated using entries in successive levels of the configuration database. The full syntax of *filter* is: "*Manufacturer::Platform::Project type::Execution environment::Activity::Connection type*".

If you specify an incomplete *filter*, then `--cdb-list` shows the entries in the next level of the configuration database. So if you do not specify a *filter*, `--cdb-list` shows the *Manufacturer* entries from the first level of the configuration database. If you specify a *filter* using entries from the first and second levels of the database, then `--cdb-list` shows the *Project type* entries within the specified *Platform*. If you specify the complete *filter* then `--cdb-list` lists the parameters that need to be specified using `--cdb-list-param`.

————— **Note** —————

The entries in the configuration database are case-sensitive.

————— **Note** —————

Connection type refers to DSTREAM or RVI, so there is no *Connection type* when connecting to a model.

To list all first level entries in the configuration database, use:

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

To list all the configuration database entries within the manufacturer Altera, use:

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

`--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`.
- `--disable-semihosting`
Disables semihosting operations.
- `--disable_semihosting_console`
Disables all semihosting operations to the debugger console.
- `--enable-semihosting`
Enables semihosting operations.
- `-h` or `--help`
Displays a summary of the main command-line options.
- `-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.
- `-e arg` or `--semihosting-error arg`
Specifies a file to write semihosting stderr.
- `-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.
- `--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`.
- `--top_mem=address`
Specifies the stack base, also known as the top of memory. Top of memory is only used for semihosting operations.
- `--target-os=name`
Specifies the operating system on the target. Use this option if you want to debug the operating system on the target.
- `--target-os-list`
Lists the operating systems that DS-5 Debugger can debug.

Note

Specifying the `--cdb-entry` option is sufficient to establish a connection to a model. However to establish a connection in all other cases, for example when using DSTREAM for Linux application debug, you must specify both `--cdb-entry` and `--cdb-entry-param` options.

You must normally specify `--cdb-entry` when invoking the debugger for all other options to be valid. The exception to this are:

- `--cdb-list` and `--help` do not require `--cdb-entry`.
- `--cdb-root` can be specified with either `--cdb-list` or `--cdb-entry`.

Example 1-1 Examples

To connect to an ARM FVP Cortex-A8 model, first use `--cdb-list` to identify the entries in the configuration database within ARM FVP, then use:

```
debugger --cdb-entry "ARM FVP::Cortex-A8::Bare Metal Debug::Bare Metal Debug::Debug Cortex-A8"
```

To connect to an ARM FVP Cortex-A9x4 model and specify an image to load, first use `--cdb-list` to identify the entries in the configuration database within ARM FVP, then use:

```
debugger --cdb-entry "ARM FVP::VE_Cortex_A9x4::Bare Metal Debug::Bare Metal SMP Debug::Debug Cortex-A9x4 SMP" --image "C:\DS-5_Workspace\fireworks_A9x4-FVP\fireworks-Cortex-A9x4-FVP.axf"
```

To connect to a Pandaboard target, first use `--cdb-list` to identify the entries in the configuration database within pandaboard.org, then use:

```
debugger --cdb-entry "pandaboard.org::OMAP 4430::Bare Metal Debug::Bare Metal Debug::Debug Cortex-A9x2 SMP::RealView ICE" --cdb-entry-param "Connection=TCP:TestFarm-Panda-A9x2"
```

To connect to a Versatile Express A9x4 board using USB, first use `--cdb-list` to identify the entries in the configuration database within ARM Development Boards, then use:

```
debugger --cdb-entry "ARM Development Boards::Versatile Express A9x4::Bare Metal Debug::Bare Metal SMP Debug of all cores::Debug Cortex-A9x4 SMP::DSTREAM" --cdb-entry-param "Connection=USB:000271"
```

To connect to a Versatile Express A9x4 board using USB, first use `--cdb-list` to identify the entries in the configuration database within ARM Development Boards. If the operating system on the target is RTX, then to debug RTX on the target, use:

```
debugger --cdb-entry "ARM Development Boards::Versatile Express A9x4::Bare Metal Debug::Bare Metal SMP Debug of all cores::Debug Cortex-A9x4 SMP::DSTREAM" --cdb-entry-param "Connection=USB:000271" --target-os rtx
```

To connect and debug a Linux application on a Beagleboard target that is already running `gdbserver`, first use `--cdb-list` to identify the entries in the configuration database within beagleboard.org, then use:

```
debugger --cdb-entry "beagleboard.org::OMAP 3530::Linux Application Debug::gdbserver (TCP)::Connect to already running gdbserver" --cdb-entry-param "Address=TCP:TestFarm-Beagle-A8-Android" --cdb-entry-param "Port=5350"
```

To connect and debug a Linux kernel on a Beagleboard target, first use `--cdb-list` to identify the entries in the configuration database within beagleboard.org, then use:

```
debugger --cdb-entry "beagleboard.org::OMAP 3530::Linux Kernel and/or Device Driver Debug::Linux Kernel Debug::Debug Cortex-A8::DSTREAM" --cdb-entry-param "Connection=TCP:TestFarm-Beagle-A8-Android"
```

Note

Once a debugger connection has been established, type `quit` when you want to exit the connection.

Related concepts

[7.6 About Log4j configuration files](#) on page 7-175.

Related tasks

[1.4 Launching the debugger from Eclipse](#) on page 1-22.

Related references

[1.10 DS-5 Debugger command-line console keyboard shortcuts](#) on page 1-34.

[8.1 Exporting DS-5 Debugger commands generated during a debug session](#) on page 8-180.

[7.1 About semihosting and top of memory](#) on page 7-169.

Related information

[Log4j in Apache Logging Services.](#)

[DS-5 Debugger commands.](#)

1.7 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. To list the entries from the user specified configuration databases, use the following command-line syntax:

`debugger --cdb-list --cdb-root path_to_cdb1[;path_to_cdb2]` on Windows

`debugger --cdb-list --cdb-root path_to_cdb1[:path_to_cdb2]` on Linux

Where:

`--cdb-list`

Is the option that lists the entries from the configuration databases.

`--cdb-root`

Is the option that specifies the path to one or more configuration databases.

path_to_cdb1 and *path_to_cdb2*

Are directory paths to the configuration databases.

————— **Note** —————

DS-5 processes the databases from left to right. The information in the already processed databases are replaced with information from the databases that are processed later. 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 at the right end of the list on the command-line.

————— **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. To connect to a target in a user specified configuration database, use the following command-line syntax:

`debugger --cdb-entry "Manufacturer::Platform::Project type::Execution environment::Activity::Connection type" --cdb-root path_to_cdb1`

Where:

`--cdb-entry`

Specifies the target to connect to.

Manufacturer::Platform::Project type::Execution environment::Activity::Connection type correspond to the entries in the user specified configuration database.

Note

- To specify more than one configuration database, you must separate the directory paths using a colon or semicolon for a Linux or Windows system respectively.
 - If connection parameters are required, specify them using the `--cdb-entry-param` option.
-

Related concepts

[1.5 About DS-5 headless command-line debugger](#) on page 1-23.

Related references

[1.6 Headless command-line debugger options](#) on page 1-24.

1.8 Capturing trace using the headless debugger

To capture trace using the headless debugger you must enable trace when invoking the headless debugger.

To enable trace when invoking the headless debugger, you must specify a file containing the Debug and Trace Services Layer (DTSL) options. This task shows you how to specify DTSL options to the headless debugger. You can then run a script containing commands to start and stop trace capture. For this task, it is useful to setup the DTSL options using the graphical interface of DS-5 Debugger. When you setup the DTSL options, the debugger creates a file contain the DTSL options. You can then specify this file when invoking the headless debugger.

Procedure

1. Using the graphical interface of DS-5 Debugger, open the **Debug Configurations** dialog for your trace capable target.
2. In the **Connections** tab, click DTSL options **Edit** to open the **DTSL Configuration Editor** dialog. Select a trace capture method and enable core trace for the required processors. Apply the DTSL and trace settings. These settings are stored in a *.dtslprops file in your workspace. The filename is shown in the **Name of configuration** field. In this example, the settings are stored in default.dtslprops.

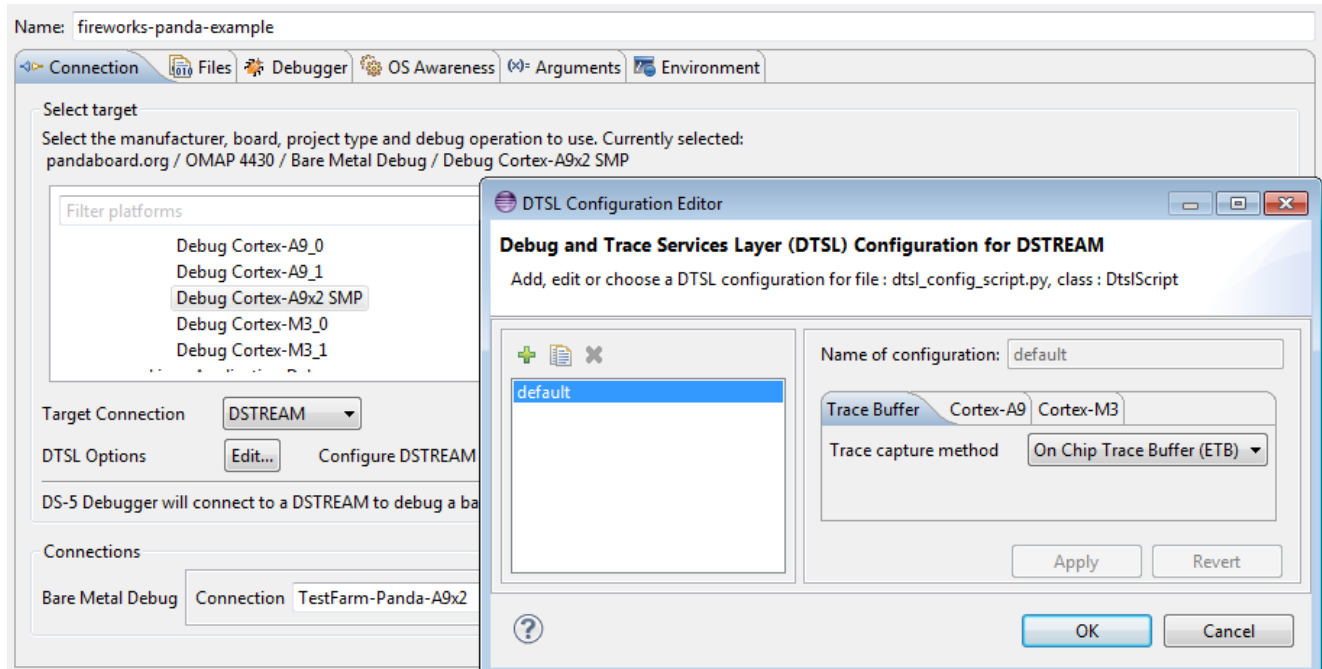


Figure 1-1 Enable trace in the DTSL options

3. Copy the DTSL settings file, for example default.dtslprops, from the workspace, for example from C:\DS-5_Workspace\metadata\plugins\com.arm.ds\DTSL\pandaboard.org++OMAP+4430\pandaboard\OMAP44xx, to a different directory and change its name, for example to C:\Headless\my_dtsl_settings.dtslprops.
4. Open the DS-5 command prompt for headless debugging.
5. Use the --cdb-list argument to identify your target configuration. Then invoke the headless debugger with the target configuration and target name. When invoking the debugger, also specify the DTSL options file using --cdb-entry-params. For example: `debugger --cdb-entry "pandaboard.org::OMAP 4430::Bare Metal Debug::Bare Metal Debug::Debug Cortex-A9x2 SMP::RealView ICE" --cdb-entry-param "Connection=TestFarm-Panda-A9x2" --cdb-entry-param "dtsl_options_file=C:\Headless\my_dtsl_settings.dtslprops"`

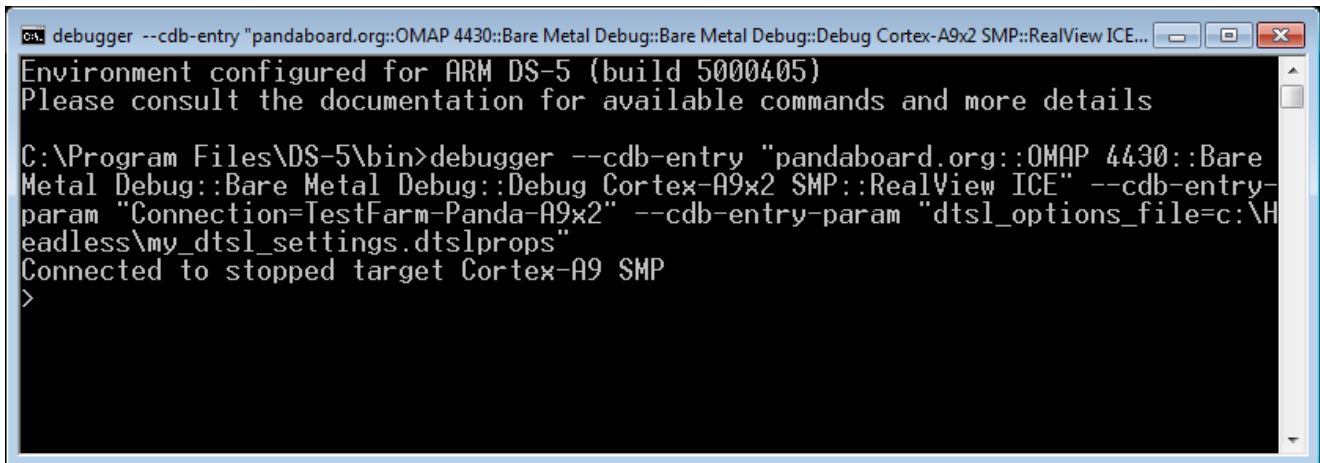


Figure 1-2 Headless debugger connection with DTSL option

6. The headless debugger connects to the target. You can now issue commands to load and run an image, and also to start and stop the trace capture. If you write a script file containing the trace capture commands, you can specify this script file when invoking the headless debugger. For example: `debugger --cdb-entry "pandaboard.org::OMAP 4430::Bare Metal Debug::Bare Metal Debug::Debug Cortex-A9x2 SMP::RealView ICE" --cdb-entry-param "Connection=TestFarm-Panda-A9x2" --cdb-entry-param "dtls_options_file=C:\Headless\my_dtls_settings.dtlsprops" --script=C:\Headless\my_script.txt`. An example script file might contain the following commands:

```
loadfile C:\DS-5_Workspace\fireworks_panda\fireworks_panda.axf # Load an image to debug
start # Start running the image after setting a temporary
breakpoint
wait # Wait for a breakpoint
trace start # Start the trace capture when the breakpoint is hit
advance plot3 # Set a temporary breakpoint at symbol plot3
wait # Wait for a breakpoint
trace stop # Stop the trace when the breakpoint at plot3 is hit
trace report FILE=report.txt # Write the trace output to report.txt
quit # Exit the headless debugging session
```

Related concepts

[1.5 About DS-5 headless command-line debugger on page 1-23.](#)

Related references

[1.6 Headless command-line debugger options on page 1-24.](#)

[10.38 Debug Configurations - Connection tab on page 10-293.](#)

[10.44 DTSL Configuration Editor dialog box on page 10-308.](#)

Related information

[DS-5 Debugger Tracing Commands.](#)

1.9 DS-5 Debug perspective keyboard shortcuts

You can use various keyboard shortcuts 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 level including stepping into all function calls where there is debug information.

ALT+F5

Step at instruction level including stepping into all function calls where there is debug information.

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

[1.4 Launching the debugger from Eclipse on page 1-22.](#)

Related references

[10.6 Commands view on page 10-214.](#)

[10.7 Debug Control view on page 10-217.](#)

1.10 DS-5 Debugger command-line console keyboard shortcuts

DS-5 Debugger provides editing features, a command history, and common keyboard shortcuts for use on the command-line console.

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, you can use the following keyboard shortcuts:

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

[1.6 Headless command-line debugger options on page 1-24.](#)

1.11 Standards compliance in DS-5 Debugger

There are different levels 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 sections:

- [2.1 Types of target connections](#) on page 2-37.
- [2.2 About Fixed Virtual Platform \(FVP\) entries in the configuration database](#) on page 2-38.
- [2.3 Configuring a connection to a Linux target using gdbserver](#) on page 2-39.
- [2.4 Configuring a connection to a Fixed Virtual Platform \(FVP\) for Linux application debug](#) on page 2-41.
- [2.5 Adding a custom model](#) on page 2-43.
- [2.6 Configuring a connection to a Linux Kernel](#) on page 2-45.
- [2.7 About configuring connections to a Linux target using Application Debug with Rewind Support](#) on page 2-47.
- [2.8 Configuring a connection to a bare-metal target](#) on page 2-52.
- [2.9 Configuring an Event Viewer connection to a bare-metal target](#) on page 2-54.
- [2.10 About the target configuration import utility](#) on page 2-56.
- [2.11 Adding a new platform](#) on page 2-58.
- [2.12 Adding a new configuration database to DS-5](#) on page 2-60.
- [2.13 Exporting an existing launch configuration](#) on page 2-62.
- [2.14 Importing an existing launch configuration](#) on page 2-65.
- [2.15 Disconnecting from a target](#) on page 2-67.

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\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.
- A CADI-compliant connection between the debugger and a model.

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 9-194.

[6.13 About application rewind](#) on page 6-160.

Related tasks

[2.7.1 Connecting to an existing application and application rewind session](#) on page 2-47.

[2.7.2 Downloading your application and application rewind server on the target system](#) on page 2-48.

[2.7.3 Starting the application rewind server and debugging the target-resident application](#) on page 2-50.

2.2 About *Fixed Virtual Platform (FVP)* entries in the configuration database

The configuration database installed with DS-5 Debugger contains configuration entries for various targets and several ARM FVPs.

The entries in the configuration database provide information about the targets and models so that DS-5 Debugger can connect to them. Though the DS-5 configuration database contains entries for several ARM FVPs, the FVPs are not provided with DS-5 Debugger. To be able to connect to an FVP, you must obtain and install the FVP separately. When installed, ensure that the `PATH` environment variable for your OS contains the path to the installed FVP. DS-5 debugger can launch and connect to the installed FVP if it has an entry in the DS-5 configuration database.

If you have a model that does not have an entry in the DS-5 configuration database, you must create an entry before DS-5 Debugger can connect to it.

Related tasks

[2.4 Configuring a connection to a Fixed Virtual Platform \(FVP\) for Linux application debug](#) on page 2-41.

[2.5 Adding a custom model](#) on page 2-43.

[2.12 Adding a new configuration database to DS-5](#) on page 2-60.

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

- [2.13 Exporting an existing launch configuration on page 2-62.](#)
- [2.14 Importing an existing launch configuration on page 2-65.](#)
- [2.4 Configuring a connection to a Fixed Virtual Platform \(FVP\) for Linux application debug on page 2-41.](#)
- [2.6 Configuring a connection to a Linux Kernel on page 2-45.](#)
- [2.8 Configuring a connection to a bare-metal target on page 2-52.](#)
- [2.9 Configuring an Event Viewer connection to a bare-metal target on page 2-54.](#)

Related references

- [10.38 Debug Configurations - Connection tab on page 10-293.](#)
- [10.39 Debug Configurations - Files tab on page 10-296.](#)
- [10.40 Debug Configurations - Debugger tab on page 10-300.](#)
- [10.41 Debug Configurations - OS Awareness tab on page 10-303.](#)
- [10.42 Debug Configurations - Arguments tab on page 10-304.](#)
- [10.43 Debug Configurations - Environment tab on page 10-306.](#)
- [10.50 Target management terminal for serial and SSH connections on page 10-332.](#)
- [11.1 ARM Linux problems and solutions on page 11-346.](#)
- [11.3 Target connection problems and solutions on page 11-348.](#)

2.4 Configuring a connection to a *Fixed Virtual Platform (FVP)* for Linux application debug

DS-5 supports connecting to an FVP. Use the described options to configure your connection to the Cortex®-A8 (preconfigured to boot ARM embedded Linux) FVP supplied with DS-5.

Procedure

1. From the main menu, select **Window > Open Perspective > DS-5 Debug**.
2. Select **Debug Configurations...** from the **Run** menu.
3. Select **DS-5 Debugger** from the configuration tree and then click **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 **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

- [2.13 Exporting an existing launch configuration on page 2-62.](#)
- [2.14 Importing an existing launch configuration on page 2-65.](#)
- [2.3 Configuring a connection to a Linux target using gdbserver on page 2-39.](#)
- [2.6 Configuring a connection to a Linux Kernel on page 2-45.](#)
- [2.8 Configuring a connection to a bare-metal target on page 2-52.](#)
- [2.9 Configuring an Event Viewer connection to a bare-metal target on page 2-54.](#)

Related references

- [10.38 Debug Configurations - Connection tab on page 10-293.](#)
- [10.39 Debug Configurations - Files tab on page 10-296.](#)
- [10.40 Debug Configurations - Debugger tab on page 10-300.](#)
- [10.41 Debug Configurations - OS Awareness tab on page 10-303.](#)
- [10.42 Debug Configurations - Arguments tab on page 10-304.](#)
- [10.43 Debug Configurations - Environment tab on page 10-306.](#)

Related information

[Model Shell options for Fast Models.](#)

2.5 Adding a custom model

If you have built your own custom CADI-compliant model, to connect to it using DS-5, you have to create a new entry in the DS-5 configuration database. Depending on how the model is defined in the configuration database, DS-5 can either launch the model or can connect to an already running model.

Note

ARM FVPs not provided with DS-5 installation should be defined in the PATH environment variable of your OS to be available for DS-5.

Add the <DS-5 install directory>/bin directory to your PATH environment variable and restart Eclipse.

- For Windows, enter `set PATH=<your model path>\bin;%PATH%`
- For Linux, enter `export PATH=<your model path>/bin:$PATH`

To make the change permanent, so that the modified path is available in future sessions:

- For Windows, right-click **My Computer > Properties > Advanced system settings > Environment Variables** and under **User Variables**, create a PATH variable with the value <your model path> \bin, or else append ;<your model path>\bin to any existing PATH variable.
- For Linux, set up the PATH in the appropriate shell configuration file. For example, in .bashrc, add the line `export PATH=<your model path>/bin:$PATH`

For the models you build yourself, follow the instructions as described in this topic.

Procedure

1. Launch your model and start the CADI server.
 - If your model is a library file:
 - On Windows, select **Start > All Programs > ARM DS-5 > DS-5 Command Prompt** and enter `model_shell -m <your model path and name> -S`.
 - On Linux, open a new terminal and run: `<DS-5 installation folder>/bin/model_shell -m <your model path and name> -S`
 - If your model is an executable file, at the command prompt, enter <your model path and name> -S.

Note

For more information about options available with the `model_shell` utility in DS-5, enter `model_shell --help` at the DS-5 command prompt.

2. Launch the `cdbimporter` utility and use it to create a new entry in the configuration database.
 - a. On Windows, at the DS-5 command prompt, enter `cdbimporter -m` to launch the `cdbimporter` utility and import the model.
 - b. Enter the **DS-5 source configuration path**. If accepting the default source path, press **Enter** on your keyboard.
 - c. Enter the **DS-5 destination configuration path**. If accepting the default configuration path, press **Enter** on your keyboard.
 - d. If there is more than one active CADI-compliant model simulation, DS-5 lists the available connections. Select the one you want to connect to. If accepting the default model connection, press **Enter** on your keyboard.
 - e. If needed, select a core to modify. If accepting the default, press **Enter** on your keyboard.
 - f. If needed, enter the name of the **Platform Manufacturer**. If accepting the default, press **Enter** on your keyboard.

Note

If you do not enter a name for the platform manufacturer, you can find the platform you added listed under **Imported** in the list of platforms in the Debug Configurations dialog.

- g. Enter the name of the **Platform**. If accepting the default, press **Enter** on your keyboard.
 - h. If you want to enable DS-5 to launch the model when starting a debug session, enter the path to your model. If you plan to launch the model separately, enter n at the prompt.
3. Launch DS-5 and rebuild the debugger database:
 - From the DS-5 menu, select **Window > Preferences > DS-5 > Configuration Database** and click **Rebuild database**.

Your model is now available as one of the targets in the configuration database. Use the Debug Configurations dialog to create, manage, and run configurations for this target.

Related concepts

[2.1 Types of target connections on page 2-37.](#)

[2.10 About the target configuration import utility on page 2-56.](#)

Related tasks

[2.4 Configuring a connection to a Fixed Virtual Platform \(FVP\) for Linux application debug on page 2-41.](#)

[2.3 Configuring a connection to a Linux target using gdbserver on page 2-39.](#)

[2.6 Configuring a connection to a Linux Kernel on page 2-45.](#)

[2.8 Configuring a connection to a bare-metal target on page 2-52.](#)

[1.7 Specifying a custom configuration database using the headless command-line debugger on page 1-29.](#)

Related information

[Component Architecture Debug Interface Developer Guide.](#)

2.6 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.

Prerequisites

Before connecting, you must:

- Obtain the target IP address or name for the connection between the debugger and the debug hardware adapter.
- If required, set up a Remote Systems Explorer (RSE) connection to the target.

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.

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 debug configuration.
4. In the **Name** field, enter a suitable name for the new debug 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, press **F1** on your keyboard to display dynamic context help.

Related concepts

- [6.10.2 About debugging a Linux kernel on page 6-153.](#)
- [6.10.3 About debugging Linux kernel modules on page 6-154.](#)
- [6.7 About debugging bare-metal symmetric multiprocessing systems on page 6-147.](#)

Related tasks

- [2.13 Exporting an existing launch configuration on page 2-62.](#)
- [2.14 Importing an existing launch configuration on page 2-65.](#)
- [2.4 Configuring a connection to a Fixed Virtual Platform \(FVP\) for Linux application debug on page 2-41.](#)
- [2.3 Configuring a connection to a Linux target using gdbserver on page 2-39.](#)
- [2.8 Configuring a connection to a bare-metal target on page 2-52.](#)
- [2.9 Configuring an Event Viewer connection to a bare-metal target on page 2-54.](#)

Related references

- [10.38 Debug Configurations - Connection tab on page 10-293.](#)
- [10.39 Debug Configurations - Files tab on page 10-296.](#)
- [10.40 Debug Configurations - Debugger tab on page 10-300.](#)
- [10.41 Debug Configurations - OS Awareness tab on page 10-303.](#)
- [10.42 Debug Configurations - Arguments tab on page 10-304.](#)
- [10.43 Debug Configurations - Environment tab on page 10-306.](#)
- [10.50 Target management terminal for serial and SSH connections on page 10-332.](#)
- [11.1 ARM Linux problems and solutions on page 11-346.](#)
- [11.3 Target connection problems and solutions on page 11-348.](#)

Related information

- [Debugging a loadable kernel module.](#)

2.7 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.

This section contains the following subsections:

- [2.7.1 Connecting to an existing application and application rewind session on page 2-47.](#)
- [2.7.2 Downloading your application and application rewind server on the target system on page 2-48.](#)
- [2.7.3 Starting the application rewind server and debugging the target-resident application on page 2-50.](#)

2.7.1 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\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

[6.13 About application rewind on page 6-160.](#)

Related references

- [10.38 Debug Configurations - Connection tab on page 10-293.](#)
- [10.39 Debug Configurations - Files tab on page 10-296.](#)
- [10.40 Debug Configurations - Debugger tab on page 10-300.](#)
- [10.41 Debug Configurations - OS Awareness tab on page 10-303.](#)
- [10.42 Debug Configurations - Arguments tab on page 10-304.](#)
- [10.43 Debug Configurations - Environment tab on page 10-306.](#)
- [2.1 Types of target connections on page 2-37.](#)

2.7.2 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

[6.13 About application rewind on page 6-160.](#)

Related references

[10.38 Debug Configurations - Connection tab on page 10-293.](#)

[10.39 Debug Configurations - Files tab on page 10-296.](#)

[10.40 Debug Configurations - Debugger tab on page 10-300.](#)

[10.41 Debug Configurations - OS Awareness tab on page 10-303.](#)

[10.42 Debug Configurations - Arguments tab on page 10-304.](#)

[10.43 Debug Configurations - Environment tab on page 10-306.](#)

[2.1 Types of target connections on page 2-37.](#)

2.7.3 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

[6.13 About application rewind on page 6-160.](#)

Related references

- [10.38 Debug Configurations - Connection tab on page 10-293.](#)
- [10.39 Debug Configurations - Files tab on page 10-296.](#)
- [10.40 Debug Configurations - Debugger tab on page 10-300.](#)
- [10.41 Debug Configurations - OS Awareness tab on page 10-303.](#)
- [10.42 Debug Configurations - Arguments tab on page 10-304.](#)
- [10.43 Debug Configurations - Environment tab on page 10-306.](#)
- [2.1 Types of target connections on page 2-37.](#)

2.8 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

- [2.13 Exporting an existing launch configuration on page 2-62.](#)
- [2.14 Importing an existing launch configuration on page 2-65.](#)
- [2.4 Configuring a connection to a Fixed Virtual Platform \(FVP\) for Linux application debug on page 2-41.](#)
- [2.3 Configuring a connection to a Linux target using gdbserver on page 2-39.](#)
- [2.6 Configuring a connection to a Linux Kernel on page 2-45.](#)
- [2.9 Configuring an Event Viewer connection to a bare-metal target on page 2-54.](#)

Related references

- [10.38 Debug Configurations - Connection tab on page 10-293.](#)

- 10.39 Debug Configurations - Files tab on page 10-296.*
- 10.40 Debug Configurations - Debugger tab on page 10-300.*
- 10.41 Debug Configurations - OS Awareness tab on page 10-303.*
- 10.42 Debug Configurations - Arguments tab on page 10-304.*
- 10.43 Debug Configurations - Environment tab on page 10-306.*

2.9 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

- [2.13 Exporting an existing launch configuration on page 2-62.](#)
- [2.14 Importing an existing launch configuration on page 2-65.](#)
- [2.4 Configuring a connection to a Fixed Virtual Platform \(FVP\) for Linux application debug on page 2-41.](#)
- [2.3 Configuring a connection to a Linux target using gdbserver on page 2-39.](#)
- [2.6 Configuring a connection to a Linux Kernel on page 2-45.](#)
- [2.8 Configuring a connection to a bare-metal target on page 2-52.](#)

Related references

- [10.38 Debug Configurations - Connection tab on page 10-293.](#)
- [10.39 Debug Configurations - Files tab on page 10-296.](#)
- [10.40 Debug Configurations - Debugger tab on page 10-300.](#)
- [10.41 Debug Configurations - OS Awareness tab on page 10-303.](#)
- [10.42 Debug Configurations - Arguments tab on page 10-304.](#)
- [10.43 Debug Configurations - Environment tab on page 10-306.](#)

2.10 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.
- big.LITTLE™ configurations for cores that support big.LITTLE operation, such as Cortex-A15/Cortex-A7.

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 (DSTREAMtrace 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.

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, at the DS-5 Command Prompt, enter either `cdbimporter --list-cores` or `cdbimporter -l`.

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

Related tasks

[2.11 Adding a new platform on page 2-58.](#)

[2.12 Adding a new configuration database to DS-5 on page 2-60.](#)

Related references

[10.45 Configuration Database panel on page 10-310.](#)

2.11 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 *DS-5 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 (.rvc). 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

[2.10 About the target configuration import utility on page 2-56.](#)

Related tasks

[2.12 Adding a new configuration database to DS-5 on page 2-60.](#)

Related references

[10.45 Configuration Database panel on page 10-310.](#)

2.12 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.

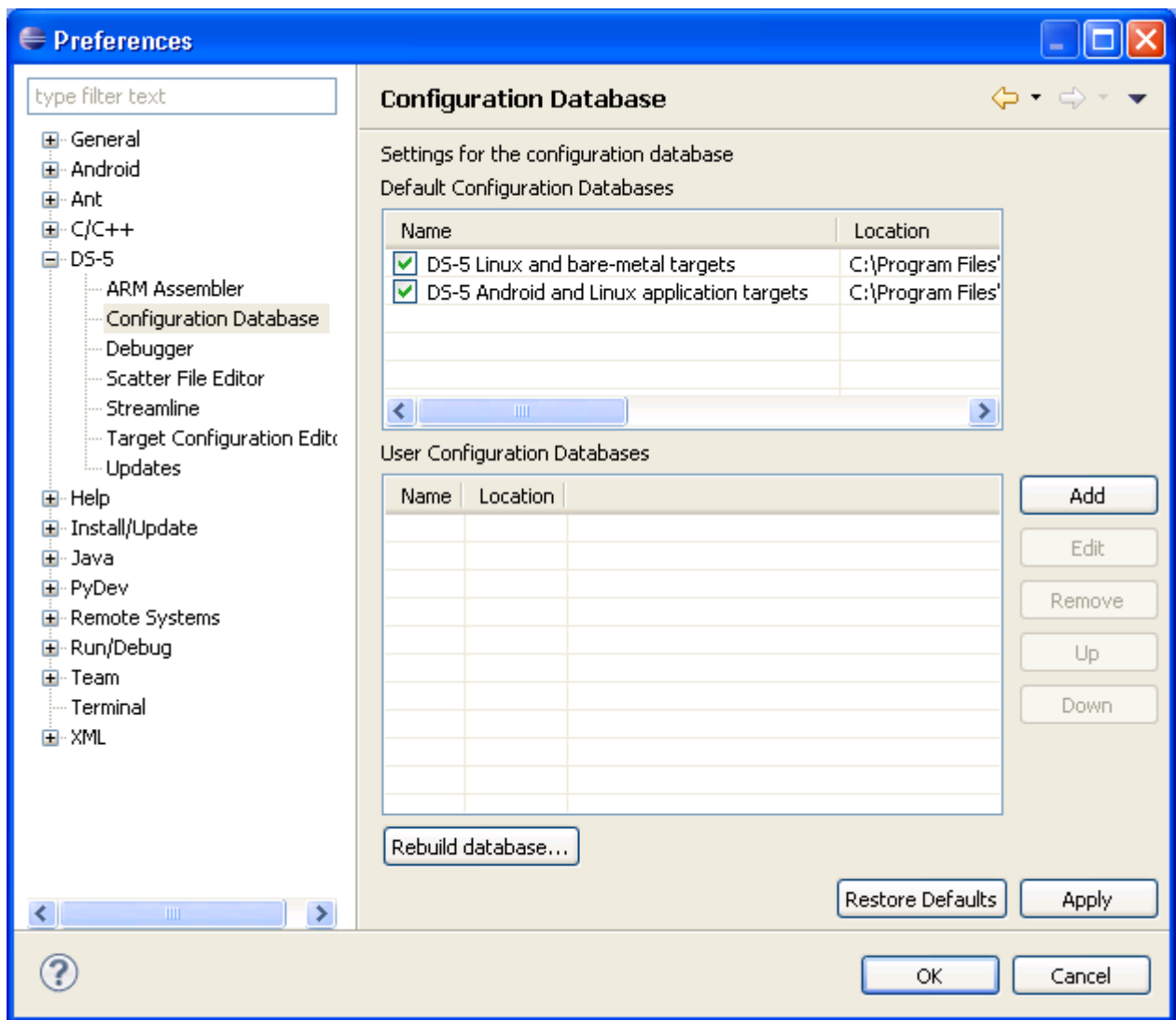


Figure 2-1 Adding a new configuration database

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

[2.10 About the target configuration import utility on page 2-56.](#)

Related tasks

[2.11 Adding a new platform on page 2-58.](#)

Related references

[10.45 Configuration Database panel on page 10-310.](#)

2.13 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**.

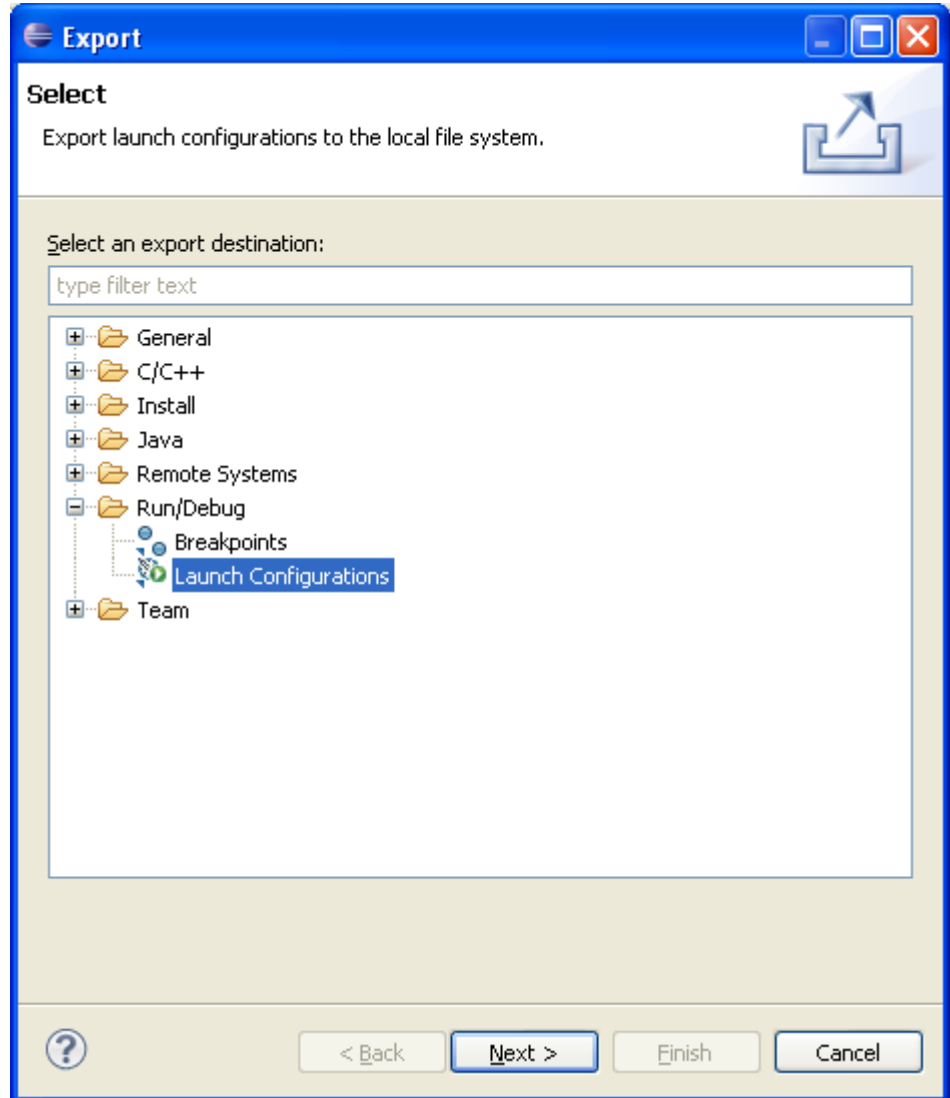


Figure 2-2 Export launch configuration dialog box

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**.

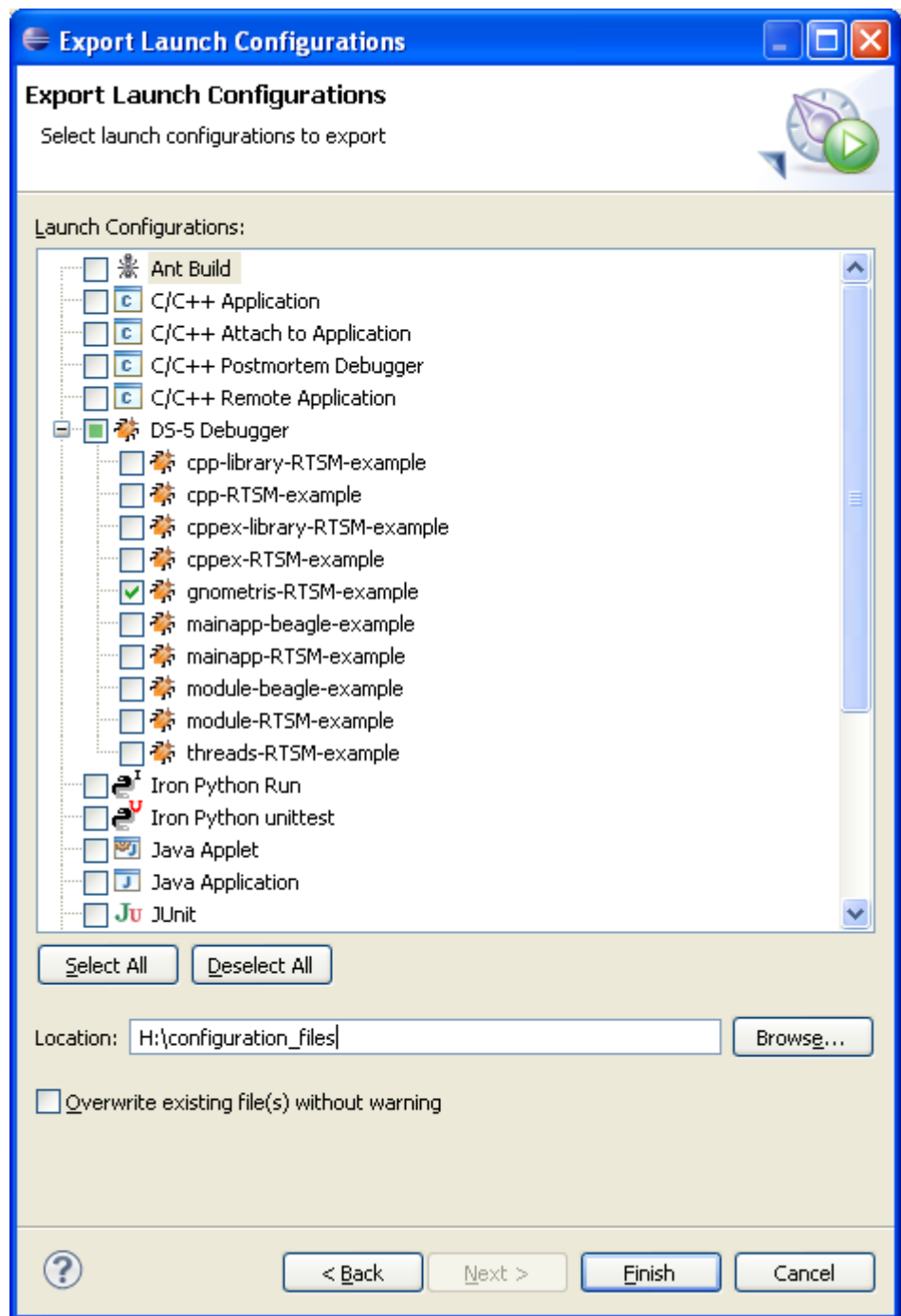


Figure 2-3 Launch configuration selection panels

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

Related tasks

- [2.14 Importing an existing launch configuration on page 2-65.](#)
- [2.4 Configuring a connection to a Fixed Virtual Platform \(FVP\) for Linux application debug on page 2-41.](#)
- [2.3 Configuring a connection to a Linux target using gdbserver on page 2-39.](#)
- [2.6 Configuring a connection to a Linux Kernel on page 2-45.](#)

2.8 Configuring a connection to a bare-metal target on page 2-52.

2.9 Configuring an Event Viewer connection to a bare-metal target on page 2-54.

2.14 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**.

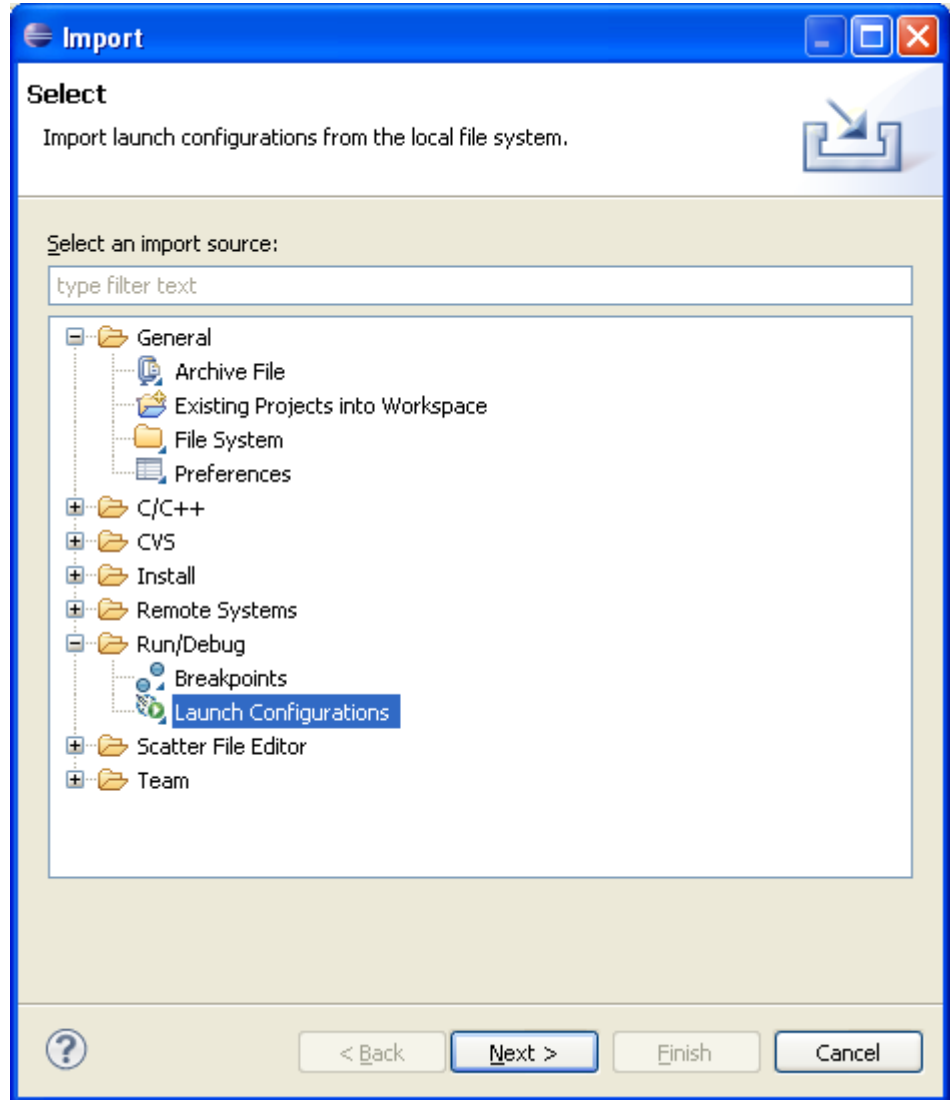


Figure 2-4 Import launch configuration dialog box

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).

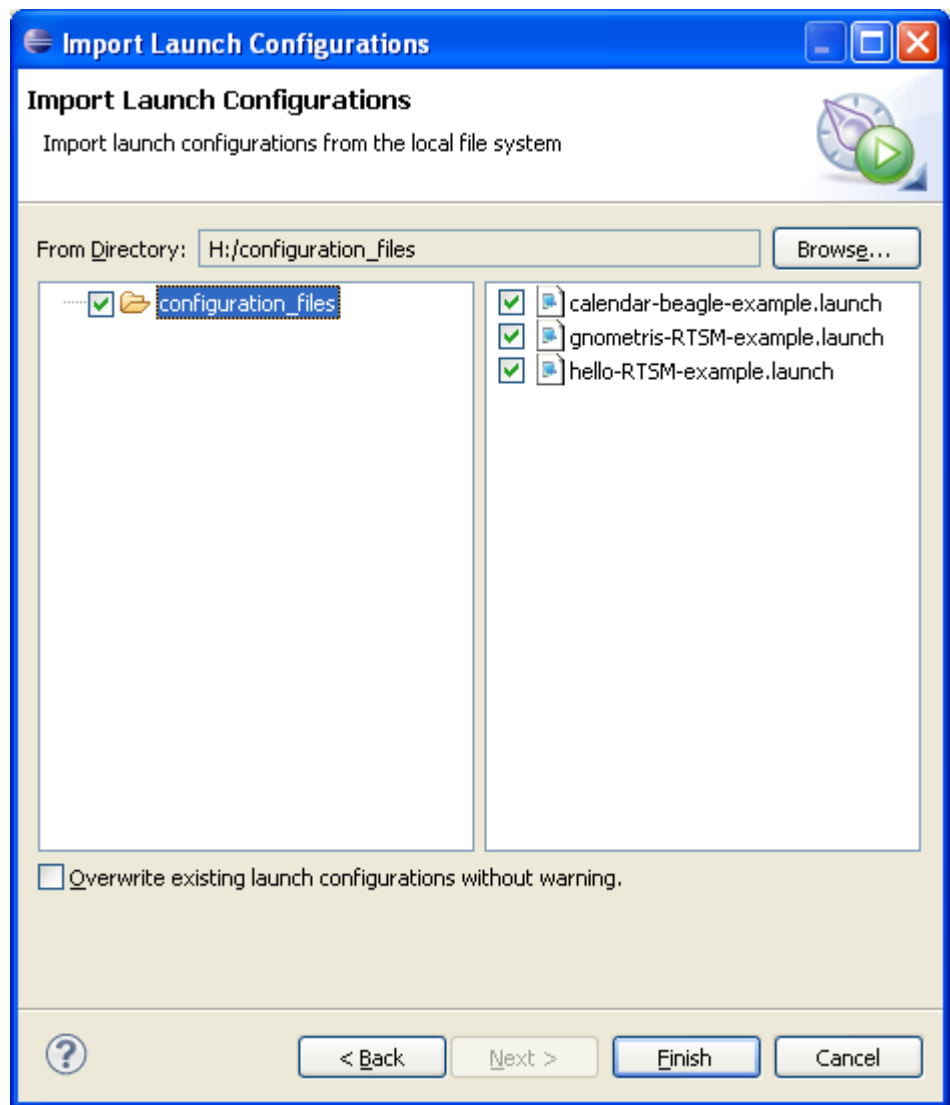


Figure 2-5 Launch configuration file selection panels

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

- [2.13 Exporting an existing launch configuration on page 2-62.](#)
- [2.4 Configuring a connection to a Fixed Virtual Platform \(FVP\) for Linux application debug on page 2-41.](#)
- [2.3 Configuring a connection to a Linux target using gdbserver on page 2-39.](#)
- [2.6 Configuring a connection to a Linux Kernel on page 2-45.](#)
- [2.8 Configuring a connection to a bare-metal target on page 2-52.](#)
- [2.9 Configuring an Event Viewer connection to a bare-metal target on page 2-54.](#)

2.15 Disconnecting from a target

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

Procedure

1. 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 10-214.](#)

[10.7 Debug Control view on page 10-217.](#)

[10.57 DS-5 Debugger menu and toolbar icons on page 10-342.](#)

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 sections:

- [3.1 About the target configuration editor on page 3-69.](#)
- [3.2 Target configuration editor - Overview tab on page 3-70.](#)
- [3.3 Target configuration editor - Memory tab on page 3-72.](#)
- [3.4 Target configuration editor - Peripherals tab on page 3-74.](#)
- [3.5 Target configuration editor - Registers tab on page 3-76.](#)
- [3.6 Target configuration editor - Group View tab on page 3-78.](#)
- [3.7 Target configuration editor - Enumerations tab on page 3-80.](#)
- [3.8 Target configuration editor - Configurations tab on page 3-82.](#)
- [3.9 Scenario demonstrating how to create a new target configuration file on page 3-84.](#)
- [3.10 Creating a power domain for a target on page 3-95.](#)
- [3.11 Creating a Group list on page 3-96.](#)
- [3.12 Importing an existing target configuration file on page 3-98.](#)
- [3.13 Exporting a target configuration file on page 3-100.](#)

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.

To configure the target peripherals, you must provide the TCF files to DS-5 Debugger before connecting to the target. You can specify directories containing TCF files in the **Debug Configurations** window by selecting **Add peripheral description files from directory** in the **Files** tab.

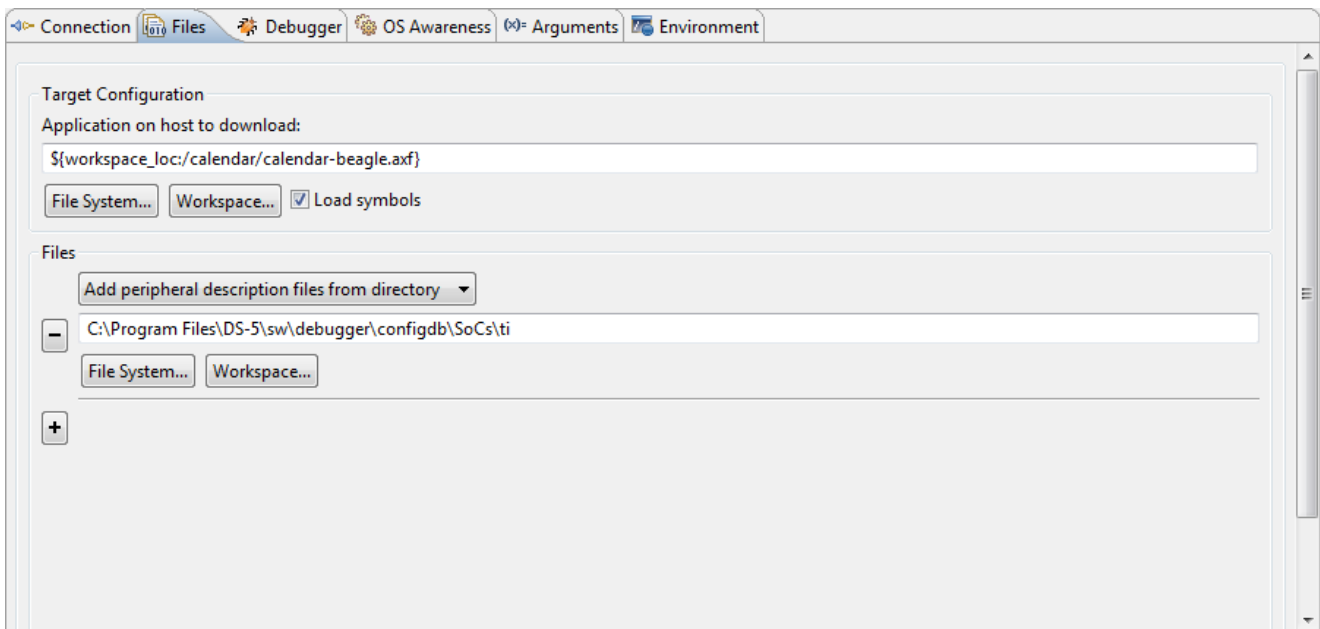


Figure 3-1 Specifying TCF files in the Debug Configurations window

Related references

- [3.2 Target configuration editor - Overview tab on page 3-70.](#)
- [3.3 Target configuration editor - Memory tab on page 3-72.](#)
- [3.4 Target configuration editor - Peripherals tab on page 3-74.](#)
- [3.5 Target configuration editor - Registers tab on page 3-76.](#)
- [3.6 Target configuration editor - Group View tab on page 3-78.](#)
- [3.7 Target configuration editor - Enumerations tab on page 3-80.](#)
- [3.8 Target configuration editor - Configurations tab on page 3-82.](#)
- [10.39 Debug Configurations - Files tab on page 10-296.](#)
- [3.9 Scenario demonstrating how to create a new target configuration file on page 3-84.](#)

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.

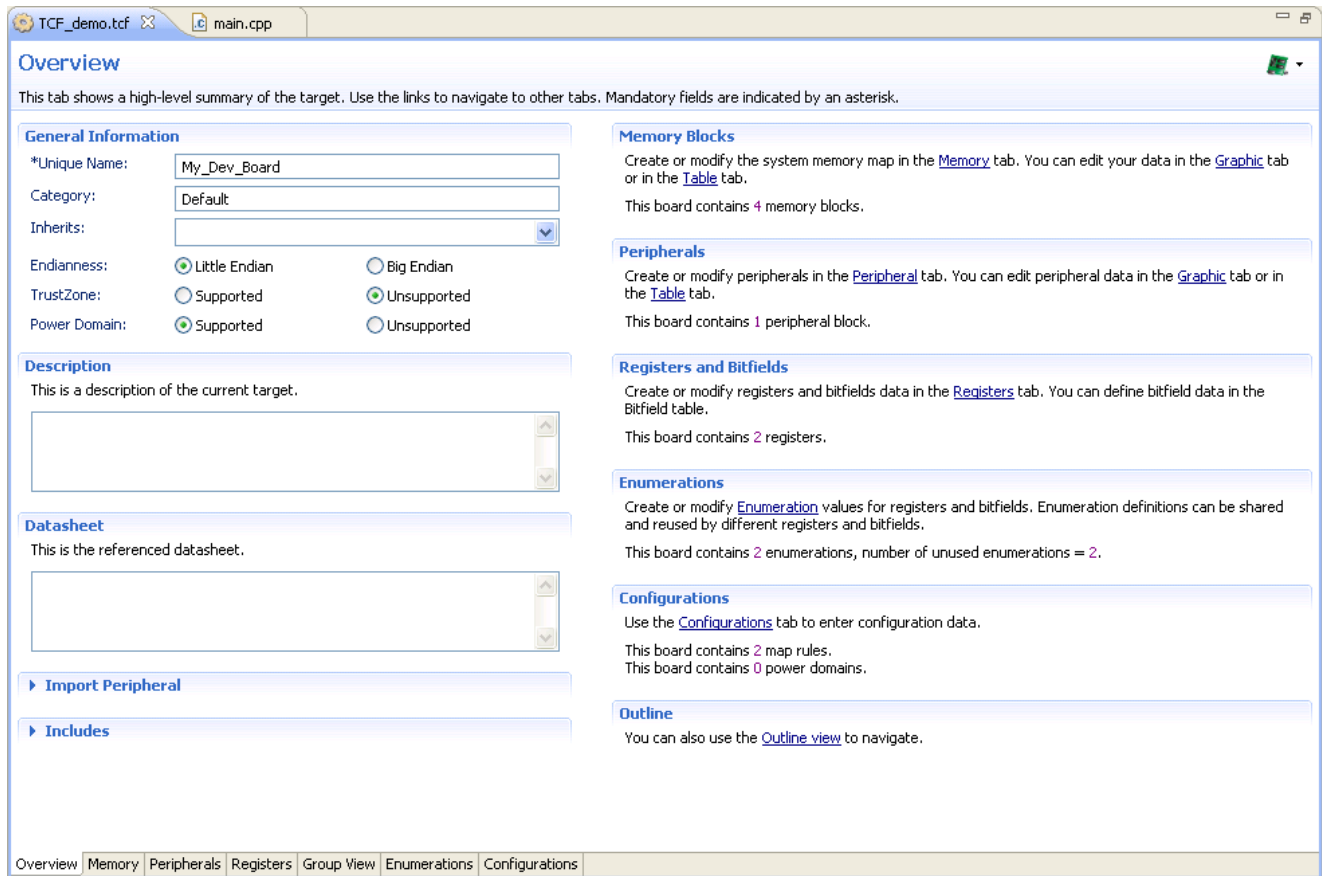


Figure 3-2 Target configuration editor - Overview tab

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 3-69.

Related tasks

[3.10 Creating a power domain for a target](#) on page 3-95.

Related references

[3.3 Target configuration editor - Memory tab](#) on page 3-72.

[3.4 Target configuration editor - Peripherals tab](#) on page 3-74.

[3.5 Target configuration editor - Registers tab](#) on page 3-76.

[3.6 Target configuration editor - Group View tab](#) on page 3-78.

[3.7 Target configuration editor - Enumerations tab](#) on page 3-80.

[3.8 Target configuration editor - Configurations tab](#) on page 3-82.

[3.9 Scenario demonstrating how to create a new target configuration file](#) on page 3-84.

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 0x0.

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.

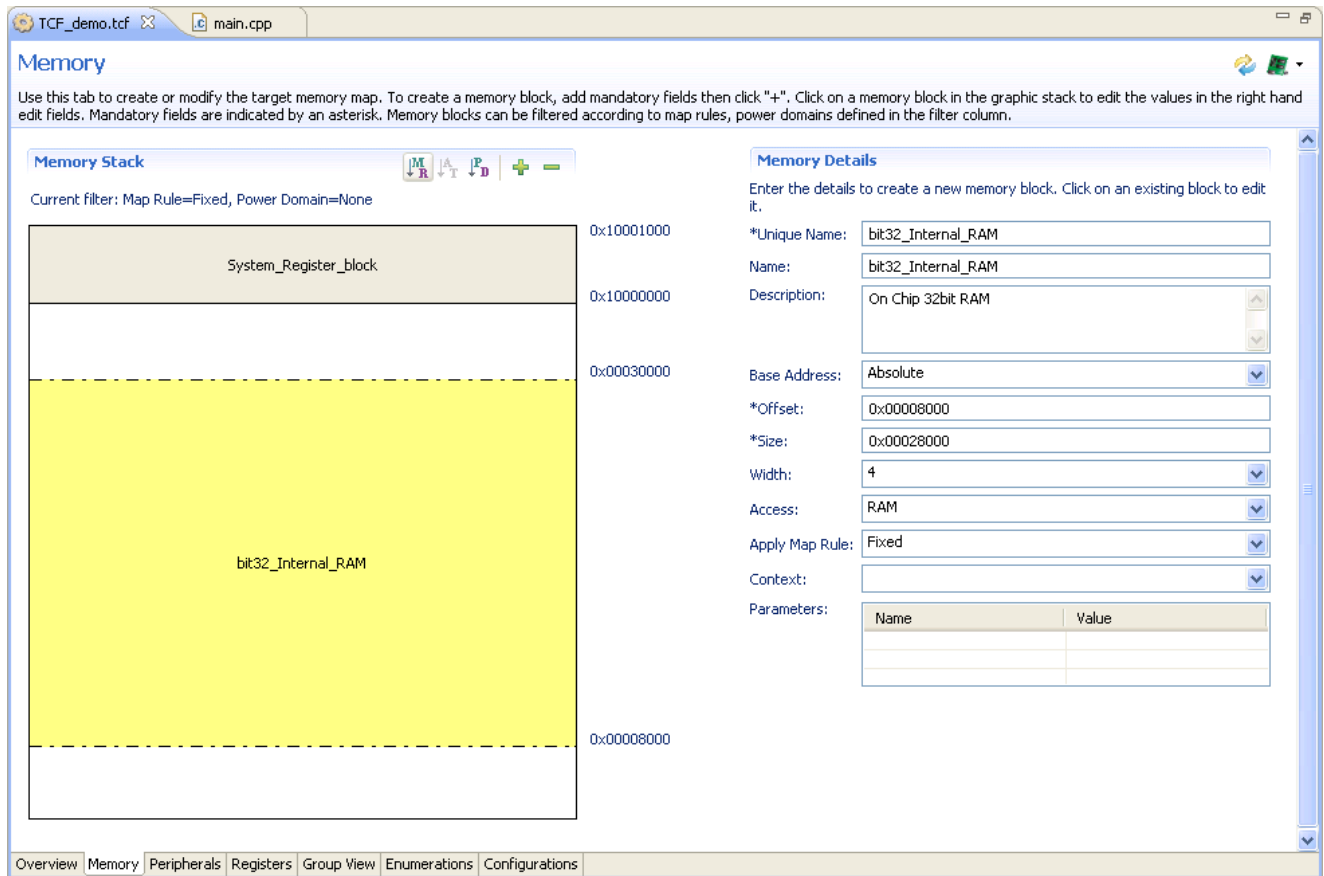


Figure 3-3 Target configuration editor - Memory tab

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 3-69.](#)

Related tasks

[3.9.1 Creating a memory map on page 3-85.](#)

[3.9.8 Creating a memory region for remapping by a control register on page 3-91.](#)

[3.9.9 Applying the map rules to the overlapping memory regions on page 3-92.](#)

Related references

[3.2 Target configuration editor - Overview tab on page 3-70.](#)

[3.4 Target configuration editor - Peripherals tab on page 3-74.](#)

[3.5 Target configuration editor - Registers tab on page 3-76.](#)

[3.6 Target configuration editor - Group View tab on page 3-78.](#)

[3.7 Target configuration editor - Enumerations tab on page 3-80.](#)

[3.8 Target configuration editor - Configurations tab on page 3-82.](#)

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 0x0.

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.

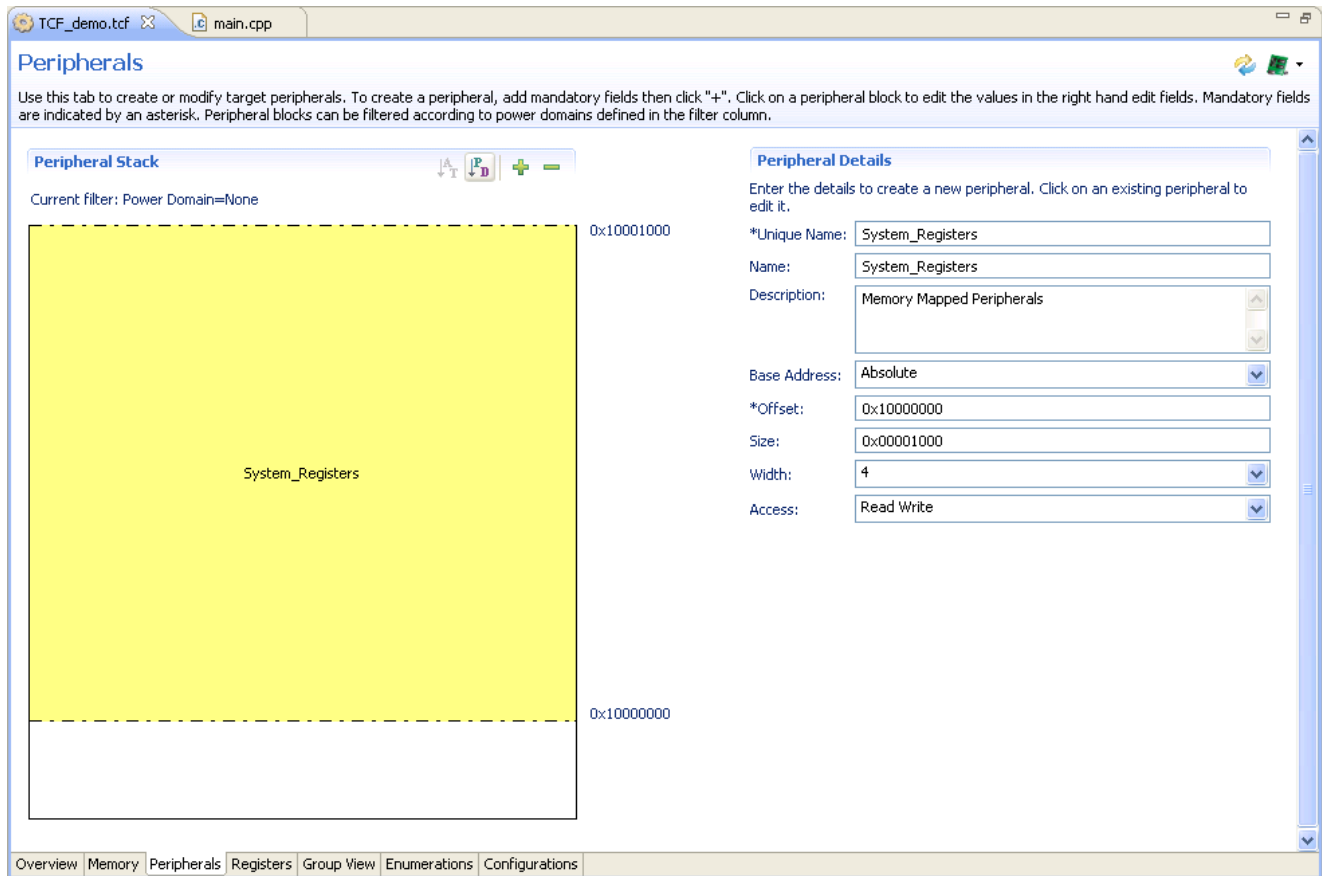


Figure 3-4 Target configuration editor - Peripherals tab

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 3-69.](#)

Related tasks

[3.9.2 Creating a peripheral on page 3-86.](#)

Related references

[3.2 Target configuration editor - Overview tab on page 3-70.](#)

[3.3 Target configuration editor - Memory tab on page 3-72.](#)

[3.5 Target configuration editor - Registers tab on page 3-76.](#)

[3.6 Target configuration editor - Group View tab on page 3-78.](#)

[3.7 Target configuration editor - Enumerations tab on page 3-80.](#)

[3.8 Target configuration editor - Configurations tab on page 3-82.](#)

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.

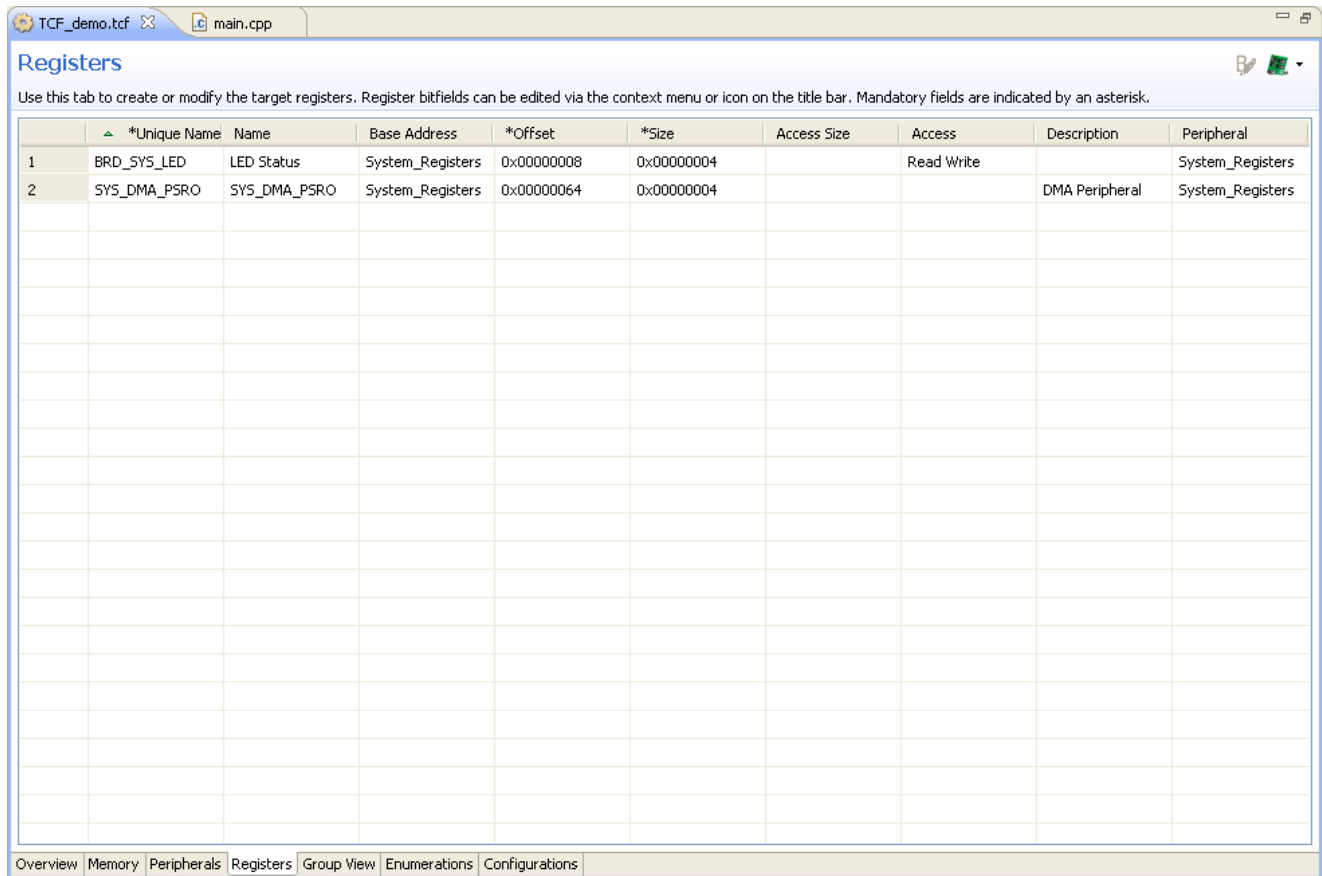


Figure 3-5 Target configuration editor - Registers tab

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 3-69.

Related tasks

3.9.3 Creating a standalone register on page 3-87.

3.9.4 Creating a peripheral register on page 3-88.

3.9.6 Assigning enumerations to a peripheral register on page 3-89.

3.9.5 Creating enumerations for use with a peripheral register on page 3-88.

Related references

3.2 Target configuration editor - Overview tab on page 3-70.

3.3 Target configuration editor - Memory tab on page 3-72.

3.4 Target configuration editor - Peripherals tab on page 3-74.

3.6 Target configuration editor - Group View tab on page 3-78.

3.7 Target configuration editor - Enumerations tab on page 3-80.

3.8 Target configuration editor - Configurations tab on page 3-82.

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**.

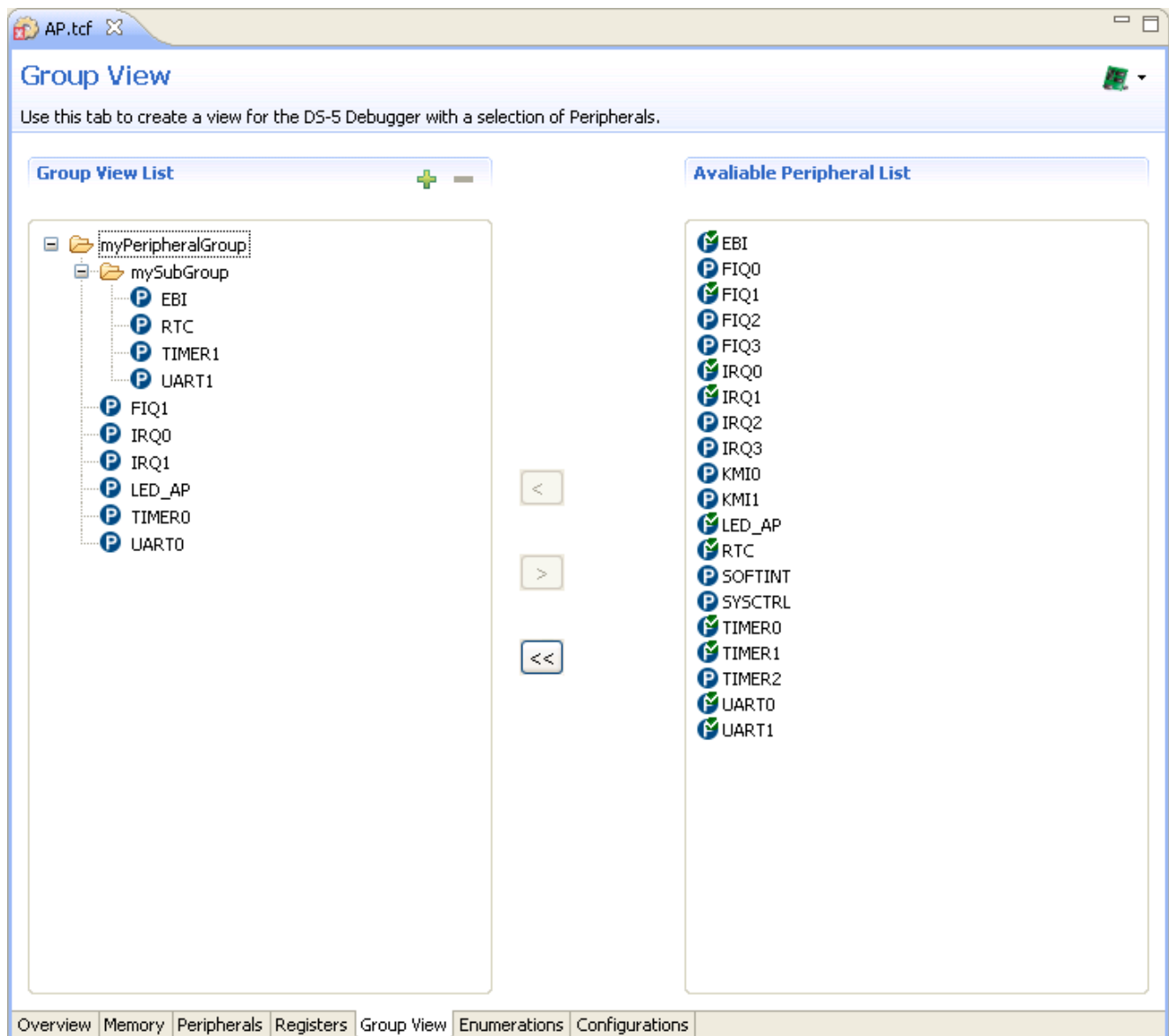


Figure 3-6 Target configuration editor - Group View tab

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 3-69.

Related tasks

[3.11 Creating a Group list](#) on page 3-96.

Related references

[3.2 Target configuration editor - Overview tab](#) on page 3-70.

[3.3 Target configuration editor - Memory tab](#) on page 3-72.

[3.4 Target configuration editor - Peripherals tab](#) on page 3-74.

[3.5 Target configuration editor - Registers tab](#) on page 3-76.

[3.7 Target configuration editor - Enumerations tab](#) on page 3-80.

[3.8 Target configuration editor - Configurations tab](#) on page 3-82.

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.

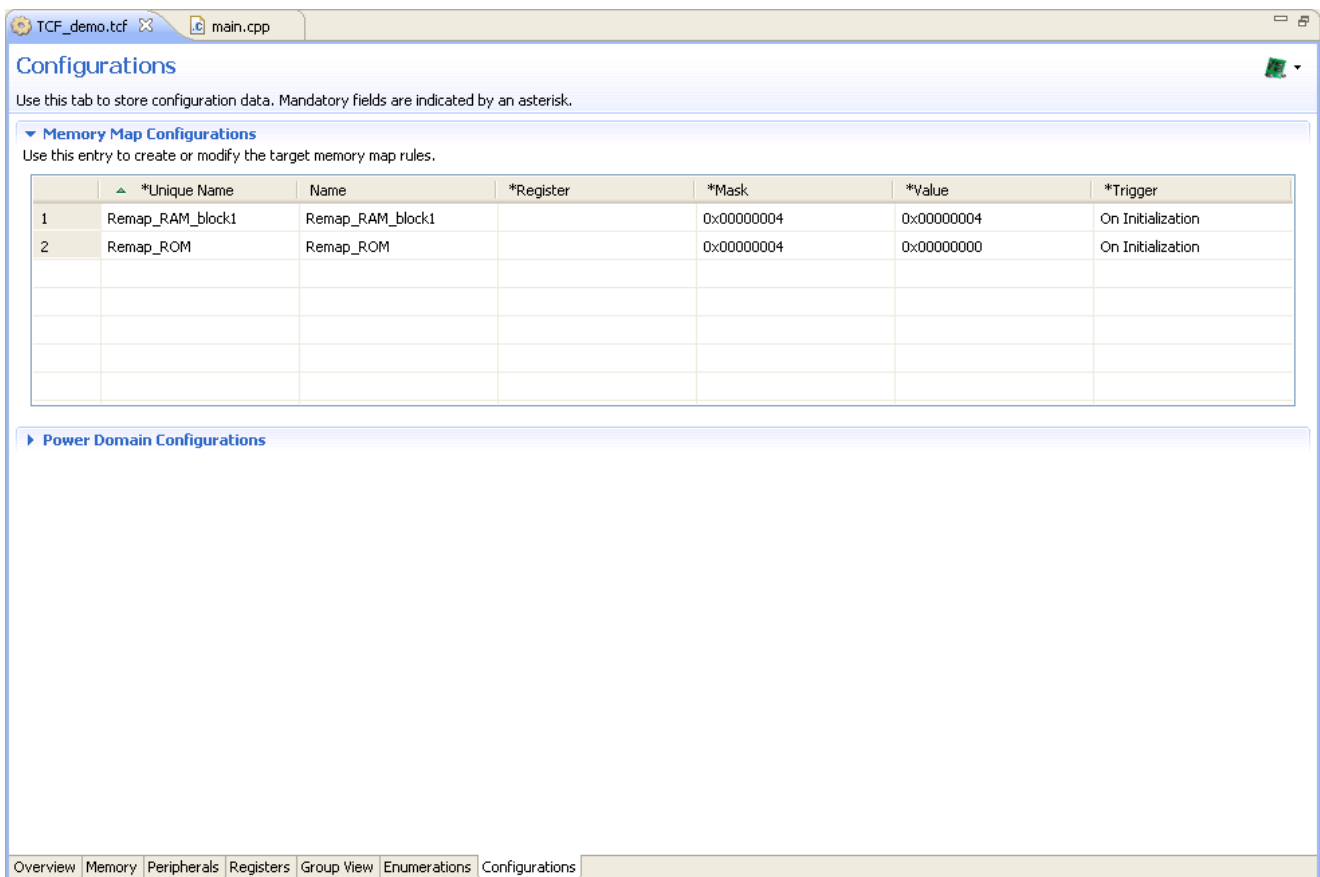


Figure 3-7 Target configuration editor - Enumerations tab

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 3-69.](#)

Related tasks

[3.9.5 Creating enumerations for use with a peripheral register](#) on page 3-88.

[3.9.6 Assigning enumerations to a peripheral register](#) on page 3-89.

Related references

[3.2 Target configuration editor - Overview tab](#) on page 3-70.

[3.3 Target configuration editor - Memory tab](#) on page 3-72.

[3.4 Target configuration editor - Peripherals tab](#) on page 3-74.

[3.5 Target configuration editor - Registers tab](#) on page 3-76.

[3.6 Target configuration editor - Group View tab](#) on page 3-78.

[3.8 Target configuration editor - Configurations tab](#) on page 3-82.

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.

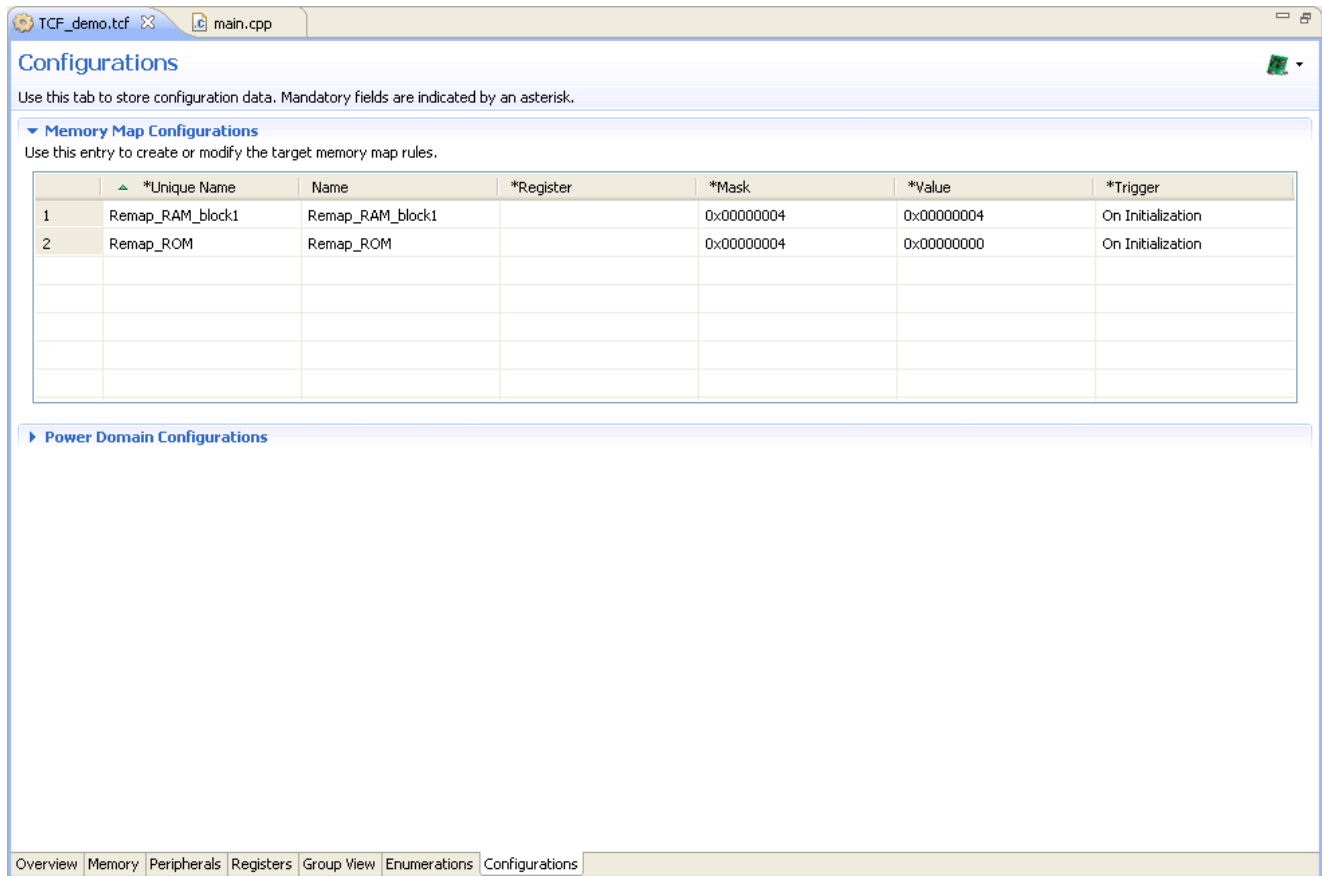


Figure 3-8 Target configuration editor - Configuration tab

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 3-69.

Related tasks

[3.9.3 Creating a standalone register](#) on page 3-87.

[3.9.7 Creating remapping rules for a control register](#) on page 3-90.

[3.10 Creating a power domain for a target](#) on page 3-95.

Related references

[3.2 Target configuration editor - Overview tab](#) on page 3-70.

[3.3 Target configuration editor - Memory tab](#) on page 3-72.

[3.4 Target configuration editor - Peripherals tab](#) on page 3-74.

[3.5 Target configuration editor - Registers tab](#) on page 3-76.

[3.6 Target configuration editor - Group View tab](#) on page 3-78.

[3.7 Target configuration editor - Enumerations tab](#) on page 3-80.

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.

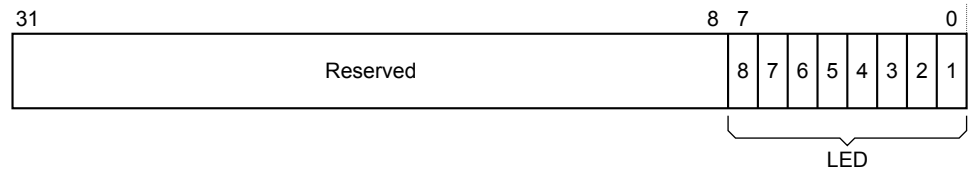


Figure 3-9 LED register and bitfields

- 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.

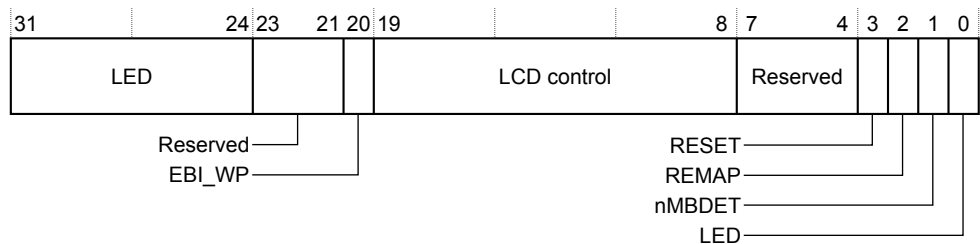


Figure 3-10 Core module and LCD control register

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

This section contains the following subsections:

- [3.9.1 Creating a memory map on page 3-85.](#)
- [3.9.2 Creating a peripheral on page 3-86.](#)
- [3.9.3 Creating a standalone register on page 3-87.](#)
- [3.9.4 Creating a peripheral register on page 3-88.](#)
- [3.9.5 Creating enumerations for use with a peripheral register on page 3-88.](#)
- [3.9.6 Assigning enumerations to a peripheral register on page 3-89.](#)
- [3.9.7 Creating remapping rules for a control register on page 3-90.](#)
- [3.9.8 Creating a memory region for remapping by a control register on page 3-91.](#)
- [3.9.9 Applying the map rules to the overlapping memory regions on page 3-92.](#)

3.9.1 Creating a memory map

Describes how to create a new memory map.

Procedure

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

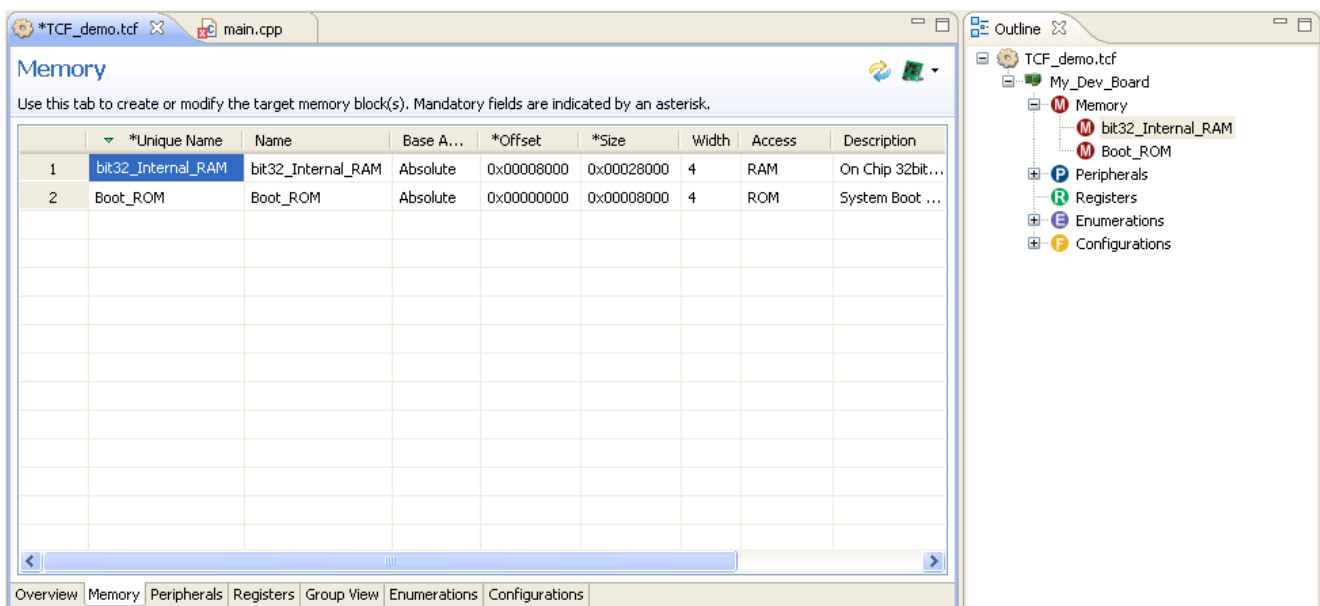


Figure 3-11 Creating a Memory map

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 3-86.](#)
- [3.9.3 Creating a standalone register on page 3-87.](#)
- [3.9.4 Creating a peripheral register on page 3-88.](#)
- [3.9.5 Creating enumerations for use with a peripheral register on page 3-88.](#)
- [3.9.6 Assigning enumerations to a peripheral register on page 3-89.](#)
- [3.9.7 Creating remapping rules for a control register on page 3-90.](#)
- [3.9.8 Creating a memory region for remapping by a control register on page 3-91.](#)
- [3.9.9 Applying the map rules to the overlapping memory regions on page 3-92.](#)

Related references

- [3.9 Scenario demonstrating how to create a new target configuration file on page 3-84.](#)
- [3.3 Target configuration editor - Memory tab on page 3-72.](#)

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.

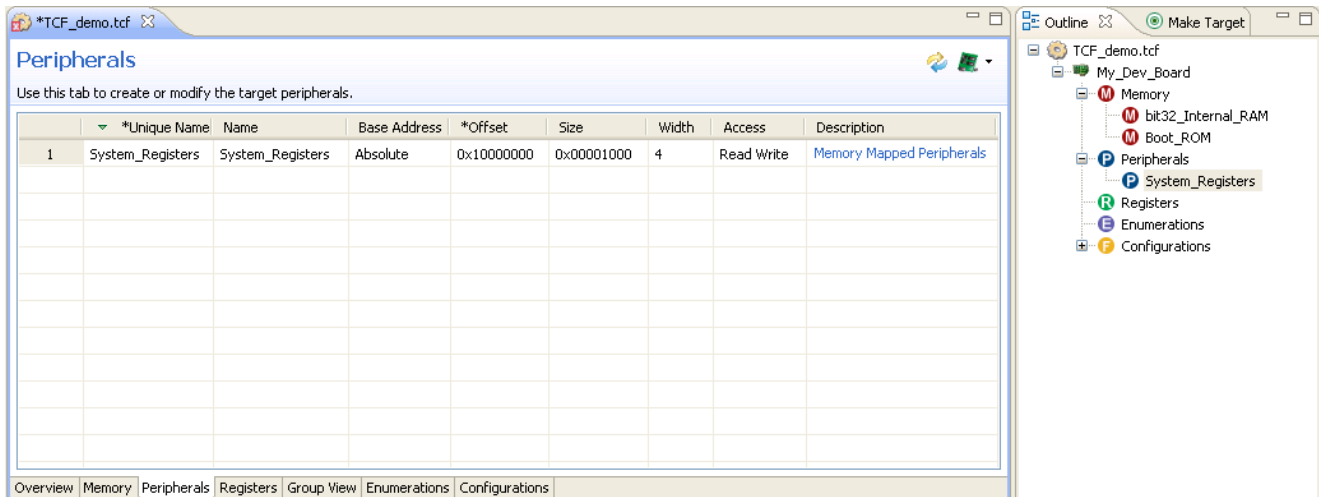


Figure 3-12 Creating a peripheral

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 3-85.](#)
- [3.9.3 Creating a standalone register on page 3-87.](#)
- [3.9.4 Creating a peripheral register on page 3-88.](#)
- [3.9.5 Creating enumerations for use with a peripheral register on page 3-88.](#)
- [3.9.6 Assigning enumerations to a peripheral register on page 3-89.](#)
- [3.9.7 Creating remapping rules for a control register on page 3-90.](#)
- [3.9.8 Creating a memory region for remapping by a control register on page 3-91.](#)
- [3.9.9 Applying the map rules to the overlapping memory regions on page 3-92.](#)

Related references

[3.9 Scenario demonstrating how to create a new target configuration file on page 3-84.](#)

[3.4 Target configuration editor - Peripherals tab on page 3-74.](#)

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.

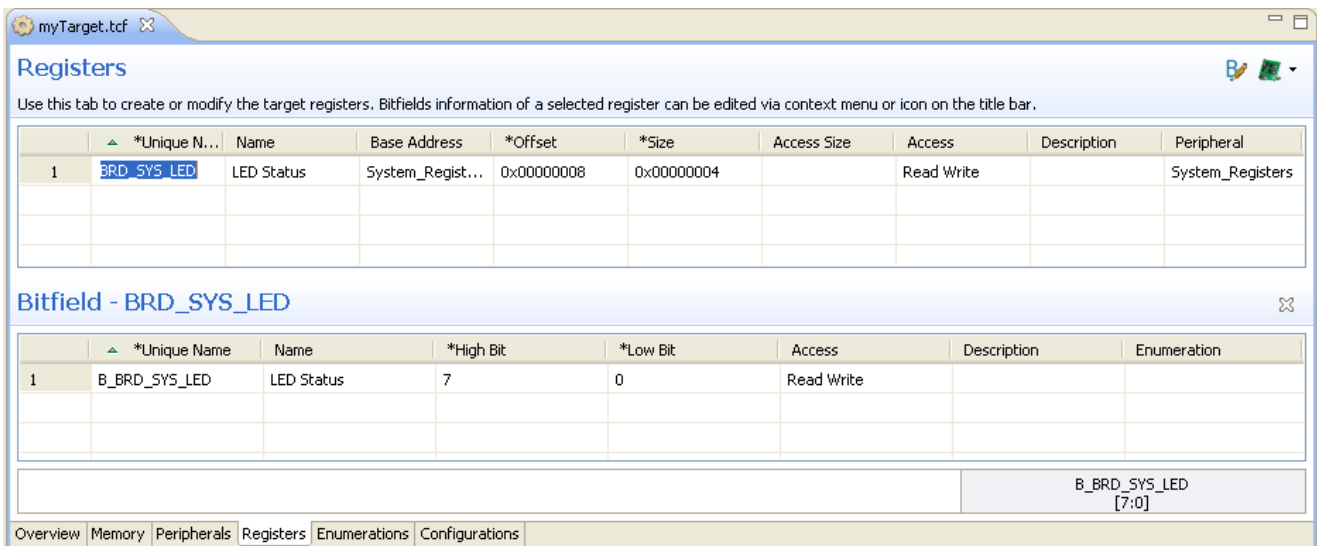


Figure 3-13 Creating a standalone register

On completion, close the floating table.

Related tasks

[3.9.1 Creating a memory map on page 3-85.](#)

[3.9.2 Creating a peripheral on page 3-86.](#)

[3.9.4 Creating a peripheral register on page 3-88.](#)

[3.9.5 Creating enumerations for use with a peripheral register on page 3-88.](#)

[3.9.6 Assigning enumerations to a peripheral register on page 3-89.](#)

[3.9.7 Creating remapping rules for a control register on page 3-90.](#)

[3.9.8 Creating a memory region for remapping by a control register on page 3-91.](#)

[3.9.9 Applying the map rules to the overlapping memory regions on page 3-92.](#)

Related references

[3.9 Scenario demonstrating how to create a new target configuration file on page 3-84.](#)

[3.5 Target configuration editor - Registers tab on page 3-76.](#)

[3.8 Target configuration editor - Configurations tab on page 3-82.](#)

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.

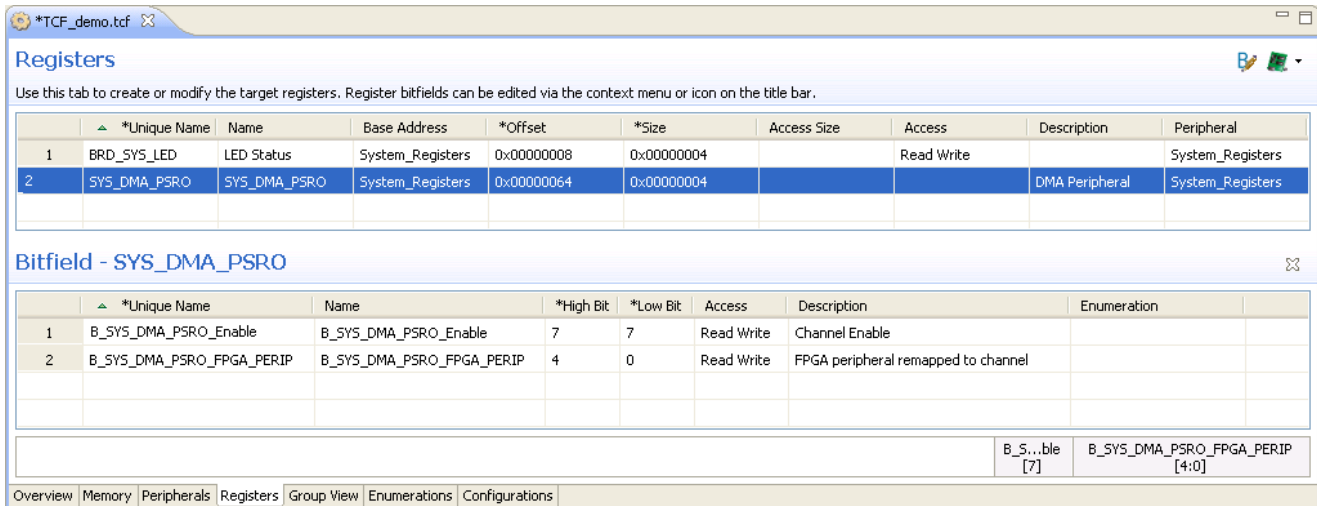


Figure 3-14 Creating a peripheral register

Related tasks

- [3.9.1 Creating a memory map on page 3-85.](#)
- [3.9.2 Creating a peripheral on page 3-86.](#)
- [3.9.3 Creating a standalone register on page 3-87.](#)
- [3.9.5 Creating enumerations for use with a peripheral register on page 3-88.](#)
- [3.9.6 Assigning enumerations to a peripheral register on page 3-89.](#)
- [3.9.7 Creating remapping rules for a control register on page 3-90.](#)
- [3.9.8 Creating a memory region for remapping by a control register on page 3-91.](#)
- [3.9.9 Applying the map rules to the overlapping memory regions on page 3-92.](#)

Related references

- [3.9 Scenario demonstrating how to create a new target configuration file on page 3-84.](#)
- [3.5 Target configuration editor - Registers tab on page 3-76.](#)

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.

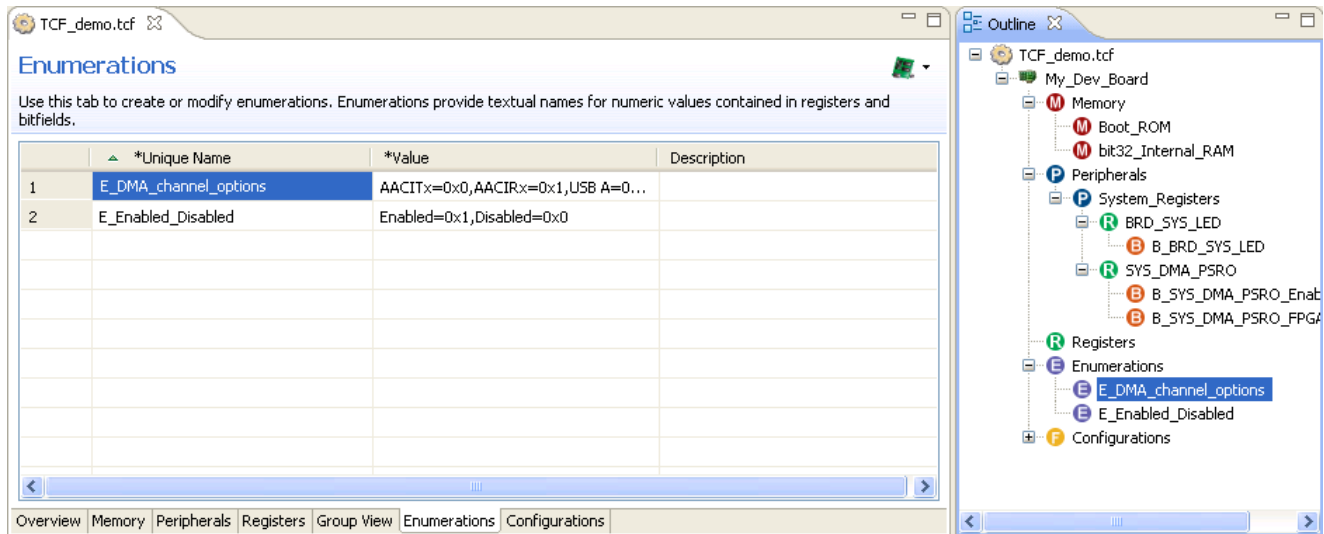


Figure 3-15 Creating enumerations

Related tasks

- [3.9.1 Creating a memory map on page 3-85.](#)
- [3.9.2 Creating a peripheral on page 3-86.](#)
- [3.9.3 Creating a standalone register on page 3-87.](#)
- [3.9.4 Creating a peripheral register on page 3-88.](#)
- [3.9.6 Assigning enumerations to a peripheral register on page 3-89.](#)
- [3.9.7 Creating remapping rules for a control register on page 3-90.](#)
- [3.9.8 Creating a memory region for remapping by a control register on page 3-91.](#)
- [3.9.9 Applying the map rules to the overlapping memory regions on page 3-92.](#)

Related references

- [3.9 Scenario demonstrating how to create a new target configuration file on page 3-84.](#)
- [3.5 Target configuration editor - Registers tab on page 3-76.](#)
- [3.7 Target configuration editor - Enumerations tab on page 3-80.](#)

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.

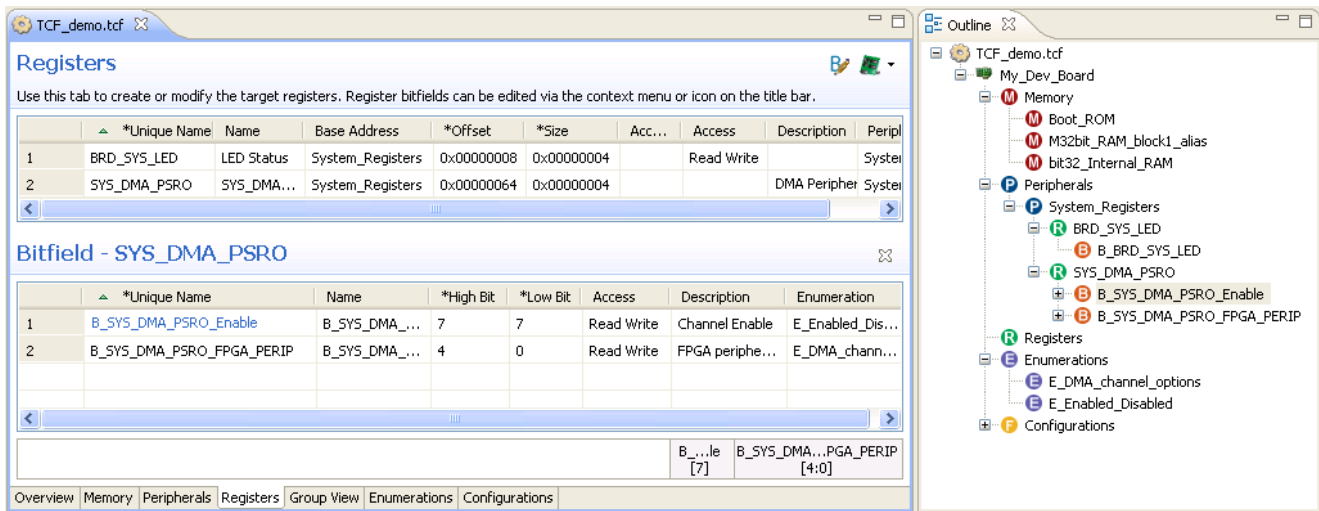


Figure 3-16 Assigning enumerations

Related tasks

- [3.9.1 Creating a memory map on page 3-85.](#)
- [3.9.2 Creating a peripheral on page 3-86.](#)
- [3.9.3 Creating a standalone register on page 3-87.](#)
- [3.9.4 Creating a peripheral register on page 3-88.](#)
- [3.9.5 Creating enumerations for use with a peripheral register on page 3-88.](#)
- [3.9.7 Creating remapping rules for a control register on page 3-90.](#)
- [3.9.8 Creating a memory region for remapping by a control register on page 3-91.](#)
- [3.9.9 Applying the map rules to the overlapping memory regions on page 3-92.](#)

Related references

- [3.9 Scenario demonstrating how to create a new target configuration file on page 3-84.](#)
- [3.5 Target configuration editor - Registers tab on page 3-76.](#)
- [3.7 Target configuration editor - Enumerations tab on page 3-80.](#)

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.

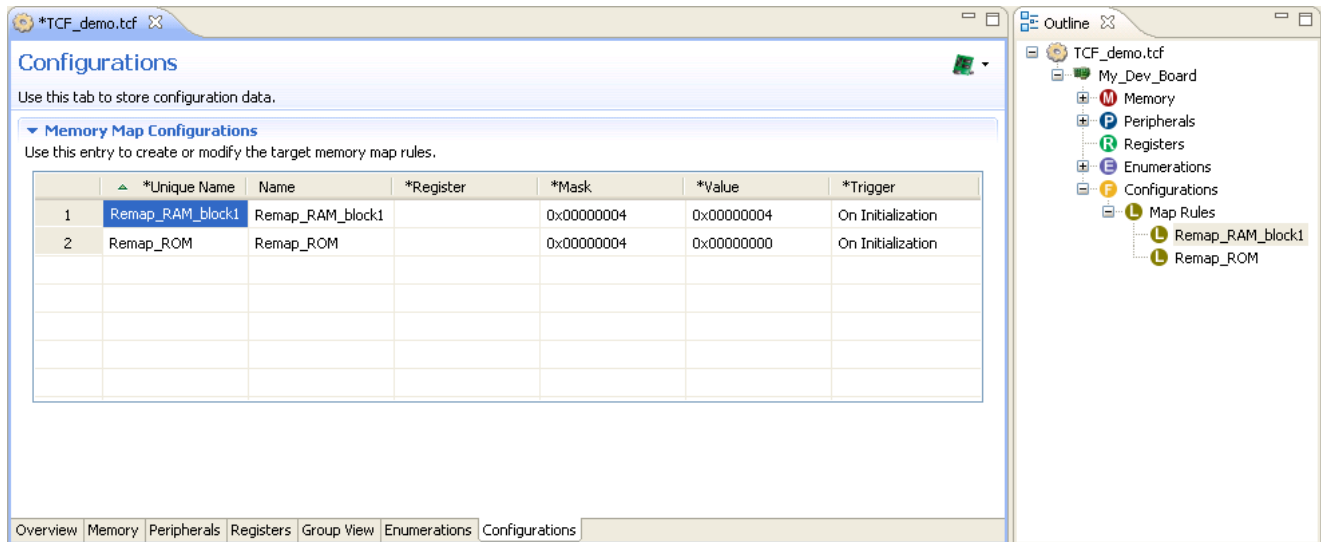


Figure 3-17 Creating remapping rules

Related tasks

- [3.9.1 Creating a memory map on page 3-85.](#)
- [3.9.2 Creating a peripheral on page 3-86.](#)
- [3.9.3 Creating a standalone register on page 3-87.](#)
- [3.9.4 Creating a peripheral register on page 3-88.](#)
- [3.9.5 Creating enumerations for use with a peripheral register on page 3-88.](#)
- [3.9.6 Assigning enumerations to a peripheral register on page 3-89.](#)
- [3.9.8 Creating a memory region for remapping by a control register on page 3-91.](#)
- [3.9.9 Applying the map rules to the overlapping memory regions on page 3-92.](#)

Related references

- [3.9 Scenario demonstrating how to create a new target configuration file on page 3-84.](#)
- [3.8 Target configuration editor - Configurations tab on page 3-82.](#)

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.

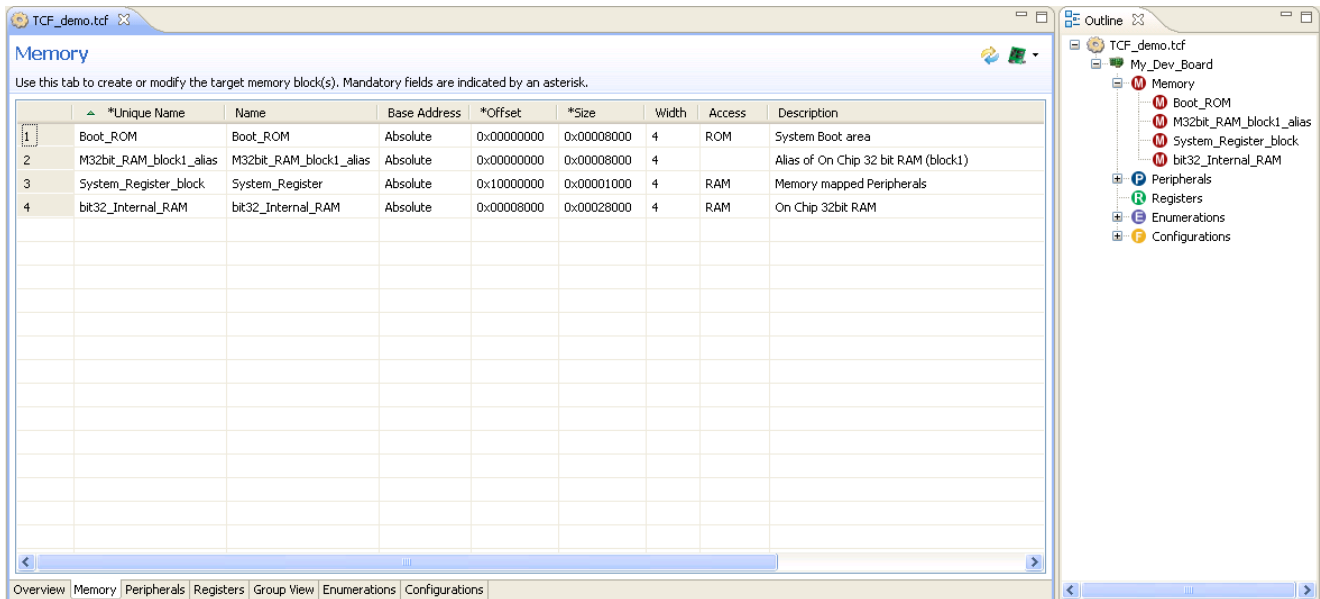


Figure 3-18 Creating a memory region for remapping by a control register

Related tasks

- [3.9.1 Creating a memory map on page 3-85.](#)
- [3.9.2 Creating a peripheral on page 3-86.](#)
- [3.9.3 Creating a standalone register on page 3-87.](#)
- [3.9.4 Creating a peripheral register on page 3-88.](#)
- [3.9.5 Creating enumerations for use with a peripheral register on page 3-88.](#)
- [3.9.6 Assigning enumerations to a peripheral register on page 3-89.](#)
- [3.9.7 Creating remapping rules for a control register on page 3-90.](#)
- [3.9.9 Applying the map rules to the overlapping memory regions on page 3-92.](#)

Related references

- [3.9 Scenario demonstrating how to create a new target configuration file on page 3-84.](#)
- [3.3 Target configuration editor - Memory tab on page 3-72.](#)

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.

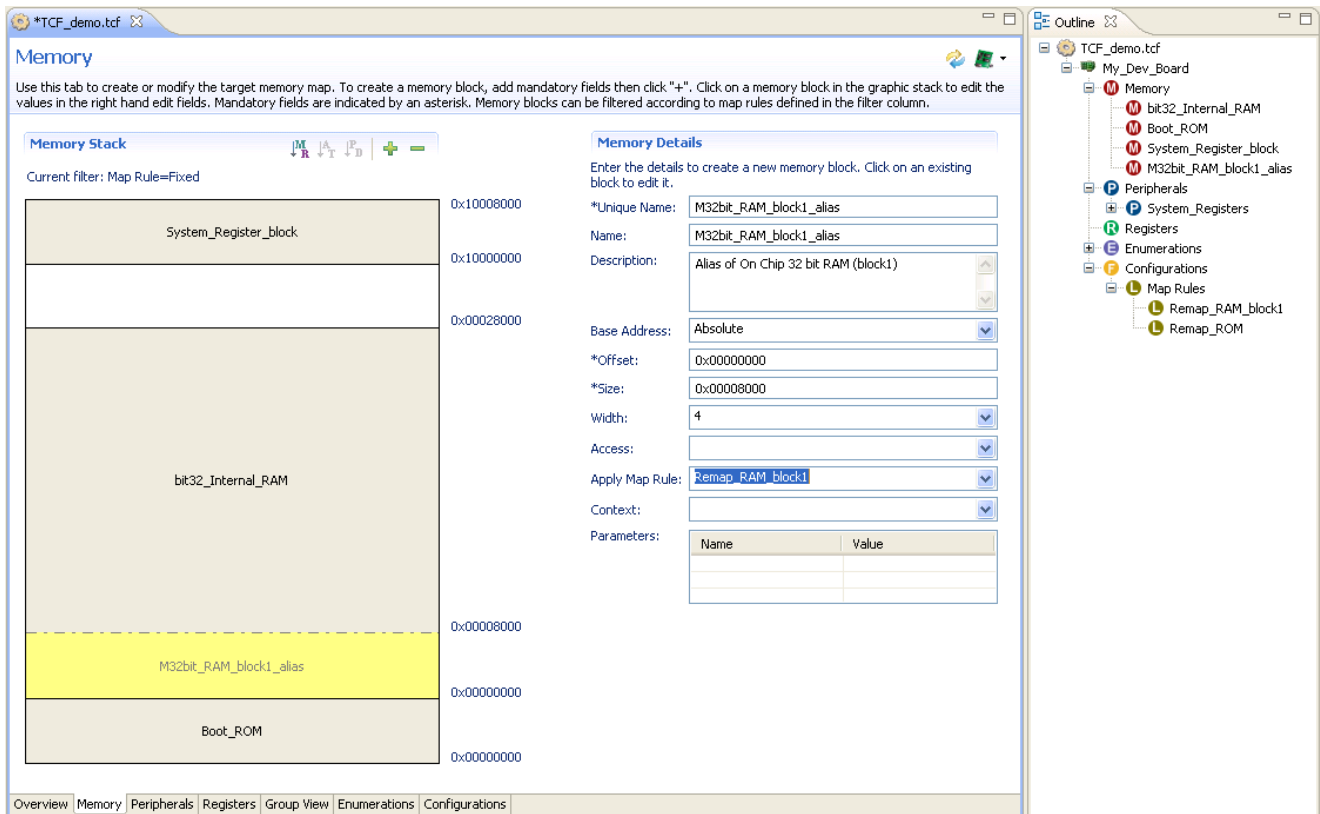


Figure 3-19 Applying the Remap_RAM_block1 map rule

3. To apply the other map rule, you must select **Remap_ROM** in 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.

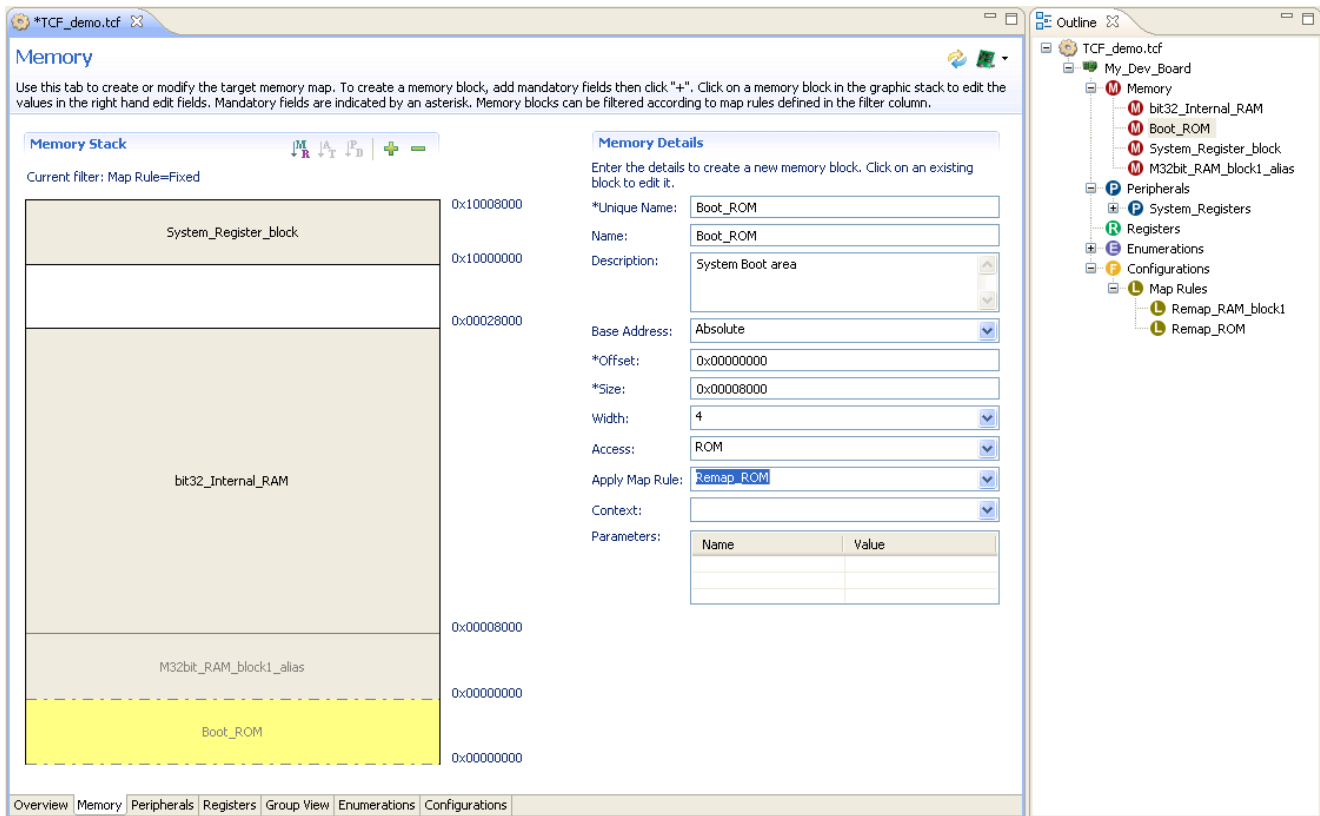


Figure 3-20 Applying the Remap_ROM map rule

5. Save the file.

Related tasks

- [3.9.1 Creating a memory map on page 3-85.](#)
- [3.9.2 Creating a peripheral on page 3-86.](#)
- [3.9.3 Creating a standalone register on page 3-87.](#)
- [3.9.4 Creating a peripheral register on page 3-88.](#)
- [3.9.5 Creating enumerations for use with a peripheral register on page 3-88.](#)
- [3.9.6 Assigning enumerations to a peripheral register on page 3-89.](#)
- [3.9.7 Creating remapping rules for a control register on page 3-90.](#)
- [3.9.8 Creating a memory region for remapping by a control register on page 3-91.](#)

Related references

- [3.9 Scenario demonstrating how to create a new target configuration file on page 3-84.](#)
- [3.3 Target configuration editor - Memory tab on page 3-72.](#)

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.

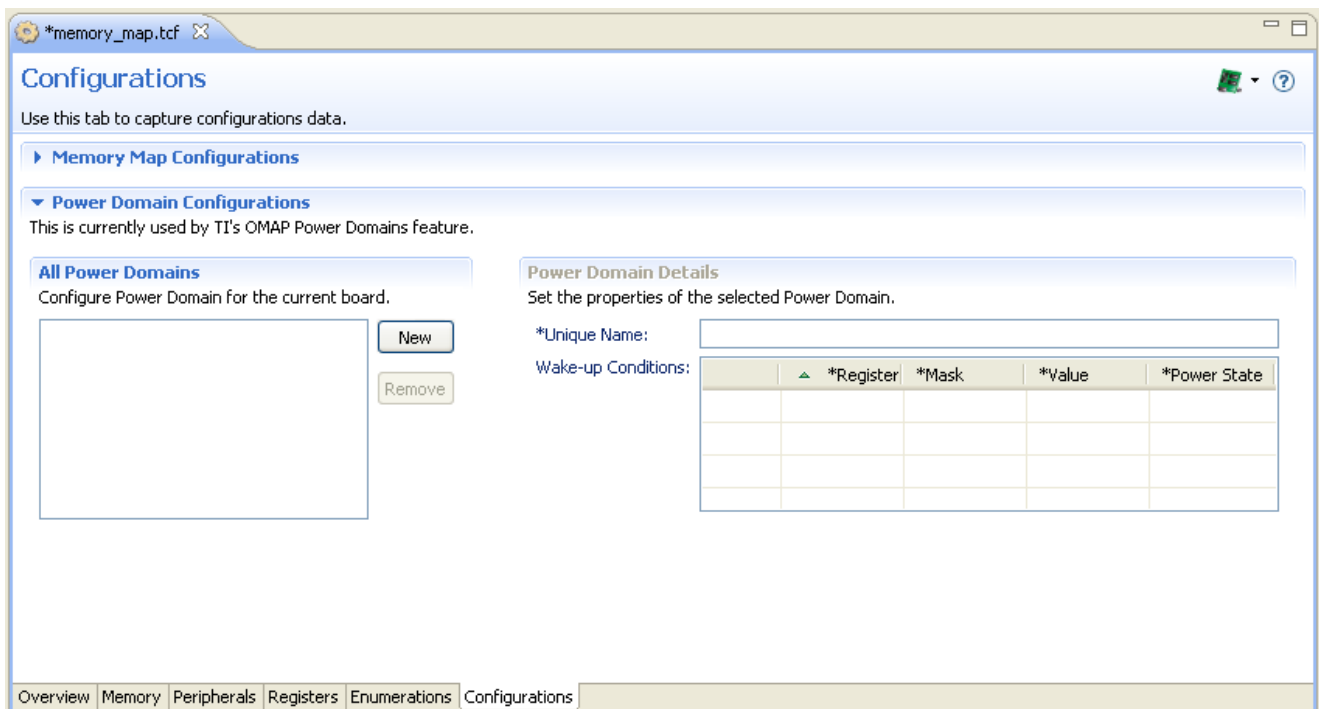


Figure 3-21 Power Domain Configurations

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 3-87.](#)

Related references

[3.2 Target configuration editor - Overview tab on page 3-70.](#)

[3.8 Target configuration editor - Configurations tab on page 3-82.](#)

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.

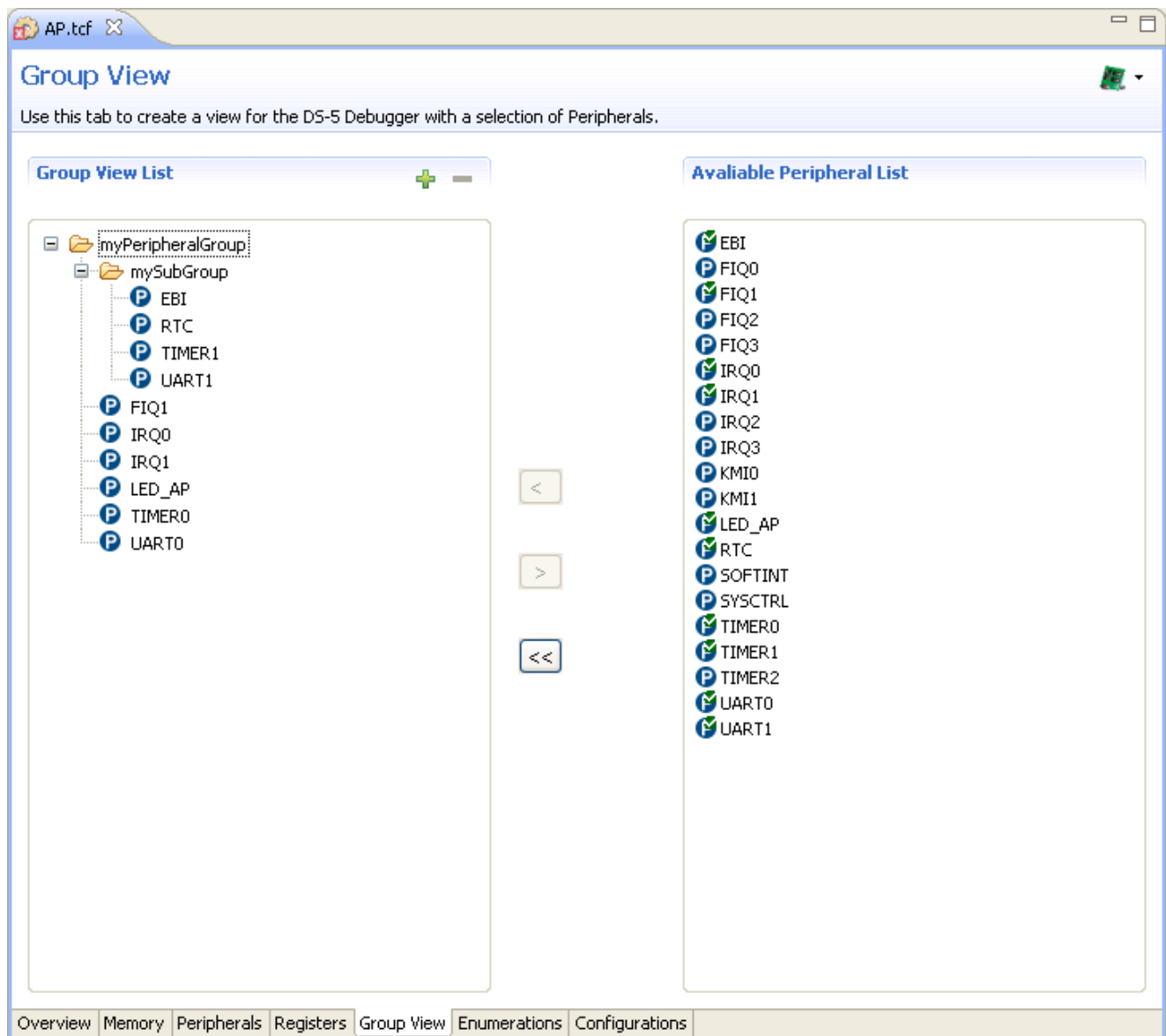


Figure 3-22 Creating a group list

Related references

[3.6 Target configuration editor - Group View tab on page 3-78.](#)

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.

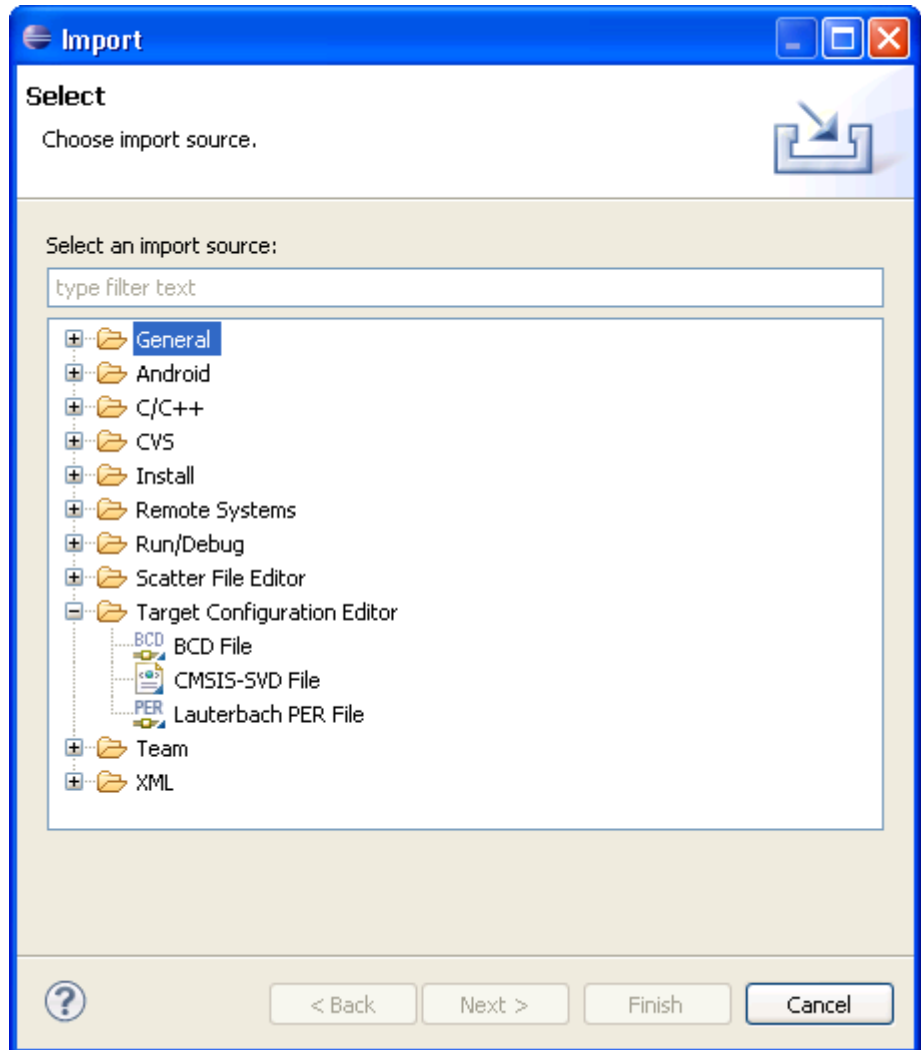


Figure 3-23 Selecting an existing target configuration file

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

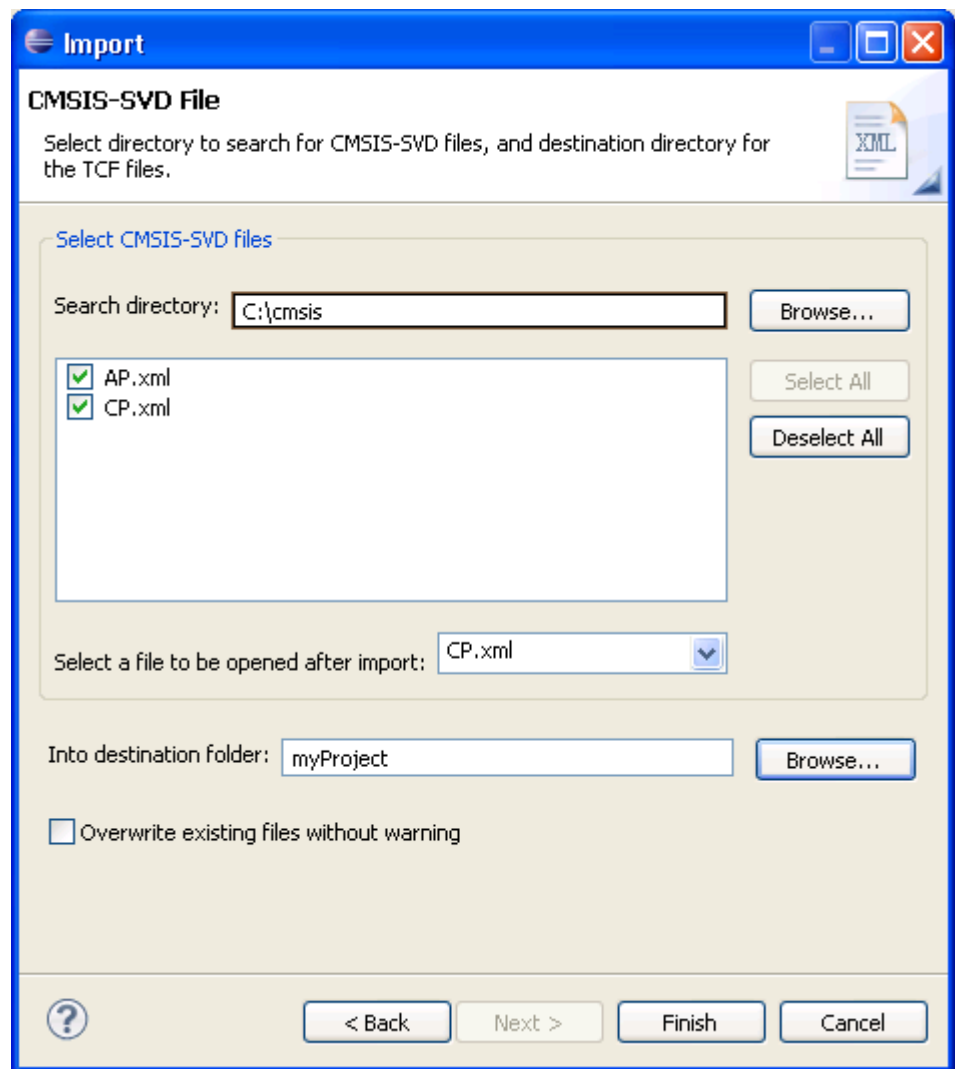


Figure 3-24 Importing the target configuration file

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 3-100.](#)

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**.

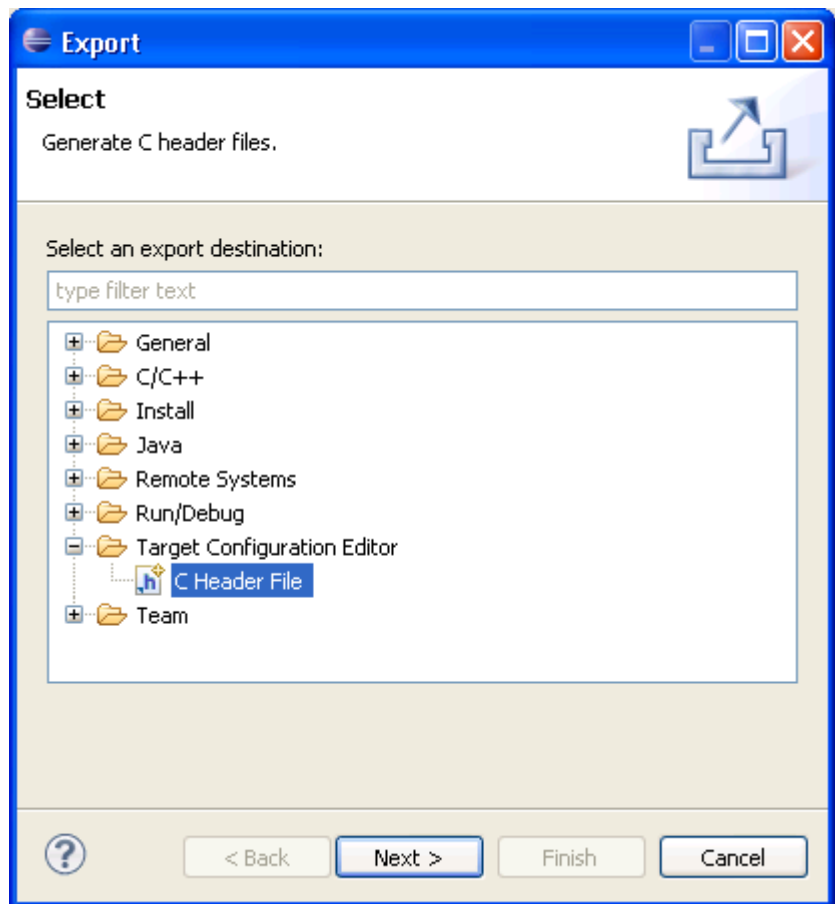


Figure 3-25 Exporting to C header file

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**.

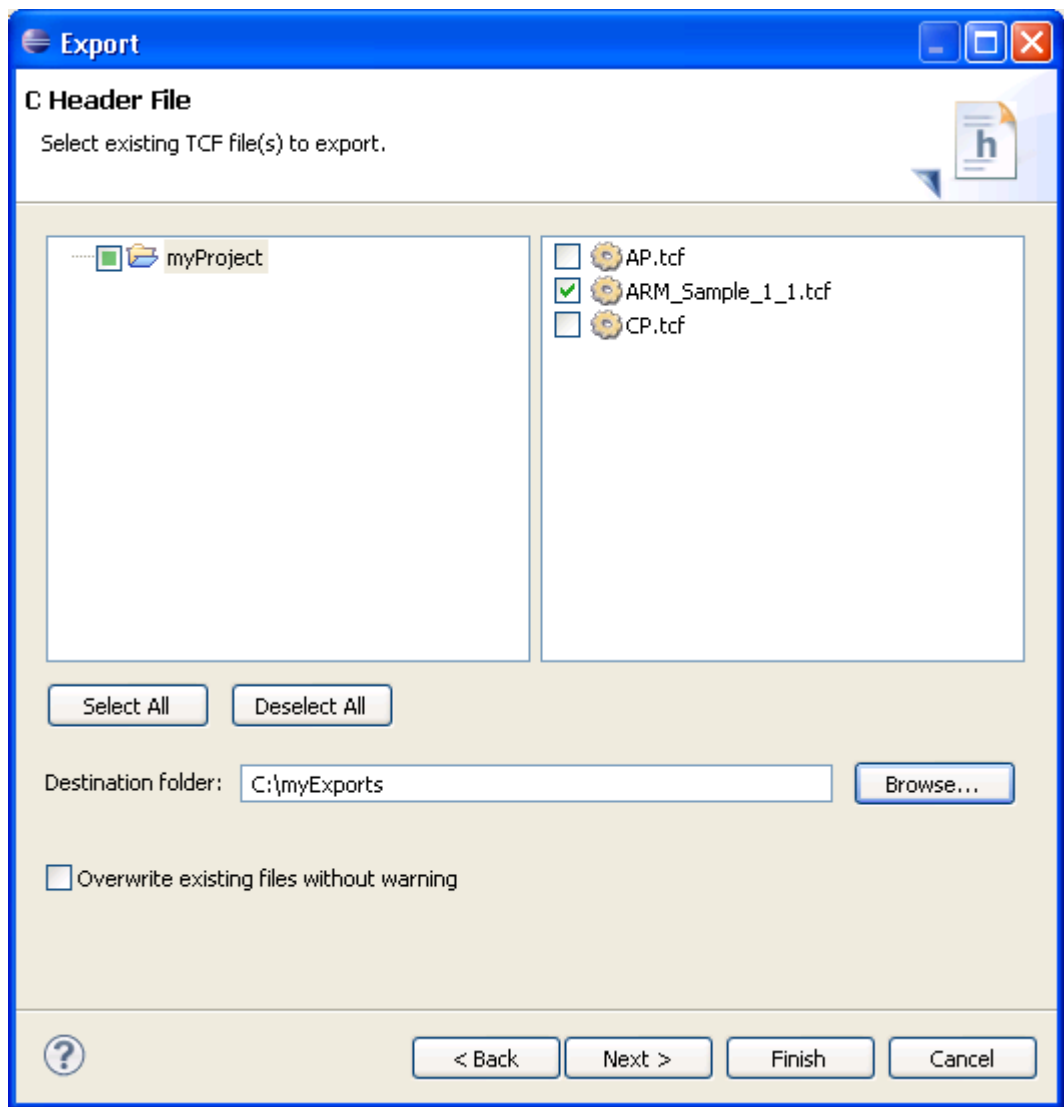


Figure 3-26 Selecting the files

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 3-98.](#)

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 sections:

- *4.1 About loading an image on to the target on page 4-103.*
- *4.2 About loading debug information into the debugger on page 4-105.*
- *4.3 About passing arguments to main() on page 4-107.*
- *4.4 Running an image on page 4-108.*
- *4.5 Working with breakpoints and watchpoints on page 4-109.*
- *4.6 Working with conditional breakpoints on page 4-116.*
- *4.7 About pending breakpoints and watchpoints on page 4-120.*
- *4.8 Setting a tracepoint on page 4-121.*
- *4.9 Setting Streamline start and stop points on page 4-122.*
- *4.10 Stepping through an application on page 4-123.*
- *4.11 Handling Unix signals on page 4-125.*
- *4.12 Handling processor exceptions on page 4-127.*
- *4.13 Configuring the debugger path substitution rules on page 4-129.*

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 not 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 loads the 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.

Set PC to entry point

Sets the PC to the entry point when loading image or debug information so that the code runs from the beginning.

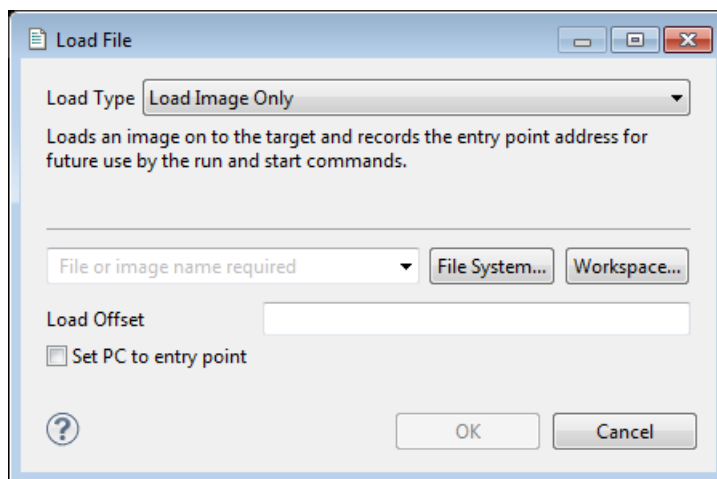


Figure 4-1 Load File dialog box

Related concepts

[4.2 About loading debug information into the debugger on page 4-105.](#)

Related tasks

[2.4 Configuring a connection to a Fixed Virtual Platform \(FVP\) for Linux application debug on page 2-41.](#)

[2.3 Configuring a connection to a Linux target using gdbserver on page 2-39.](#)

[2.6 Configuring a connection to a Linux Kernel on page 2-45.](#)

[2.8 Configuring a connection to a bare-metal target on page 2-52.](#)

[2.9 Configuring an Event Viewer connection to a bare-metal target on page 2-54.](#)

Related references

[10.6 Commands view on page 10-214.](#)

[10.7 Debug Control view on page 10-217.](#)

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.

Loading debug information increases memory use and can take a long time. To minimize these costs, the debugger loads debug information incrementally as it is needed. This is called on-demand loading. Certain operations, such as listing all the symbols in an image, load additional data into the debugger and therefore incur a small delay. Loading of debug information can occur at any time, on-demand, so you must ensure that your images remain accessible to the debugger and do not change during your debug session.

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 loads the 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.

Set PC to entry point

Sets the PC to the entry point when loading image or debug information so that the code runs from the beginning.

Note

The option is not available for the **Add Symbols File** option.

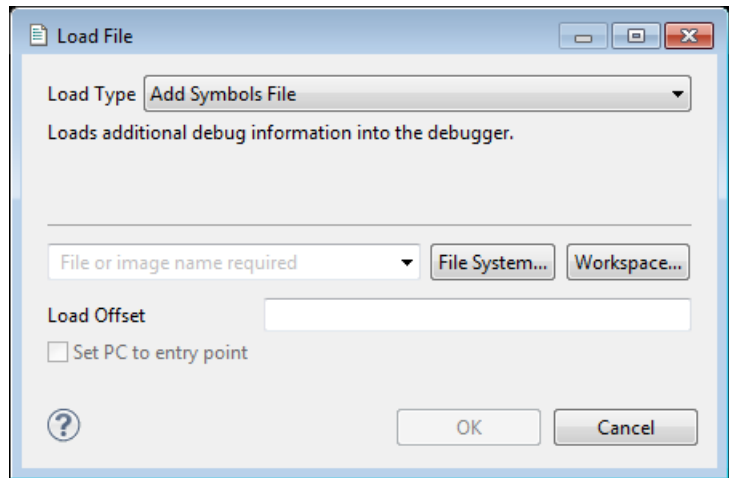


Figure 4-2 Load additional debug information dialog box

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 4-103.](#)

Related tasks

[2.4 Configuring a connection to a Fixed Virtual Platform \(FVP\) for Linux application debug on page 2-41.](#)

[2.3 Configuring a connection to a Linux target using gdbserver on page 2-39.](#)

[2.6 Configuring a connection to a Linux Kernel on page 2-45.](#)

[2.8 Configuring a connection to a bare-metal target on page 2-52.](#)

[2.9 Configuring an Event Viewer connection to a bare-metal target on page 2-54.](#)

Related references

[10.6 Commands view on page 10-214.](#)

[10.7 Debug Control view on page 10-217.](#)

[4.13 Configuring the debugger path substitution rules on page 4-129.](#)

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 7-169.

[7.2 Working with semihosting](#) on page 7-171.

[7.3 Enabling automatic semihosting support in the debugger](#) on page 7-172.

[7.4 Controlling semihosting messages using the command-line console](#) on page 7-173.

[10.42 Debug Configurations - Arguments tab](#) on page 10-304.

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 reside 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 10-214.](#)

[10.7 Debug Control view on page 10-217.](#)

[10.38 Debug Configurations - Connection tab on page 10-293.](#)

[10.39 Debug Configurations - Files tab on page 10-296.](#)

[10.40 Debug Configurations - Debugger tab on page 10-300.](#)

[10.42 Debug Configurations - Arguments tab on page 10-304.](#)

[10.43 Debug Configurations - Environment tab on page 10-306.](#)

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.

This section contains the following subsections:

- [4.5.1 Setting or deleting an execution breakpoint on page 4-110.](#)
- [4.5.2 Setting or removing a data watchpoint on page 4-111.](#)
- [4.5.3 Viewing the properties of a data watchpoint on page 4-112.](#)
- [4.5.4 Importing DS-5 breakpoint settings from a file on page 4-113.](#)
- [4.5.5 Exporting DS-5 breakpoint settings to a file on page 4-113.](#)

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.

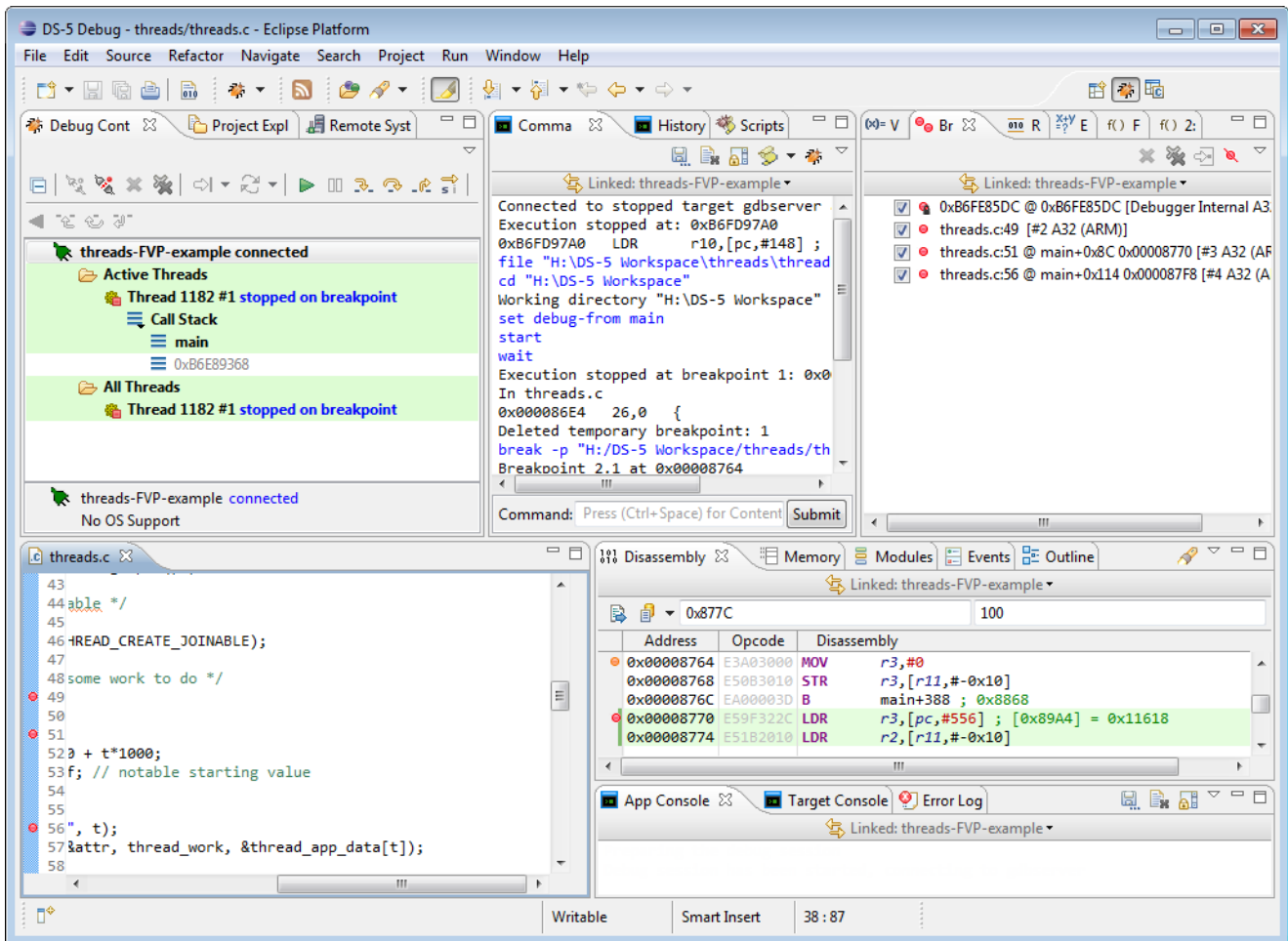


Figure 4-3 Setting an execution breakpoint

Related references

- [4.5 Working with breakpoints and watchpoints on page 4-109.](#)
- [4.5.3 Viewing the properties of a data watchpoint on page 4-112.](#)
- [4.8 Setting a tracepoint on page 4-121.](#)
- [4.9 Setting Streamline start and stop points on page 4-122.](#)
- [4.6 Working with conditional breakpoints on page 4-116.](#)
- [4.6.1 Assigning conditions to an existing breakpoint on page 4-116.](#)
- [4.7 About pending breakpoints and watchpoints on page 4-120.](#)
- [4.5.5 Exporting DS-5 breakpoint settings to a file on page 4-113.](#)
- [4.5.4 Importing DS-5 breakpoint settings from a file on page 4-113.](#)

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

- 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.

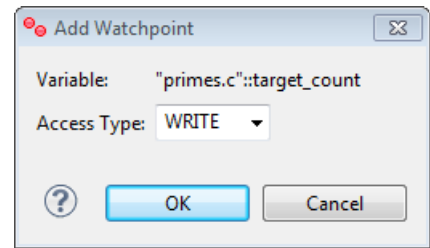


Figure 4-4 Setting a data watchpoint

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:

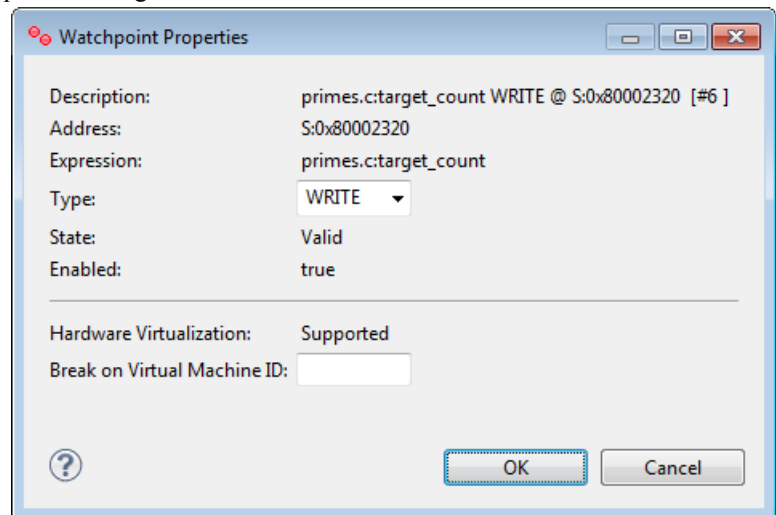


Figure 4-5 Viewing the properties of a data watchpoint

- 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 4-109.](#)
- [4.5.1 Setting or deleting an execution breakpoint on page 4-110.](#)
- [4.8 Setting a tracepoint on page 4-121.](#)
- [4.9 Setting Streamline start and stop points on page 4-122.](#)
- [4.6 Working with conditional breakpoints on page 4-116.](#)
- [4.6.1 Assigning conditions to an existing breakpoint on page 4-116.](#)
- [4.7 About pending breakpoints and watchpoints on page 4-120.](#)

[4.5.5 Exporting DS-5 breakpoint settings to a file on page 4-113.](#)

[4.5.4 Importing DS-5 breakpoint settings from a file on page 4-113.](#)

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 4-109.](#)

[4.5.1 Setting or deleting an execution breakpoint on page 4-110.](#)

[4.5.3 Viewing the properties of a data watchpoint on page 4-112.](#)

[4.8 Setting a tracepoint on page 4-121.](#)

[4.9 Setting Streamline start and stop points on page 4-122.](#)

[4.6 Working with conditional breakpoints on page 4-116.](#)

[4.6.1 Assigning conditions to an existing breakpoint on page 4-116.](#)

[4.7 About pending breakpoints and watchpoints on page 4-120.](#)

[4.5.5 Exporting DS-5 breakpoint settings to a file on page 4-113.](#)

[10.3 ARM assembler editor on page 10-205.](#)

[10.4 Breakpoints view on page 10-207.](#)

[10.5 C/C++ editor on page 10-211.](#)

[10.6 Commands view on page 10-214.](#)

[10.8 Disassembly view on page 10-221.](#)

[10.11 Expressions view on page 10-229.](#)

[10.14 Memory view on page 10-237.](#)

[10.17 Registers view on page 10-248.](#)

[10.26 Variables view on page 10-271.](#)

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 4-109.
[4.5.1 Setting or deleting an execution breakpoint](#) on page 4-110.
[4.5.3 Viewing the properties of a data watchpoint](#) on page 4-112.
[4.8 Setting a tracepoint](#) on page 4-121.
[4.9 Setting Streamline start and stop points](#) on page 4-122.
[4.6 Working with conditional breakpoints](#) on page 4-116.
[4.6.1 Assigning conditions to an existing breakpoint](#) on page 4-116.
[4.7 About pending breakpoints and watchpoints](#) on page 4-120.
[4.5.4 Importing DS-5 breakpoint settings from a file](#) on page 4-113.
[10.3 ARM assembler editor](#) on page 10-205.
[10.4 Breakpoints view](#) on page 10-207.
[10.5 C/C++ editor](#) on page 10-211.
[10.6 Commands view](#) on page 10-214.
[10.8 Disassembly view](#) on page 10-221.
[10.11 Expressions view](#) on page 10-229.
[10.14 Memory view](#) on page 10-237.
[10.17 Registers view](#) on page 10-248.
[10.26 Variables view](#) on page 10-271.

Related information

[DS-5 Debugger commands](#).

Related references

[4.5.1 Setting or deleting an execution breakpoint](#) on page 4-110.
[4.5.3 Viewing the properties of a data watchpoint](#) on page 4-112.
[4.8 Setting a tracepoint](#) on page 4-121.
[4.9 Setting Streamline start and stop points](#) on page 4-122.
[4.6 Working with conditional breakpoints](#) on page 4-116.
[4.6.1 Assigning conditions to an existing breakpoint](#) on page 4-116.
[4.7 About pending breakpoints and watchpoints](#) on page 4-120.
[4.5.5 Exporting DS-5 breakpoint settings to a file](#) on page 4-113.
[4.5.4 Importing DS-5 breakpoint settings from a file](#) on page 4-113.
[10.3 ARM assembler editor](#) on page 10-205.
[10.4 Breakpoints view](#) on page 10-207.
[10.5 C/C++ editor](#) on page 10-211.
[10.6 Commands view](#) on page 10-214.
[10.8 Disassembly view](#) on page 10-221.
[10.11 Expressions view](#) on page 10-229.
[10.14 Memory view](#) on page 10-237.

10.17 Registers view on page 10-248.

10.26 Variables view on page 10-271.

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.

This section contains the following subsections:

- [4.6.1 Assigning conditions to an existing breakpoint on page 4-116.](#)

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.

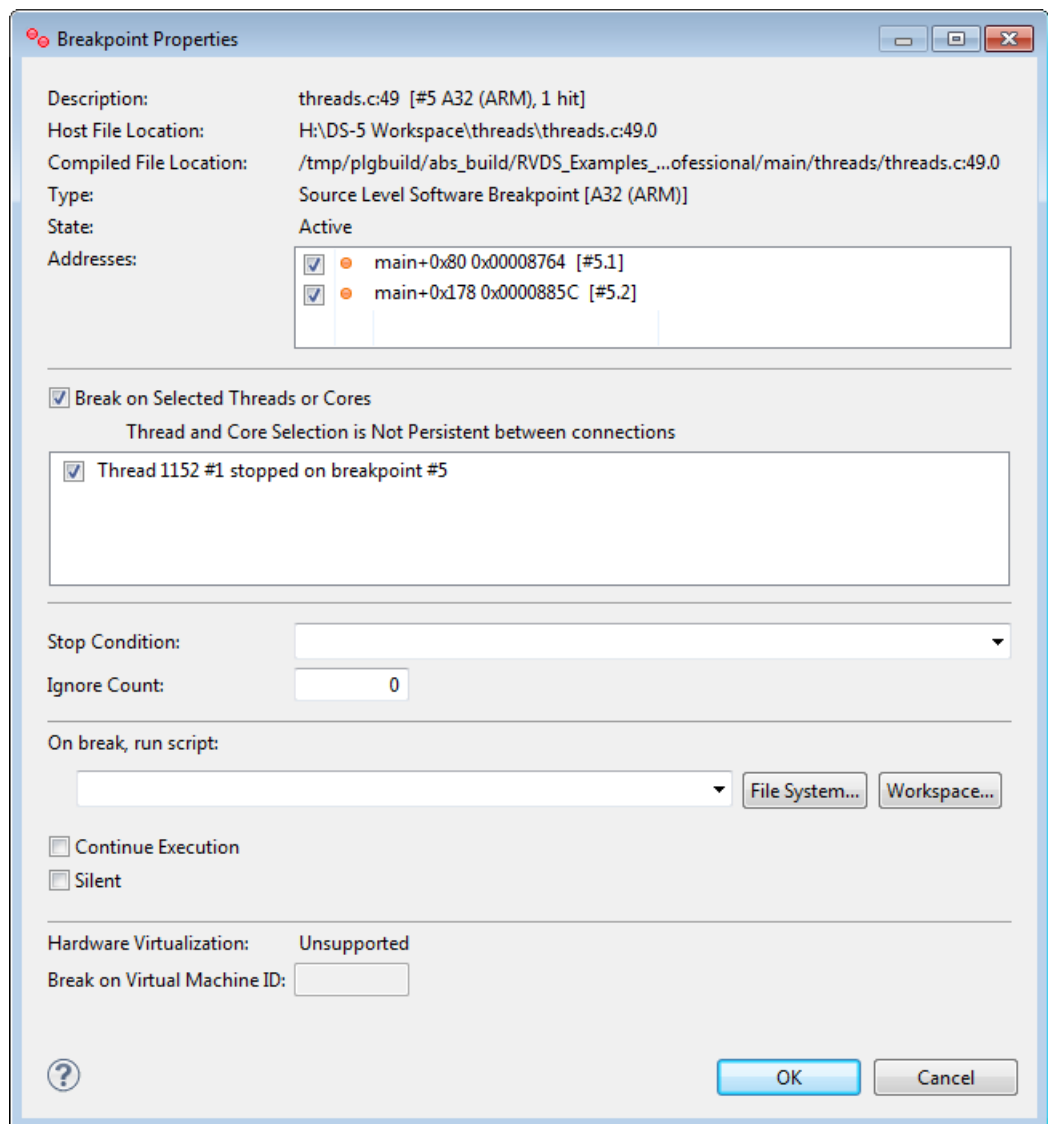


Figure 4-6 Breakpoint Properties dialog

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 4-109.](#)
[4.5.1 Setting or deleting an execution breakpoint on page 4-110.](#)
[4.5.3 Viewing the properties of a data watchpoint on page 4-112.](#)
[4.8 Setting a tracepoint on page 4-121.](#)
[4.9 Setting Streamline start and stop points on page 4-122.](#)
[4.6 Working with conditional breakpoints on page 4-116.](#)
[4.7 About pending breakpoints and watchpoints on page 4-120.](#)
[4.5.5 Exporting DS-5 breakpoint settings to a file on page 4-113.](#)
[4.5.4 Importing DS-5 breakpoint settings from a file on page 4-113.](#)
[10.3 ARM assembler editor on page 10-205.](#)
[10.4 Breakpoints view on page 10-207.](#)
[10.5 C/C++ editor on page 10-211.](#)
[10.6 Commands view on page 10-214.](#)
[10.8 Disassembly view on page 10-221.](#)
[10.11 Expressions view on page 10-229.](#)
[10.14 Memory view on page 10-237.](#)
[10.17 Registers view on page 10-248.](#)
[10.26 Variables view on page 10-271.](#)

Related information

DS-5 Debugger commands.

Related references

[4.5 Working with breakpoints and watchpoints on page 4-109.](#)
[4.5.1 Setting or deleting an execution breakpoint on page 4-110.](#)
[4.5.3 Viewing the properties of a data watchpoint on page 4-112.](#)
[4.8 Setting a tracepoint on page 4-121.](#)
[4.9 Setting Streamline start and stop points on page 4-122.](#)
[4.6.1 Assigning conditions to an existing breakpoint on page 4-116.](#)
[4.7 About pending breakpoints and watchpoints on page 4-120.](#)
[4.5.5 Exporting DS-5 breakpoint settings to a file on page 4-113.](#)
[4.5.4 Importing DS-5 breakpoint settings from a file on page 4-113.](#)
[10.3 ARM assembler editor on page 10-205.](#)
[10.4 Breakpoints view on page 10-207.](#)
[10.5 C/C++ editor on page 10-211.](#)

10.6 Commands view on page 10-214.
10.8 Disassembly view on page 10-221.
10.11 Expressions view on page 10-229.
10.14 Memory view on page 10-237.
10.17 Registers view on page 10-248.
10.26 Variables view on page 10-271.

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.

Example 4-1 Examples

To manually set a pending 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 4-109.](#)
- [4.5.1 Setting or deleting an execution breakpoint on page 4-110.](#)
- [4.5.3 Viewing the properties of a data watchpoint on page 4-112.](#)
- [4.8 Setting a tracepoint on page 4-121.](#)
- [4.9 Setting Streamline start and stop points on page 4-122.](#)
- [4.6 Working with conditional breakpoints on page 4-116.](#)
- [4.6.1 Assigning conditions to an existing breakpoint on page 4-116.](#)
- [4.5.5 Exporting DS-5 breakpoint settings to a file on page 4-113.](#)
- [4.5.4 Importing DS-5 breakpoint settings from a file on page 4-113.](#)
- [10.3 ARM assembler editor on page 10-205.](#)
- [10.4 Breakpoints view on page 10-207.](#)
- [10.5 C/C++ editor on page 10-211.](#)
- [10.6 Commands view on page 10-214.](#)
- [10.8 Disassembly view on page 10-221.](#)
- [10.11 Expressions view on page 10-229.](#)
- [10.14 Memory view on page 10-237.](#)
- [10.17 Registers view on page 10-248.](#)
- [10.26 Variables view on page 10-271.](#)

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 4-109.](#)
- [4.5.1 Setting or deleting an execution breakpoint on page 4-110.](#)
- [4.5.3 Viewing the properties of a data watchpoint on page 4-112.](#)
- [4.9 Setting Streamline start and stop points on page 4-122.](#)
- [4.6 Working with conditional breakpoints on page 4-116.](#)
- [4.6.1 Assigning conditions to an existing breakpoint on page 4-116.](#)
- [4.7 About pending breakpoints and watchpoints on page 4-120.](#)
- [4.5.5 Exporting DS-5 breakpoint settings to a file on page 4-113.](#)
- [4.5.4 Importing DS-5 breakpoint settings from a file on page 4-113.](#)
- [10.3 ARM assembler editor on page 10-205.](#)
- [10.4 Breakpoints view on page 10-207.](#)
- [10.5 C/C++ editor on page 10-211.](#)
- [10.6 Commands view on page 10-214.](#)
- [10.8 Disassembly view on page 10-221.](#)
- [10.11 Expressions view on page 10-229.](#)
- [10.14 Memory view on page 10-237.](#)
- [10.17 Registers view on page 10-248.](#)
- [10.26 Variables view on page 10-271.](#)

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 4-109.](#)
- [4.5.1 Setting or deleting an execution breakpoint on page 4-110.](#)
- [4.5.3 Viewing the properties of a data watchpoint on page 4-112.](#)
- [4.8 Setting a tracepoint on page 4-121.](#)
- [4.6 Working with conditional breakpoints on page 4-116.](#)
- [4.6.1 Assigning conditions to an existing breakpoint on page 4-116.](#)
- [4.7 About pending breakpoints and watchpoints on page 4-120.](#)
- [4.5.5 Exporting DS-5 breakpoint settings to a file on page 4-113.](#)
- [4.5.4 Importing DS-5 breakpoint settings from a file on page 4-113.](#)
- [10.3 ARM assembler editor on page 10-205.](#)
- [10.5 C/C++ editor on page 10-211.](#)

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.

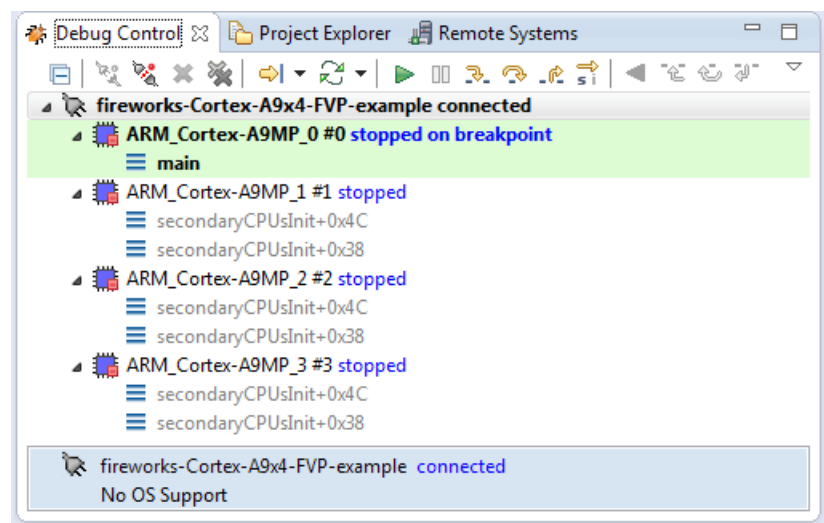


Figure 4-7 Debug Control view

Example 4-2 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
stepi 5          # Execute five instructions
```

Related concepts

[6.9 About debugging shared libraries on page 6-150.](#)

6.10.2 About debugging a Linux kernel on page 6-153.

6.10.3 About debugging Linux kernel modules on page 6-154.

Related references

5.1 Examining the target execution environment on page 5-132.

5.2 Examining the call stack on page 5-134.

4.11 Handling Unix signals on page 4-125.

4.12 Handling processor exceptions on page 4-127.


4.11 Handling Unix signals

For Linux applications, ARM processors have the facility to trap Unix signals.

To manage Unix signals in the debugger, either:

- Select **Manage Signals** from the **Breakpoints** view menu. Select the individual **Signal** you want to **Stop** or **Print** information, and click **OK**. The results are displayed in the Command view.
- Use the `handle` command and view the results in the Command view.

Tip

 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.

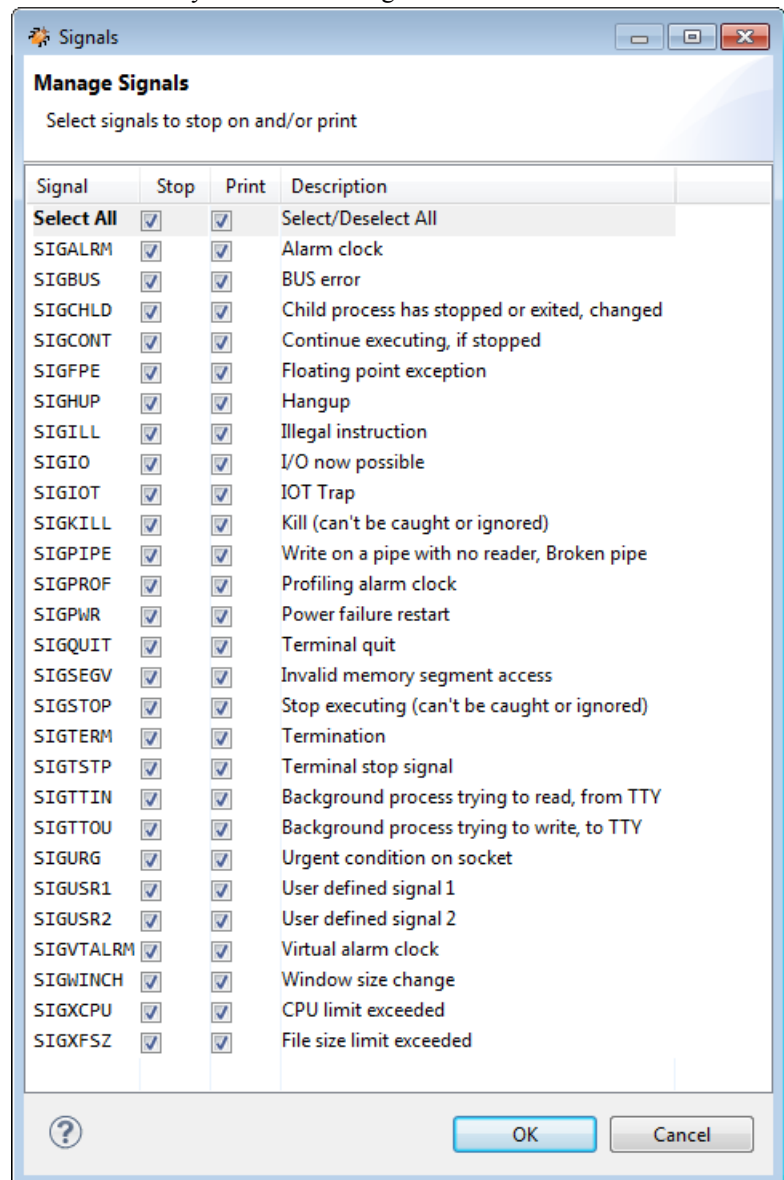


Figure 4-8 Manage signals dialog (Unix signals)

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.

Example 4-3 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 debugged application ignores the signal and continues to operate.

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

[6.9 About debugging shared libraries](#) on page 6-150.

[6.10.2 About debugging a Linux kernel](#) on page 6-153.

[6.10.3 About debugging Linux kernel modules](#) on page 6-154.

Related references

[4.10 Stepping through an application](#) on page 4-123.

[5.1 Examining the target execution environment](#) on page 5-132.

[5.2 Examining the call stack](#) on page 5-134.

[4.12 Handling processor exceptions](#) on page 4-127.

[10.4 Breakpoints view](#) on page 10-207.

[10.6 Commands view](#) on page 10-214.

[10.35 Manage Signals dialog box](#) on page 10-289.

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. Select the individual **Signal** you want to **Stop** or **Print** information, and click **OK**. The results are displayed in the Command view.
- Use the `handle` command and view the results in the Command view.

Tip

 You can also use the `info signals` command to display the current handler settings.

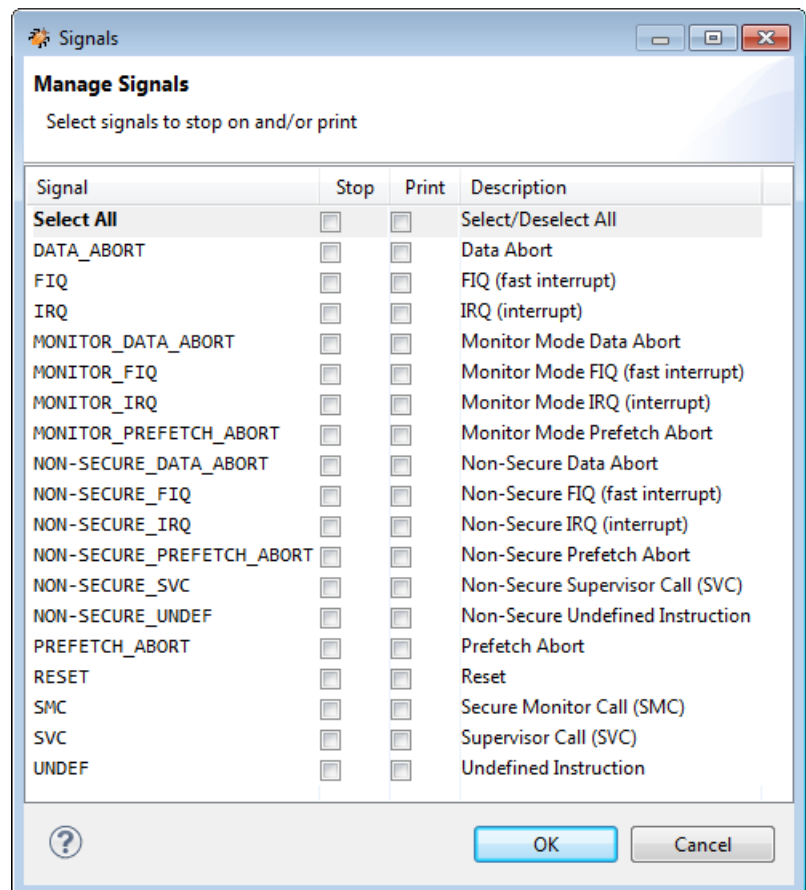


Figure 4-9 Manage signals dialog

Note

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

Example 4-4 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

[6.9 About debugging shared libraries](#) on page 6-150.

[6.10.2 About debugging a Linux kernel](#) on page 6-153.

[6.10.3 About debugging Linux kernel modules](#) on page 6-154.

Related references

[4.10 Stepping through an application](#) on page 4-123.

[5.1 Examining the target execution environment](#) on page 5-132.

[5.2 Examining the call stack](#) on page 5-134.

[4.11 Handling Unix signals](#) on page 4-125.

[10.4 Breakpoints view](#) on page 10-207.

[10.6 Commands view](#) on page 10-214.

[10.35 Manage Signals dialog box](#) on page 10-289.

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.

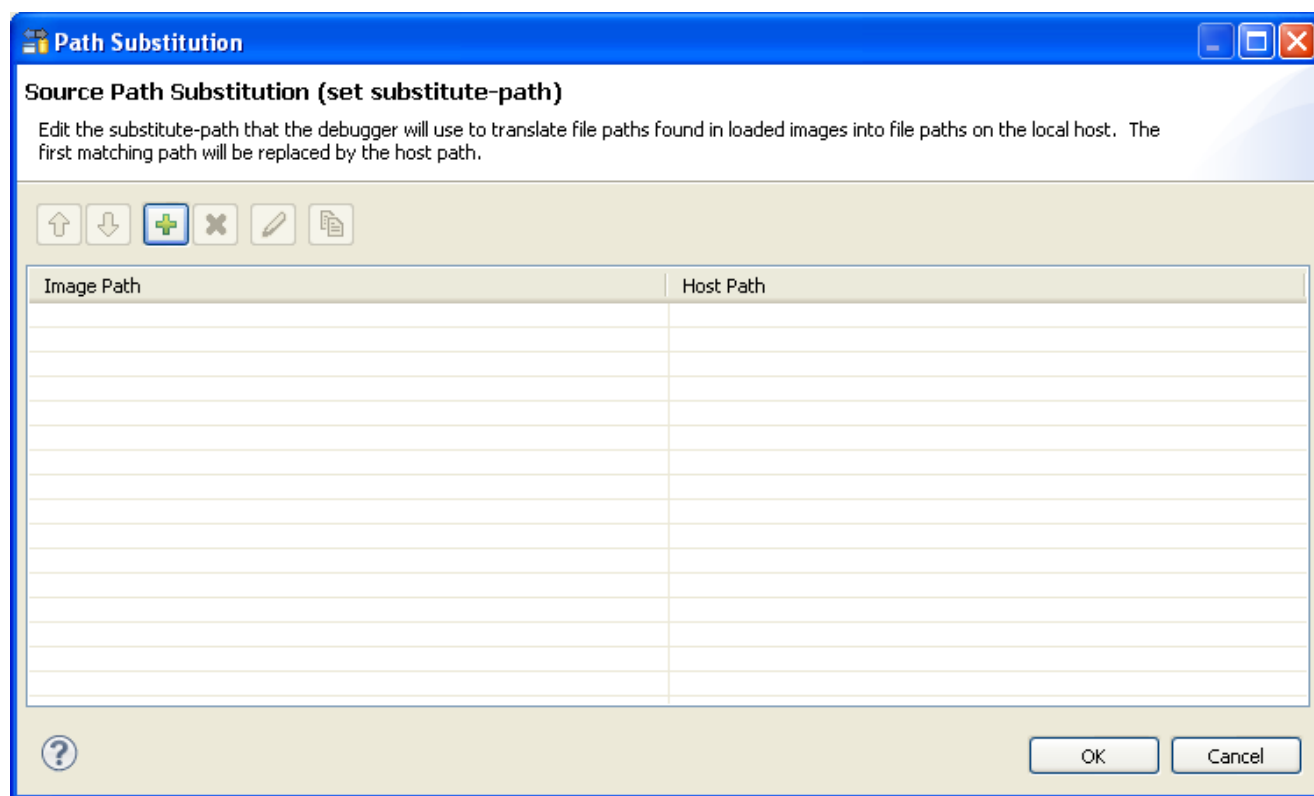


Figure 4-10 Path Substitution dialog box

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**.

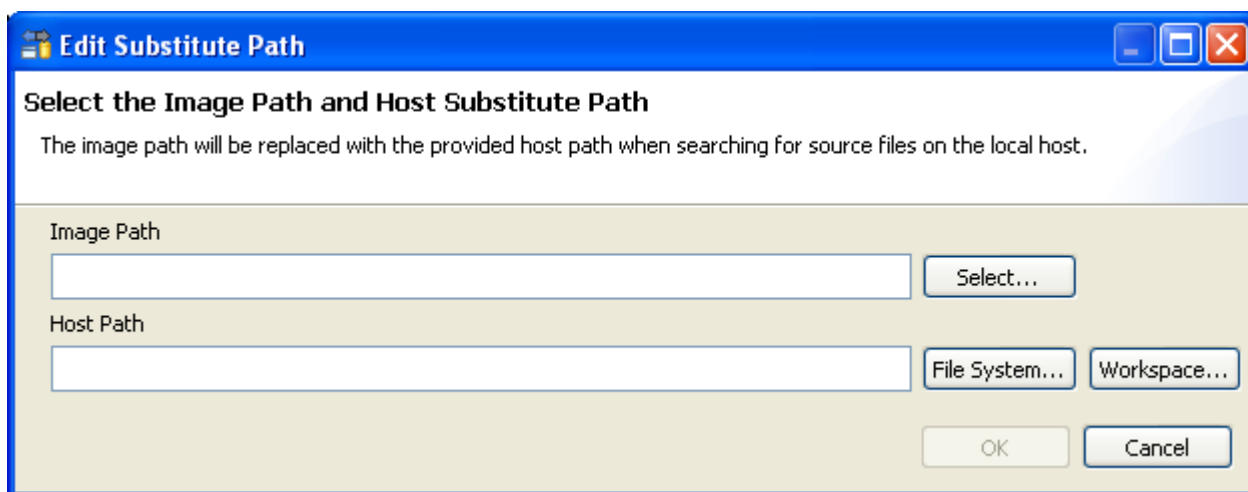


Figure 4-11 Edit Substitute Path dialog box

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 4-105.

Chapter 5

Examining the Target

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

It contains the following sections:

- [5.1 Examining the target execution environment on page 5-132.](#)
- [5.2 Examining the call stack on page 5-134.](#)
- [5.3 About trace support on page 5-135.](#)
- [5.4 About post-mortem debugging of trace data on page 5-138.](#)

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.

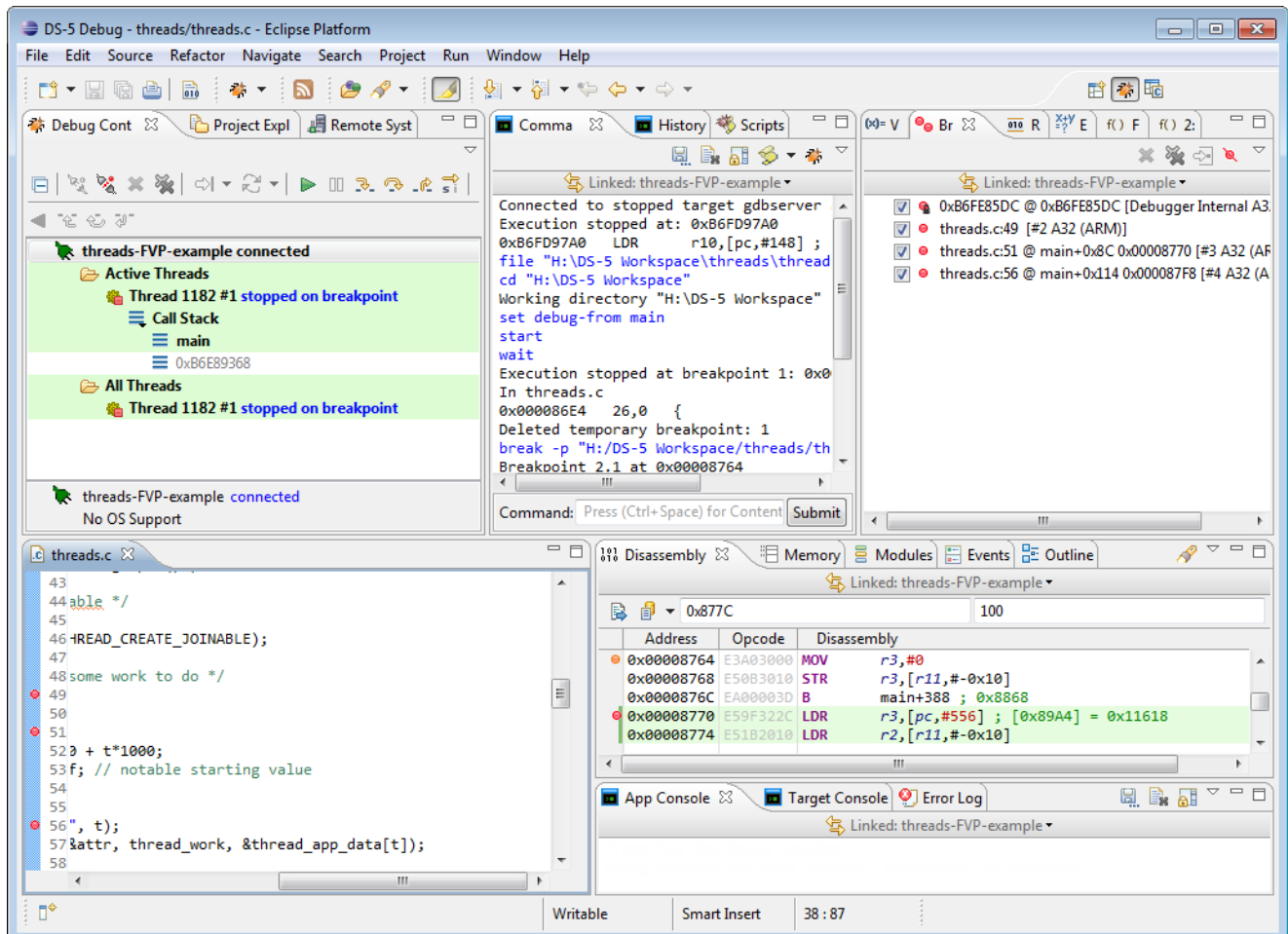


Figure 5-1 Target execution environment

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

- [6.9 About debugging shared libraries on page 6-150.](#)
- [6.10.2 About debugging a Linux kernel on page 6-153.](#)
- [6.10.3 About debugging Linux kernel modules on page 6-154.](#)

Related references

- [4.10 Stepping through an application on page 4-123.](#)
- [5.2 Examining the call stack on page 5-134.](#)

4.11 Handling Unix signals on page 4-125.
4.12 Handling processor exceptions on page 4-127.

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.

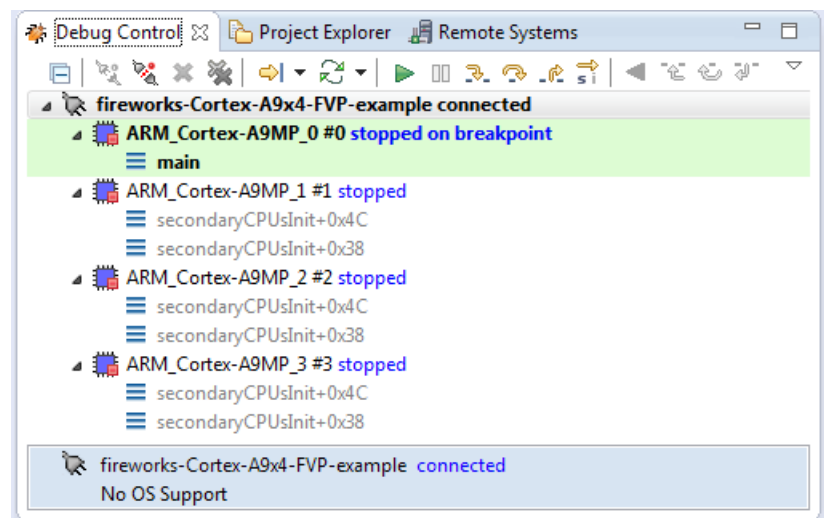


Figure 5-2 Debug Control view

Related concepts

[6.9 About debugging shared libraries](#) on page 6-150.

[6.10.2 About debugging a Linux kernel](#) on page 6-153.

[6.10.3 About debugging Linux kernel modules](#) on page 6-154.

Related references

[4.10 Stepping through an application](#) on page 4-123.

[5.1 Examining the target execution environment](#) on page 5-132.

[4.11 Handling Unix signals](#) on page 4-125.

[4.12 Handling processor exceptions](#) on page 4-127.

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 your source code 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 your source code 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 5-138.](#)

[1.5 About DS-5 headless command-line debugger on page 1-23.](#)

Related tasks

[1.7 Specifying a custom configuration database using the headless command-line debugger on page 1-29.](#)

Related references

[1.6 Headless command-line debugger options on page 1-24.](#)

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 5-135.](#)

[1.5 About DS-5 headless command-line debugger on page 1-23.](#)

Related tasks

[1.7 Specifying a custom configuration database using the headless command-line debugger on page 1-29.](#)

Related references

[1.6 Headless command-line debugger options on page 1-24.](#)

Chapter 6

Debugging Embedded Systems

Gives an introduction to debugging embedded systems.

It contains the following sections:

- [6.1 About endianness on page 6-140.](#)
- [6.2 About accessing AHB, APB, and AXI buses on page 6-141.](#)
- [6.3 About virtual and physical memory on page 6-142.](#)
- [6.4 About address spaces on page 6-143.](#)
- [6.5 About debugging hypervisors on page 6-145.](#)
- [6.6 About debugging big.LITTLE systems on page 6-146.](#)
- [6.7 About debugging bare-metal symmetric multiprocessing systems on page 6-147.](#)
- [6.8 About debugging multi-threaded applications on page 6-149.](#)
- [6.9 About debugging shared libraries on page 6-150.](#)
- [6.10 About OS awareness on page 6-152.](#)
- [6.11 About debugging TrustZone enabled targets on page 6-157.](#)
- [6.12 About debugging a Unified Extensible Firmware Interface \(UEFI\) on page 6-159.](#)
- [6.13 About application rewind on page 6-160.](#)
- [6.14 About debugging MMUs on page 6-161.](#)
- [6.15 About Debug and Trace Services Layer \(DTSL\) on page 6-163.](#)
- [6.16 About CoreSight™ Target Access Library on page 6-164.](#)
- [6.17 About debugging caches on page 6-165.](#)

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
```

Each address space has a size, which is the number of bits that comprise the address. Common address space size on embedded and low-end devices is 32-bits, higher-end devices that require more memory might use > 32-bits. As an example, some devices based around ARM architecture ARMv7 make use of LPAE (Large Physical Address Extensions) to extend physical addresses on the AXI bus to 40-bits, even though virtual addresses within the processor are 32-bits.

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 accesses 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 10-214.](#)

[10.7 Debug Control view on page 10-217.](#)

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 the 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 10-214.](#)

[10.7 Debug Control view on page 10-217.](#)

Related information

[DS-5 Debugger commands.](#)

6.4 About address spaces

You can use address space prefixes in DS-5 Debugger to refer to different addresses spaces. You can use these address space prefixes for various debugging activities such as to:

- set breakpoint in a specific memory space
- read or write memory
- load symbols associated with a specific memory space.

DS-5 Debugger also uses these prefixes when reporting the current memory space where the execution stopped.

Note

The address spaces can be different on different targets. The availability of an address space depends on what architecture features are implemented, such as security extensions.

Address spaces in ARMv7 based processors

The following address space prefixes might be available for ARMv7 based processors:

- S: This corresponds to the secure address space.
- H: This corresponds to the hypervisor address space.
- N: This corresponds to the non-secure address space.
- SP: This corresponds to secure world physical memory.
- NP: This corresponds to non-secure world physical memory.

The following are examples of DS-5 Debugger commands with address spaces for ARMv7 based processors:

- `break S:main`
- `x N:0x80000000`
- `add-symbol-file foo.axf SP:0.`

When execution stops, DS-5 Debugger reports the current memory space, for example:

- Execution stopped in SVC mode at S:0x80000000
- Execution stopped in SYS mode at breakpoint 1: S:0x8000BA8.

Address spaces in ARMv8 based processors

The following address space prefixes might be available for ARMv8 based processors when in the AArch64 execution state:

- EL3: This corresponds to the EL3 translation regime. This is a secure address space.
- EL2: This corresponds to the EL2 translation regime. This is a non-secure address space.
- EL1S: This corresponds to the Secure EL1 and Secure EL0 translation regimes.
- EL1N: This corresponds to the Non-secure EL1 and Non-secure EL0 translation regimes.
- SP: This corresponds to Secure world physical memory.
- NP: This corresponds to Non-secure world physical memory.

The following address space prefixes might be available for ARMv8 based processors when in the AArch32 execution state:

- S: This corresponds to the EL3, Secure EL1, and Secure EL0 translation regimes.
- H: This corresponds to the EL2 translation regime. This is a Non-secure address space.
- N: This corresponds to the Non-secure EL1 and Non-secure EL0 translation regimes.
- SP: This corresponds to Secure world physical memory.
- NP: This corresponds to Non-secure world physical memory.

The following are examples of DS-5 Debugger commands with address spaces for ARMv8 based processors:

- `break EL1N:main`
- `x EL1S:0x80000000`
- `add-symbol-file foo.axf SP:0.`

When execution stops, DS-5 Debugger reports the current memory space, for example:

- Execution stopped in EL3h mode at: `EL3:0x0000000080001500`
- Execution stopped in EL1h mode at breakpoint 2.2: `EL1N:0x000000008000F6C`

If the core is stopped in exception level EL3, the debugger cannot reliably determine whether the translation regime at EL1/EL0 is configured for secure or non-secure access. This is because the secure monitor can change this at run-time when switching between secure and non-secure worlds. Memory accesses from EL3, such as setting software breakpoints at EL1S: or EL1N: addresses, might cause corruption or the target to lockup.

The memory spaces for the EL1 and EL0 exception levels have the same prefix because the same translation tables are used for both EL0 and EL1. These translation tables are used for either Secure EL1/EL0 or Non-secure EL1/EL0. The consequence of this is that if the core, in AArch64 state, is stopped in EL0 in secure state, then the debugger reports: Execution stopped in EL0h mode at: `EL1S:0x0000000000000000`.

————— **Note** —————

The reported `EL1S:` here refers to the memory space that is common to EL0 and EL1. It does not refer to the exception level.

Related concepts

[6.2 About accessing AHB, APB, and AXI buses on page 6-141.](#)

[6.3 About virtual and physical memory on page 6-142.](#)

[6.5 About debugging hypervisors on page 6-145.](#)

6.5 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 10-214.](#)

[10.7 Debug Control view on page 10-217.](#)

Related information

[DS-5 Debugger commands.](#)

6.6 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

[6.7 About debugging bare-metal symmetric multiprocessing systems on page 6-147.](#)

Related references

[10.6 Commands view on page 10-214.](#)

[10.7 Debug Control view on page 10-217.](#)

Related information

[DS-5 Debugger commands.](#)

6.7 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.

Note

It is recommended to always use an SMP connection when debugging an SMP system. Using a single-core connection instead of an SMP connection might result in other cores halting on software breakpoints with no way to resume them.

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

- [6.6 About debugging big.LITTLE systems on page 6-146.](#)
- [4.1 About loading an image on to the target on page 4-103.](#)
- [4.2 About loading debug information into the debugger on page 4-105.](#)

Related references

- [4.5 Working with breakpoints and watchpoints on page 4-109.](#)
- [4.8 Setting a tracepoint on page 4-121.](#)
- [4.9 Setting Streamline start and stop points on page 4-122.](#)
- [4.6 Working with conditional breakpoints on page 4-116.](#)
- [4.6.1 Assigning conditions to an existing breakpoint on page 4-116.](#)
- [4.7 About pending breakpoints and watchpoints on page 4-120.](#)
- [4.10 Stepping through an application on page 4-123.](#)
- [10.4 Breakpoints view on page 10-207.](#)
- [10.6 Commands view on page 10-214.](#)
- [10.7 Debug Control view on page 10-217.](#)
- [10.8 Disassembly view on page 10-221.](#)
- [10.14 Memory view on page 10-237.](#)
- [10.16 Modules view on page 10-246.](#)
- [10.17 Registers view on page 10-248.](#)
- [10.26 Variables view on page 10-271.](#)

Related information

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

6.8 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.

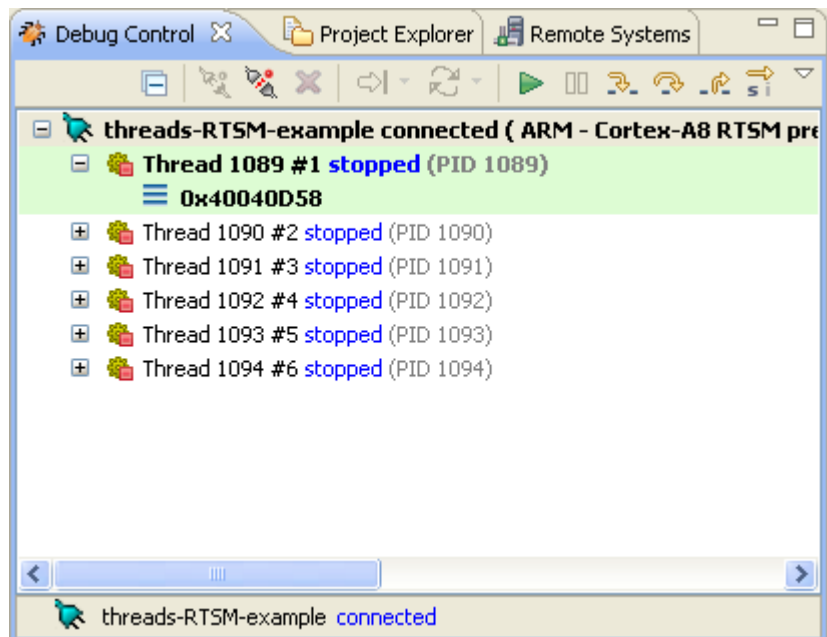


Figure 6-1 Threading call stacks in the Debug Control view

Related references

- [10.4 Breakpoints view on page 10-207.](#)
- [10.6 Commands view on page 10-214.](#)
- [10.7 Debug Control view on page 10-217.](#)
- [10.8 Disassembly view on page 10-221.](#)
- [10.14 Memory view on page 10-237.](#)
- [10.16 Modules view on page 10-246.](#)
- [10.17 Registers view on page 10-248.](#)
- [10.26 Variables view on page 10-271.](#)

6.9 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.

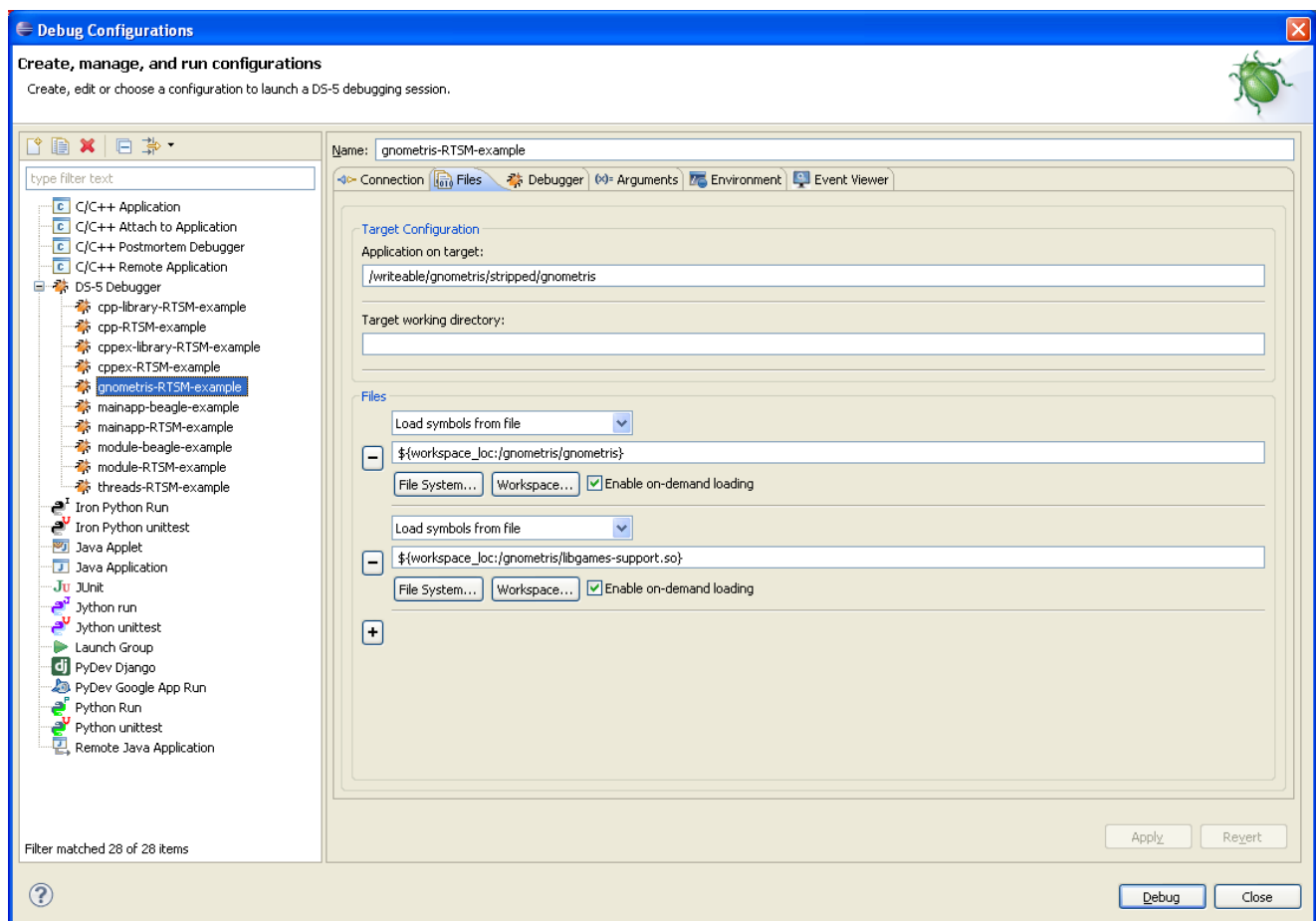


Figure 6-2 Adding individual shared library files

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.

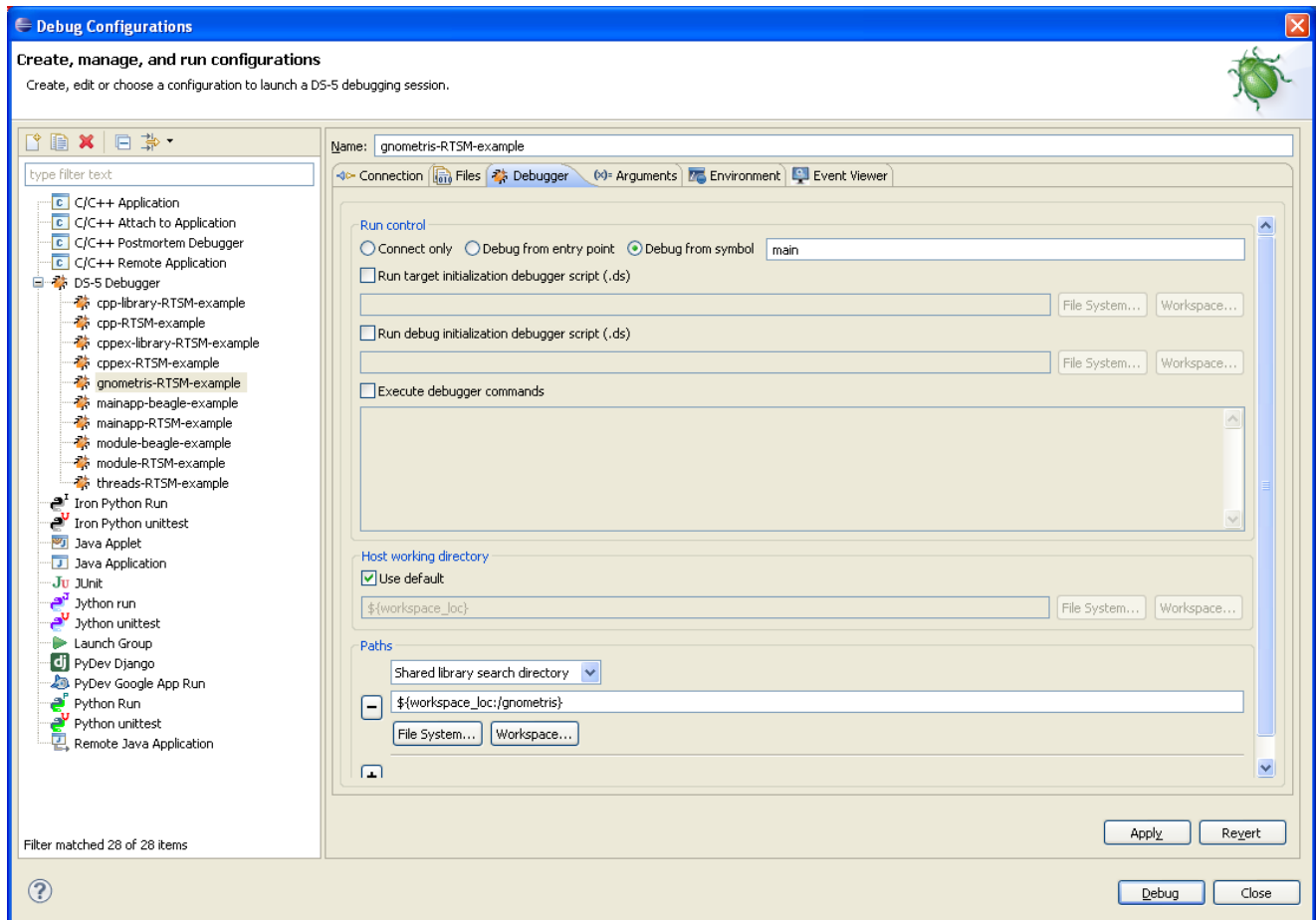


Figure 6-3 Modifying the shared library search paths

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 4-123.](#)
- [5.1 Examining the target execution environment on page 5-132.](#)
- [5.2 Examining the call stack on page 5-134.](#)
- [4.11 Handling Unix signals on page 4-125.](#)
- [4.12 Handling processor exceptions on page 4-127.](#)
- [10.4 Breakpoints view on page 10-207.](#)
- [10.6 Commands view on page 10-214.](#)
- [10.7 Debug Control view on page 10-217.](#)
- [10.8 Disassembly view on page 10-221.](#)
- [10.14 Memory view on page 10-237.](#)
- [10.16 Modules view on page 10-246.](#)
- [10.17 Registers view on page 10-248.](#)
- [10.26 Variables view on page 10-271.](#)

6.10 About OS awareness

DS-5 provides support for a number of operating systems that can run on the target. This is called OS awareness and it provides a representation of the operating system threads and other relevant data structures.

The OS awareness support in DS-5 Debugger depends on the OS version and the processor architecture on the target.

DS-5 Debugger provides OS awareness for:

- ThreadX 5.6: ARMv5, ARMv5T, ARMv5TE, ARMv5TEJ, ARMv6M, ARMv7M, ARMv7R, ARMv7A
- µC/OS-II 2.92: ARMv6M, ARMv7M, ARMv7R, ARMv7A
- µC/OS-III 3.04: ARMv6M, ARMv7M, ARMv7R, ARMv7A
- embOS 3.88: ARMv5, ARMv5T, ARMv5TE, ARMv5TEJ, ARMv6M, ARMv7M, ARMv7R, ARMv7A
- Keil CMSIS-RTOS RTX 4.7: Cortex-M0, Cortex-M0+, Cortex-M1, Cortex-M3, Cortex-M4, and Cortex-A9 processors
- FreeRTOS 8.0: ARMv6M, ARMv7M, ARMv7R, ARMv7A
- Freescale MQX 4.0: Freescale-based Cortex-M4 and Cortex-A5 processors
- Quadros RTXC 1.0.2: ARMv5, ARMv5T, ARMv5TE, ARMv5TEJ, ARMv7M, ARMv7R, ARMv7A.
- Nucleus RTOS 2014.06: ARMv5, ARMv5T, ARMv5TE, ARMv5TEJ, ARMv6M, ARMv7M, ARMv7R, ARMv7A.

Note

- By default, OS awareness is not present for an architecture or processor that is not listed above.
 - OS awareness support for newer versions of the OS depends on the scope of changes to their internal data structures.
 - OS awareness in DS-5 Debugger does not support the original non-CMSIS Keil RTX.
-

The Linux kernels that DS-5 Debugger provides OS awareness for are:

- Linux 2.6.28, ARMv7A
- Linux 2.6.38: ARMv7A
- Linux 3.0: ARMv7A
- Linux 3.11.0-rc6: ARMv7A
- Linux 3.13.0-rc3: ARMv7A
- Linux 3.6.0-rc6: ARMv7A
- Linux 3.7.0: ARMv7A
- Linux 3.9.0-rc3: ARMv7A
- Linux 3.11.0-rc6: ARMv8A.

Note

Later versions of Linux are expected to work on ARMv7A and ARMv8A architectures.

This section contains the following subsections:

- [6.10.1 About debugging FreeRTOS™ on page 6-152.](#)
- [6.10.2 About debugging a Linux kernel on page 6-153.](#)
- [6.10.3 About debugging Linux kernel modules on page 6-154.](#)
- [6.10.4 About debugging ThreadX on page 6-156.](#)

6.10.1 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.

6.10.2 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` Enables the kernel debug options.
 - `CONFIG_DEBUG_INFO=y` Builds `vmlinux` with debugging information.
 - `CONFIG_DEBUG_INFO_REDUCED=n` Includes full debugging information when compiling the kernel.
 - `CONFIG_PERF_EVENTS=n`
Disables the performance events subsystem. Some implementations of the performance events subsystem internally make use of hardware breakpoints, disrupting the use of hardware breakpoints set by the debugger. It is recommended to disable this option if you observe the debugger failing to hit hardware breakpoints or failing to report kernel module load and unload events.

Note

If you are working with Streamline, this option must be enabled.

Compiling the kernel source generates a Linux kernel image and symbol files which contain debug information.

Note

Be aware that:

- Other options might be required depending on the type of debugging you want to perform. Check the kernel documentation for details.
- 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.

————— **Note** —————

If the Linux kernel you are debugging runs on multiple cores, then it is recommended to select an SMP connection type when connecting the debugger. Using a single-core connection instead of an SMP connection might result in other cores halting on software breakpoints with no way to resume them.

-
4. Debug the kernel as required.

Related concepts

[6.10.3 About debugging Linux kernel modules on page 6-154.](#)

[6.7 About debugging bare-metal symmetric multiprocessing systems on page 6-147.](#)

Related tasks

[2.6 Configuring a connection to a Linux Kernel on page 2-45.](#)

Related references

[4.10 Stepping through an application on page 4-123.](#)

[5.1 Examining the target execution environment on page 5-132.](#)

[5.2 Examining the call stack on page 5-134.](#)

[4.11 Handling Unix signals on page 4-125.](#)

[4.12 Handling processor exceptions on page 4-127.](#)

[10.39 Debug Configurations - Files tab on page 10-296.](#)

[10.40 Debug Configurations - Debugger tab on page 10-300.](#)

[10.4 Breakpoints view on page 10-207.](#)

[10.6 Commands view on page 10-214.](#)

[10.7 Debug Control view on page 10-217.](#)

[10.8 Disassembly view on page 10-221.](#)

[10.14 Memory view on page 10-237.](#)

[10.16 Modules view on page 10-246.](#)

[10.17 Registers view on page 10-248.](#)

[10.26 Variables view on page 10-271.](#)

Related information

[Debugging a loadable kernel module.](#)

6.10.3 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 after loading the module debug information into the debugger. For example, you can load the debug information using `add-symbol-file modex.ko`. To set a source-level breakpoint in a module before it is loaded into the kernel, use `break -p` to create a pending breakpoint. When the kernel loads the module, the debugger loads the symbols and applies the pending breakpoint.

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 `rmmod` 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

[6.10.2 About debugging a Linux kernel on page 6-153.](#)

[6.7 About debugging bare-metal symmetric multiprocessing systems on page 6-147.](#)

Related tasks

[2.6 Configuring a connection to a Linux Kernel on page 2-45.](#)

Related references

[4.10 Stepping through an application on page 4-123.](#)

[5.1 Examining the target execution environment on page 5-132.](#)

[5.2 Examining the call stack on page 5-134.](#)

[4.11 Handling Unix signals on page 4-125.](#)

[4.12 Handling processor exceptions on page 4-127.](#)

[10.4 Breakpoints view on page 10-207.](#)

[10.6 Commands view on page 10-214.](#)

[10.7 Debug Control view on page 10-217.](#)

[10.8 Disassembly view on page 10-221.](#)

[10.14 Memory view on page 10-237.](#)

[10.16 Modules view on page 10-246.](#)

[10.17 Registers view on page 10-248.](#)

[10.26 Variables view on page 10-271.](#)

Related information

[Debugging a loadable kernel module.](#)

6.10.4 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.

6.11 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, with a prefix. For example `N:0x1000` or `S: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 10-207.](#)
- [10.6 Commands view on page 10-214.](#)
- [10.7 Debug Control view on page 10-217.](#)
- [10.8 Disassembly view on page 10-221.](#)
- [10.14 Memory view on page 10-237.](#)
- [10.16 Modules view on page 10-246.](#)
- [10.17 Registers view on page 10-248.](#)
- [10.26 Variables view on page 10-271.](#)

Related information

DS-5 Debugger commands.

ARM Security Technology Building a Secure System using TrustZone Technology.

Technical Reference Manual.

Architecture Reference Manual.

6.12 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 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](#)

6.13 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 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

[2.7.1 Connecting to an existing application and application rewind session on page 2-47.](#)

[2.7.2 Downloading your application and application rewind server on the target system on page 2-48.](#)

[2.7.3 Starting the application rewind server and debugging the target-resident application on page 2-50.](#)

Related references

[2.1 Types of target connections on page 2-37.](#)

6.14 About debugging MMUs

DS-5 Debugger provides various features to debug Memory Management Unit (MMU) related issues.

A Memory Management Unit is a hardware feature that controls virtual to physical address translation, access permissions, and memory attributes. The MMU is configured by system control registers and translation tables stored in memory.

A device can contain any number of MMUs. If a device has cascaded MMUs, then the output address of one MMU is used as the input address to another MMU. A given translation depends on the context in which it occurs and the set of MMUs that it passes through.

For example, a processor that implements the ARMv7A hypervisor extensions, such as Cortex-A15, includes at least three MMUs. Typically one is used for hypervisor memory, one for virtualization and one for normal memory accesses within an OS. When in hypervisor state, memory accesses pass only through the hypervisor MMU. When in normal state, memory accesses pass first through the normal MMU and then through the virtualization MMU. For more information see the *ARM Architecture Reference Manual*.

DS-5 Debugger provides visibility of MMU translation tables for some versions of the ARM Architecture. To help you debug MMU related issues, DS-5 Debugger enables you to:

- Convert a virtual address to a physical address.
- Convert a physical address to a virtual address.
- View the MMU configuration registers and override their values.
- View the translation tables as a tree structure.
- View the virtual memory layout and attributes as a table.

You can access these features using the MMU view in the graphical debugger or using the MMU commands from the command line.

Cache and MMU data in DS-5 Debugger

In some specific circumstances, DS-5 Debugger cannot provide a fully accurate view of the translation tables due to its limited visibility of the target state.

The MMU hardware on the target performs a translation table walk by doing one or more translation table lookups. These lookups require accessing memory by physical address (or intermediate physical address for two stage translations). However, to read or modify translation table entries, the CPU accesses memory by virtual address. In each of these cases, the accessed translation table entries are permitted to reside in the CPU's data caches. This means that if a translation table entry resides in a region of memory marked as write-back cacheable and the CPU's data cache is enabled, then any modification to a translation table entry might not be written to the physical memory immediately. This is not a problem for the MMU hardware, which has awareness of the CPU's data caches.

To perform translation tables walks, DS-5 Debugger must also access memory by physical address. It does this by disabling the MMU. Because the MMU is disabled, these memory accesses might not take into account the contents of CPU's data caches. Hence these physical memory accesses might return stale data.

To avoid stale translation tables entries in DS-5 Debugger:

- When walking translation tables where the debugger has data cache awareness, you can enable cache-aware physical memory accesses. Use the command:

```
set mmu use-cache-for-phys-reads true
```
- If you think that the translation table entries contain stale data, then you can use the debugger to manually clean and invalidate the contents of the CPU caches. Use the command:

cache flush

————— **Note** —————

Flushing large caches might take a long time.

Related concepts

[6.17 About debugging caches](#) on page 6-165.

Related references

[10.15 MMU view](#) on page 10-242.

Related information

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

6.15 About *Debug and Trace Services Layer* (DTSL)

Debug and Trace Services Layer (DTSL) is a software layer within the DS-5 Debugger stack. It sits between the debugger and the RDDI target access API.

DTSL takes responsibility for:

- Low level debugger component creation and configuration. For example, CoreSight component configuration, which can also involve live re-configuration.
- Target access and debug control.
- Capture and control of trace data with:
 - in-target trace capture components, such as ETB
 - off-target trace capture device, such as DSTREAM™.
- Delivery of trace streams to the debugger or other 3rd party trace consumers.

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.

A simple example of this is when DTSL connects to a simple platform containing a Cortex-A8, ETM, and ETB. When the DTSL connection is activated it runs a Jython script to create the DTSL configuration. This configuration is populated with a Java *Device* object called *Cortex-A8*, a *TraceSource* object called *ETM*, and a *TraceCaptureDevice* object called *ETB*. The Debugger, or another program using DTSL, can then access the DTSL configuration to retrieve these objects and perform debug and trace operations.

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.

ARM has made DTSL available for your own use so that you can create Java or Jython programs to access and control the target platform.

For details, see the DTSL documents and files provided with DS-5 here:

<DS-5 Install folder>\sw\DTSL

Related references

[Chapter 14 Debug and Trace Services Layer \(DTSL\) on page 14-389.](#)

6.16 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 can 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.

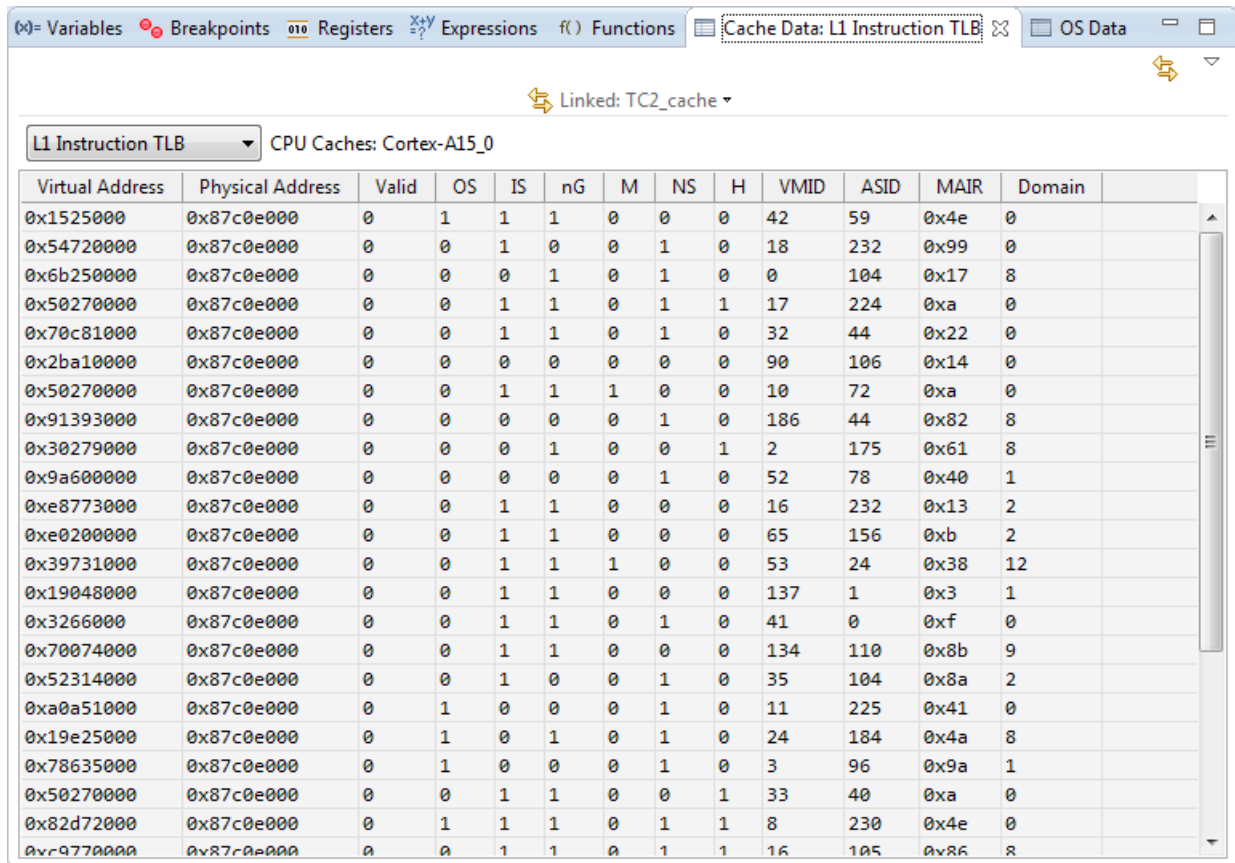
For details, see the CoreSight example provided with DS-5 here:

<DS-5 Install folder>/examples/CoreSight_Access_Library.zip

6.17 About debugging caches

DS-5 Debugger allows you to view contents of caches in your system. For example, L1 cache or TLB cache.

You can either view information about the caches in the Cache Data view or by using the `cache list` and `cache print` commands in the Commands view.



Virtual Address	Physical Address	Valid	OS	IS	nG	M	NS	H	VMID	ASID	MAIR	Domain
0x1525000	0x87c0e000	0	1	1	1	0	0	0	42	59	0x4e	0
0x5472000	0x87c0e000	0	0	1	0	0	1	0	18	232	0x99	0
0x6b25000	0x87c0e000	0	0	0	1	0	1	0	0	104	0x17	8
0x5027000	0x87c0e000	0	0	1	1	0	1	1	17	224	0xa	0
0x70c81000	0x87c0e000	0	0	1	1	0	1	0	32	44	0x22	0
0x2ba10000	0x87c0e000	0	0	0	0	0	0	0	90	106	0x14	0
0x50270000	0x87c0e000	0	0	1	1	1	0	0	10	72	0xa	0
0x91393000	0x87c0e000	0	0	0	0	0	1	0	186	44	0x82	8
0x30279000	0x87c0e000	0	0	0	1	0	0	1	2	175	0x61	8
0x9a600000	0x87c0e000	0	0	0	0	0	1	0	52	78	0x40	1
0xe8773000	0x87c0e000	0	0	1	1	0	0	0	16	232	0x13	2
0xe0200000	0x87c0e000	0	0	1	1	0	0	0	65	156	0xb	2
0x39731000	0x87c0e000	0	0	1	1	1	0	0	53	24	0x38	12
0x19048000	0x87c0e000	0	0	1	1	0	0	0	137	1	0x3	1
0x3266000	0x87c0e000	0	0	1	1	0	1	0	41	0	0xf	0
0x70074000	0x87c0e000	0	0	1	1	0	0	0	134	110	0x8b	9
0x52314000	0x87c0e000	0	0	1	0	0	1	0	35	104	0x8a	2
0xa0a51000	0x87c0e000	0	1	0	0	0	1	0	11	225	0x41	0
0x19e25000	0x87c0e000	0	1	0	1	0	1	0	24	184	0x4a	8
0x78635000	0x87c0e000	0	1	0	0	0	1	0	3	96	0x9a	1
0x50270000	0x87c0e000	0	0	1	1	0	0	1	33	40	0xa	0
0x82d72000	0x87c0e000	0	1	1	1	0	1	1	8	230	0x4e	0
0xc9770000	0x87c0e000	0	0	1	1	0	1	1	16	105	0x86	8

Figure 6-4 Cache Data view (showing L1 TLB cache)

Note

Cache awareness is dependent on the exact device and connection method.

The **Cache debug mode** option in the DTSL Configuration Editor dialog enables or disables the reading of cache RAMs in the **Cache Data** view. Selecting this option enables the reading of cache RAMs every time the target stops, if the **Cache Data** view is suitably configured.

Enabling the **Preserve cache contents in debug state** option in the DTSL Configuration Editor preserves the cache contents while the core is stopped. If this option is disabled, there is no guarantee that the cache contents will be preserved when the core is stopped.

Note

For the most accurate results, enable the **Preserve cache contents in debug state** option in the DTSL Configuration Editor dialog. When this option is not enabled, the information presented might be less accurate due to debugger interactions with the target.

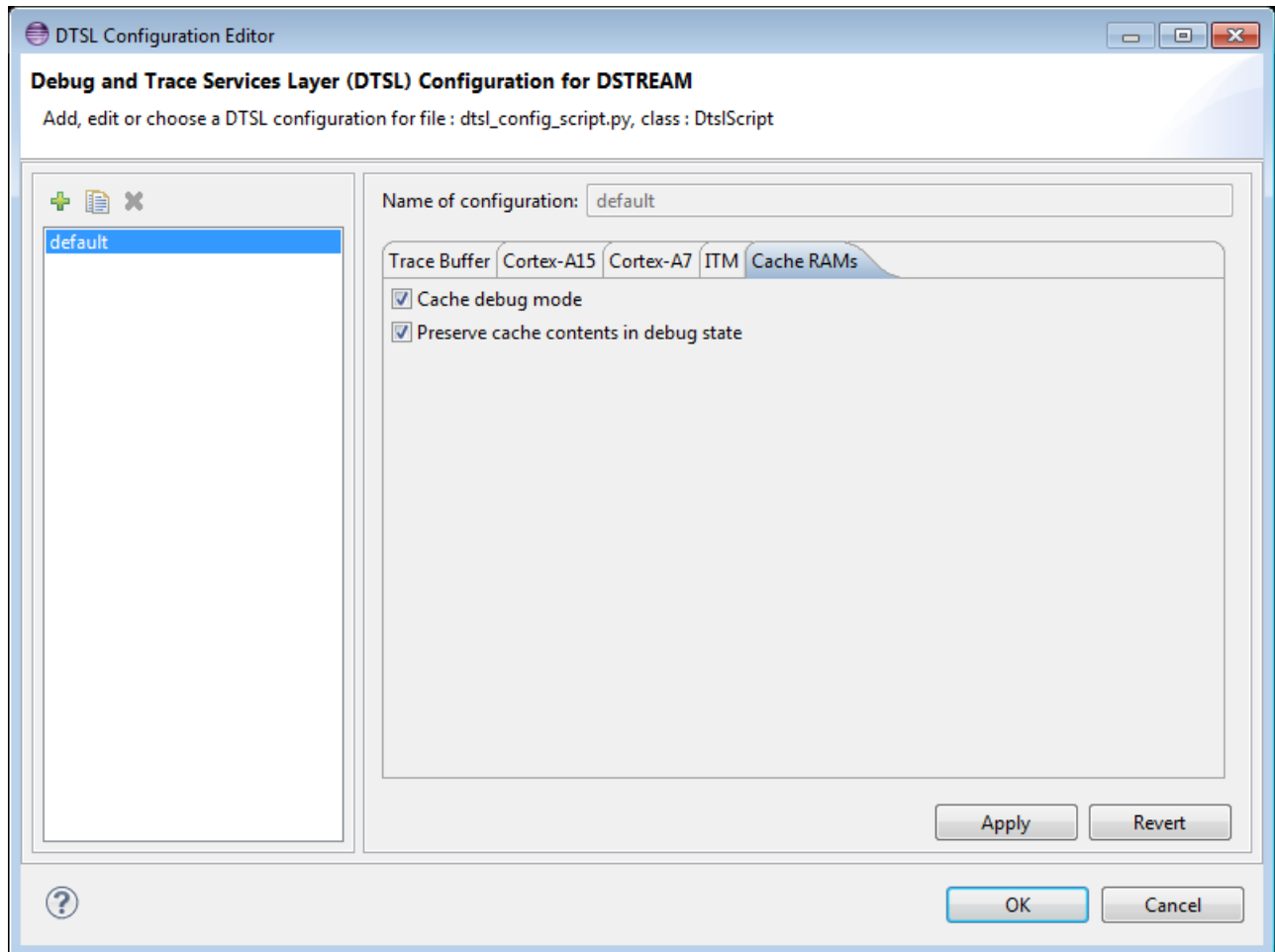


Figure 6-5 DTSL Configuration Editor (Cache RAMs configuration tab)

Note

For processors based on the ARMv8 architecture, there are restrictions on cache preservation:

- Cache preservation is not possible when the MMU is configured to use the short descriptor translation table format.
- When using the long descriptor translation table format, cache preservation is possible but the TLB contents cannot be preserved.

You can either enable the options prior to connecting to the target from the Debug Configurations dialog, or after connecting from the Debug Control view context menu.

Note

On some devices, reading cache data can be very slow. To avoid issues, do not enable DTSL options which are not required. Also, if not required, close any cache views in the user interface.

You can use the **Memory** view to display the target memory from the perspective of the different caches present on the target. On the command line, to display or read the memory from the perspective of a cache, prefix the memory address with `<cacheViewID=value>:.` For the Cortex-A15 processor, possible values of `cacheViewID` are:

- L1I
- L1D
- L2
- L3.

For example:

```
# Display memory from address 0x9000 from the perspective of the L1D cache.  
x/16w N<cacheViewID=L1D>:0x9000  
  
# Dump memory to myFile.bin, from address 0x80009000 from the perspective of the L2 cache.  
dump binary memory myFile.bin S<cacheViewID=L2>:0x80009000 0x10000  
  
# Append to myFile.bin, memory from address 0x80009000 from the perspective of the L3 cache.  
append memory myFile.bin <cacheViewID=L3>:0x80009000 0x10000
```

Related references

[10.19 Cache Data view on page 10-255.](#)

[10.14 Memory view on page 10-237.](#)

[10.44 DTSL Configuration Editor dialog box on page 10-308.](#)

Related information

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

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

Chapter 7

Controlling Runtime Messages

Describes semihosting and how to control runtime messages.

It contains the following sections:

- [7.1 About semihosting and top of memory on page 7-169.](#)
- [7.2 Working with semihosting on page 7-171.](#)
- [7.3 Enabling automatic semihosting support in the debugger on page 7-172.](#)
- [7.4 Controlling semihosting messages using the command-line console on page 7-173.](#)
- [7.5 Controlling the output of logging messages on page 7-174.](#)
- [7.6 About Log4j configuration files on page 7-175.](#)
- [7.7 Customizing the output of logging messages from the debugger on page 7-176.](#)

7.1 About semihosting and top of memory

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.

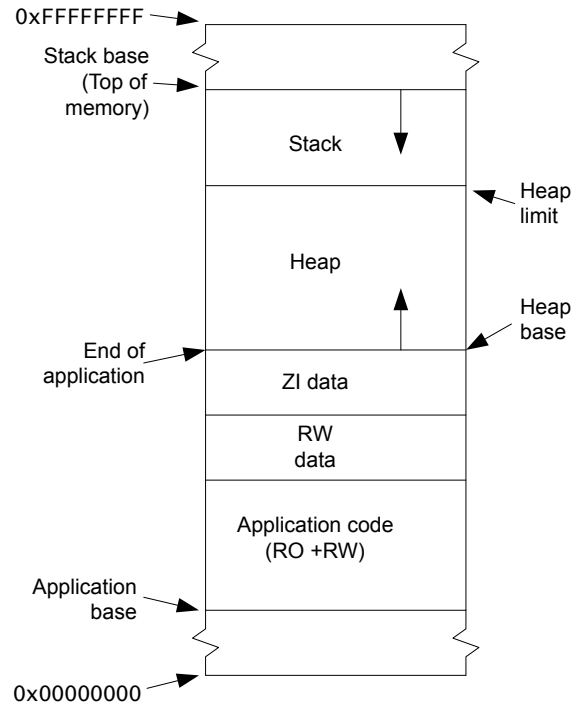


Figure 7-1 Typical layout between top of memory, stack, and heap

Semihosting implementation

DS-5 supports semihosting for both AArch64 and AArch32 states on software models and real target hardware. DS-5 Debugger handles semihosting by intercepting `HLT 0xF000` in AArch64, or `SVC` instructions in AArch32 (either `SVC 0x123456` in ARM state or `SVC 0xAB` in Thumb state).

- For AArch64 code running on real target hardware, the target halts on the `HLT` instruction and the debugger handles the semihosting automatically.
- For AArch32 code running on real target hardware or when using a software model in either AArch32 or AArch64 states, you must explicitly set a semihosting trap. Otherwise the debugger reports this error: `ERROR(TAB180): The semihosting breakpoint address has not been specified.` This error is reported when the debugger tries to enable semihosting, either when an image is loaded that contains the special symbols `__auto_semihosting` or `__semihosting_library_function`, or if you explicitly try to enable semihosting using `set semihosting enabled on`.

You can set a semihosting trap in the debugger by executing the CLI command: `set semihosting vector <trap_address>`

This instructs the debugger to set a breakpoint at this address, and when this breakpoint is hit, the debugger takes control to perform the semihosting operation.

How execution gets to this address from the HLT (AArch64) or SVC (AArch32) semihosting instruction depends on the program used, the exception level (EL) the program is executing at, how exceptions are set up to propagate, and similar settings.

It is your responsibility to ensure that execution reaches this address. This is typically done by setting the semihosting vector address to an appropriate offset in the appropriate vector table, or by creating an explicit entry in the vector table that, perhaps conditionally, branches to a known offset.

In a mixed AArch64 and AArch32 system, with semihosting used in both execution states, you must arrange for the trapping to occur at a single AArch64 trap address. The AArch64 trap address must be at an exception level that is higher than the AArch32 semihosting calling code because exceptions can only be taken from AArch32 to a higher exception level running in AArch64.

Note

The AArch64 semihosting calling code must be at the same or lower exception level than the AArch64 trap address. For example, EL3 AArch64 startup code that switches to EL2 AArch64 then starts an AArch32 application at EL1 could all make use of a semihosting trap in EL3 AArch64.

Related references

[4.3 About passing arguments to main\(\) on page 4-107.](#)

[7.2 Working with semihosting on page 7-171.](#)

[7.3 Enabling automatic semihosting support in the debugger on page 7-172.](#)

[7.4 Controlling semihosting messages using the command-line console on page 7-173.](#)

[10.42 Debug Configurations - Arguments tab on page 10-304.](#)

[10.1 App Console view on page 10-202.](#)

Related information

[DS-5 Debugger commands.](#)

7.2 Working with semihosting

Semihosting is supported by the debugger in both the command-line console and from the user interface.

Using semihosting from the 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` option to send input messages 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.

Using semihosting from the user interface

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 4-107.](#)

[7.1 About semihosting and top of memory on page 7-169.](#)

[7.3 Enabling automatic semihosting support in the debugger on page 7-172.](#)

[7.4 Controlling semihosting messages using the command-line console on page 7-173.](#)

[10.42 Debug Configurations - Arguments tab on page 10-304.](#)

[10.1 App Console view on page 10-202.](#)

Related information

[DS-5 Debugger commands.](#)

7.3 Enabling automatic 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. If required, the linker automatically adds this symbol.

Example 7-1 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 4-107.](#)
- [7.1 About semihosting and top of memory on page 7-169.](#)
- [7.2 Working with semihosting on page 7-171.](#)
- [7.4 Controlling semihosting messages using the command-line console on page 7-173.](#)
- [10.42 Debug Configurations - Arguments tab on page 10-304.](#)
- [10.1 App Console view on page 10-202.](#)

Related information

[DS-5 Debugger commands.](#)

7.4 Controlling semihosting messages using the command-line console

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 4-107.](#)
- [7.1 About semihosting and top of memory on page 7-169.](#)
- [7.2 Working with semihosting on page 7-171.](#)
- [7.3 Enabling automatic semihosting support in the debugger on page 7-172.](#)
- [10.42 Debug Configurations - Arguments tab on page 10-304.](#)
- [10.1 App Console view on page 10-202.](#)

Related information

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

7.5 Controlling the output of 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 7-175.](#)

Related tasks

[7.7 Customizing the output of logging messages from the debugger on page 7-176.](#)

Related information

[Log4j in Apache Logging Services.](#)

7.6 About Log4j configuration files

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. It 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 7-176.](#)

Related references

[7.5 Controlling the output of logging messages on page 7-174.](#)

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 7-2 Examples

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 7-175.](#)

Related references

[7.5 Controlling the output of logging messages on page 7-174.](#)

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.

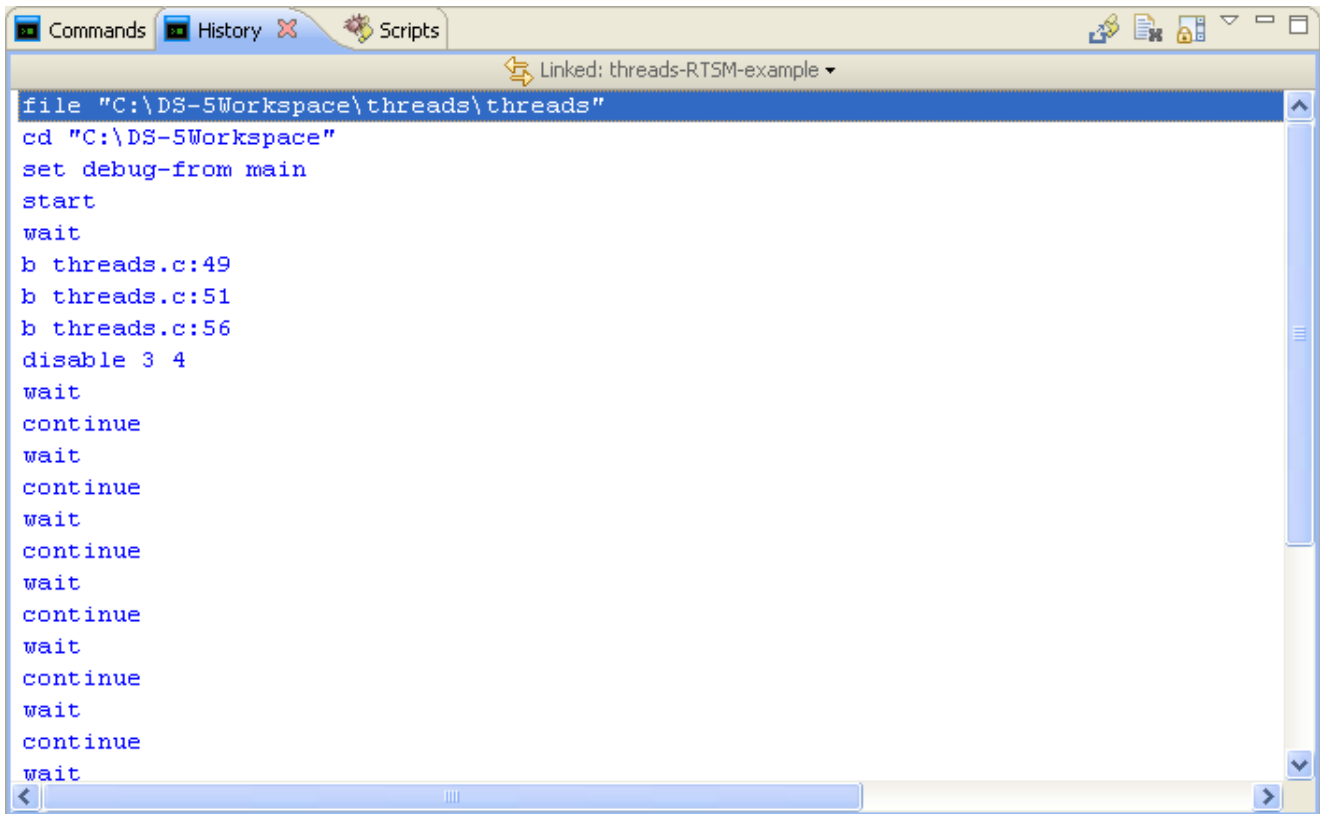
It contains the following sections:

- [8.1 Exporting DS-5 Debugger commands generated during a debug session](#) on page 8-180.
- [8.2 Creating a DS-5 Debugger script](#) on page 8-181.
- [8.3 Creating a CMM-style script](#) on page 8-182.
- [8.4 About Jython scripts](#) on page 8-183.
- [8.5 Jython script concepts and interfaces](#) on page 8-185.
- [8.6 Creating Jython projects in Eclipse for DS-5](#) on page 8-186.
- [8.7 Creating a Jython script](#) on page 8-189.
- [8.8 Running a script](#) on page 8-191.

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.



The screenshot shows the DS-5 Debugger interface with the 'History' tab selected. The 'Commands' tab is also visible. The 'Scripts' tab is active, showing a list of commands generated during a debug session. The commands are as follows:

```
file "C:\DS-5Workspace\threads\threads"
cd "C:\DS-5Workspace"
set debug-from main
start
wait
b threads.c:49
b threads.c:51
b threads.c:56
disable 3 4
wait
continue
wait
continue
wait
continue
wait
continue
wait
continue
wait
continue
wait
```

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()
# Initialization 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.getIdentifer(), 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 8-186.](#)

[8.6.2 Configuring an existing project to use the DS-5 Jython interpreter](#) on page 8-187.

[8.7 Creating a Jython script](#) on page 8-189.

[8.8 Running a script](#) on page 8-191.

Related references

[10.21 Scripts view](#) on page 10-260.

[8.5 Jython script concepts and interfaces](#) on page 8-185.

[10.37 Script Parameters dialog box](#) on page 10-292.

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 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.executedDSCommand('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 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.

This section contains the following subsections:

- [8.6.1 Creating a new Jython project in Eclipse for DS-5 on page 8-186.](#)
- [8.6.2 Configuring an existing project to use the DS-5 Jython interpreter on page 8-187.](#)

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**.

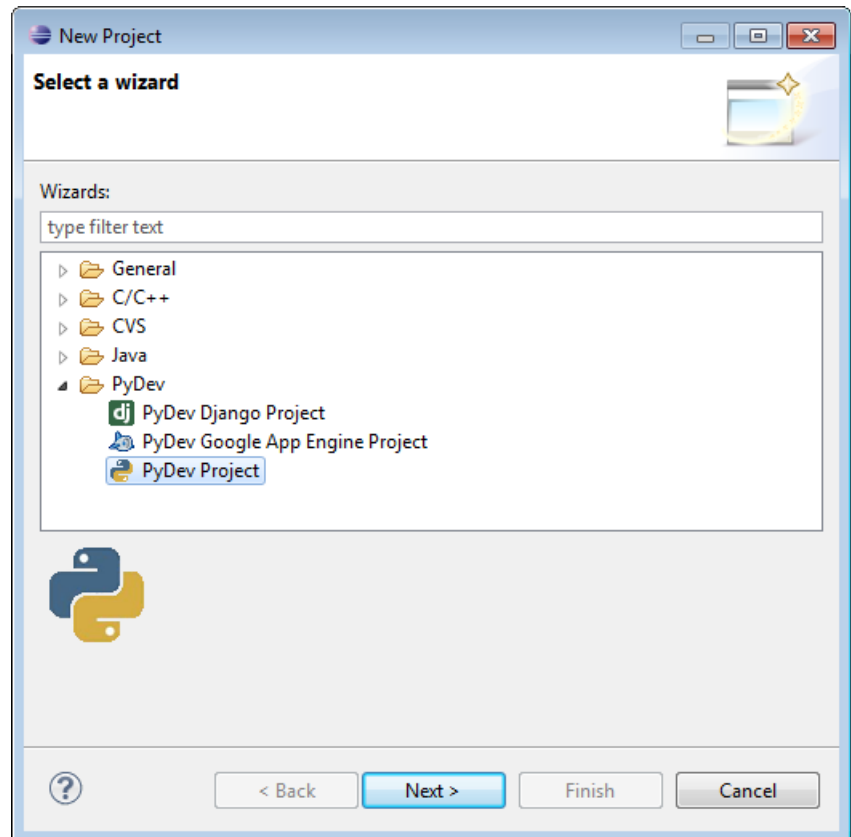


Figure 8-2 PyDev project wizard

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**.

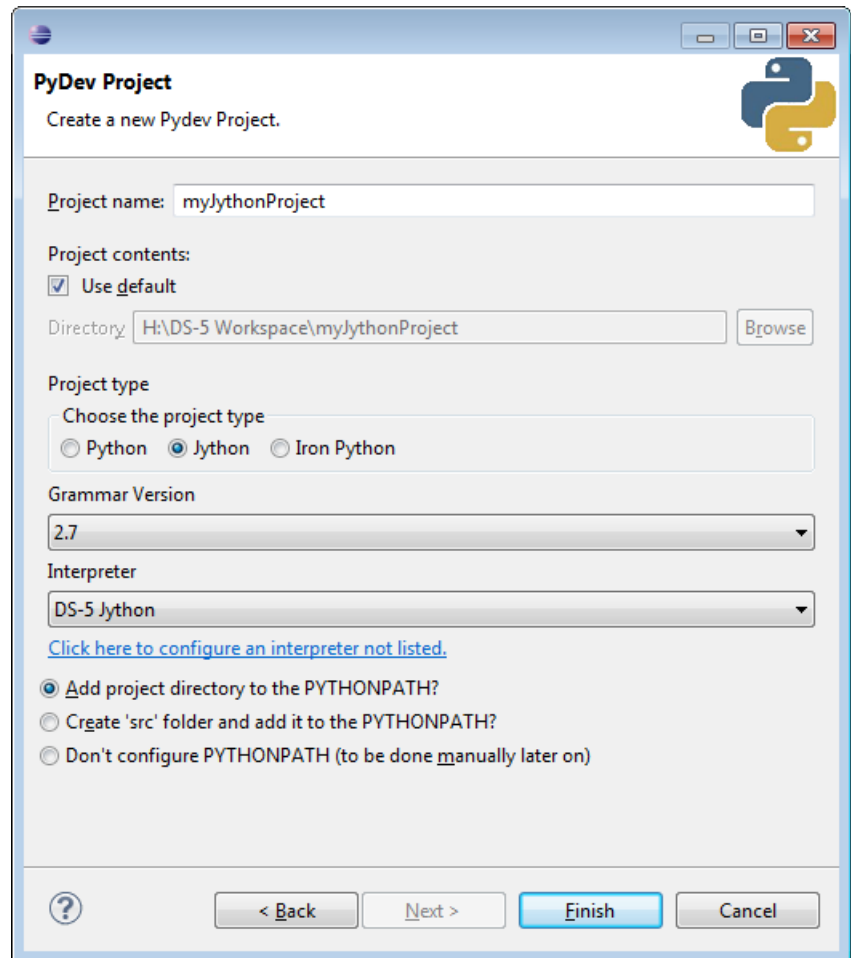


Figure 8-3 PyDev project settings

6. Click **Finish** to create the project.

Related concepts

[8.4 About Jython scripts on page 8-183.](#)

Related tasks

[8.6.2 Configuring an existing project to use the DS-5 Jython interpreter on page 8-187.](#)

[8.7 Creating a Jython script on page 8-189.](#)

[8.8 Running a script on page 8-191.](#)

Related references

[10.21 Scripts view on page 10-260.](#)

[8.5 Jython script concepts and interfaces on page 8-185.](#)

[10.37 Script Parameters dialog box on page 10-292.](#)

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 8-183.

Related tasks

[8.6.1 Creating a new Jython project in Eclipse for DS-5](#) on page 8-186.

[8.7 Creating a Jython script](#) on page 8-189.

[8.8 Running a script](#) on page 8-191.

Related references

[10.21 Scripts view](#) on page 10-260.

[8.5 Jython script concepts and interfaces](#) on page 8-185.

[10.37 Script Parameters dialog box](#) on page 10-292.

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.

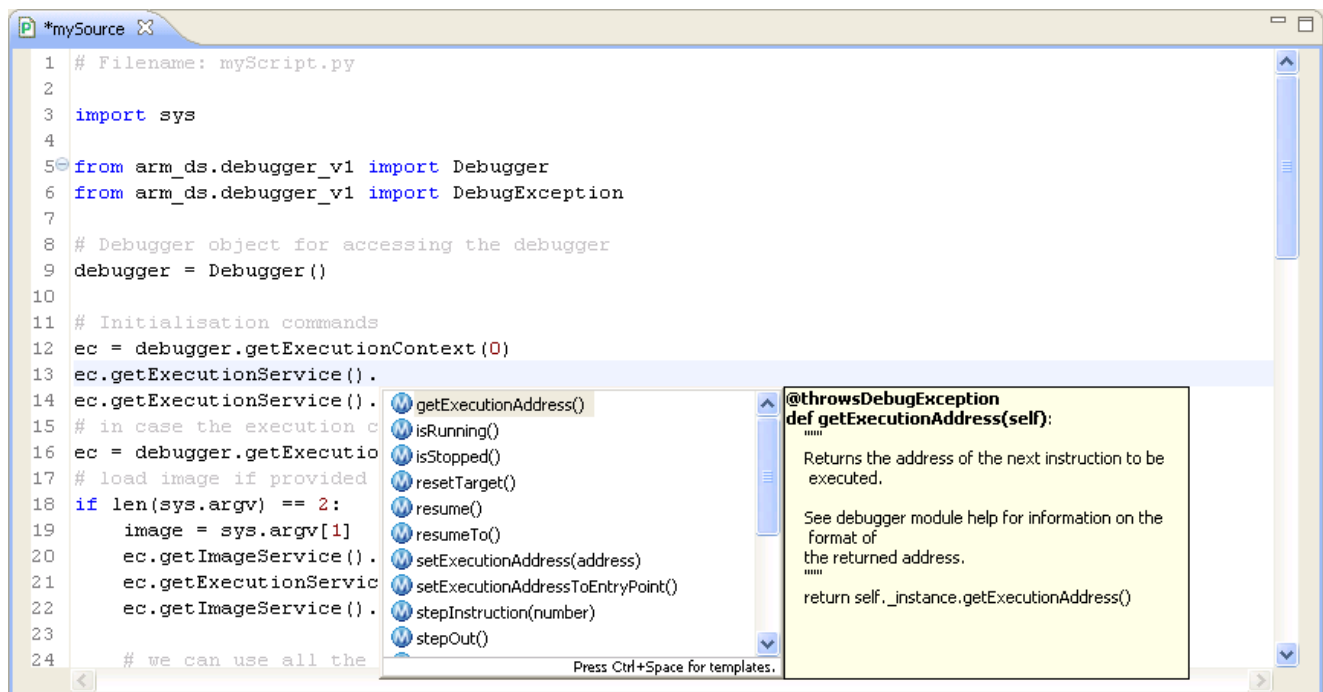


Figure 8-4 Jython auto-completion and help

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 8-183.](#)

Related tasks

[8.6.1 Creating a new Jython project in Eclipse for DS-5 on page 8-186.](#)

[8.6.2 Configuring an existing project to use the DS-5 Jython interpreter on page 8-187.](#)

[8.8 Running a script on page 8-191.](#)

Related references

10.21 Scripts view on page 10-260.

8.5 Jython script concepts and interfaces on page 8-185.

10.37 Script Parameters dialog box on page 10-292.

8.8 Running a script

Describes how to run a script from Eclipse.

Procedure

1. 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.

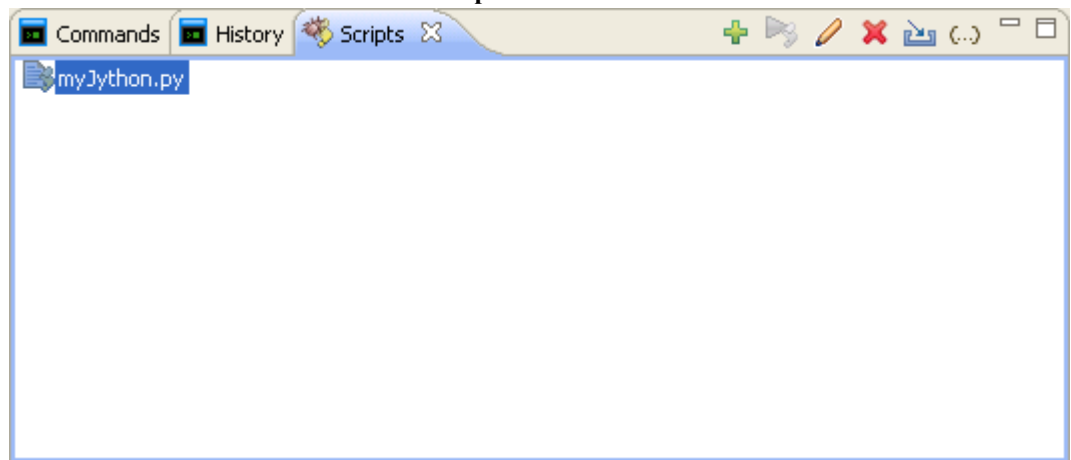


Figure 8-5 Scripts view

— In the **Commands** view you can use the DS-5 Debugger source command.

Related concepts

[8.4 About Jython scripts on page 8-183.](#)

Related tasks

[8.6.1 Creating a new Jython project in Eclipse for DS-5 on page 8-186.](#)

[8.6.2 Configuring an existing project to use the DS-5 Jython interpreter on page 8-187.](#)

[8.7 Creating a Jython script on page 8-189.](#)

Related references

[8.1 Exporting DS-5 Debugger commands generated during a debug session on page 8-180.](#)

[8.2 Creating a DS-5 Debugger script on page 8-181.](#)

[8.3 Creating a CMM-style script on page 8-182.](#)

[10.6 Commands view on page 10-214.](#)

[10.13 History view on page 10-235.](#)

[10.21 Scripts view on page 10-260.](#)

[8.5 Jython script concepts and interfaces on page 8-185.](#)

[10.37 Script Parameters dialog box on page 10-292.](#)

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 sections:

- [9.1 About the Snapshot Viewer](#) on page 9-194.
- [9.2 Components of a Snapshot Viewer initialization file](#) on page 9-196.
- [9.3 Connecting to the Snapshot Viewer](#) on page 9-198.
- [9.4 Considerations when creating debugger scripts for the Snapshot Viewer](#) on page 9-199.

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.

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 9-198.](#)

Related references

[9.2 Components of a Snapshot Viewer initialization file on page 9-196.](#)

[9.4 Considerations when creating debugger scripts for the Snapshot Viewer on page 9-199.](#)

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.

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.

Example 9-1 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 9-194.](#)

Related tasks

[9.3 Connecting to the Snapshot Viewer on page 9-198.](#)

Related references

[9.4 Considerations when creating debugger scripts for the Snapshot Viewer on page 9-199.](#)

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
```

Related concepts

[9.1 About the Snapshot Viewer on page 9-194.](#)

Related tasks

[1.6 Headless command-line debugger options on page 1-24.](#)

Related references

[9.2 Components of a Snapshot Viewer initialization file on page 9-196.](#)

[9.4 Considerations when creating debugger scripts for the Snapshot Viewer on page 9-199.](#)

[1.10 DS-5 Debugger command-line console keyboard shortcuts on page 1-34.](#)

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.

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 9-194.](#)

Related tasks

[9.3 Connecting to the Snapshot Viewer on page 9-198.](#)

Related references

[9.2 Components of a Snapshot Viewer initialization file on page 9-196.](#)

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 sections:

- [10.1 App Console view on page 10-202.](#)
- [10.2 ARM Asm Info view on page 10-204.](#)
- [10.3 ARM assembler editor on page 10-205.](#)
- [10.4 Breakpoints view on page 10-207.](#)
- [10.5 C/C++ editor on page 10-211.](#)
- [10.6 Commands view on page 10-214.](#)
- [10.7 Debug Control view on page 10-217.](#)
- [10.8 Disassembly view on page 10-221.](#)
- [10.9 Events view on page 10-225.](#)
- [10.10 Event Viewer Settings dialog box on page 10-227.](#)
- [10.11 Expressions view on page 10-229.](#)
- [10.12 Functions view on page 10-233.](#)
- [10.13 History view on page 10-235.](#)
- [10.14 Memory view on page 10-237.](#)
- [10.15 MMU view on page 10-242.](#)
- [10.16 Modules view on page 10-246.](#)
- [10.17 Registers view on page 10-248.](#)
- [10.18 OS Data view on page 10-254.](#)
- [10.19 Cache Data view on page 10-255.](#)
- [10.20 Screen view on page 10-257.](#)
- [10.21 Scripts view on page 10-260.](#)
- [10.22 Target Console view on page 10-262.](#)

- *10.23 Target view on page 10-263.*
- *10.24 Trace view on page 10-265.*
- *10.25 Trace Control view on page 10-268.*
- *10.26 Variables view on page 10-271.*
- *10.27 Timed Auto-Refresh Properties dialog box on page 10-278.*
- *10.28 Memory Exporter dialog box on page 10-279.*
- *10.29 Memory Importer dialog box on page 10-280.*
- *10.30 Fill Memory dialog box on page 10-281.*
- *10.31 Export Trace Report dialog box on page 10-282.*
- *10.32 Breakpoint Properties dialog box on page 10-284.*
- *10.33 Watchpoint Properties dialog box on page 10-287.*
- *10.34 Tracepoint Properties dialog box on page 10-288.*
- *10.35 Manage Signals dialog box on page 10-289.*
- *10.36 Functions Filter dialog box on page 10-291.*
- *10.37 Script Parameters dialog box on page 10-292.*
- *10.38 Debug Configurations - Connection tab on page 10-293.*
- *10.39 Debug Configurations - Files tab on page 10-296.*
- *10.40 Debug Configurations - Debugger tab on page 10-300.*
- *10.41 Debug Configurations - OS Awareness tab on page 10-303.*
- *10.42 Debug Configurations - Arguments tab on page 10-304.*
- *10.43 Debug Configurations - Environment tab on page 10-306.*
- *10.44 DTSL Configuration Editor dialog box on page 10-308.*
- *10.45 Configuration Database panel on page 10-310.*
- *10.46 Configuration Perspective on page 10-312.*
- *10.47 About the Remote System Explorer on page 10-329.*
- *10.48 Remote Systems view on page 10-330.*
- *10.49 Remote System Details view on page 10-331.*
- *10.50 Target management terminal for serial and SSH connections on page 10-332.*
- *10.51 Remote Scratchpad view on page 10-333.*
- *10.52 Remote Systems terminal for SSH connections on page 10-334.*
- *10.53 Terminal Settings dialog box on page 10-335.*
- *10.54 Debug Hardware Configure IP view on page 10-337.*
- *10.55 Debug Hardware Firmware Installer view on page 10-339.*
- *10.56 Connection Browser dialog box on page 10-341.*
- *10.57 DS-5 Debugger menu and toolbar icons on page 10-342.*

10.1 App Console view

Use the **App Console** view 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.

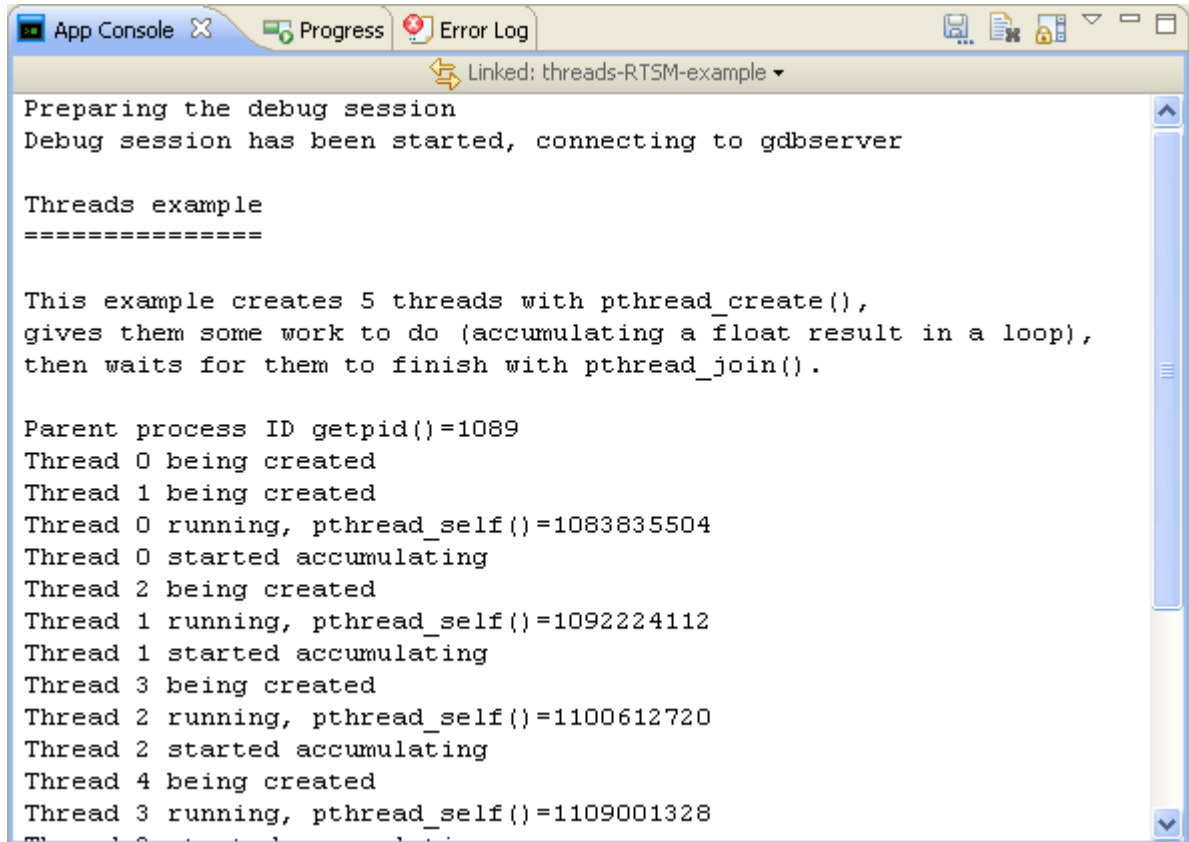


Figure 10-1 App Console view

Note

Default settings for this view, for example the maximum number of lines to display, are controlled by the DS-5 Debugger option in the Preferences dialog box. 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 different 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.

Toggles Scroll Lock

Enables or disables the automatic scrolling of messages in the **App Console** view.

View Menu

This menu contains the following options:

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 7-169.

[7.2 Working with semihosting](#) on page 7-171.

[7.3 Enabling automatic semihosting support in the debugger](#) on page 7-172.

[Chapter 10 DS-5 Debug Perspectives and Views](#) on page 10-200.

10.2 ARM Asm Info view

Use the **ARM Asm Info** view to display the documentation for an ARM or Thumb® instruction or directive.

When you are editing assembly language source files, which have a `.s` extension, 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 shown when you press **F3**.

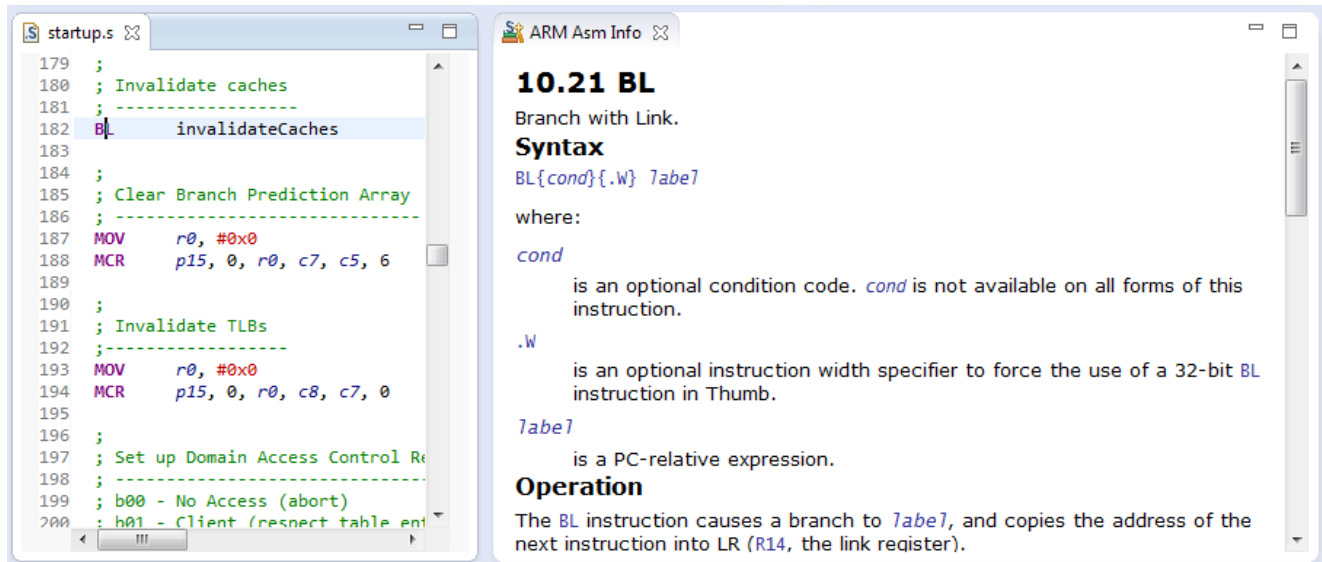


Figure 10-2 ARM Asm Info view

To manually show 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

[Chapter 10 DS-5 Debug Perspectives and Views](#) on page 10-200.

[10.3 ARM assembler editor](#) on page 10-205.

10.3 ARM assembler editor

Use the ARM assembler editor to view and edit ARM assembly language source code. It provides syntax highlighting, code formatting, and content assist for auto-completion.

This editor also enables you to:

- Select an instruction or directive and press **F3** to view the related ARM assembler reference information.
- 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.

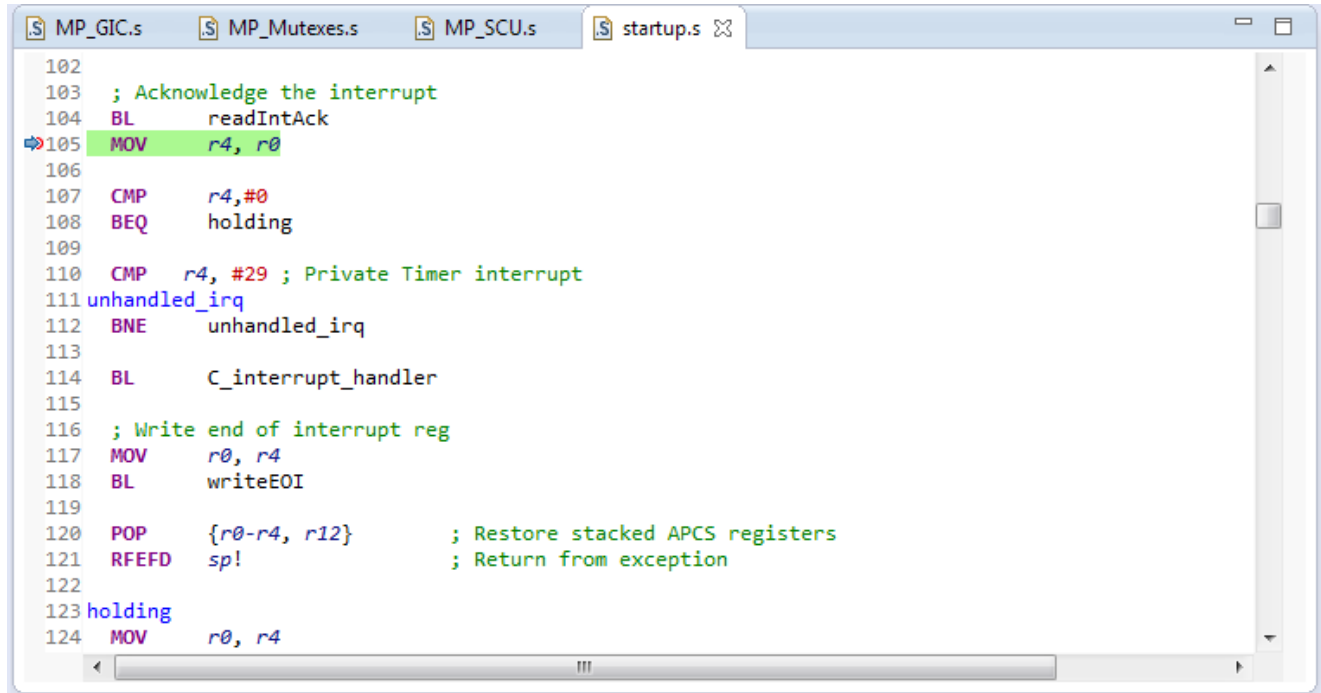


Figure 10-3 ARM assembler editor

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 or removes a breakpoint.

Toggle Hardware Breakpoint

Sets or removes a hardware breakpoint.

Resolve Breakpoint

Resolves a pending breakpoint.

Disable Breakpoint, Enable Breakpoint

Disables or enables the selected breakpoint.

Toggle Trace Start Point

Sets or removes a trace start point.

Toggle Trace Stop Point

Sets or removes a trace stop point.

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.

Breakpoint Properties...

Displays the Breakpoint Properties dialog box for the selected breakpoint. This enables you to control breakpoint activation.

Default Breakpoint Type

The following breakpoint options are available:

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.

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

[4.5 Working with breakpoints and watchpoints](#) on page 4-109.

[4.8 Setting a tracepoint](#) on page 4-121.

[4.6 Working with conditional breakpoints](#) on page 4-116.

[4.6.1 Assigning conditions to an existing breakpoint](#) on page 4-116.

[4.7 About pending breakpoints and watchpoints](#) on page 4-120.

[4.5.5 Exporting DS-5 breakpoint settings to a file](#) on page 4-113.

[4.5.4 Importing DS-5 breakpoint settings from a file](#) on page 4-113.

[4.9 Setting Streamline start and stop points](#) on page 4-122.

[Chapter 10 DS-5 Debug Perspectives and Views](#) on page 10-200.

[5.1 Examining the target execution environment](#) on page 5-132.

[5.2 Examining the call stack](#) on page 5-134.

[10.2 ARM Asm Info view](#) on page 10-204.

10.4 Breakpoints view

Use the **Breakpoints** view to display the breakpoints, watchpoints, and tracepoints you have set in your program.

It also enables you to:

- Disable, enable, or delete breakpoints, watchpoints, and tracepoints.
- 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.

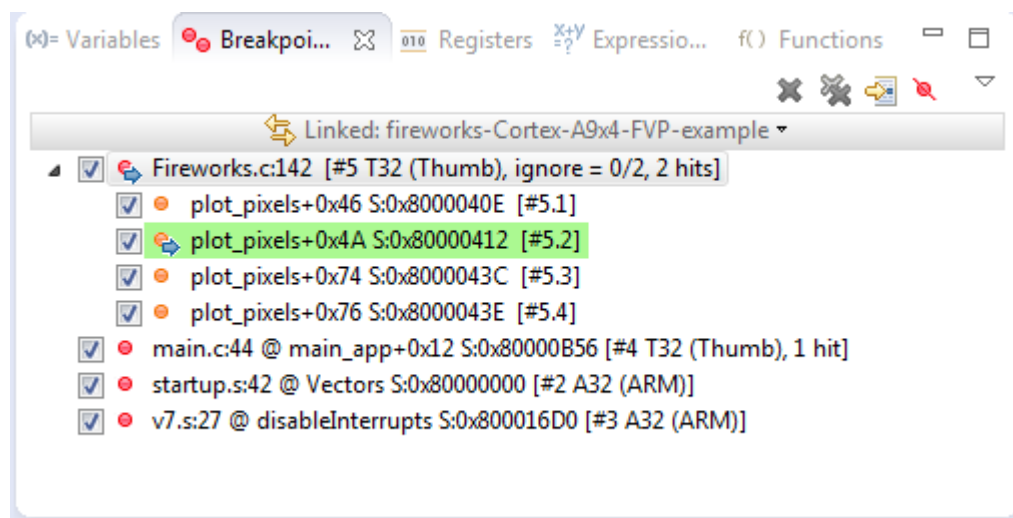


Figure 10-4 Breakpoints view

Syntax of a breakpoint entry

A breakpoint entry has the following syntax:

```
source_file:linenum @ function+offset address [#ID instruction_set, 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, for example `main.c:44`.

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, `main_app+0x12` shows that the breakpoint is 18 bytes from the start of the `main_app()` function.

address

The address at which 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 syntax of a sub-breakpoint entry is:

```
function+offset address [#ID]
```

instruction_set

The instruction set of the instruction at the address of the breakpoint, A32 (ARM) or T32 (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.

Syntax of a watchpoint entry

A watchpoint entry has the following syntax:

```
name type @ address [#ID]
```

where:

name

The name of the variable where the watchpoint is set.

type

The access type of the watchpoint.

address

The address at which the watchpoint is set.

ID

The watchpoint ID number.

Syntax of a tracepoint entry

A tracepoint entry has the following syntax:

```
source_file:linenum
```

where:

source_file:linenum

If the source file is available, the file name and line number in the file where the tracepoint is set.

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 different 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, watchpoints, and tracepoints.

Delete All

Removes all breakpoints, watchpoints, and tracepoints.

Go to File

Displays the source file containing the line of code where the selected breakpoint or tracepoint is set. This option is disabled for a watchpoint.

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.

Show in Disassembly

Displays the disassembly where the selected breakpoint is set. This option is disabled for a tracepoint.

Show in Memory

Displays the memory where the selected watchpoint is set. This option is disabled for a tracepoint.

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.

Enable Breakpoints

Enables the selected breakpoints, watchpoints, and tracepoints.

Disable Breakpoints

Disables the selected breakpoints, watchpoints, and tracepoints.

Copy

Copies the selected breakpoints, watchpoints, and tracepoints. You can also use the standard keyboard shortcut to do this.

Paste

Pastes the copied breakpoints, watchpoints, and tracepoints. They are enabled by default. You can also use the standard keyboard shortcut to do this.

Select all

Selects all breakpoints, watchpoints, and tracepoints. You can also use the standard keyboard shortcut to do this.

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.

View Menu

The following **View Menu** options are available:

New Breakpoints View

Displays a new instance of the **Breakpoints** view.

Import Breakpoints

Imports a list of breakpoints and watchpoints from a file.

Export Breakpoints

Exports the current list of breakpoints and watchpoints to 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.

Auto Update Breakpoint Line Numbers

Automatically updates breakpoint line numbers in the **Breakpoints** view when changes occur in the source file.

Manage Signals

Displays the Manage Signals dialog box.

Related concepts

[6.8 About debugging multi-threaded applications on page 6-149.](#)

[6.9 About debugging shared libraries on page 6-150.](#)

[6.10.2 About debugging a Linux kernel on page 6-153.](#)

[6.10.3 About debugging Linux kernel modules on page 6-154.](#)

[6.11 About debugging TrustZone enabled targets on page 6-157.](#)

Related references

- [4.5 Working with breakpoints and watchpoints](#) on page 4-109.
- [4.8 Setting a tracepoint](#) on page 4-121.
- [4.6 Working with conditional breakpoints](#) on page 4-116.
- [4.6.1 Assigning conditions to an existing breakpoint](#) on page 4-116.
- [4.7 About pending breakpoints and watchpoints](#) on page 4-120.
- [4.5.5 Exporting DS-5 breakpoint settings to a file](#) on page 4-113.
- [4.5.4 Importing DS-5 breakpoint settings from a file](#) on page 4-113.
- [Chapter 10 DS-5 Debug Perspectives and Views](#) on page 10-200.
- [5.1 Examining the target execution environment](#) on page 5-132.
- [5.2 Examining the call stack](#) on page 5-134.

10.5 C/C++ editor

Use the C/C++ editor to view and edit C and C++ source code. It provides syntax highlighting, code formatting, and content assist (**Ctrl+Space**) for auto-completion.

This editor also enables you to:

- View interactive help when hovering over C library functions.
- 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.

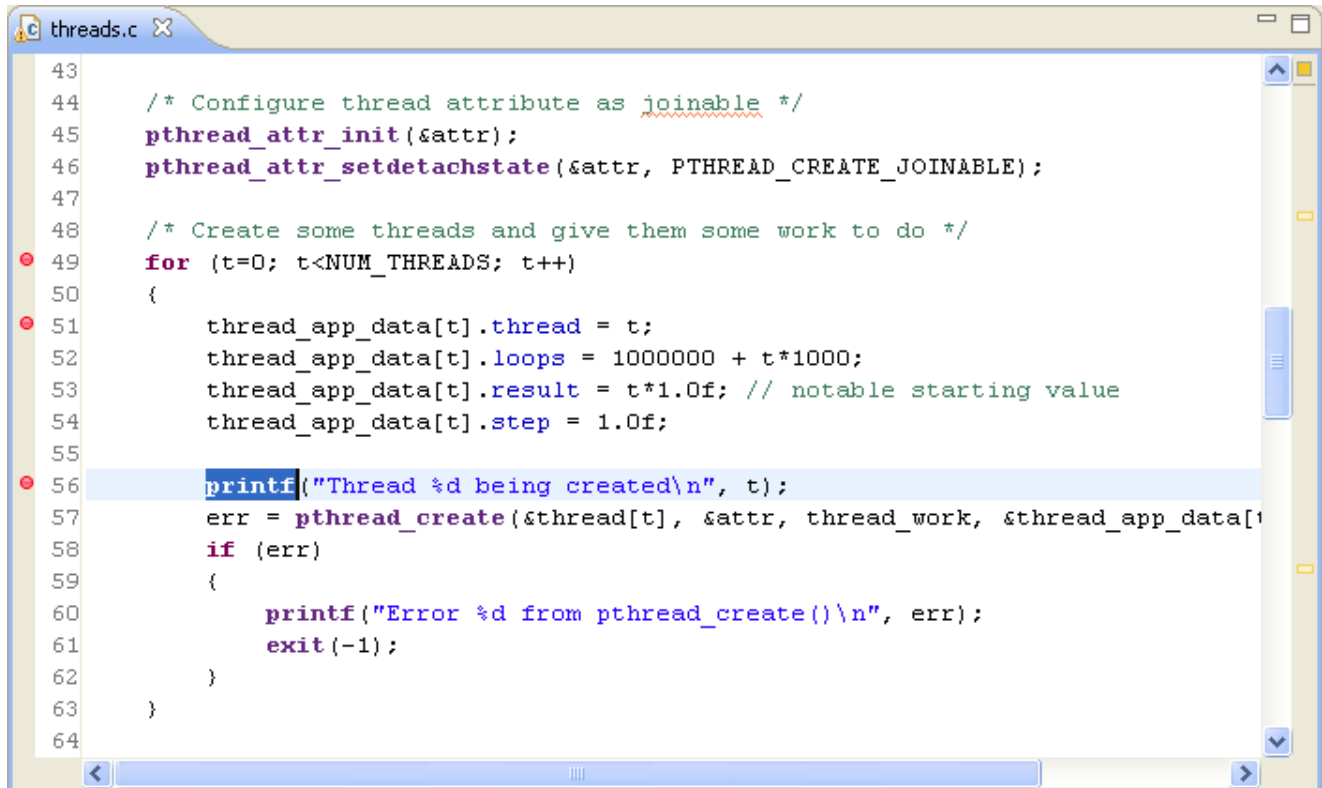


Figure 10-5 C/C++ editor

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.

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.

Breakpoint Properties...

Displays the Breakpoint Properties dialog box for the selected breakpoint. This enables you to control breakpoint activation.

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.

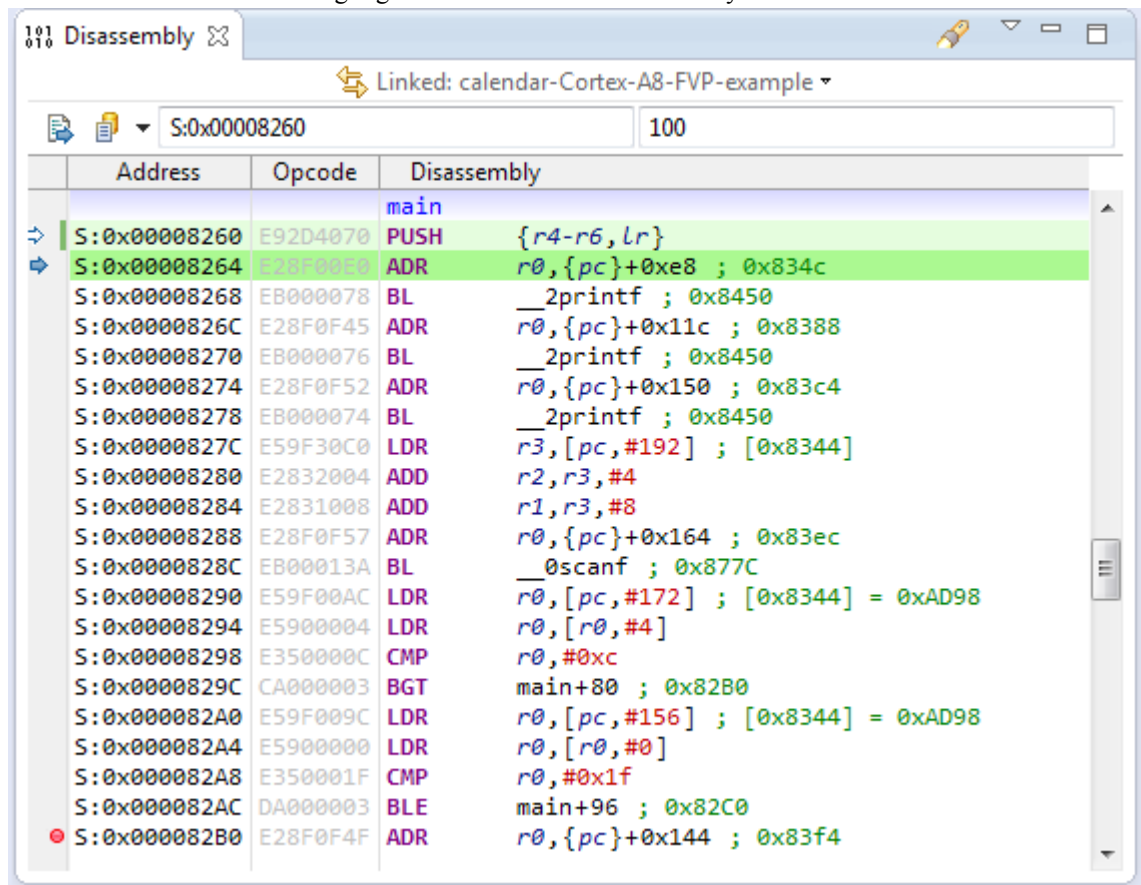


Figure 10-6 Show disassembly for selected source line

For more information on the other options not listed here, see the dynamic help.

Related references

- [4.5 Working with breakpoints and watchpoints on page 4-109.](#)
- [4.8 Setting a tracepoint on page 4-121.](#)
- [4.6 Working with conditional breakpoints on page 4-116.](#)
- [4.6.1 Assigning conditions to an existing breakpoint on page 4-116.](#)
- [4.7 About pending breakpoints and watchpoints on page 4-120.](#)
- [4.5.5 Exporting DS-5 breakpoint settings to a file on page 4-113.](#)
- [4.5.4 Importing DS-5 breakpoint settings from a file on page 4-113.](#)
- [4.9 Setting Streamline start and stop points on page 4-122.](#)
- [Chapter 10 DS-5 Debug Perspectives and Views on page 10-200.](#)

10.6 Commands view

Use the **Commands** view to display DS-5 Debugger commands and the messages output by the debugger. It enables you to enter commands, run a command script, and save the contents of the view to a text file.

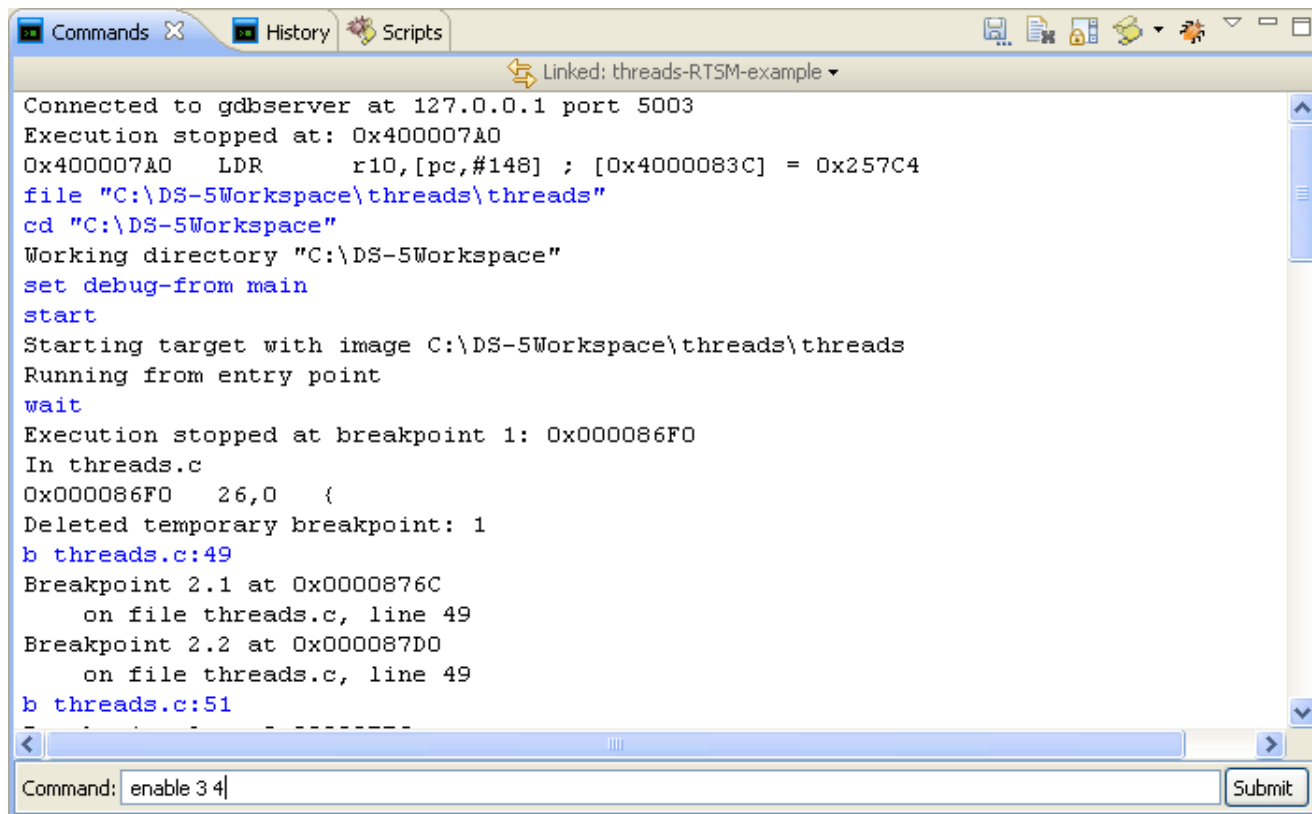


Figure 10-7 Commands view

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 script 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 different 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.

Toggles 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 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

[6.8 About debugging multi-threaded applications on page 6-149.](#)

[6.9 About debugging shared libraries on page 6-150.](#)

[6.10.2 About debugging a Linux kernel on page 6-153.](#)

[6.10.3 About debugging Linux kernel modules on page 6-154.](#)

[6.11 About debugging TrustZone enabled targets on page 6-157.](#)

Related references

[1.9 DS-5 Debug perspective keyboard shortcuts on page 1-33.](#)

[10.7 Debug Control view on page 10-217.](#)

4.5 Working with breakpoints and watchpoints on page 4-109.
4.8 Setting a tracepoint on page 4-121.
4.6 Working with conditional breakpoints on page 4-116.
4.6.1 Assigning conditions to an existing breakpoint on page 4-116.
4.7 About pending breakpoints and watchpoints on page 4-120.
4.5.5 Exporting DS-5 breakpoint settings to a file on page 4-113.
4.5.4 Importing DS-5 breakpoint settings from a file on page 4-113.
Chapter 10 DS-5 Debug Perspectives and Views on page 10-200.

Related information

DS-5 Debugger commands.

10.7 Debug Control view

Use the **Debug Control** view to display 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.

This view enables you to:

- 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 onto the target.
- Load debug information when required by the debugger.
- Reset the target.
- Start, run and stop the application.
- 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.

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, as shown in the following figure:

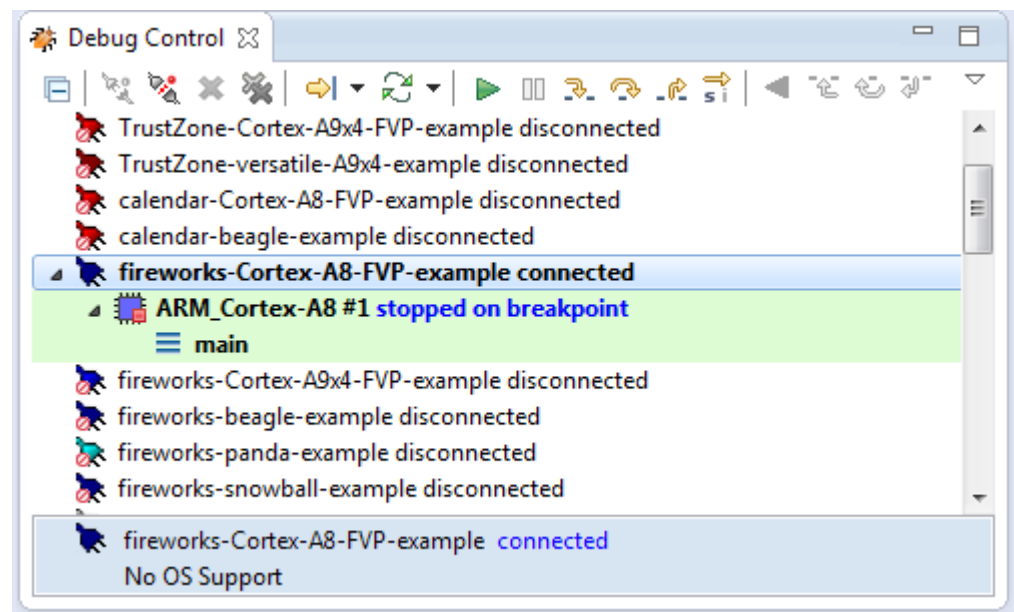


Figure 10-8 Debug Control view

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.

Remove Connection

Removes the selected target connection from the **Debug Control** view.

Remove All Connections

Removes all target connections from the **Debug Control** view, except any that are connected to the target.

Debug from menu

This menu lists the different 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.

Reverse Continue

Continues running backwards through the code.

Reverse Step Source Line

Reverse Step Instruction

This option depends on the stepping mode selected:

- If source line mode is selected, steps backwards at the source level including stepping into all function calls where there is debug information.
- If instruction mode is selected, steps backwards at the instruction level including stepping into all function calls.

Reverse Step Over Source Line

Reverse Step Over Instruction

This option depends on the stepping mode selected:

- If source line mode is selected, steps backwards at the source level but stepping over all function calls.
- If instruction mode is selected, steps backwards at the instruction level but stepping over all function calls.

Reverse Step Out

Continues running backwards to the instruction before the start of the selected stack frame.

Debug Configurations...

Displays the Debug Configurations dialog box, with the configuration for the selected connection displayed.

Launch in background

If this option is disabled, the Progress Information dialog box is displayed when the application launches.

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.

Reset DS-5 Views to 'Linked'

Resets DS-5 views to link to the selected connection in the **Debug Control** view.

View CPU Caches

Displays the **Cache Data** view for a connected configuration.

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 disconnected.

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.

Threads Presentation

Displays either a flat or hierarchical presentation of the threads in the stack trace.

Auto Expand Call Stack

Controls whether to automatically display an expanded stack when selecting a connection.

Default Stack Depth

Controls the maximum number of stack frames to display in the **Debug Control** view. Increasing the number of displayed stack frames might slow the debugger performance.

Always Show Cores

Displays the available processors.

DTSL options

Opens the **DTSL Configuration Editor** dialog to specify the DTSL options for the target connection.

Related concepts

[6.8 About debugging multi-threaded applications](#) on page 6-149.

[6.9 About debugging shared libraries](#) on page 6-150.

[6.10.2 About debugging a Linux kernel](#) on page 6-153.

[6.10.3 About debugging Linux kernel modules](#) on page 6-154.

[6.11 About debugging TrustZone enabled targets](#) on page 6-157.

Related references

[1.9 DS-5 Debug perspective keyboard shortcuts](#) on page 1-33.

[10.6 Commands view](#) on page 10-214.

[Chapter 10 DS-5 Debug Perspectives and Views](#) on page 10-200.

10.8 Disassembly view

Use the **Disassembly** view to display a disassembly of the code in the running application.

It also enables you to:

- Specify the start address for the disassembly. You can use expressions in this field, for example \$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 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.

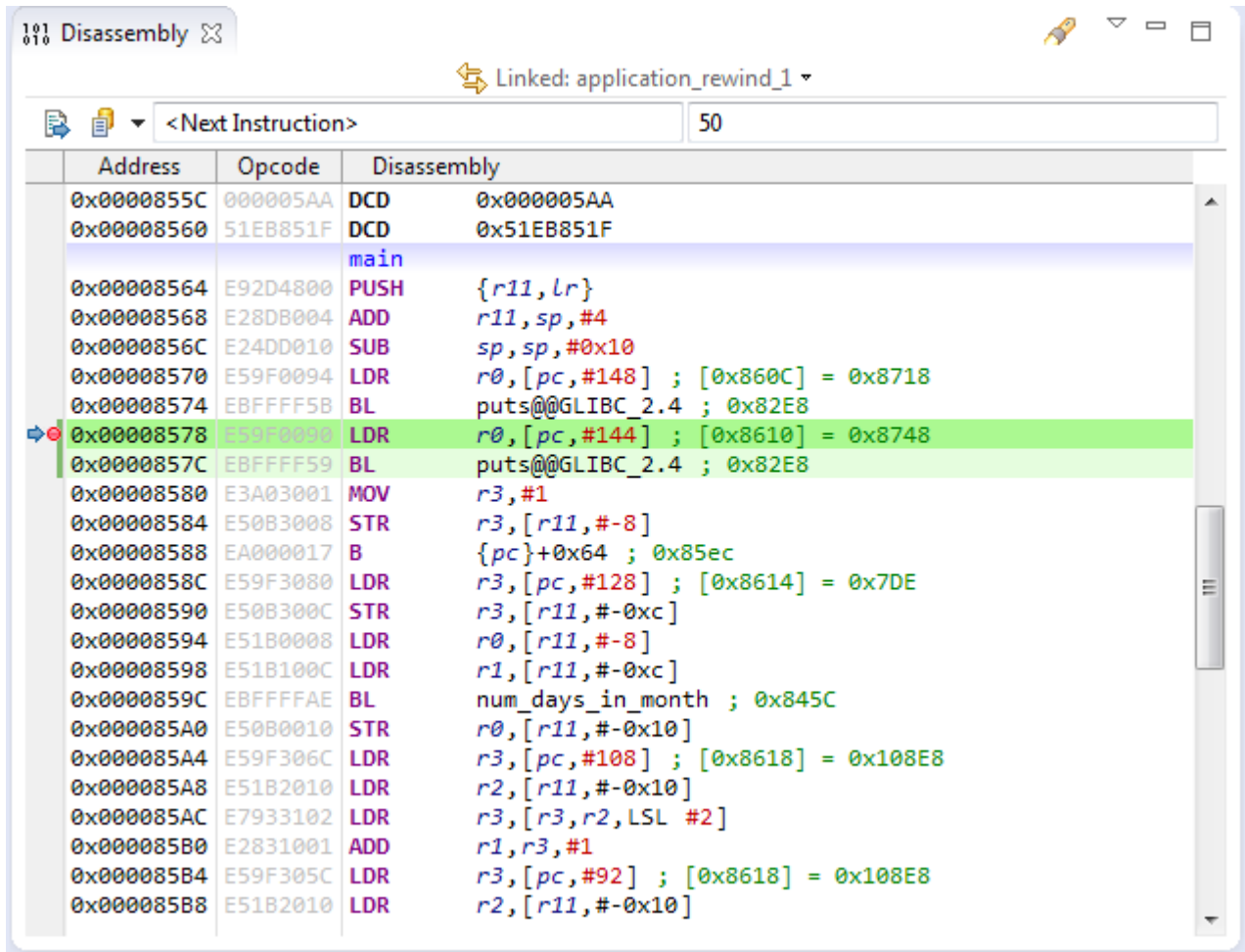


Figure 10-9 Disassembly view

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 different 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.

Next Instruction

Shows the disassembly for the instruction at the address of the program counter.

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 for which you want to view the disassembly. You can specify the address as a hex number or as an expression, for example `$PC+256`, `$1r`, or `main`.

Context menu options are available for editing this field.

Size field

The number of instructions to display before and after the location specified in the Address field.

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.

A32 (ARM)

ARM instruction set.

T32 (Thumb)

Thumb instruction set.

T32EE (ThumbEE)

ThumbEE 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.

Update View When Hidden

Enables the updating of the view when it is hidden behind other views. By default this view does not update when hidden.

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.

Translate Address <address>

Displays the **MMU** view and translates the selected address.

Toggle Watchpoint

Sets or removes a watchpoint at the selected address.

Toggle Breakpoint

Sets or removes a breakpoint at the selected address.

Toggle Hardware Breakpoint

Sets or removes a hardware 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.

Select All

Selects all the text.

Related concepts

[6.8 About debugging multi-threaded applications](#) on page 6-149.

[6.9 About debugging shared libraries](#) on page 6-150.

[6.10.2 About debugging a Linux kernel](#) on page 6-153.

[6.10.3 About debugging Linux kernel modules](#) on page 6-154.

[6.11 About debugging TrustZone enabled targets](#) on page 6-157.

Related references

[4.5 Working with breakpoints and watchpoints](#) on page 4-109.

[4.8 Setting a tracepoint](#) on page 4-121.

[4.6 Working with conditional breakpoints](#) on page 4-116.

[4.6.1 Assigning conditions to an existing breakpoint](#) on page 4-116.

[4.7 About pending breakpoints and watchpoints](#) on page 4-120.

[4.5.5 Exporting DS-5 breakpoint settings to a file](#) on page 4-113.

[4.5.4 Importing DS-5 breakpoint settings from a file](#) on page 4-113.

[Chapter 10 DS-5 Debug Perspectives and Views](#) on page 10-200.

10.9 Events view

Use the **Events** view to view the output generated by the *System Trace Macrocell* (STM) and *Instruction Trace Macrocell* (ITM) events.

Data is captured from your application when it runs. However, no data appears in the Events view until you stop the application.

To stop the target, either click the **Interrupt** icon in the **Debug Control** view, or use the **stop** command in the **Commands** view. When your application stops, any captured logging information is automatically appended to the open views.

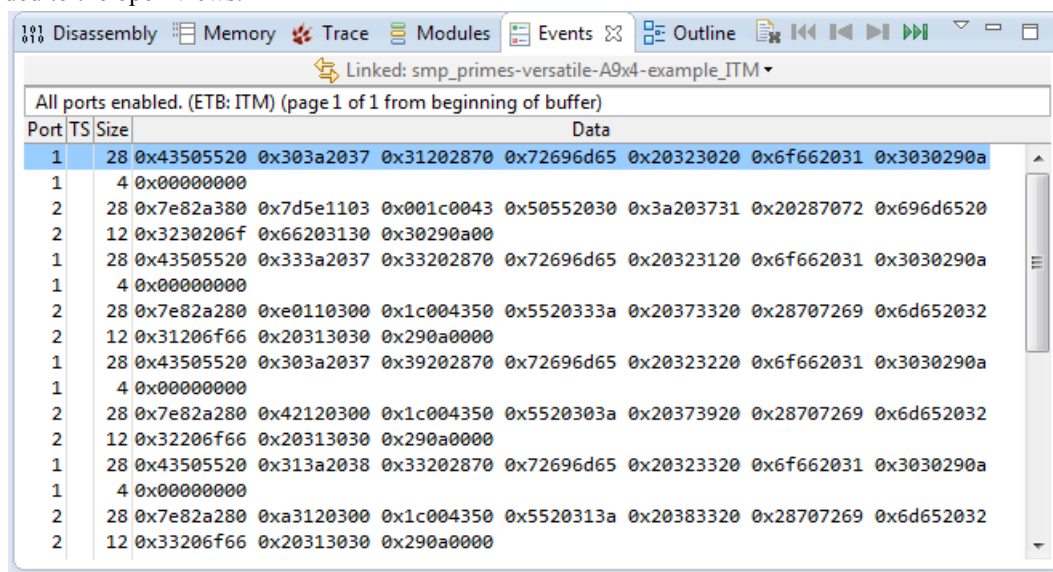


Figure 10-10 Events view (Shown with all ports enabled for an ETB:ITM trace source)

Note

Use the Event Viewer Settings dialog to select a **Trace Source** as well as to set up **Ports** (if ITM is the trace source) or **Masters** (if STM is the trace source) 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 different 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.

Clear Trace

Clears the debug hardware device buffer and all trace sources. The views might retain data, but after clearing trace, refreshing the views clears them as well.

Start of page

Displays events from the beginning of the trace buffer.

Page back

Moves one page back in the trace buffer.

Page forward

Moves one page forward in the trace buffer.

End of page

Displays events from the end of the trace buffer.

View Menu

The following **View Menu** options are available:

New Events View

Displays a new instance of the **Events** view.

Find Timestamp...

Displays the Search by Timestamp dialog which allows you to search through the entire trace buffer. The search results opens up the page where the timestamp is found and selects the closest timestamp.

Update View When Hidden

Enables the updating of the view when it is hidden behind other views. By default this view does not update when hidden.

Refresh

Refreshes the view.

Freeze Data

Toggles the freezing of data in the current view.

Events Settings...

Displays the Settings dialog where you can select a trace source and set options for the selected trace source.

Open Trace Control View

Opens the Trace Control View.

Events content menu

Timestamps

Timestamps represent the approximate time when events are generated.

Synchronize Timestamps

Synchronizes all Trace and Events views to display data around the same timestamp.

Set Timestamp Origin

Sets the selected event record as the timestamp origin.

Note

For a given connection, the timestamp origin is global for all Trace and Events views.

Clear Timestamp Origin

Clears the timestamp origin.

Timestamp Format: Numeric

Sets the timestamp in numeric format. This is the raw timestamp value received from the ITM/STM protocol.

Timestamp Format: (h:m:s)

Sets the timestamp in **hours:minutes:seconds** format.

Related references

[Chapter 10 DS-5 Debug Perspectives and Views on page 10-200.](#)

[10.10 Event Viewer Settings dialog box on page 10-227.](#)

Related information

[CoreSight™ Components Technical Reference Manual.](#)

[CoreSight™ System Trace Macrocell Technical Reference Manual.](#)

10.10 Event Viewer Settings dialog box

Use the **Event Viewer Settings** dialog to select a **Trace Source** as well as to set up **Ports** (if ITM is the trace source) or **Masters** (if STM is the trace source) to display in the Events view.

General settings

Select a Trace Source

Selects the required trace source from the list.

Height

The number of lines to display per results page. The default is 100 lines.

Width

The number of characters to display per line. The default is 80 characters.

Import

Imports an existing Event Viewer Settings configuration file. This file contains details about the **Trace Source** and **Ports** (in the case of ITM trace) or **Masters** and **Channels** (in the case of STM trace) used to create the configuration.

Export

Exports the currently displayed Event Viewer Settings configuration to use with a different Events view.

OK

Reorganizes the current channels into a canonical form, saves the settings, and closes the dialog box.

Cancel

Enables you to cancel unsaved changes.

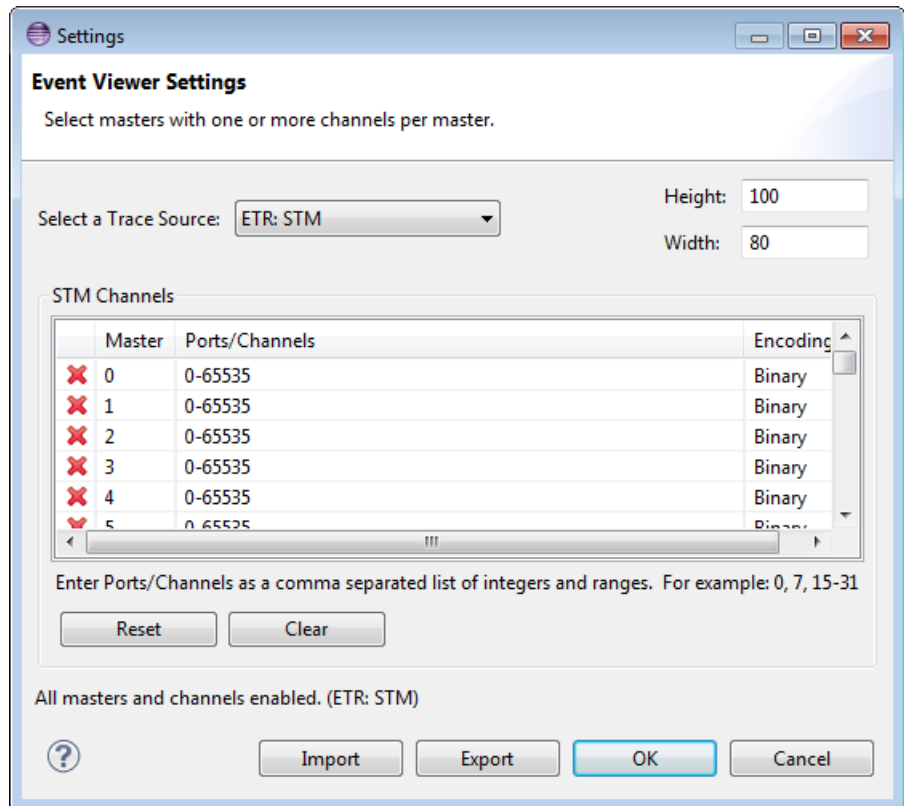


Figure 10-11 Event Viewer Settings (Shown with all Masters and Channels enabled for an ETR:STM trace source)

For ITM trace sources

Ports

Click to add or delete a **Port**.

Encoding

Select the type of encoding you want for the data associated with the port. The options available are **Binary**, **Text**, and **TAE**.

Reset

Click **Reset** to display and enable all available Ports for the selected ITM trace source.

————— **Note** —————

This clears any custom settings.

Clear

Click to clear all **Ports**.

For STM trace sources

Masters

Click to add or delete **Masters** and **Channels** that you want to display in the **Events** view.

————— **Note** —————

Masters are only available for STM trace.

Encoding

Select the type of encoding you want for the data associated with the channels. The options available are **Binary** and **Text**.

Reset

Click **Reset** to display and enable all available **Masters** and **Channels** for the selected STM trace source.

————— **Note** —————

This clears any custom settings.

Clear

Click to clear all **Masters** and **Channels**.

Related references

[10.9 Events view on page 10-225.](#)

Related information

[CoreSight™ Components Technical Reference Manual.](#)

[CoreSight™ System Trace Macrocell Technical Reference Manual.](#)

10.11 Expressions view

Use the Expressions view to create and work with expressions.

Name	Value	Type	Count	Size	Location	Access
(struct thread_info*)(SSP_SVC & ~0x1FFF)	0x00000000	struct thread_info*	1	32		RO
flags	3892248592	long unsigned int		32	S:0x00000000	R/W
preempt_count	-402593792	int		32	S:0x00000004	R/W
addr_limit	3892248592	mm_segment_t		32	S:0x00000008	R/W
task	0xE800E800	struct task_struct*	1	32	S:0x0000000C	R/W
exec_domain	0xE7FF0010	struct exec_domain*	1	32	S:0x00000010	R/W
cpu	3892373504	_u32		32	S:0x00000014	R/W
cpu_domain	3892248592	_u32		32	S:0x00000018	R/W
cpu_context		struct cpu_context_save		384	S:0x0000001C	R/W
syscall	3892373504	_u32		32	S:0x0000004C	R/W
used_cp	"\x10"	_u8[16]	16	128	S:0x00000050	R/W
tp_value	3892248592	long unsigned int		32	S:0x00000060	R/W
crunchstate		struct crunch_state		1472	S:0x00000064	R/W
fpstate		union fp_state		1120	S:0x00000120	R/W
vfpstate		union vfp_state		2240	S:0x000001B0	R/W
restart_block		struct restart_block		320	S:0x000002C8	R/W

Enter new expression here.

Figure 10-12 Expressions view

You can:

Add expressions

Enter your expression in the **Enter new expression here** field and press Enter on your keyboard. This adds the expression to the view and displays its value.

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`.


Edit expressions

You can edit the values of expressions in the **Value** field. Select the value and edit it.


Delete expressions

In the Expressions view, select the expression you want to remove from view, and click  to remove the selected expression. If you want to remove all expressions, click .

Tip

 You can also use the **Delete** key on your keyboard to delete the registers.

Refresh view

To refresh or update the values in the view, click .

Example 10-1 Examples

When debugging the Linux kernel, to view its internal thread structure, use these expressions:

For ARMv7 in SVC mode, with 8K stack size:

```
(struct thread_info*)($SP_SVC & ~0x1FFF)
```

For ARMv8 AArch64 in EL1, with 16K stack size:

```
(struct thread_info*)($SP_EL1 & ~0x3FFF)
```

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 different 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.

Remove Selected Expression

Removes the selected expression from the list.

Remove All Expressions

Removes all expressions from the list.

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 Registers

If the expression corresponds to a register, this displays the Registers view with that register selected.

Show in Memory

Where enabled, this displays the Memory view with the address set to either:

- The value of the selected expression, if the value translates to an address, for example the address of an array, `&name`
- The location of the expression, for example the name of an array, `name`.

The memory size is set to the size of the expression, using the **sizeof** keyword.

Show Dereference in Memory

If the selected expression is a pointer, this displays the Memory view with the address set to the value of the expression.

Show in Disassembly

Where enabled, this displays the Disassembly view with the address set to the location of the expression.

Show Dereference in Disassembly

If the selected expression is a pointer, this displays the Disassembly view, with the address set to the value of the expression.

Translate Variable Address

Displays the MMU view and translates the address of the variable.

Toggle Watchpoint

Displays the Add Watchpoint dialog to set a watchpoint on the selected variable, or removes the watchpoint if one has been set.

Enable Watchpoint

Enables the watchpoint, if a watchpoint has been set on the selected variable.

Disable Watchpoint

Disables the watchpoint, if a watchpoint has been set on the selected variable.

Resolve Watchpoint

If a watchpoint has been set on the selected variable, this re-evaluates the address of the watchpoint. If the address can be resolved the watchpoint is set, otherwise it remains pending.

Watchpoint Properties

Displays the Watchpoint Properties dialog box. This enables you to control watchpoint activation.

Send to

Enables you to add register 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 expression value.

View Menu

The following **View Menu** options are available:

New Expressions View

Displays a new instance of the Expressions view.

Update View When Hidden

Enables the updating of the view when it is hidden behind other views. By default, this view does not update when hidden.

Freeze Data

Toggles the freezing of data in the current view. This also disables and enables the **Refresh** option.

Adding a new column header

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`, or a register, for example `$R1`.

Value

The value of the expression. You can modify a value that has a white background. A yellow background indicates that the value has changed. This might result from you either performing a debug action such as stepping or by you editing the value directly.

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 the expression in bits.

Location

The address in hexadecimal identified by the expression, or the name of a register, if the expression contains only a single register name.

Access

The access type of the expression.

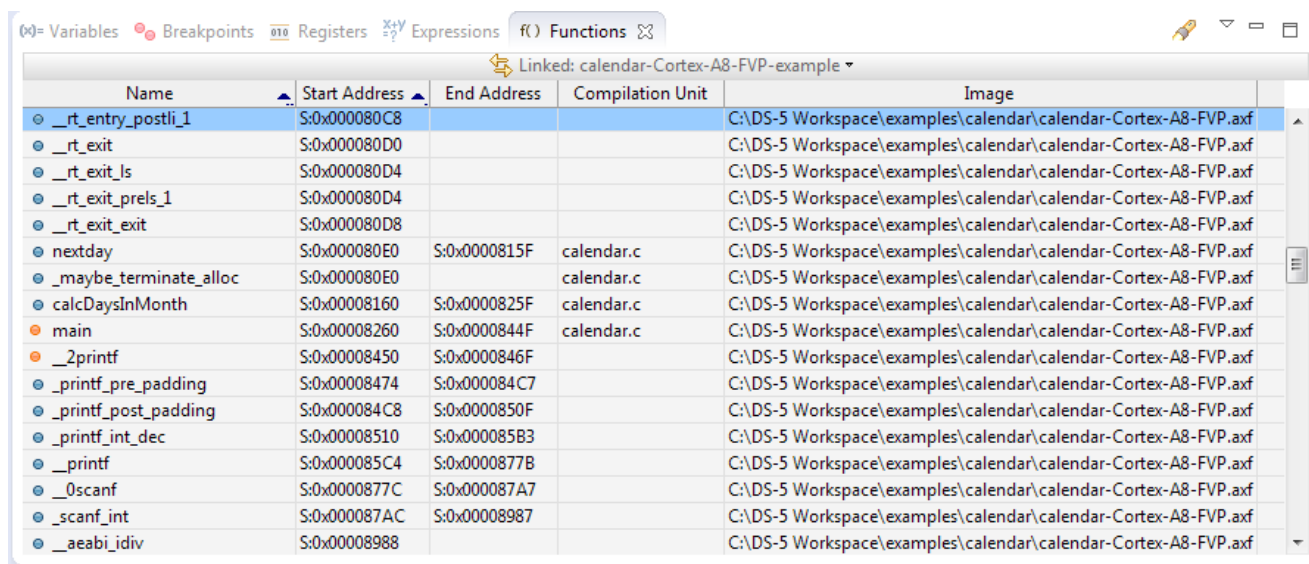
All columns are displayed by default.

Related references

4.5 Working with breakpoints and watchpoints on page 4-109.
4.8 Setting a tracepoint on page 4-121.
4.6 Working with conditional breakpoints on page 4-116.
4.6.1 Assigning conditions to an existing breakpoint on page 4-116.
4.7 About pending breakpoints and watchpoints on page 4-120.
4.5.5 Exporting DS-5 breakpoint settings to a file on page 4-113.
4.5.4 Importing DS-5 breakpoint settings from a file on page 4-113.
Chapter 10 DS-5 Debug Perspectives and Views on page 10-200.

10.12 Functions view

Use the **Functions** view to display the ELF data associated with function symbols for the loaded images. You can freeze the view to prevent the information being updated by a running target.



Name	Start Address	End Address	Compilation Unit	Image
• _rt_entry_postli_1	S:0x000080C8			C:\DS-5 Workspace\examples\calendar\calendar-Cortex-A8-FVP.axf
• _rt_exit	S:0x000080D0			C:\DS-5 Workspace\examples\calendar\calendar-Cortex-A8-FVP.axf
• _rt_exit_ls	S:0x000080D4			C:\DS-5 Workspace\examples\calendar\calendar-Cortex-A8-FVP.axf
• _rt_exit_prels_1	S:0x000080D4			C:\DS-5 Workspace\examples\calendar\calendar-Cortex-A8-FVP.axf
• _rt_exit_exit	S:0x000080D8			C:\DS-5 Workspace\examples\calendar\calendar-Cortex-A8-FVP.axf
• nextday	S:0x000080E0	S:0x0000815F	calendar.c	C:\DS-5 Workspace\examples\calendar\calendar-Cortex-A8-FVP.axf
• _maybe_terminate_alloc	S:0x000080E0		calendar.c	C:\DS-5 Workspace\examples\calendar\calendar-Cortex-A8-FVP.axf
• calcDaysInMonth	S:0x00008160	S:0x0000825F	calendar.c	C:\DS-5 Workspace\examples\calendar\calendar-Cortex-A8-FVP.axf
• main	S:0x00008260	S:0x0000844F	calendar.c	C:\DS-5 Workspace\examples\calendar\calendar-Cortex-A8-FVP.axf
• _2printf	S:0x00008450	S:0x0000846F		C:\DS-5 Workspace\examples\calendar\calendar-Cortex-A8-FVP.axf
• _printf_pre_padding	S:0x00008474	S:0x000084C7		C:\DS-5 Workspace\examples\calendar\calendar-Cortex-A8-FVP.axf
• _printf_post_padding	S:0x000084C8	S:0x0000850F		C:\DS-5 Workspace\examples\calendar\calendar-Cortex-A8-FVP.axf
• _printf_int_dec	S:0x00008510	S:0x000085B3		C:\DS-5 Workspace\examples\calendar\calendar-Cortex-A8-FVP.axf
• _printf	S:0x000085C4	S:0x0000877B		C:\DS-5 Workspace\examples\calendar\calendar-Cortex-A8-FVP.axf
• _0scanf	S:0x0000877C	S:0x000087A7		C:\DS-5 Workspace\examples\calendar\calendar-Cortex-A8-FVP.axf
• _scanf_int	S:0x000087AC	S:0x00008987		C:\DS-5 Workspace\examples\calendar\calendar-Cortex-A8-FVP.axf
• __aeabi_idiv	S:0x00008988			C:\DS-5 Workspace\examples\calendar\calendar-Cortex-A8-FVP.axf

Figure 10-13 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 name of the compilation unit containing the function.

Image

The location of the ELF image containing the function.

Show All Columns

Displays all columns.

Reset Columns

Resets the columns displayed and their widths to the default.

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 different 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.

Toggle Hardware Breakpoint

Sets or removes a hardware 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 Functions View

Displays a new instance of the **Functions** view.

Update View When Hidden

Enables the updating of the view when it is hidden behind other views. By default this view does not update when hidden.

Refresh

Refreshes the view.

Freeze Data

Toggles the freezing of data in the current view. This also disables or enables the **Refresh** option.

Filters...

Displays the Functions Filter dialog box. This enables you to filter the functions displayed in the view.

Related references

[Chapter 10 DS-5 Debug Perspectives and Views on page 10-200.](#)

10.13 History view

Use the **History** view to display a list of the commands generated during the current debug session.

It also enables you to:

- Clear its contents.
- Select commands and save them as a script file. You can 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.

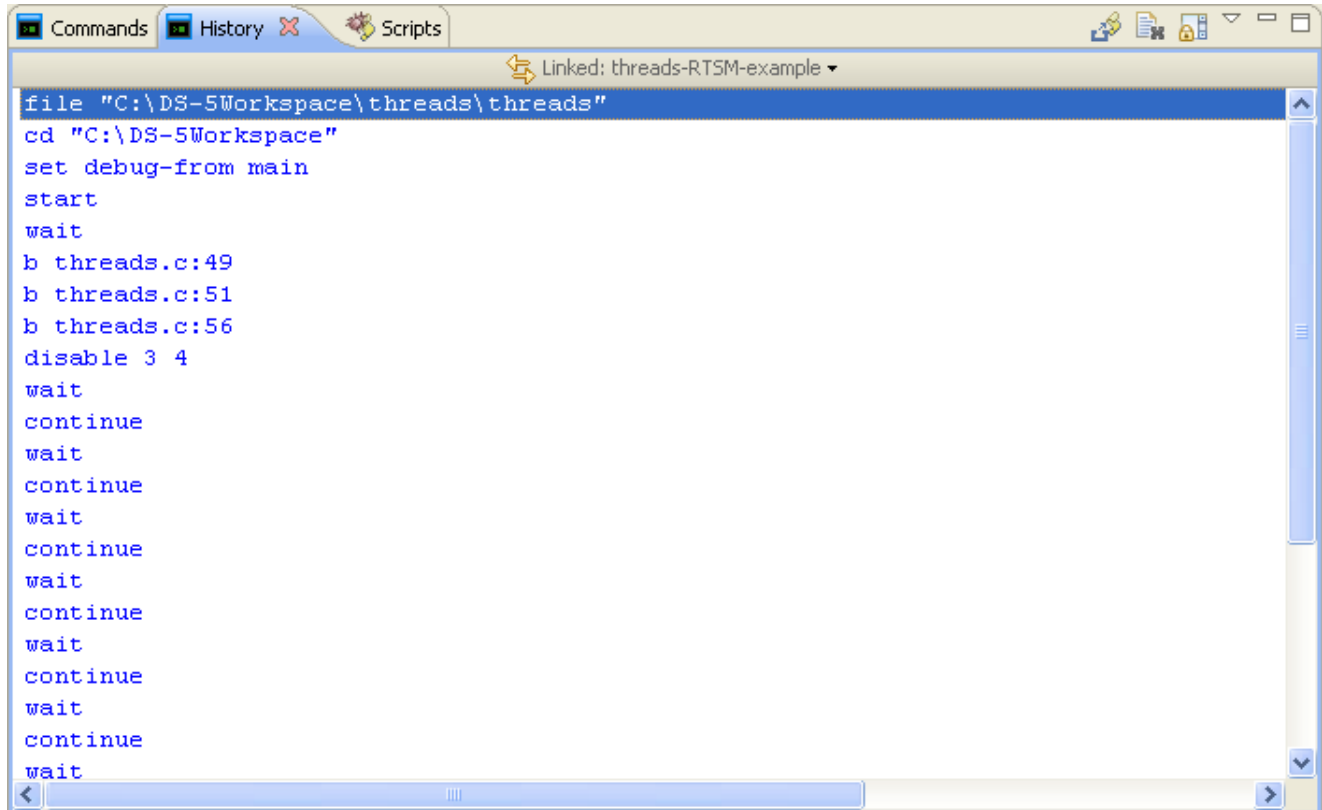


Figure 10-14 History view

Note

Default settings for this view are controlled by a DS-5 Debugger setting in the Preferences dialog box. For example, the default location 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:

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 different 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 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

[Chapter 10 DS-5 Debug Perspectives and Views on page 10-200.](#)

10.14 Memory view

Use the **Memory** view to display and modify the contents of memory.

This view enables you to:

- Specify the start address for the view, either as an absolute address or as an expression, for example `$pc+256`. You can also specify an address held in a register by dragging and dropping the register from the **Registers** view into the **Memory** view. Previous entries are listed in a drop-down list which is cleared when you exit Eclipse.
- Specify the display size of the **Memory** view in bytes, as an offset value from the start address.
- 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 view to prevent it from being updated by a running target.

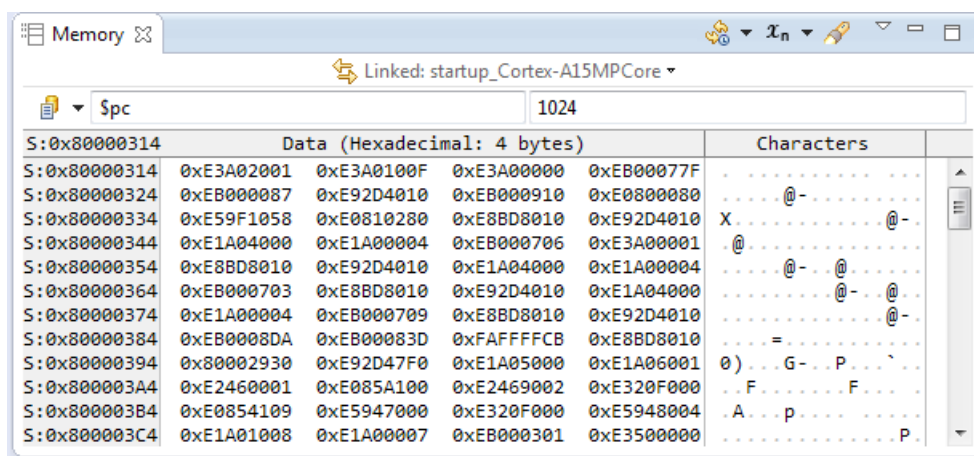


Figure 10-15 Memory view

The **Memory** view only provides the facility to modify how memory is displayed in this view. It does not enable you to change the access width for the memory region. 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 different 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.

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.

Format

Click to cycle through the memory cell formats and cell widths, or select a format from the drop-down menu. The default is hexadecimal with a display width of 4 bytes.

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.

Addresses and expressions you specify are added to the drop down history 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.

Update View When Hidden

Enables the updating of the view when it is hidden behind other views. By default this view does not update when hidden.

Show Tooltips

Toggles the display of tooltips on memory cell values.

Auto Alignment

Aligns the memory view to the currently selected data width.

Show Compressed Addresses

Shows the least significant bytes of the address that are not repeating.

Show Cache

Shows how the core views the memory from the perspective of the different caches on the target. This is disabled by default. When showing cache, the view is auto-aligned to the cache-line size. When showing cache, the memory view shows a column for each cache. The cache columns display the state of each cache-line if it is populated.

Click on a cache column header or select a cache from the **Cache Data** menu to display the data as viewed from that cache. The Memory (non-cached) option from the **Cache Data** menu shows the data in memory, as if all caches are disabled.

Note

In multiprocessor systems it is common to have caches dedicated to particular cores. For example, a dual-core system might have per-core L1 caches, but share a single L2 cache. Cache snooping is a hardware feature that allows per-core caches to be accessed from other cores. In such cases the **Cache Data** field shows all the caches that are accessible to each core, whether directly or through snooping.

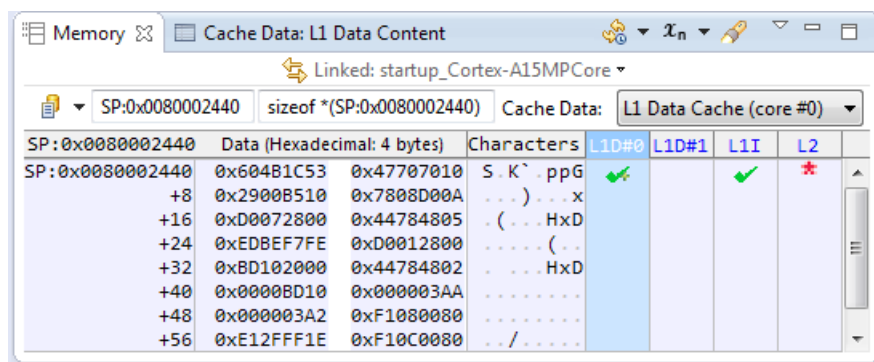


Figure 10-16 Memory view with Show Cache

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 context menu of the column header enables you to toggle the display of the individual columns.

Reset Columns

Displays the default columns.

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.

Select All

Selects all the addresses.

The following additional options are available on the context menu when you select a memory cell value:

Toggle Watchpoint

Sets or removes a watchpoint at the selected address.

Toggle Breakpoint

Sets or removes a breakpoint at the selected address.

Toggle Hardware Breakpoint

Sets or removes a hardware 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.

Translate Address <address>

Displays the **MMU** view and translates the address of the selected value in memory.

The following additional options are available on the context menu when you select a memory cell with a breakpoint:

Enable Breakpoint

Enables the breakpoint at the selected address.

Disable Breakpoint

Disables the breakpoint at the selected address.

Remove Breakpoint

Removes the breakpoint at the selected address.

Resolve Breakpoint

Resolves a pending breakpoint at the selected address.

Breakpoint Properties...

Displays and lets you change the breakpoint properties.

Related concepts

[6.8 About debugging multi-threaded applications](#) on page 6-149.

[6.9 About debugging shared libraries](#) on page 6-150.

[6.10.2 About debugging a Linux kernel](#) on page 6-153.

[6.10.3 About debugging Linux kernel modules](#) on page 6-154.

[6.11 About debugging TrustZone enabled targets](#) on page 6-157.

Related references

[4.5 Working with breakpoints and watchpoints](#) on page 4-109.

[4.8 Setting a tracepoint](#) on page 4-121.

[4.6 Working with conditional breakpoints](#) on page 4-116.

[4.6.1 Assigning conditions to an existing breakpoint](#) on page 4-116.

[4.7 About pending breakpoints and watchpoints](#) on page 4-120.

[4.5.5 Exporting DS-5 breakpoint settings to a file](#) on page 4-113.

4.5.4 Importing DS-5 breakpoint settings from a file on page 4-113.
Chapter 10 DS-5 Debug Perspectives and Views on page 10-200.

10.15 MMU view

Use the **MMU** view to perform address translations or for an overview of the translation tables and virtual memory map.

This view enables you to:

- Perform simple virtual to physical address translation.
- Perform simple physical to virtual address translation.
- Perform MMU page table walks.
- See an overview of the virtual memory map.
- Freeze the view to prevent it from being updated by a running target.

MMU Translation tab

The Translation tab enables you to translate:

- Virtual address to physical address.
- Physical address to one or more virtual addresses.

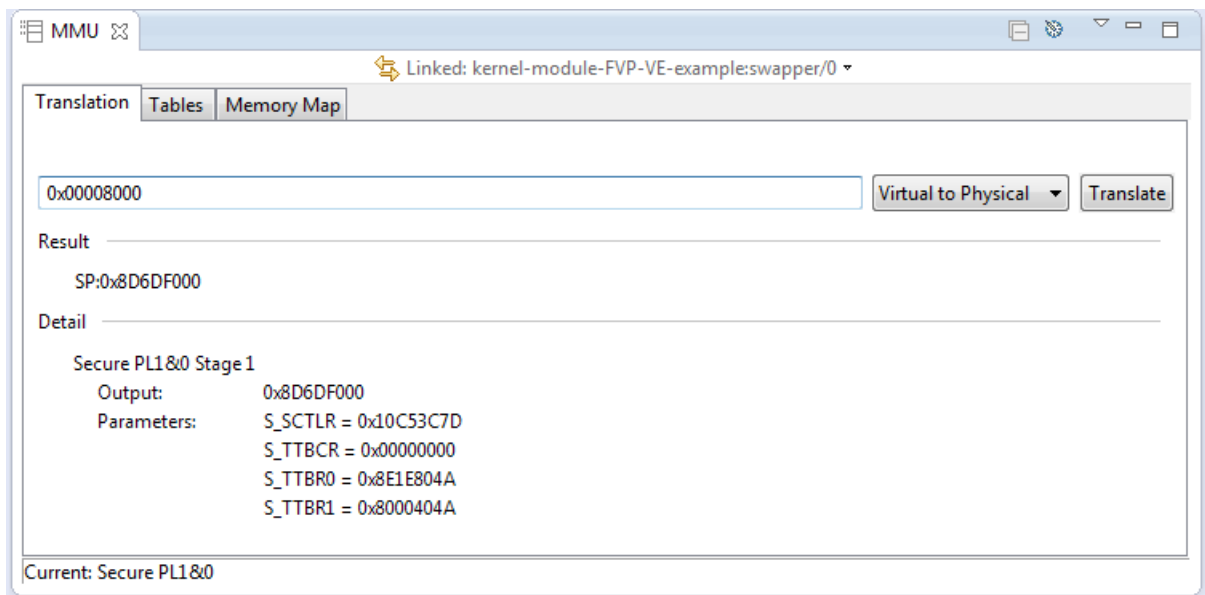


Figure 10-17 MMU Translation tab view

To perform an address translation in the Translation tab:

1. Enter a physical or virtual address in the address field. You can also enter an expression that evaluates to an address.
2. Select **Physical to Virtual** or **Virtual to Physical** depending on the translation type.
3. Click **Translate** to perform the address translation.

The Result shows the output address after the translation. The view also shows the details of the translation regime and parameters. You can customize these parameters using the MMU Settings dialog.

MMU Tables tab

Use the Tables tab to see the translation tables used by the selected translation regime. You can change the translation regime using the **MMU Settings** dialog.

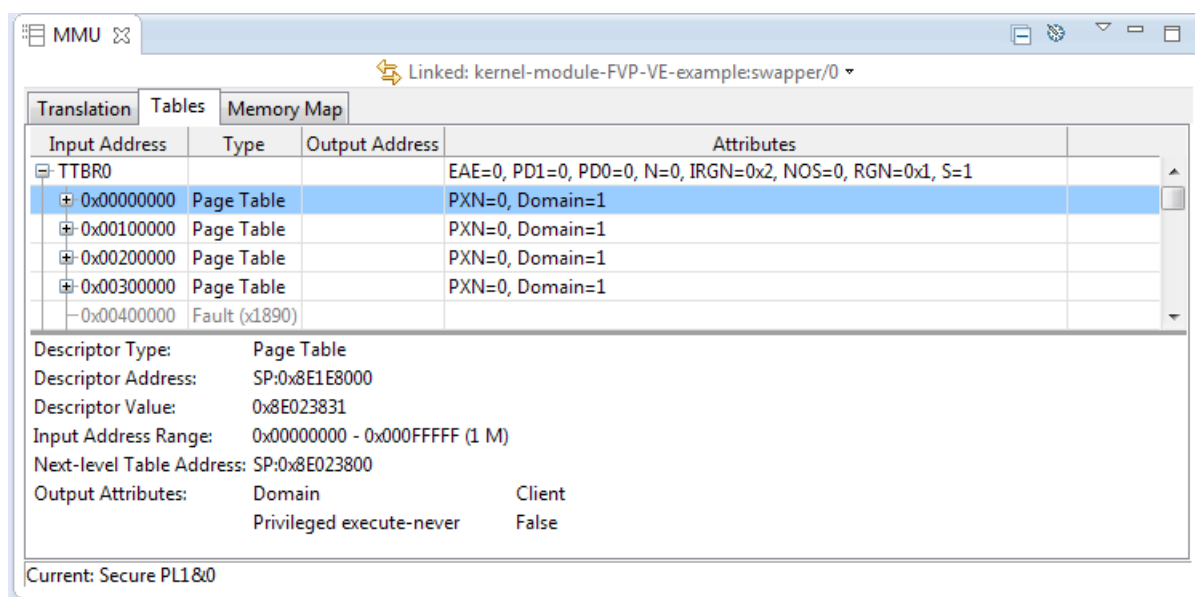


Figure 10-18 MMU Tables tab view

The Tables tab contains the following columns:

Input Address

Specifies the input address to the translation table. This is usually the virtual address, but it can also be an intermediate physical address.

Type

Specifies the type of entry in the translation table, for example Page Table, Section, Super Section, Small Page, or Large Page.

Output Address

Specifies the output address from the translation table. This is usually the physical address, but it can also be an intermediate physical address.

Attributes

Specifies the memory attributes for the memory region.

The Tables tab also provides additional information for each row of the translation table:

Descriptor Address

Specifies the address of the selected translation table location.

Descriptor Value

Specifies the content of the selected translation table location.

Input Address Range

Specifies the range of input addresses that are mapped by the selected translation table location.

Next-level Table Address

Specifies the Descriptor Address for the next level of lookup in the translation table.

Memory Map tab

The Memory Map provides a view of the virtual memory layout by combining translation table entries that map to contiguous regions of physical memory with common memory type, cacheability, shareability, and access attributes.

Virtual Range	Physical Range	Type	AP	C	S	X
SP:0x00000000-0x2BFFFFFF	<unmapped>					
SP:0x2C000000-0x2C0FFFFFF	SP:0x2C000000-0x2C0FFFFFF	Device	RW		✓	
SP:0x2C100000-0x7FFFFFFF	<unmapped>					
SP:0x80000000-0x800FFFFFF	SP:0x80000000-0x800FFFFFF	Normal	RW		✓	✓
SP:0x80100000-0xFFFFFFFF	<unmapped>					

Current: Secure PL10

Figure 10-19 Memory Map tab 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 different 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.

MMU settings

This enables you to change the translation regime and input parameters. It contains:

MMU Settings

Translation Regimes and Stages

<Follow System>

☒ Use current translation settings

☐ Use custom translation settings

Parameters

N_TTBR1	
N_TTBCR	
N_SCTLR	
N_TTBR0	
VTTBR	
VTCR	
HSCTLR	

OK Cancel

Figure 10-20 MMU settings

The MMU Settings dialog contains:

Translation Regimes and Stages

Use this to select the translation you want the debugger to use. The field lists the translation regimes and stages that the debugger is aware of. See the *ARM Architecture Reference Manual* for more information on the translation regimes.

Select <**Follow System**> to let the debugger follow the current system state. If the current system state has more than one translation stage, then DS-5 Debugger combines the translation stages when using <**Follow System**>.

Use current translation settings

Use this to instruct the debugger to use the current translation settings for the selected translation.

Use custom translation settings

Use this to instruct the debugger to override the current translation settings.

Parameters

Use this to specify override values for custom settings. For example you can change the address in TTBR0 or TTBR1.

View Menu

The following **View Menu** options are available:

New MMU View

Displays a new instance of the MMU view.

Update View When Hidden

Enables the updating of the view when it is hidden behind other views. By default this view does not update when hidden.

Refresh

Refreshes the view.

Freeze Data

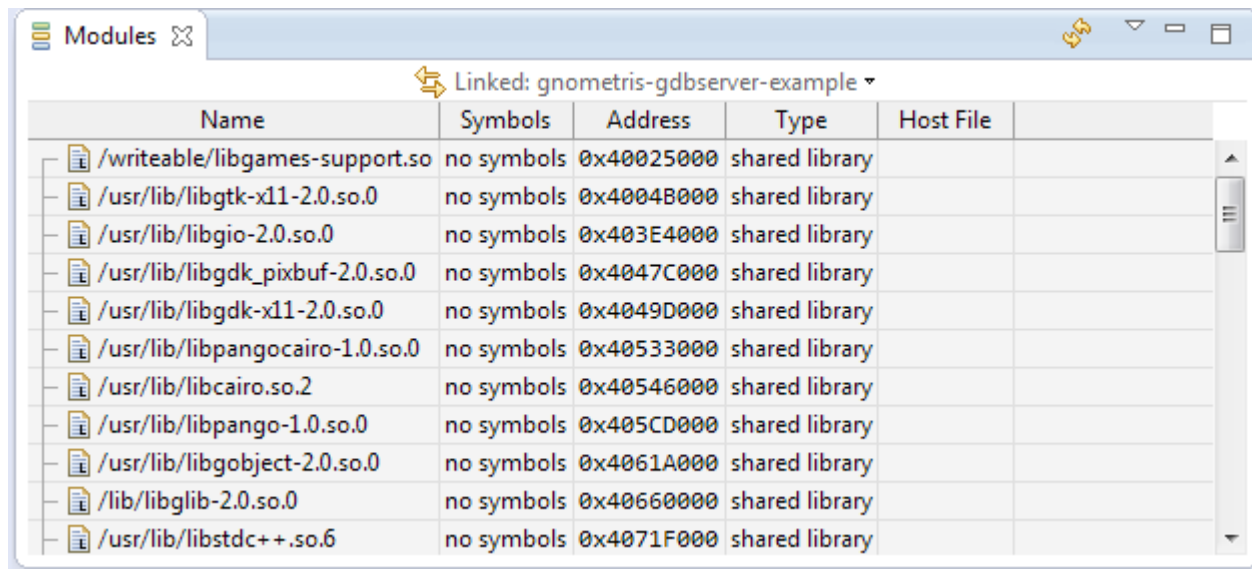
Toggles the freezing of data in the current view. This also disables or enables the **Refresh** option.

Coalesce Invalid Entries

Condenses the contiguous rows of faulty or invalid input addresses into a single row in the **Tables** tab.

10.16 Modules view

Use the **Modules** view to display a tabular view of the shared libraries and dynamically loaded *Operating System* (OS) modules used by the application. It is only populated when connected to a Linux target.



The screenshot shows the 'Modules' view in the DS-5 debugger. The title bar says 'Modules' with a close button. Below the title bar, there's a dropdown menu showing 'Linked: gnometris-gdbserver-example'. The main area contains a table with the following columns: Name, Symbols, Address, Type, and Host File. The table lists several shared libraries, all of which are of type 'shared library' and have 'no symbols' loaded. The libraries are located in various paths like /writeable/, /usr/lib/, and /lib/.

Name	Symbols	Address	Type	Host File
/writeable/libgames-support.so	no symbols	0x40025000	shared library	
/usr/lib/libgtk-x11-2.0.so.0	no symbols	0x40048000	shared library	
/usr/lib/libgio-2.0.so.0	no symbols	0x403E4000	shared library	
/usr/lib/libgdk_pixbuf-2.0.so.0	no symbols	0x4047C000	shared library	
/usr/lib/libgdk-x11-2.0.so.0	no symbols	0x4049D000	shared library	
/usr/lib/libpangocairo-1.0.so.0	no symbols	0x40533000	shared library	
/usr/lib/libcairo.so.2	no symbols	0x40546000	shared library	
/usr/lib/libpango-1.0.so.0	no symbols	0x405CD000	shared library	
/usr/lib/libgobject-2.0.so.0	no symbols	0x4061A000	shared library	
/lib/libglib-2.0.so.0	no symbols	0x40660000	shared library	
/usr/lib/libstdc++.so.6	no symbols	0x4071F000	shared library	

Figure 10-21 Modules view showing shared libraries

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.

Type

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.

Show All Columns

Displays all columns.

Reset Columns

Resets the columns displayed and their widths to the default.

The Name, Symbols, Address, Type, 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 different 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.

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.

View Menu

The following **View Menu** options are available:

Update View When Hidden

Enables the updating of the view when it is hidden behind other views. By default this view does not update when hidden.

Refresh

Refreshes the view.

Related concepts

[6.8 About debugging multi-threaded applications](#) on page 6-149.

[6.9 About debugging shared libraries](#) on page 6-150.

[6.10.2 About debugging a Linux kernel](#) on page 6-153.

[6.10.3 About debugging Linux kernel modules](#) on page 6-154.

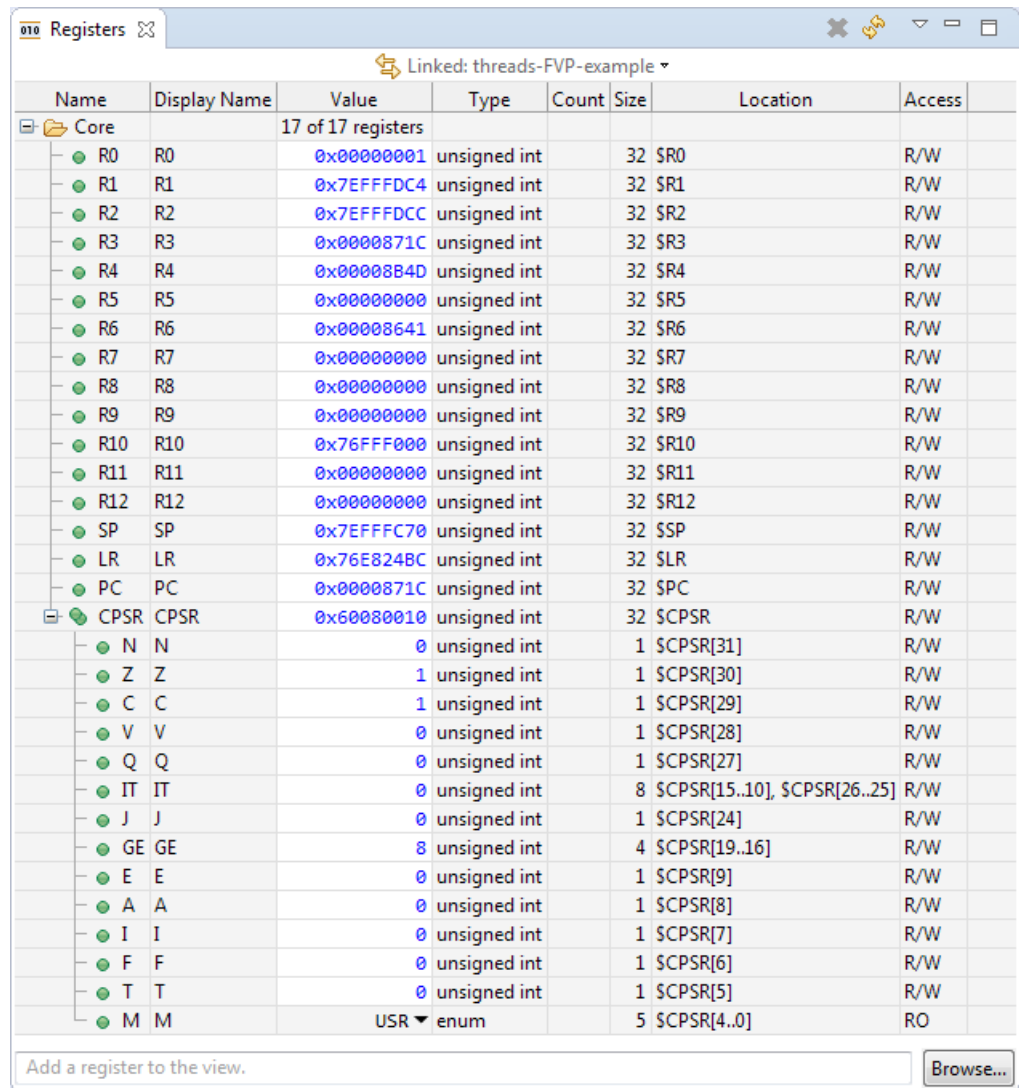
[6.11 About debugging TrustZone enabled targets](#) on page 6-157.

Related references

[Chapter 10 DS-5 Debug Perspectives and Views](#) on page 10-200.

10.17 Registers view

Use the Registers view to work with the contents of processor and peripheral registers available on your target.



Name	Display Name	Value	Type	Count	Size	Location	Access
Core 17 of 17 registers							
R0	R0	0x00000001	unsigned int		32	\$R0	R/W
R1	R1	0x7EFFFDC4	unsigned int		32	\$R1	R/W
R2	R2	0x7EFFFDC4	unsigned int		32	\$R2	R/W
R3	R3	0x0000871C	unsigned int		32	\$R3	R/W
R4	R4	0x00008B4D	unsigned int		32	\$R4	R/W
R5	R5	0x00000000	unsigned int		32	\$R5	R/W
R6	R6	0x00008641	unsigned int		32	\$R6	R/W
R7	R7	0x00000000	unsigned int		32	\$R7	R/W
R8	R8	0x00000000	unsigned int		32	\$R8	R/W
R9	R9	0x00000000	unsigned int		32	\$R9	R/W
R10	R10	0x76FFF000	unsigned int		32	\$R10	R/W
R11	R11	0x00000000	unsigned int		32	\$R11	R/W
R12	R12	0x00000000	unsigned int		32	\$R12	R/W
SP	SP	0x7EFFF000	unsigned int		32	\$SP	R/W
LR	LR	0x76E824BC	unsigned int		32	\$LR	R/W
PC	PC	0x0000871C	unsigned int		32	\$PC	R/W
CPSR	CPSR	0x60080010	unsigned int		32	\$CPSR	R/W
N	N	0	unsigned int	1		\$CPSR[31]	R/W
Z	Z	1	unsigned int	1		\$CPSR[30]	R/W
C	C	1	unsigned int	1		\$CPSR[29]	R/W
V	V	0	unsigned int	1		\$CPSR[28]	R/W
Q	Q	0	unsigned int	1		\$CPSR[27]	R/W
IT	IT	0	unsigned int	8		\$CPSR[15..10], \$CPSR[26..25]	R/W
J	J	0	unsigned int	1		\$CPSR[24]	R/W
GE	GE	8	unsigned int	4		\$CPSR[19..16]	R/W
E	E	0	unsigned int	1		\$CPSR[9]	R/W
A	A	0	unsigned int	1		\$CPSR[8]	R/W
I	I	0	unsigned int	1		\$CPSR[7]	R/W
F	F	0	unsigned int	1		\$CPSR[6]	R/W
T	T	0	unsigned int	1		\$CPSR[5]	R/W
M	M	USR	enum	5		\$CPSR[4..0]	RO

Add a register to the view. Browse...

Figure 10-22 Registers view

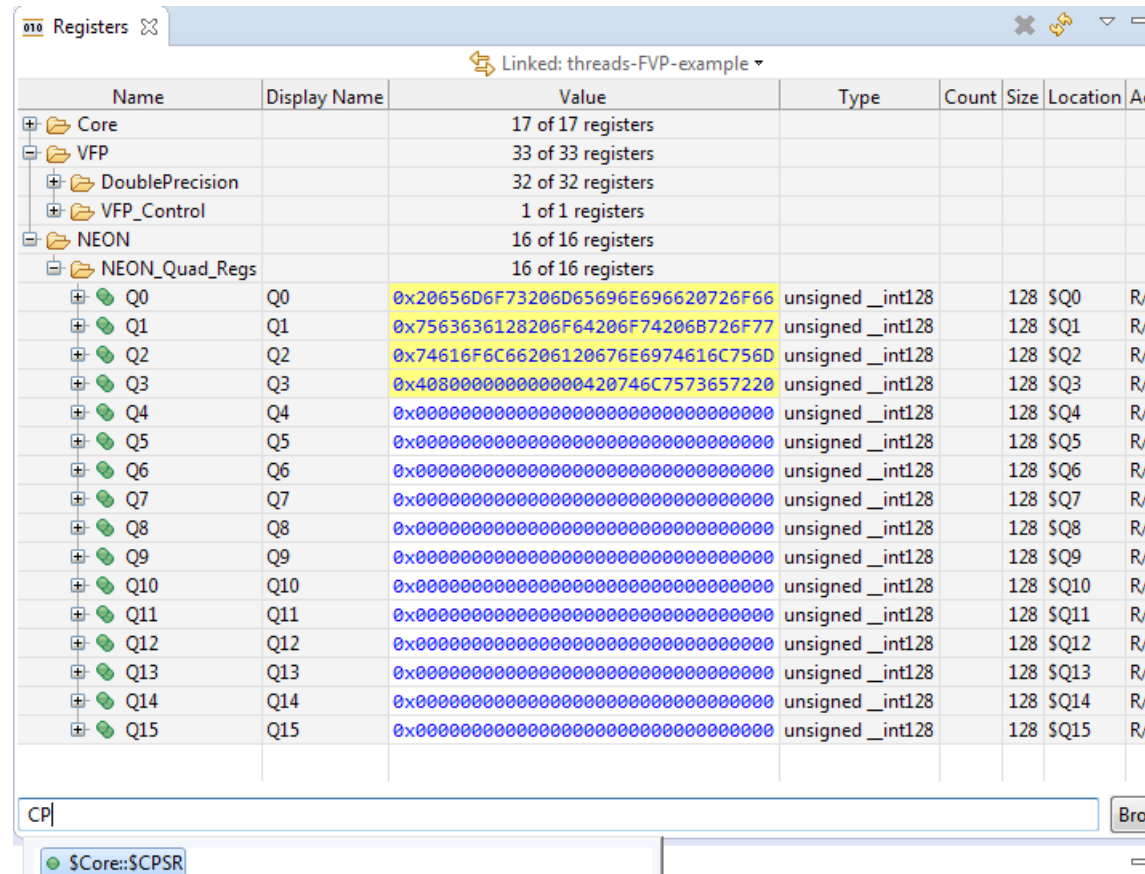
You can:

View registers available on your target

By default, the Registers view displays a common subset of the processor registers. You can add and remove other registers, either individually or in groups. Keep the set of registers in the view to the minimum to maintain good debug performance.

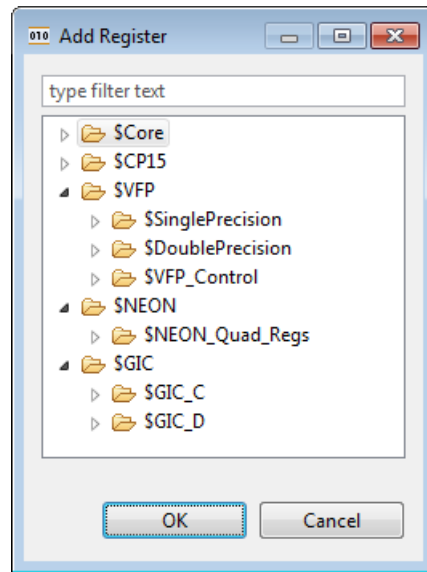
- Add a specific register or group of registers

If you know the name of the specific register or group you want to view, enter the register or group name in the **Add a register to the view** field. This lists the registers and groups that match the text you entered. For example, enter the text CP to view registers and groups with the text CP in their name. Double-click the register or group to add it to the Registers view.





- Browse and select multiple registers

If you want to view all available registers on your target click **Browse** to display the Add Register dialog. Then select one or more registers that you are interested in and click **OK** to add them to the Registers view. You can also select a register group folder if you want to view all the registers in that group. **Ctrl+A** selects all the available registers.



- Delete registers


In the Registers view, select the registers you want to remove from the view, and click  to delete the selected registers. If you want to reset the view to display the default registers again, then from the view menu select Reset to default, .

Tip



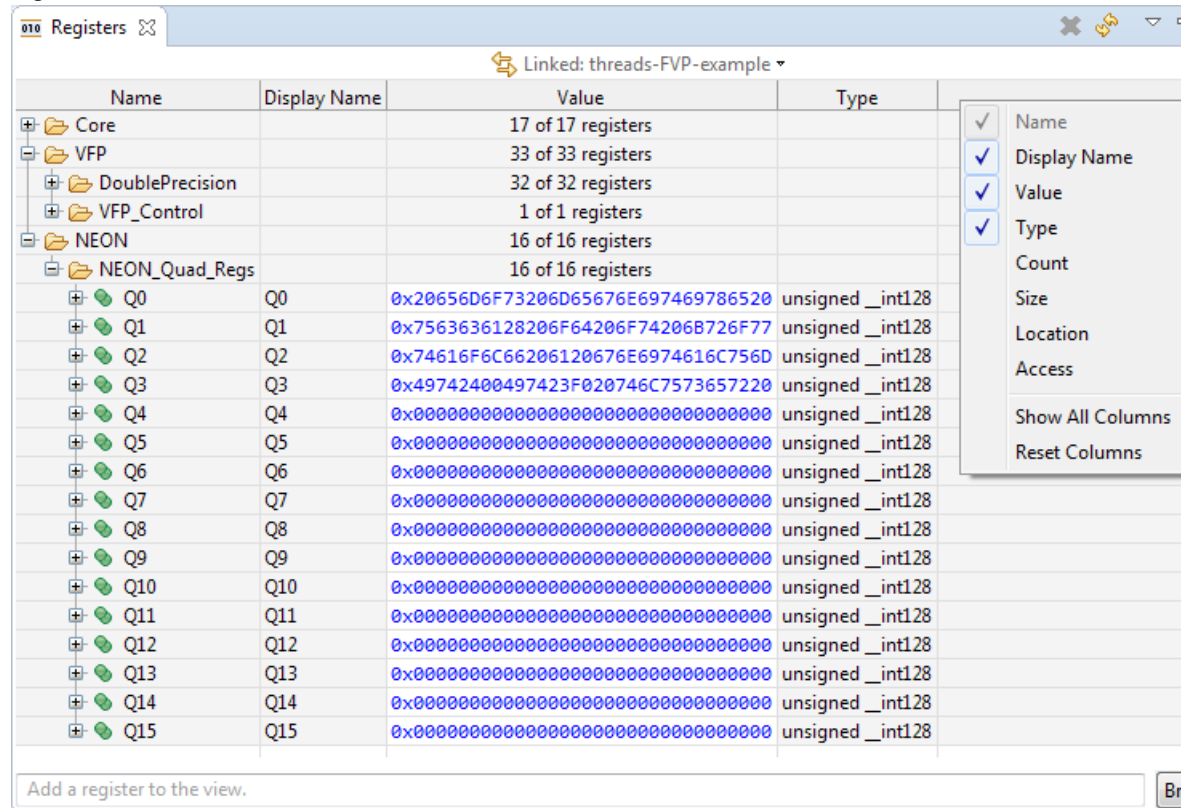
You can also use the **Delete** key on your keyboard to delete the registers.

- Refresh view

To refresh or update the values in the view, click .

Modify the value of write access registers

You can modify the values of registers with write access by clicking in the **Value** column for the register and entering a new value. Enable the **Access** column to view access rights for each register.



Drag and drop an address held in a register from the Registers view to other views

Drag and drop an address held in a register from this view into either the Memory view to see the memory at that address, or into the Disassembly view to disassemble from that address.

Change the display format of register values

You can set the format of individual bits for *Program Status Registers* (PSRs).

Freeze the selected view to prevent the values being updated by a running target

Select **Freeze Data** from the view menu to prevent values updating automatically when the view refreshes.

Toolbar and context menu options

The following options are available from the view 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 different 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.

Copy

Copies the selected registers. If a register contains bitfields, you must expand the bitfield to copy the individual bitfield values.

It can be useful to copy registers to a text editor in order to 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>

Displays the Memory view starting at the address held in the register.

Show Disassembly Pointed to By <register name>

Displays the Disassembly view starting at the address held in the register.

Translate Address in <register name>

Displays the MMU view and translates the address held in the register.

Send to <selection>

Displays a sub menu that enables you to add register filters to a specific Expressions view.

<Format list>

A list of formats you can use for the register values.

View Menu

The following **View Menu** options are available:

New Registers View

Creates a new instance of the Registers view.

Update View When Hidden

Enables the updating of the view when it is hidden behind other views. By default, this view does not update when hidden.

Reset to default registers

Resets the view to show only the default registers.

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:

Undo

Reverts the last change you made to the selected value.

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.

Select All

Selects the whole value.

Adding a new column header

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 the CPSR, enter the following command in the **Commands** view:

```
print $CPSR.M
```

Value

The value of the register. A yellow background indicates that the value has changed. This might result from you either performing a debug action such as stepping or by you editing the value directly.

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 bit range for a bitfield of a PSR. For example, bitfield M of the CPSR is displayed as \$CPSR[4..0].

Access

The access mode for the register.

Show All Columns

Displays all columns.

Reset Columns

Resets the columns displayed and their widths to the default.

The Name, Value, Size, and Access columns are displayed by default.

Related concepts

[6.8 About debugging multi-threaded applications](#) on page 6-149.

[6.9 About debugging shared libraries](#) on page 6-150.

[6.10.2 About debugging a Linux kernel](#) on page 6-153.

[6.10.3 About debugging Linux kernel modules](#) on page 6-154.

[6.11 About debugging TrustZone enabled targets](#) on page 6-157.

Related references

[4.5 Working with breakpoints and watchpoints](#) on page 4-109.

[4.8 Setting a tracepoint](#) on page 4-121.

[4.6 Working with conditional breakpoints](#) on page 4-116.

[4.6.1 Assigning conditions to an existing breakpoint](#) on page 4-116.

[4.7 About pending breakpoints and watchpoints](#) on page 4-120.

[4.5.5 Exporting DS-5 breakpoint settings to a file](#) on page 4-113.

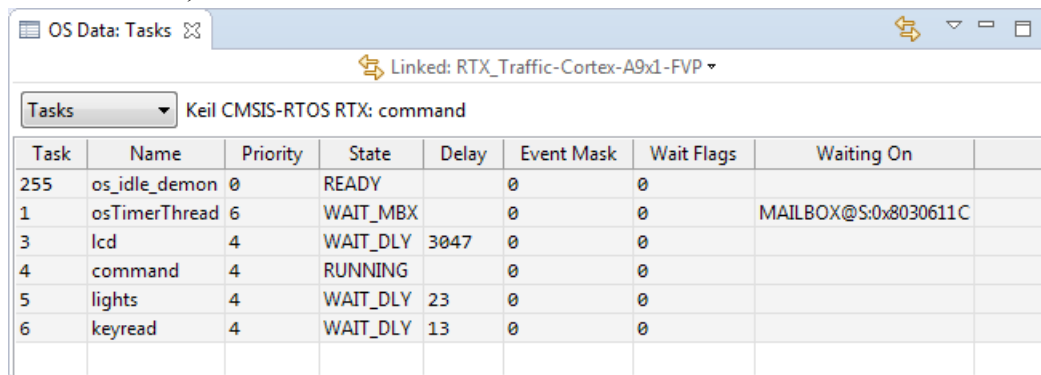
[4.5.4 Importing DS-5 breakpoint settings from a file](#) on page 4-113.

[Chapter 10 DS-5 Debug Perspectives and Views](#) on page 10-200.

10.18 OS Data view

Use the **OS Data** view to display information about the operating system, for example, tasks, semaphores, mutexes, and mailboxes.

To view the information, select a table from the list.



Task	Name	Priority	State	Delay	Event Mask	Wait Flags	Waiting On
255	os_idle_demon	0	READY		0	0	
1	osTimerThread	6	WAIT_MBX		0	0	MAILBOX@S:0x8030611C
3	lcd	4	WAIT_DLY	3047	0	0	
4	command	4	RUNNING		0	0	
5	lights	4	WAIT_DLY	23	0	0	
6	keyread	4	WAIT_DLY	13	0	0	

Figure 10-23 OS Data view (showing Keil CMSIS-RTOS RTX Tasks)

Note

Data in the OS Data view is dependent on the selected data source.

Toolbar and context menu options

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 different 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 OS Data View

Displays a new instance of the **OS Data** view.

Update View When Hidden

Enables the updating of the view when it is hidden behind other views. By default, this view does not update when hidden.

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.

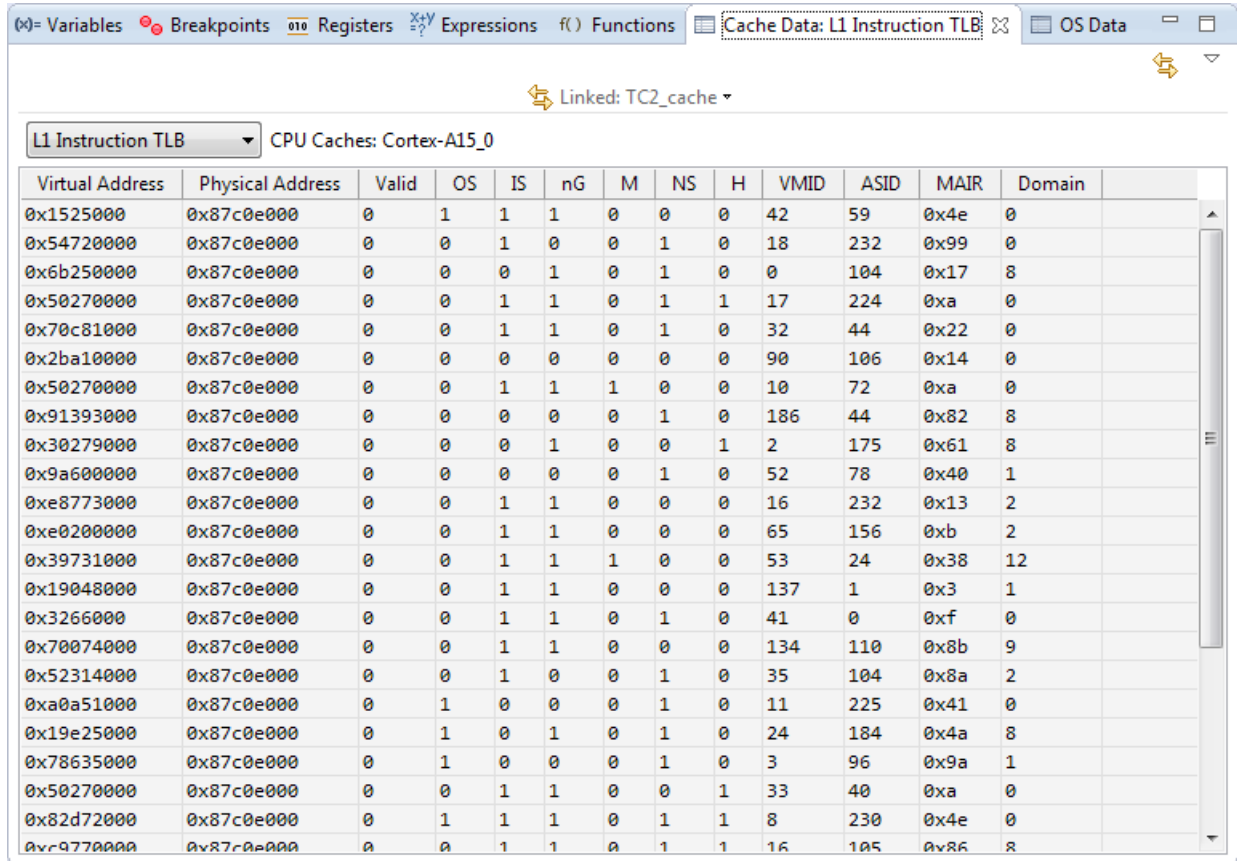
Select All

Selects all text.

10.19 Cache Data view

Use the **Cache Data** view to examine the contents of the caches in your system. For example, L1 cache or TLB cache. You must enable **Cache debug mode** in the DTSL Configuration Editor dialog.

Select the cache you want to view from the **CPU Caches** menu.



Virtual Address	Physical Address	Valid	OS	IS	nG	M	NS	H	VMID	ASID	MAIR	Domain
0x1525000	0x87c0e000	0	1	1	1	0	0	0	42	59	0x4e	0
0x5472000	0x87c0e000	0	0	1	0	0	1	0	18	232	0x99	0
0x6b25000	0x87c0e000	0	0	0	1	0	1	0	0	104	0x17	8
0x5027000	0x87c0e000	0	0	1	1	0	1	1	17	224	0xa	0
0x70c81000	0x87c0e000	0	0	1	1	0	1	0	32	44	0x22	0
0x2ba1000	0x87c0e000	0	0	0	0	0	0	0	90	106	0x14	0
0x5027000	0x87c0e000	0	0	1	1	1	0	0	10	72	0xa	0
0x91393000	0x87c0e000	0	0	0	0	0	1	0	186	44	0x82	8
0x30279000	0x87c0e000	0	0	0	1	0	0	1	2	175	0x61	8
0x9a600000	0x87c0e000	0	0	0	0	0	1	0	52	78	0x40	1
0xe8773000	0x87c0e000	0	0	1	1	0	0	0	16	232	0x13	2
0xe0200000	0x87c0e000	0	0	1	1	0	0	0	65	156	0xb	2
0x39731000	0x87c0e000	0	0	1	1	1	0	0	53	24	0x38	12
0x19048000	0x87c0e000	0	0	1	1	0	0	0	137	1	0x3	1
0x3266000	0x87c0e000	0	0	1	1	0	1	0	41	0	0xf	0
0x70074000	0x87c0e000	0	0	1	1	0	0	0	134	110	0x8b	9
0x52314000	0x87c0e000	0	0	1	0	0	1	0	35	104	0x8a	2
0xa0a51000	0x87c0e000	0	1	0	0	0	1	0	11	225	0x41	0
0x19e25000	0x87c0e000	0	1	0	1	0	1	0	24	184	0x4a	8
0x78635000	0x87c0e000	0	1	0	0	0	1	0	3	96	0x9a	1
0x5027000	0x87c0e000	0	0	1	1	0	0	1	33	40	0xa	0
0x82d72000	0x87c0e000	0	1	1	1	0	1	1	8	230	0x4e	0
0xc9770000	0x87c0e000	0	0	1	1	0	1	1	16	105	0x86	8

Figure 10-24 Cache Data view (showing L1 TLB cache)

Alternatively, you can use the `cache list` and `cache print` commands in the Commands view to show information about the caches.

Note

Cache awareness is dependent on the exact device and connection method.

Toolbar and context menu options

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 different 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 options:

New Cache Data View

Displays a new instance of the **Cache Data** view.

Update View When Hidden

Enables the updating of the view when it is hidden behind other views. By default this view does not update when hidden.

Refresh

Refreshes the view.

Freeze Data

Toggles the freezing of data in the current view. 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 right-click a value:

Copy

Copies the selected value.

Select All

Selects all text.

Related concepts

[6.17 About debugging caches on page 6-165.](#)

Related references

[10.44 DTSL Configuration Editor dialog box on page 10-308.](#)

[10.14 Memory view on page 10-237.](#)

Related information

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

10.20 Screen view

Use the **Screen** view to display the contents of the screen buffer.

This view enables you to:

- Configure when view updates should occur and the interval between updates.
- Freeze the view to prevent it being updated by the running target when it next updates.
- Set the screen buffer parameters appropriate for the target:

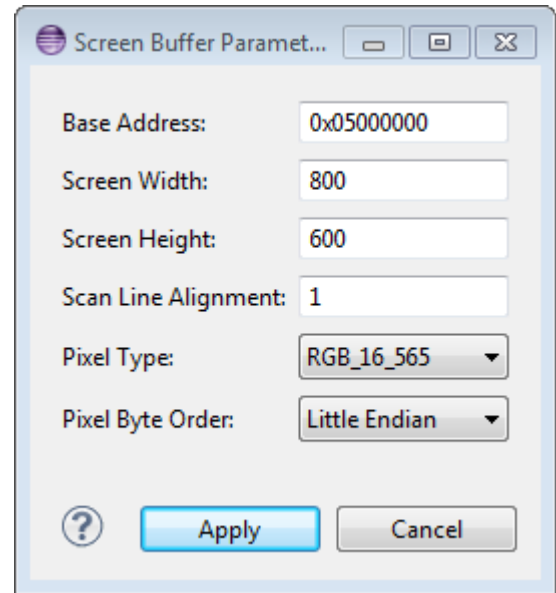


Figure 10-25 Screen buffer parameters

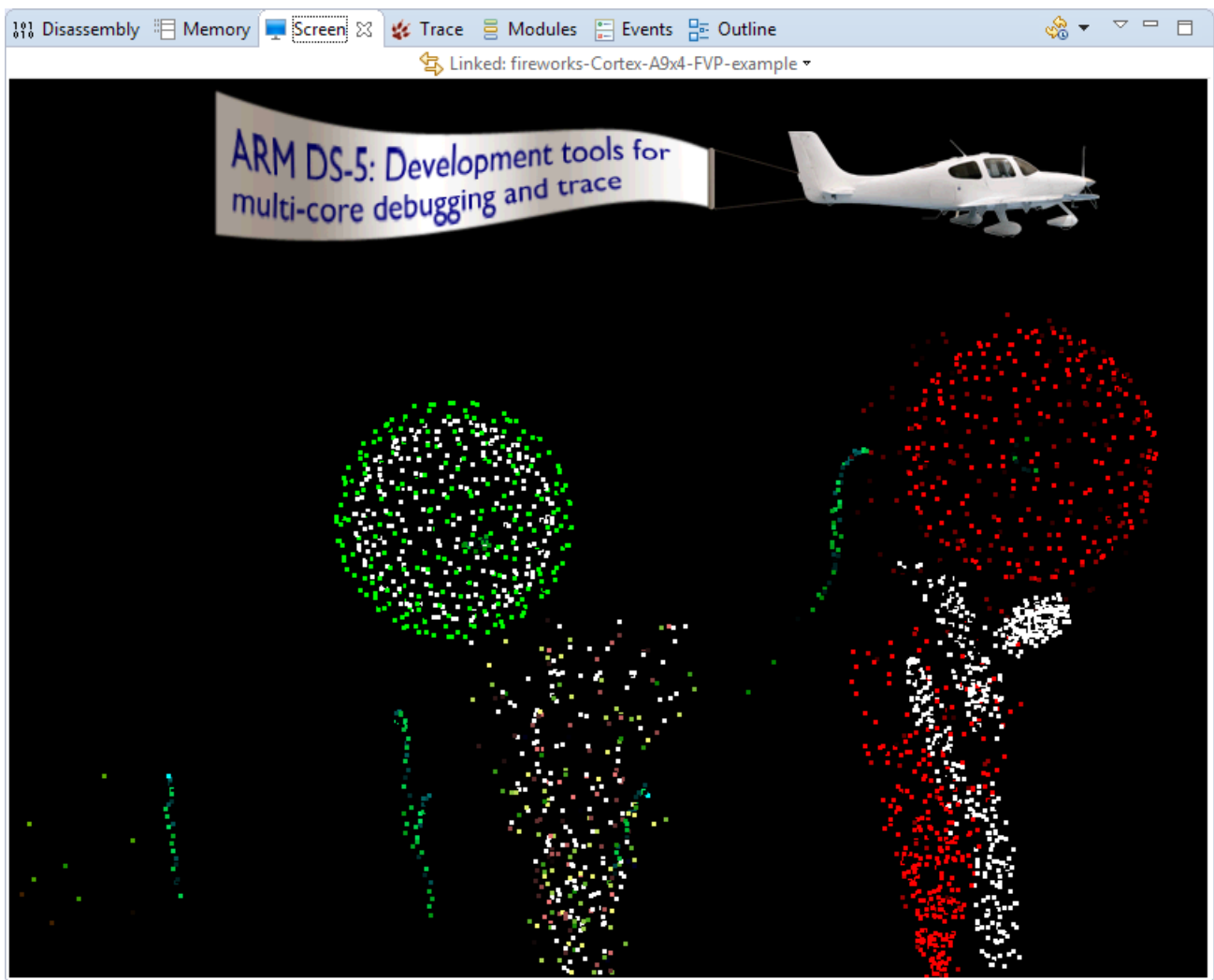


Figure 10-26 Screen view

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 different 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

- 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.

Start

Starts auto-refreshing.

Stop

Stops auto-refreshing.

Update Interval

Specifies the auto-refresh interval, in seconds or minutes.

Update When

Specifies whether updates should occur only when the target is running or stopped, or always.

Properties

Displays the Timed Auto-Refresh Properties dialog box.

New Screen View

Creates a new instance of the **Screen** 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.

Update View When Hidden

Enables the updating of the view when it is hidden behind other views. By default this view does not update when hidden.

Refresh

Refreshes the view.

Freeze Data

Toggles the freezing of data in the current view. This also disables or enables the **Refresh** option.

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

[Chapter 10 DS-5 Debug Perspectives and Views on page 10-200.](#)

10.21 Scripts view

Use the **Scripts** view to display a list of favorite scripts. It also enables you to import, run, edit, and remove scripts.

You can also import scripts into 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, drag and drop the scripts into the required order.

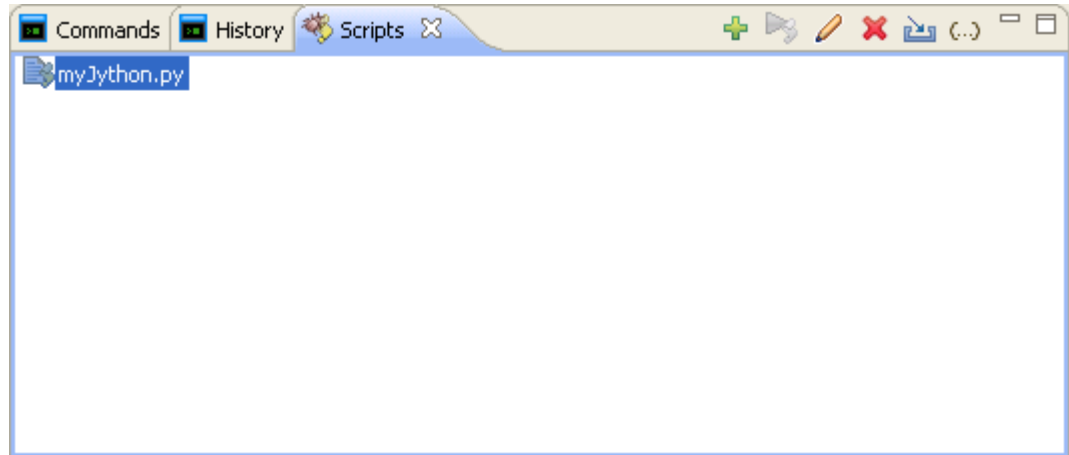


Figure 10-27 Scripts view

Note

The default location for script files is controlled by a DS-5 Debugger setting in the Preferences dialog box. You can access this setting 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 add contents to the script, select the script and click **Edit Selected Scripts**.

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. It also prompts you whether to delete the scripts from the file system.

Import Script...

Imports a script file into 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** option.

Cut

Copies the selected script filename and removes the script from the view. It also prompts you whether to delete the selected script from the file system.

Copy

Copies the selected script filename.

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. It also prompts you whether to delete the script from the file system.

Select All

Selects all script files.

Related references

[Chapter 10 DS-5 Debug Perspectives and Views](#) on page 10-200.

10.22 Target Console view

Use the **Target Console** view to display messages from the target setup scripts.

Note

Default settings for this view are controlled by DS-5 Debugger options in the Preferences dialog box. For example, the default location for the console log. 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 different 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.

Toggles Scroll Lock

Enables or disables the automatic scrolling of messages in the **Target Console** view.

View Menu

This menu contains the following options:

New Target Console View

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

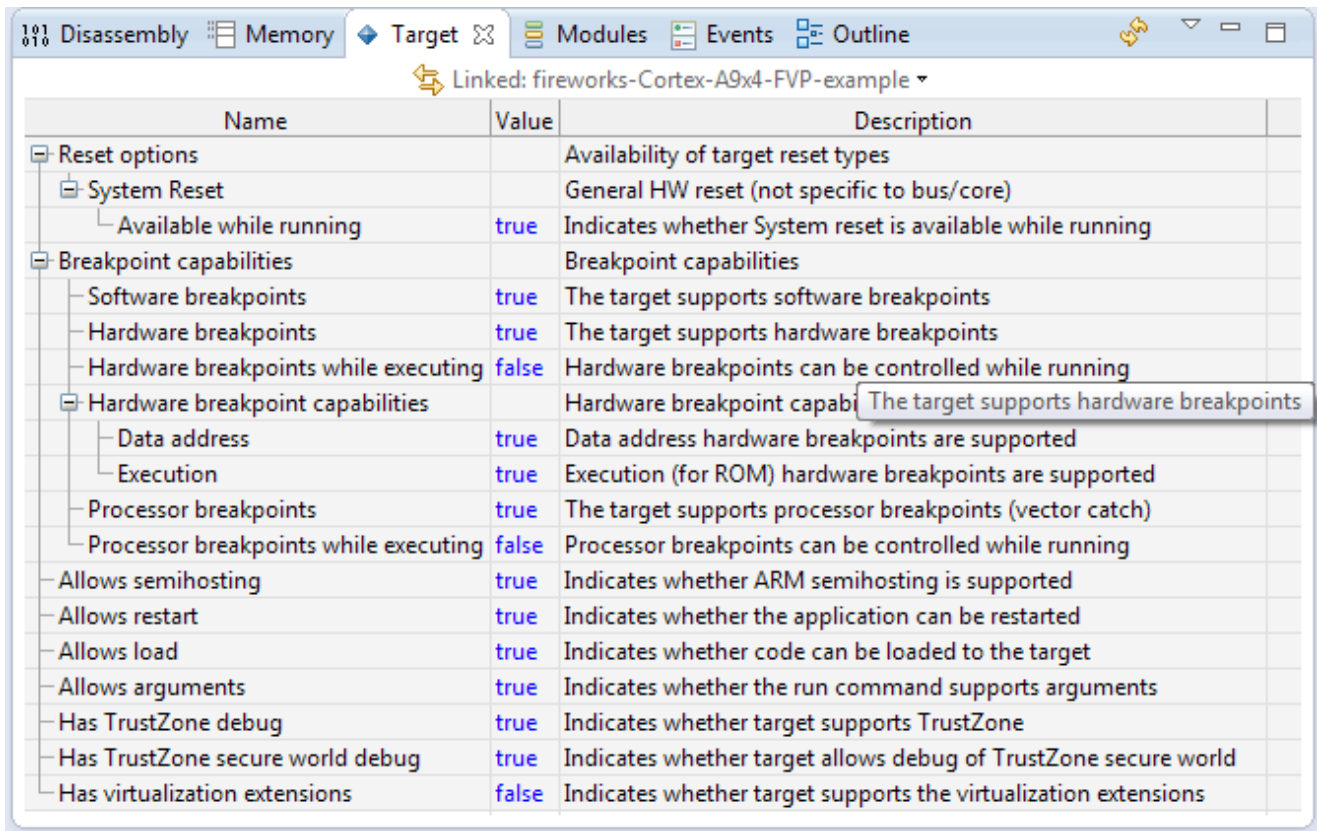
Selects all text.

Related references

[Chapter 10 DS-5 Debug Perspectives and Views on page 10-200.](#)

10.23 Target view

Use the **Target** view to display the debug capabilities of the target, for example the types of breakpoints it supports. It does not allow you to modify the capabilities.



Name	Value	Description
Reset options		Availability of target reset types
System Reset		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	false	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	false	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	true	Indicates whether target allows debug of TrustZone secure world
Has virtualization extensions	false	Indicates whether target supports the virtualization extensions

Figure 10-28 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.

Show All Columns

Displays all columns.

Reset Columns

Resets the columns displayed and their widths to the default.

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 > Target**.

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 different 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.

View Menu

This menu contains the following option:

New Target View

Displays a new instance of the **Target** view.

Copy

Copies the selected capabilities. To copy the capabilities in a group such as **Breakpoint capabilities**, you must first expand that group.

This is useful if you want to copy the capabilities to a text editor to save them for future reference.

Select All

Selects all capabilities currently expanded in the view.

Related references

[Chapter 10 DS-5 Debug Perspectives and Views on page 10-200.](#)

10.24 Trace view

Use the **Trace** view to display a graphical navigation chart that shows 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.

When a 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.

The left-hand column of the chart shows the percentages of the total trace for each function. 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 number of instructions each function executes in each timeline. The darker red color shows more instructions and the lighter yellow color shows fewer 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.

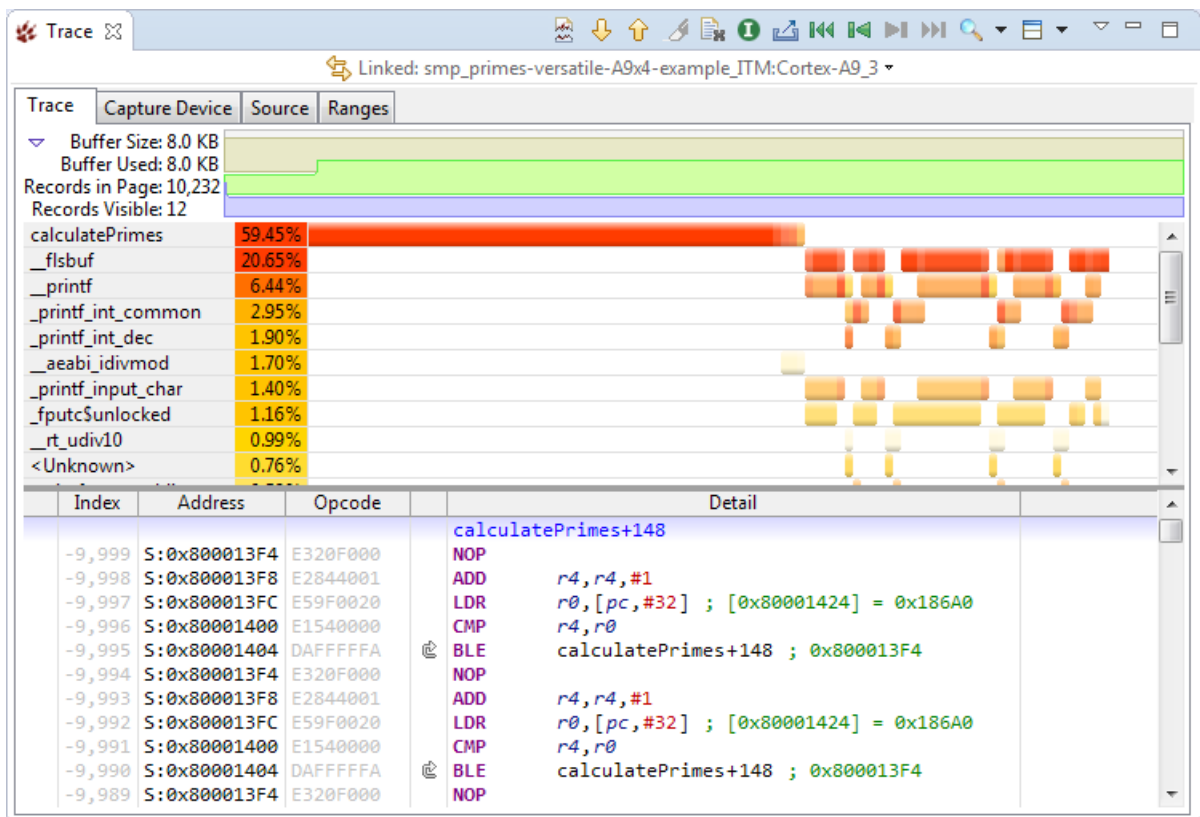


Figure 10-29 Trace view with a scale of 100:1

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 > Trace**.

The **Trace** view navigation chart contains several tabs:

- **Trace** tab shows the graphical timeline and disassembly.
- **Capture Device** tab gives information about the trace capture device and the trace buffer, and allows you to configure the trace capture.
- **Source** tab gives information about the trace source.
- **Ranges** tab allows you to limit the trace capture to a specific address range.

The **Trace** tab also shows:

Buffer Size

Size of the trace buffer to store trace records. This is determined by the trace capture device. The trace records can be instruction records or non-instruction records.

Buffer Used

Amount of the trace buffer that is already used for trace records.

Records in Page

The total number of instruction records and non-instruction records in the current **Trace** view.

Records Visible

The number of trace records visible in the disassembly area of the **Trace** 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 different connection or processor in a *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.

Updating view when hidden

Not updating view when hidden

Toggles the updating of the view when it is hidden behind other views. By default the view does not update when it is hidden, which might cause loss of trace data.

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 instructions and functions.

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 in which 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 in which you can enter the maximum number of instructions to display in the disassembly trace. The number must be within the range of 1,000 to 1,000,000 instructions.

Find Trace Trigger Event

Enables you to search for trigger events in the trace capture buffer.

Find Timestamp...

Displays a dialog box in which you can enter either a numeric timestamp as a 64 bit value or in the h:m:s format.

Find Function...

Enables you to search for a function by name in the trace buffer.

Find Instruction by Address...

Enables you to search for an instruction by address in the trace buffer.

Find ETM data access in trace buffer...

Enables you to search for a data value or range of values in the trace buffer.

Find Instruction Index...

Enables you to search for an instruction by index. A positive index is relative to the start of the trace buffer and a negative index is relative to the end.

DTSL Options...

Displays a dialog box in which you can add, edit, or choose a DTSL configuration.

————— **Note** —————

This clears the trace buffer.

Open Trace Control View

Opens the **Trace Control View**.

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 in which you can select the trace record types that you want to see in the **Trace** view.

Related tasks

[1.8 Capturing trace using the headless debugger on page 1-31.](#)

Related references

[Chapter 10 DS-5 Debug Perspectives and Views on page 10-200.](#)

10.25 Trace Control view

Use the **Trace Control** view to 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.

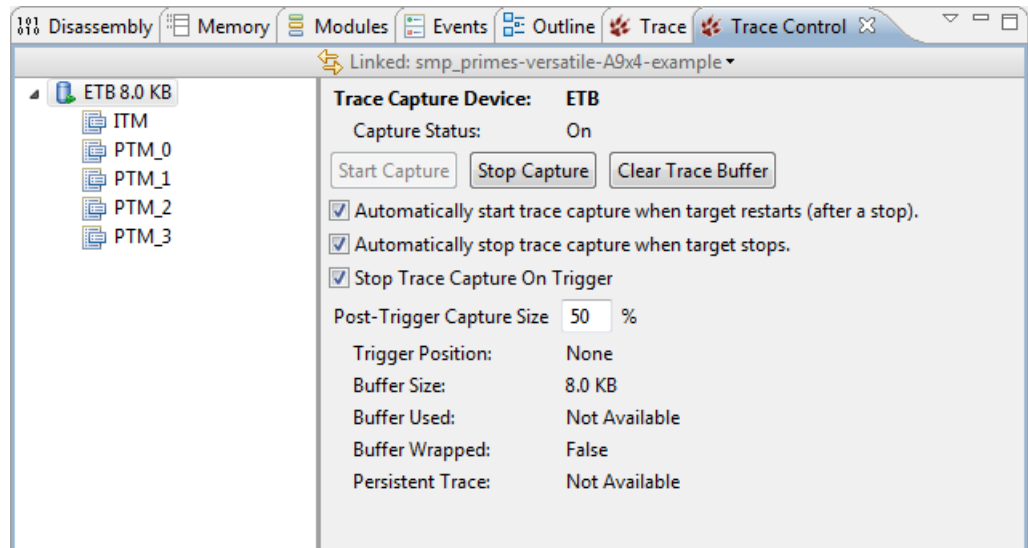


Figure 10-30 Trace Control view

The trace capture device and trace sources available in the trace capture device are listed on the left hand side of the view. Select any trace source to view additional information.

The following Trace Capture Device information is displayed in the view:

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 amount of trace data currently in the buffer.

Buffer Wrapped

The trace buffer data wraparound status.

Persistent Trace

The persistent trace data status.

The following Trace Source information is displayed in the view:

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.

Timestamp Origin

Whether a timestamp origin for the trace is set or cleared. When set, timestamps are displayed as offsets from the origin.

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.

Start trace capture when target restarts (after a stop)

Select this option to automatically start trace capture after a target restarts after a stop.

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 capture after a trace capture trigger has been hit.

Post-trigger capture size

Use this option to control the percentage of the 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 a *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.

Related references

[Chapter 10 DS-5 Debug Perspectives and Views on page 10-200.](#)

10.26 Variables view

Use the Variables view to work with the contents of local, file static, and global variables in your program.

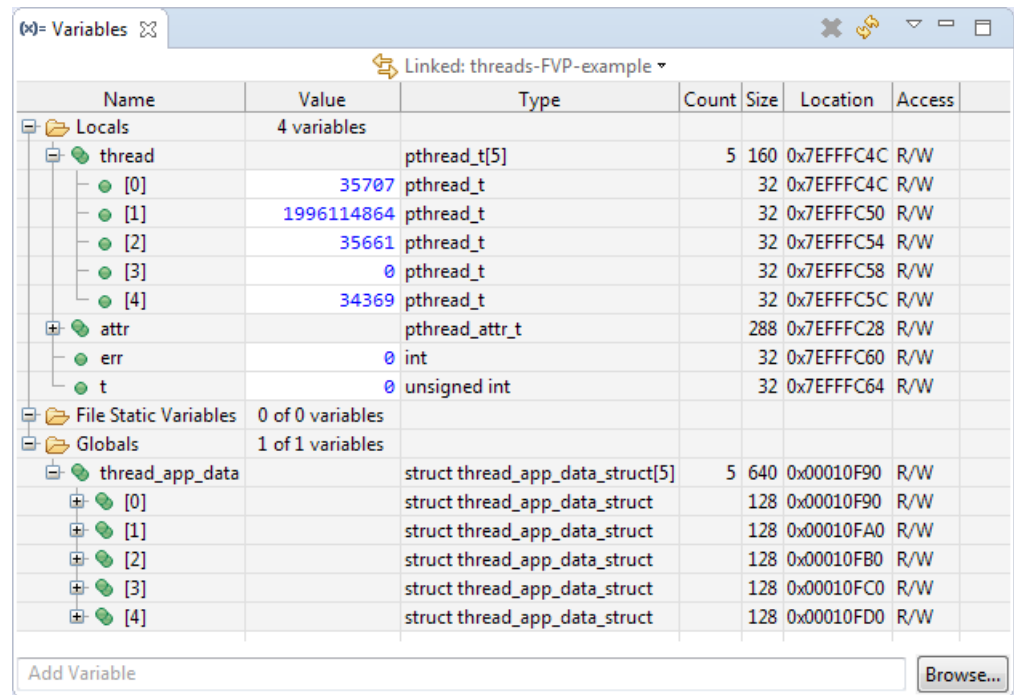


Figure 10-31 Variables view

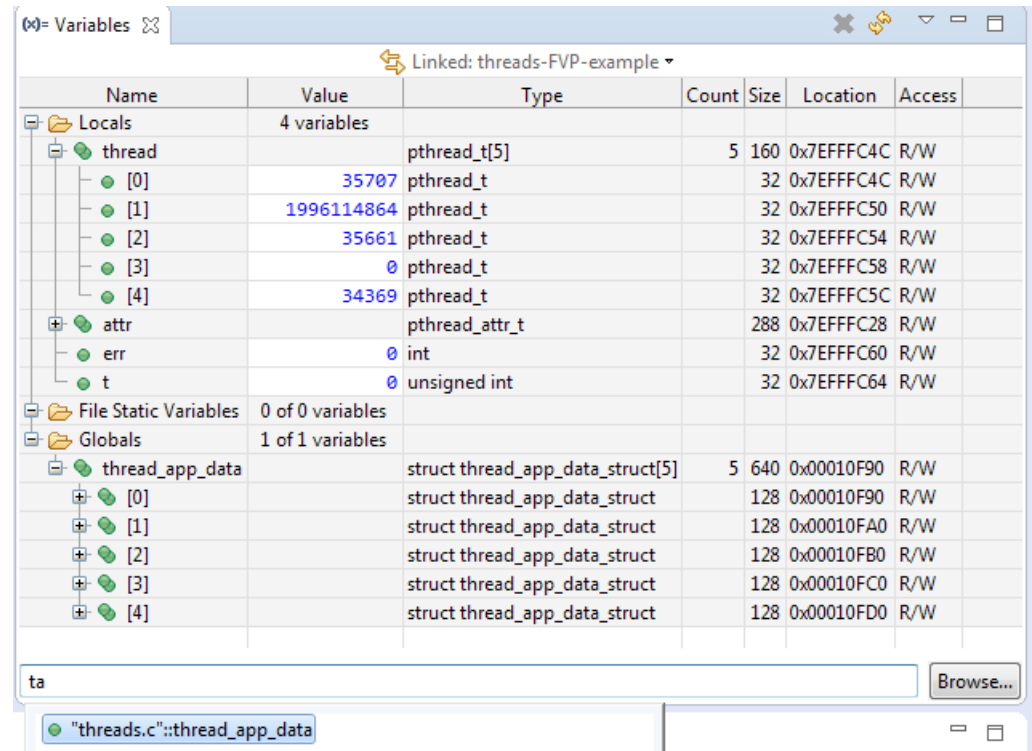
You can:

View the contents of variables that are currently in scope

By default, the Variables view displays all the local variables. It also displays the file static and global variable folder nodes. You can add or remove variables by selecting one or more variables. Keep the set of variables in the view to a minimum to maintain good debug performance.

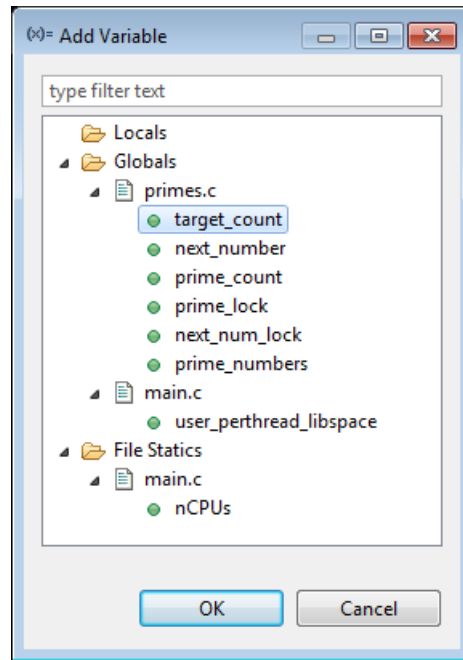
- Add a specific variable



If you know the name of the specific variable you want to view, enter the variable name in the **Add Variable** field. This lists the variables that match the text you entered. For example, enter the text **ta** to view variables with the text **ta** in their name. Double-click the variable to add it to the Variables view.




- Browse and select multiple variables

If you want to view all the available variables in your code, click **Browse** to display the Add Variable dialog. Expand the required folders and filenames to see the variables they contain. Then select one or more variables that you are interested in and click **OK** to add them to the Variables view. **Ctrl+A** selects all the variables that are visible in the dialog. Selecting a filename or folder does not automatically select its variables.




- **Delete variables**
You can remove the variables, that you added, from the variables view. In the Variables view, select the variables you want to remove from the view, and click  to remove the selected variables. If you want to reset the view to display the default variables again, then from the view menu select Reset to default, .

Tip

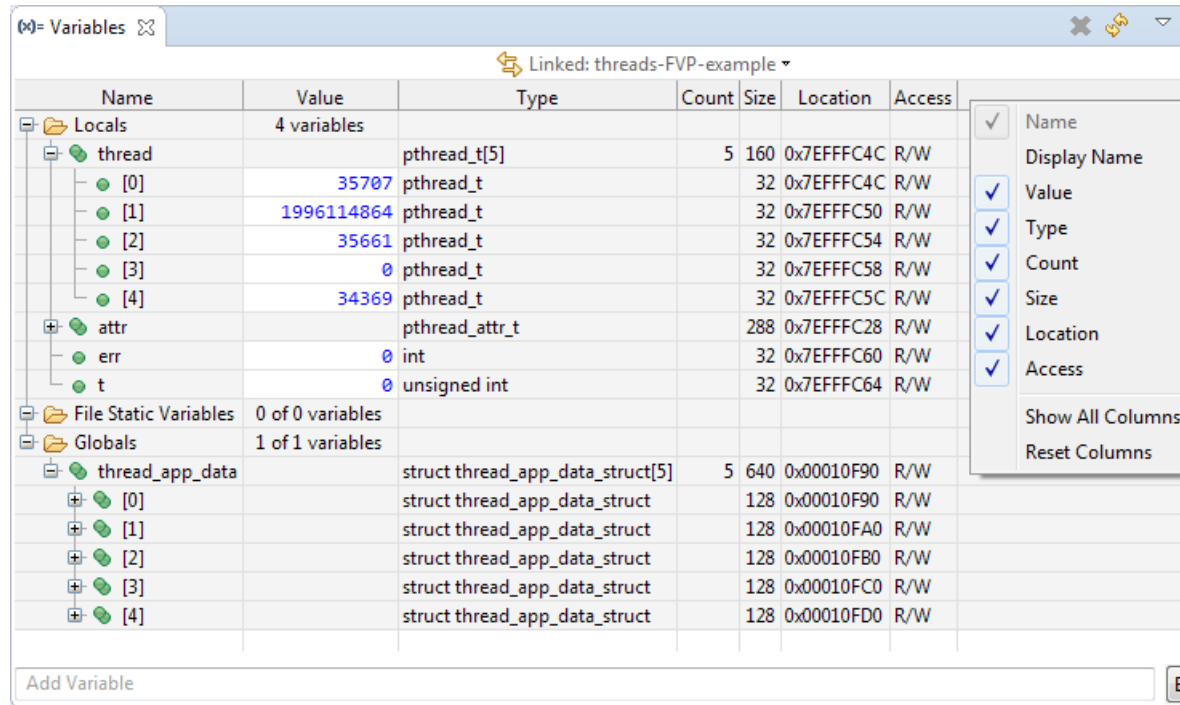
 You can also use the **Delete** key on your keyboard to delete the variables.

- **Refresh view**

To refresh or update the values in the view, click .

Modify the value of write access variables

You can modify the values of variables with write access by clicking in the **Value** column for the variable and entering a new value. Enable the **Access** column to view access rights for each variable.



Freeze the view to prevent the values being updated by a running target

Select **Freeze Data** from the view menu to prevent values updating automatically when the view refreshes.

Drag and drop a variable from the Variables view to other views

Drag and drop a variable from this view into either the Memory view to see the memory at that address, or into the Disassembly view to disassemble from that address.

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 different 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.

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 can be useful if you want to copy variables to a text editor in order to compare the values when execution stops at another location.

Select All

Selects all variables 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, for example the address of an array, &name
- The location of the variable, for example the name of an array, name.

The memory size is set to the size of the variable, using the **sizeof** keyword.

Show in Disassembly

Where enabled, displays the Disassembly view, with the address set to the location of the selected variable.

Show in Registers

If the selected variable is currently held in a register, displays the Registers view with that register selected.

Show Dereference in Memory

If the selected variable is a pointer, displays the Memory view with the address set to the value of the variable.

Show Dereference in Disassembly

If the selected variable is a pointer, displays the Disassembly view, with the address set to the value of the variable.

Translate Variable Address

Displays the MMU view and translates the address of the variable.

Toggle Watchpoint

Displays the Add Watchpoint dialog to set a watchpoint on the selected variable, or removes the watchpoint if one has been set.

Enable Watchpoint

Enables the watchpoint, if a watchpoint has been set on the selected variable.

Disable Watchpoint

Disables the watchpoint, if a watchpoint has been set on the selected variable.

Resolve Watchpoint

If a watchpoint has been set on the selected variable, re-evaluates the address of the watchpoint. If the address can be resolved the watchpoint is set, otherwise it remains pending.

Watchpoint Properties

Displays the Watchpoint Properties dialog box. This enables you to control watchpoint activation.

Send to <selection>

Enables you to add variable filters to an Expressions view. Displays a sub menu that enables you to specify an Expressions view.

<Format list>

A list of formats you can use for the variable value.

View Menu

The following **View Menu** options are available:

New Variables View

Displays a new instance of the Variables view.

Update View When Hidden

Enables the updating of the view when it is hidden behind other views. By default, this view does not update when hidden.

Reset to default variables

Resets the view to show only the default variables.

Freeze Data

Toggles the freezing of data in the current view. You cannot modify the value of a variable if the data is frozen. This option also disables or enables the **Refresh** option.

If you freeze the data before you expand an item for the first time, for example an array, the view might show Pending... Unfreeze the data to expand the item.

Editing context menu options

The following options are available on the context menu when you select a variable value for editing:

Undo

Reverts the last change you made to the selected value.

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.

Select All

Selects the value.

Adding a new column header

Right-click on the column headers to select the columns that you want to display:

Name

The name of the variable.

Value

The value of the variable.

Read-only values are displayed with a grey background. A value that you can edit is initially shown with a white background. A yellow background indicates that the value has changed. This might result from you either performing a debug action such as stepping or by you editing the value directly.

Note

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.

Access

The access mode for the variable.

Show All Columns

Displays all columns.

Reset Columns

Resets the columns displayed and their widths to the default.

Related concepts

[6.8 About debugging multi-threaded applications on page 6-149.](#)

[6.9 About debugging shared libraries on page 6-150.](#)

[6.10.2 About debugging a Linux kernel on page 6-153.](#)

[6.10.3 About debugging Linux kernel modules on page 6-154.](#)

[6.11 About debugging TrustZone enabled targets on page 6-157.](#)

Related references

[4.5 Working with breakpoints and watchpoints on page 4-109.](#)

[4.8 Setting a tracepoint on page 4-121.](#)

[4.6 Working with conditional breakpoints on page 4-116.](#)

[4.6.1 Assigning conditions to an existing breakpoint on page 4-116.](#)

[4.7 About pending breakpoints and watchpoints on page 4-120.](#)

[4.5.5 Exporting DS-5 breakpoint settings to a file on page 4-113.](#)

4.5.4 Importing DS-5 breakpoint settings from a file on page 4-113.
Chapter 10 DS-5 Debug Perspectives and Views on page 10-200.

10.27 Timed Auto-Refresh Properties dialog box

Use the **Timed Auto-Refresh Properties** dialog box to modify the update interval settings.

Update Interval

Specifies the auto refresh interval in seconds.

Update When

Specifies when to refresh the view:

Running

Refreshes the view only while the target is running.

Stopped

Refreshes the view only while the target is stopped.

Always

Always refreshes the view.

Note

When you select **Running** or **Always**, 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 through 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.

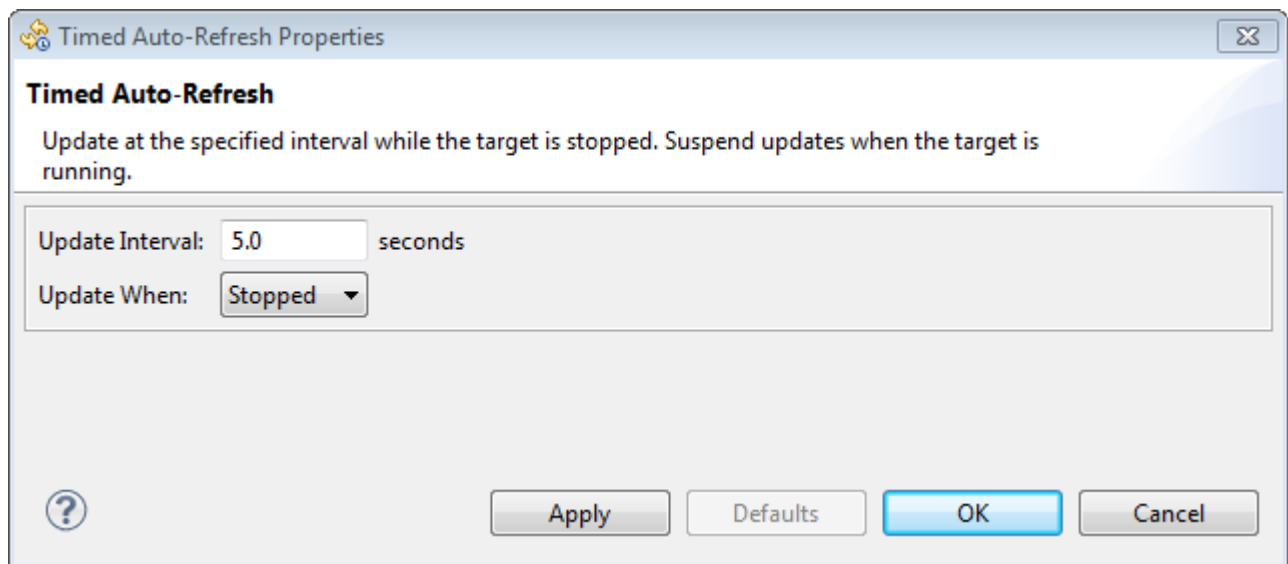


Figure 10-32 Timed Auto-Refresh Properties dialog box

10.28 Memory Exporter dialog box

Use the **Memory Exporter** dialog box to generate a file containing the data from a specific region of memory.

Memory Bounds

Specifies the memory region to export:

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 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.

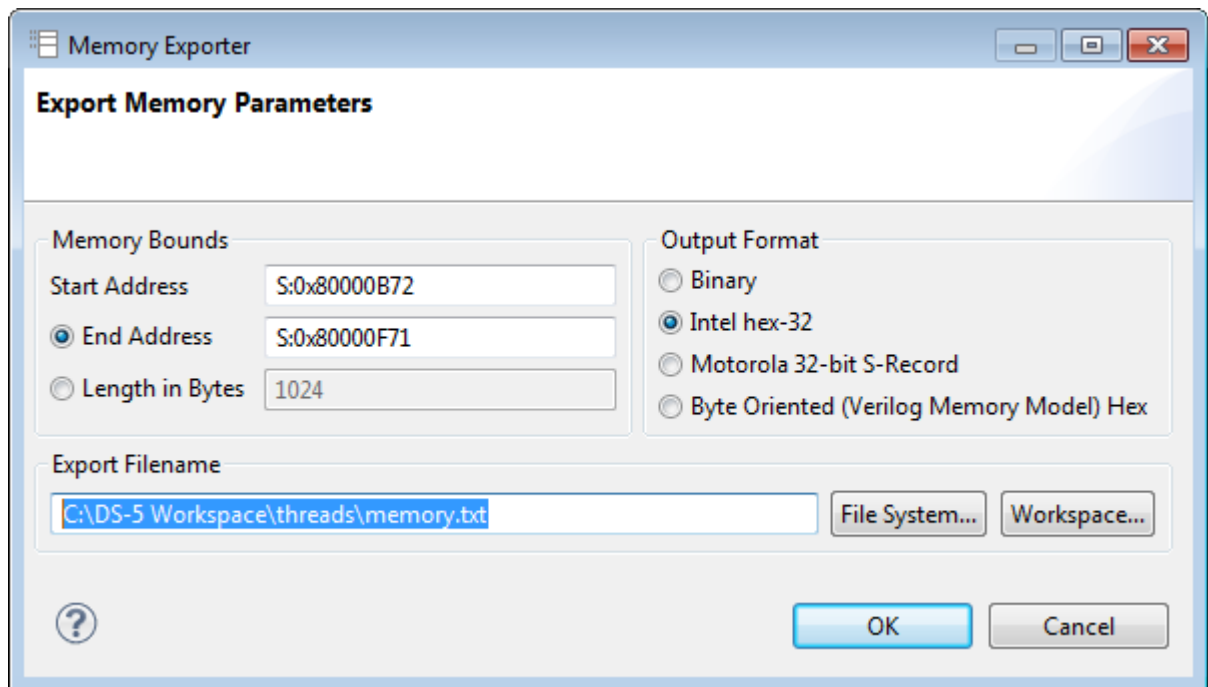


Figure 10-33 Memory Exporter dialog box

10.29 Memory Importer dialog box

Use the **Memory Importer** dialog box to import data from a file into memory.

Offset to Embedded Address

Specifies an offset that is added to all addresses in the image prior to importing it. Some image formats do not contain embedded addresses and in this case the offset is the absolute address to which the image is imported.

Memory Limit

Enables you to define a region of memory that you want to import to:

Limit to memory range

Specifies whether to limit the address range.

Start

Specifies the minimum address that can be written to. Any address prior to this is not written to. If no address is given then the default is address zero.

End

Specifies the maximum address that can be written to. Any address after this is not written to. If no address is given then the default is the end of the address space.

Import File Name

Select **Import file as binary image** if the file format is binary.

Enter the location of the file in the field provided or click on:

- **File System...** to locate the file in an external folder
- **Workspace...** to locate the file in a workspace project.

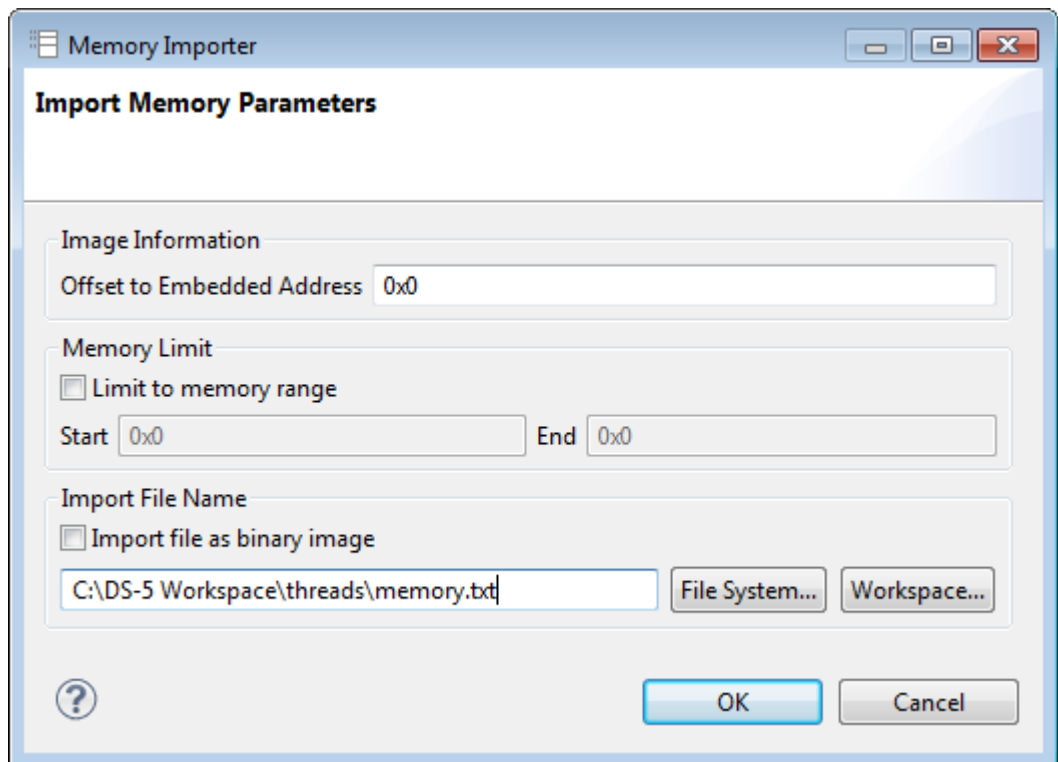


Figure 10-34 Memory Importer dialog box

10.30 Fill Memory dialog box

Use the **Fill Memory** dialog box to fill a memory region with a pattern of bytes.

Memory Bounds

Specifies the memory region:

Start Address

Specifies the start address of the memory region.

End Address

Specifies the inclusive end address of the memory region.

Length in Bytes

Specifies the number of bytes to fill.

Data Pattern

Specifies the fill pattern and its size in bytes.

Fill size

Specifies the size of the fill pattern as either 1, 2, 4, or 8 bytes.

Pattern

Specifies the pattern with which to fill the memory region.

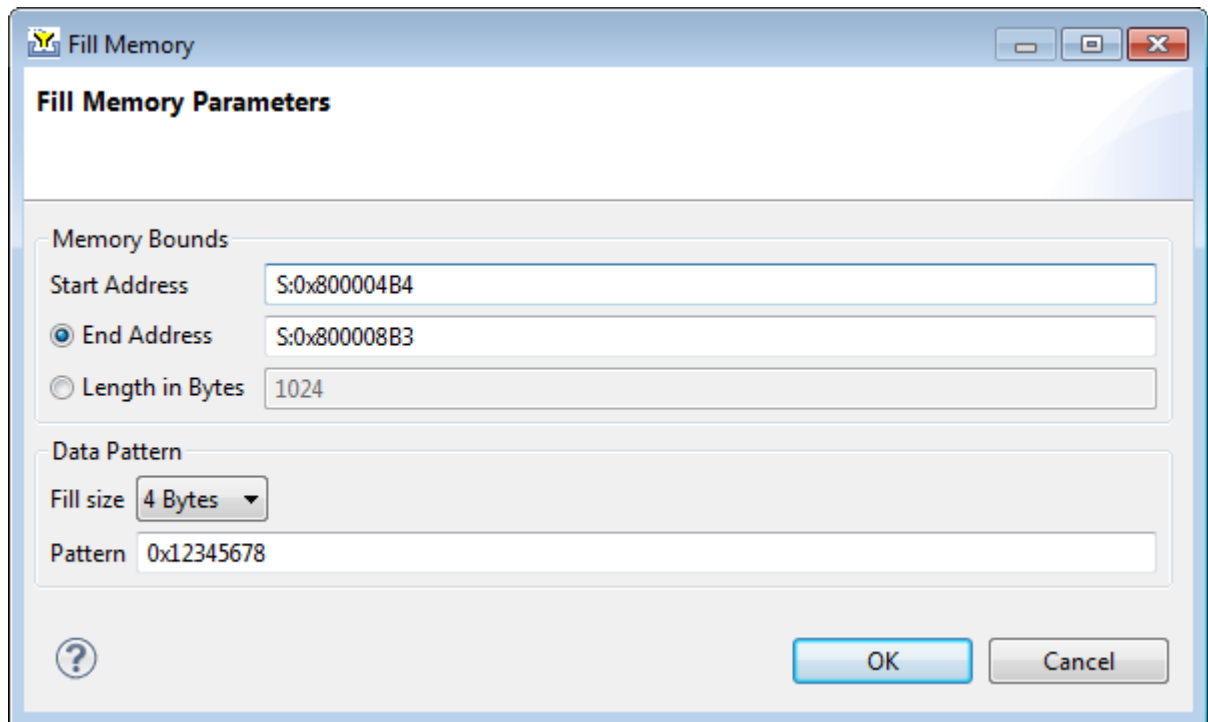


Figure 10-35 Fill Memory dialog box

10.31 Export Trace Report dialog box

Use the **Export Trace Report** dialog box to export 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.

Split Output Files

Splits the output file when it reaches the selected size.

Select source for trace report

Selects the required trace data.

Use trace view as report source

Instructions that are decoded 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

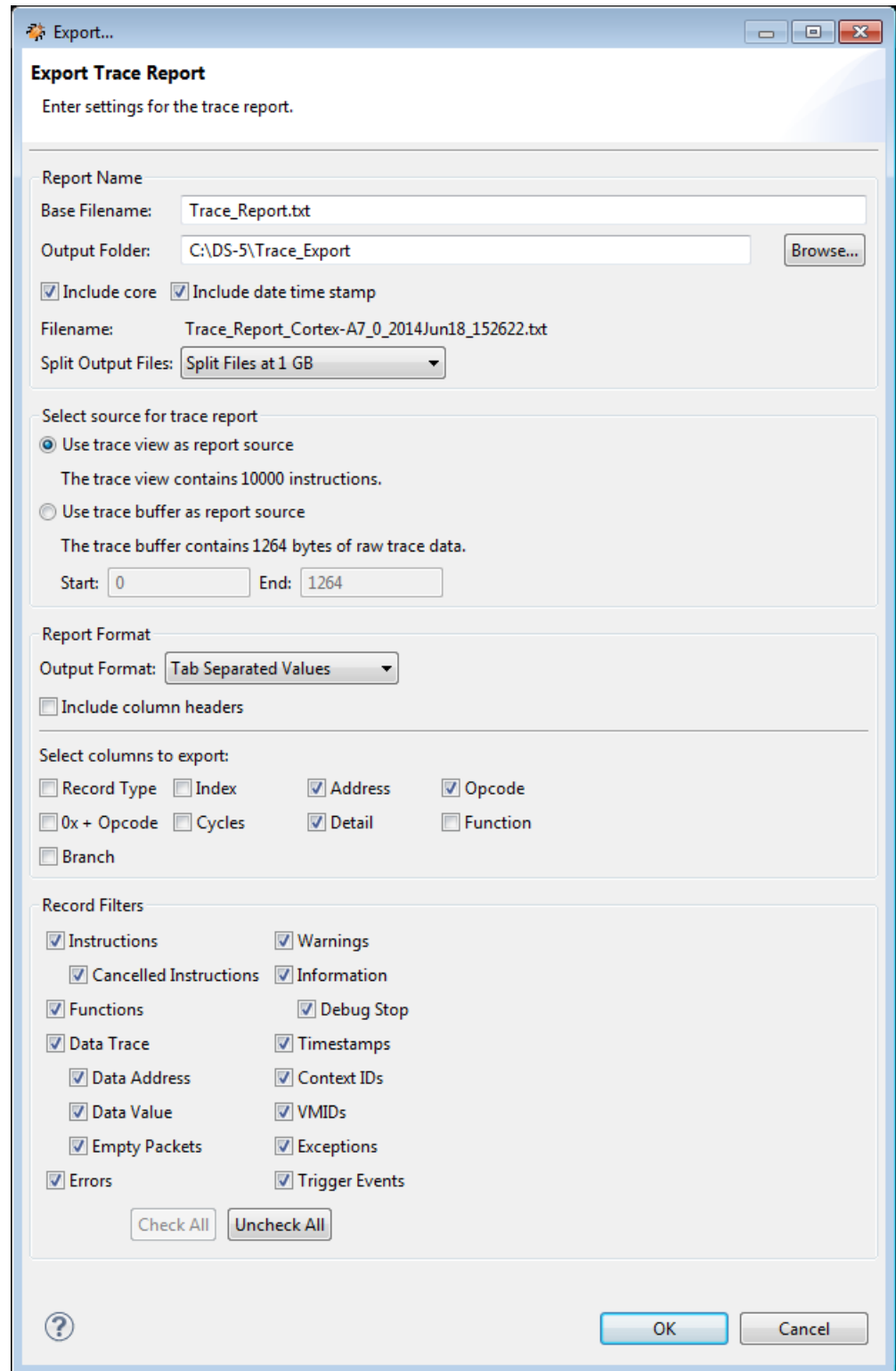
Enables or disables trace filters.

Check All

Enables you to select all the trace filters.

Uncheck All

Enables you to unselect all the trace filters.



Export...

Export Trace Report

Enter settings for the trace report.

Report Name

Base Filename:

Output Folder:

☒ Include core ☒ Include date time stamp

Filename:

Split Output Files:

Select source for trace report

☒ Use trace view as report source

The trace view contains 10000 instructions.

☐ Use trace buffer as report source

The trace buffer contains 1264 bytes of raw trace data.

Start: End:

Report Format

Output Format:

☐ Include column headers

Select columns to export:

<input type="checkbox"/> Record Type	<input type="checkbox"/> Index	<input checked="" type="checkbox"/> Address	<input checked="" type="checkbox"/> Opcode
<input type="checkbox"/> 0x + Opcode	<input type="checkbox"/> Cycles	<input checked="" type="checkbox"/> Detail	<input type="checkbox"/> Function
<input type="checkbox"/> Branch			

Record Filters

<input checked="" type="checkbox"/> Instructions	<input checked="" type="checkbox"/> Warnings
<input checked="" type="checkbox"/> Cancelled Instructions	<input checked="" type="checkbox"/> Information
<input checked="" type="checkbox"/> Functions	<input checked="" type="checkbox"/> Debug Stop
<input checked="" type="checkbox"/> Data Trace	<input checked="" type="checkbox"/> Timestamps
<input checked="" type="checkbox"/> Data Address	<input checked="" type="checkbox"/> Context IDs
<input checked="" type="checkbox"/> Data Value	<input checked="" type="checkbox"/> VMIDs
<input checked="" type="checkbox"/> Empty Packets	<input checked="" type="checkbox"/> Exceptions
<input checked="" type="checkbox"/> Errors	<input checked="" type="checkbox"/> Trigger Events

Figure 10-36 Export Trace Report dialog box

10.32 Breakpoint Properties dialog box

Use the **Breakpoint Properties** dialog box to display the properties of a breakpoint.

It also enables you to:

- Set a stop condition and an ignore count for the breakpoint.
- Specify a script file to run when the breakpoint is hit.
- Configure 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.

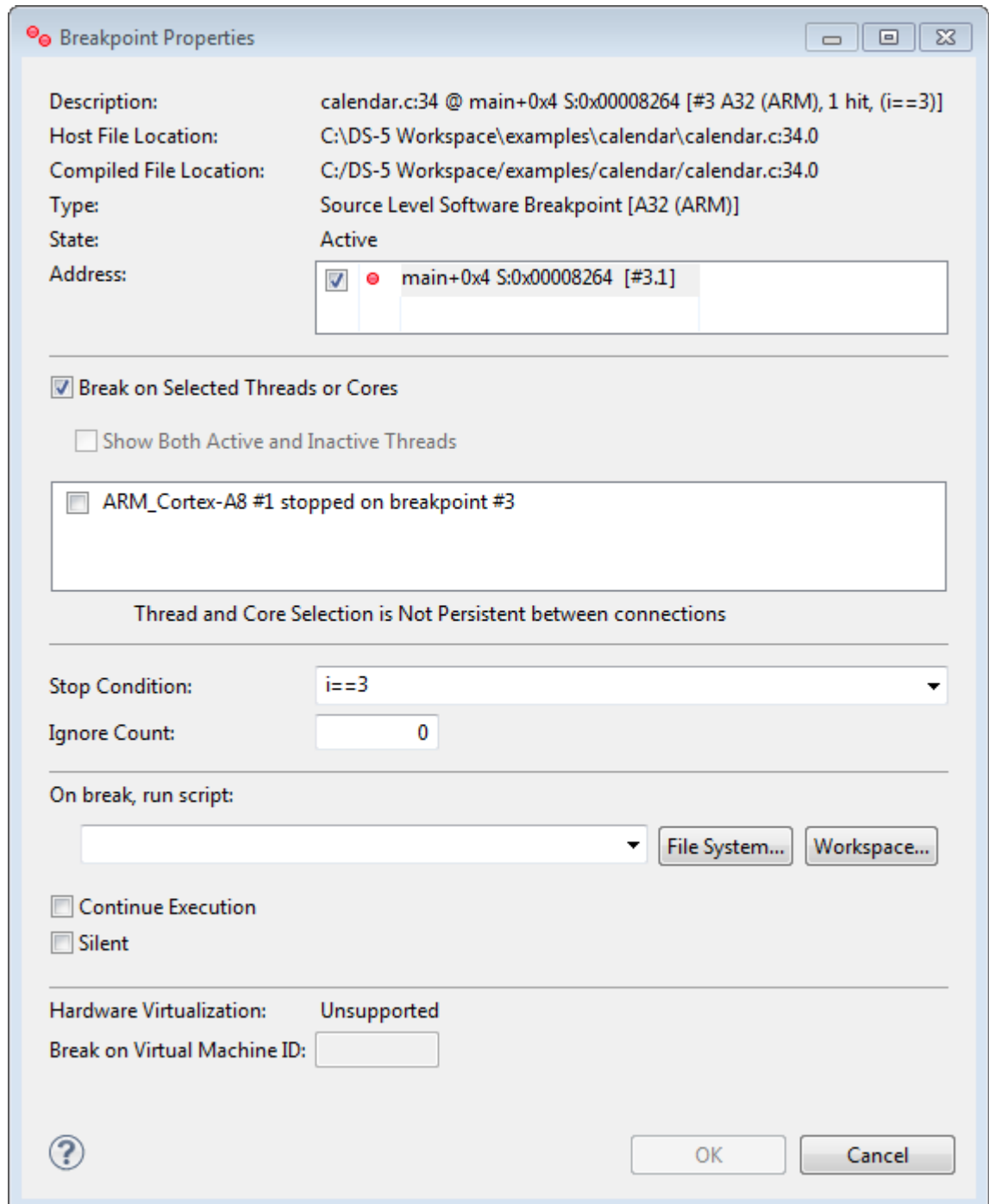


Figure 10-37 Breakpoint properties dialog box

Breakpoint information

The breakpoint information shows the basic properties of a breakpoint. It comprises:

Description

- If the source file is available, the file name and line number in the file where the breakpoint is set, for example `calendar.c:34`.
- The name of the function in which the breakpoint is set and the number of bytes from the start of the function. For example, `main+0x4` shows that the breakpoint is 4 bytes from the start of the `main()` function.
- The address at which the breakpoint is set.
- 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 instruction set at the breakpoint, A32 (ARM) or T32 (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, for example `i==3`.

Host File Location

The location of the image on the host machine.

Compiled File Location

The path that the image was compiled with. This can be relative or absolute. This location might be different from the host file location if you compile and debug the image on different machines.

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 the breakpoint, either `Software Breakpoint` or `Hardware Breakpoint`.
- The instruction set of the instruction at the address of the breakpoint, A32 (ARM) or T32 (Thumb).

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.

Disabled

The breakpoint is disabled.

No Connection

The breakpoint is in an application that is not connected to a target.

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. You can use the check boxes to enable or disable the breakpoints.

Breakpoint options

The following options are available for you to set:

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.

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 count on each pass. When it reaches zero, the breakpoint activates. Each subsequent pass causes the breakpoint to activate.

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 messages for the selected breakpoint in the **Commands** view.

Hardware Virtualization

Indicates whether Hardware Virtualization is supported.

Break on Virtual Machine ID

If Hardware Virtualization is supported, specify the *Virtual Machine ID* (VMID) of the guest operating system to which the breakpoint applies.

10.33 Watchpoint Properties dialog box

Use the **Watchpoint Properties** dialog box to display the properties of a watchpoint and to change the watchpoint type.

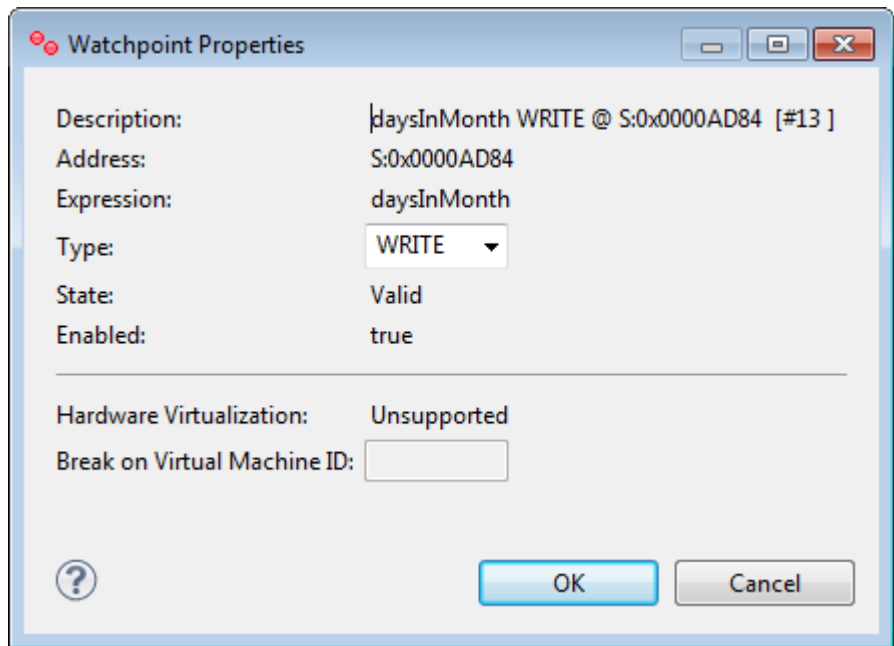


Figure 10-38 Watchpoint Properties dialog box

The following types are available:

READ

The debugger stops the target when the address is read from.

WRITE

The debugger stops the target when the address is written to.

ACCESS

The debugger stops the target when the address is read from or written to.

The following properties are displayed:

State

Indicates whether the watchpoint is valid or invalid. A watchpoint is invalid if it cannot be set for any reason. Typically this is because hardware resources have run out.

Enabled

Indicates whether the watchpoint is enabled or disabled.

Hardware Virtualization

Indicates whether Hardware Virtualization is supported.

Break on Virtual Machine ID

If Hardware Virtualization is supported, specify the *Virtual Machine ID* (VMID) of the guest operating system to which the watchpoint applies.

10.34 Tracepoint Properties dialog box

Use the **Tracepoint Properties** dialog box to display the properties of a tracepoint.

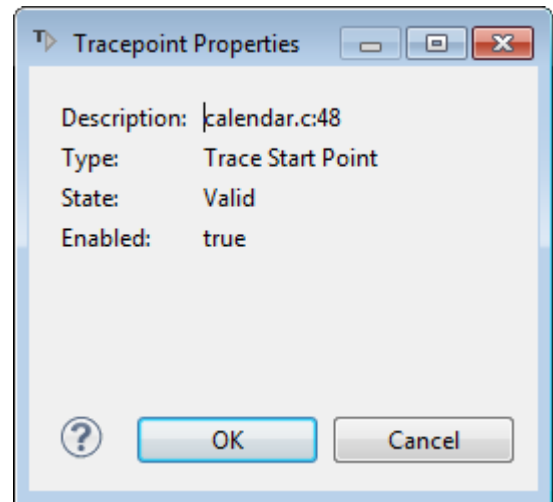


Figure 10-39 Tracepoint Properties dialog box

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.

10.35 Manage Signals dialog box

Use the **Manage Signals** dialog box 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.

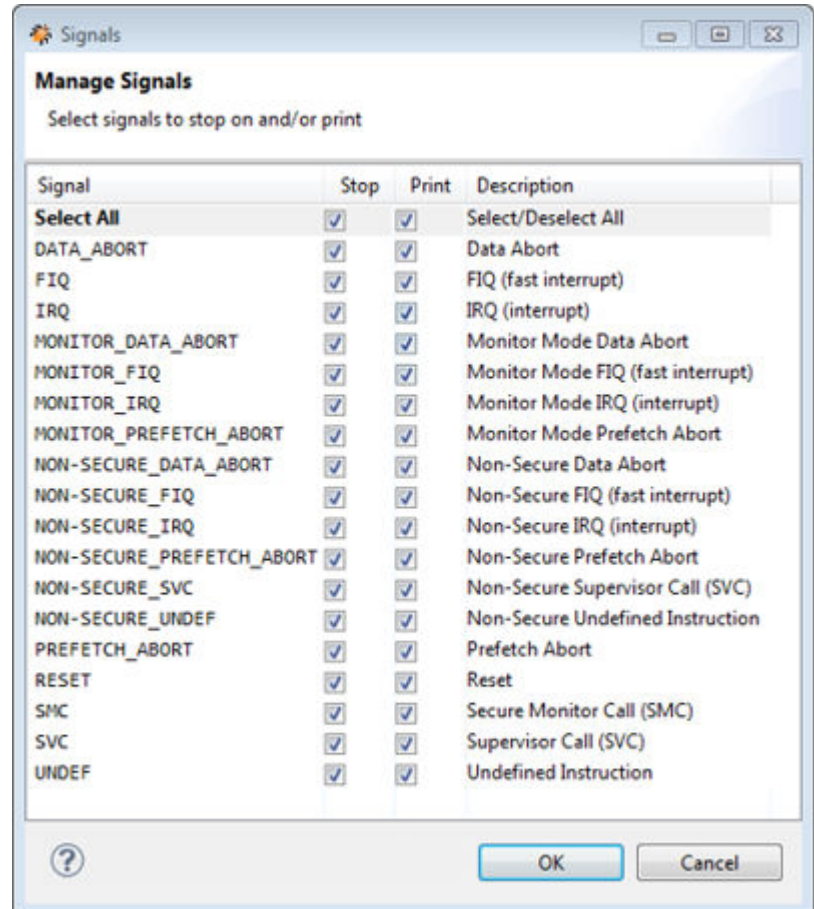


Figure 10-40 Manage Signals dialog box

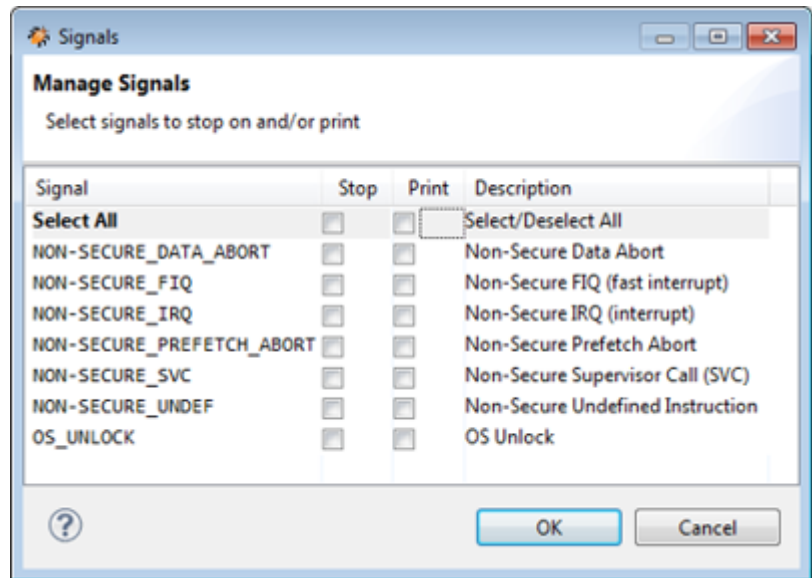


Figure 10-41 Manage exception handler settings

10.36 Functions Filter dialog box

Use the **Functions Filter** dialog box to filter the list of symbols that are displayed in the **Functions** view. You can filter functions by compilation unit or image and by function name.

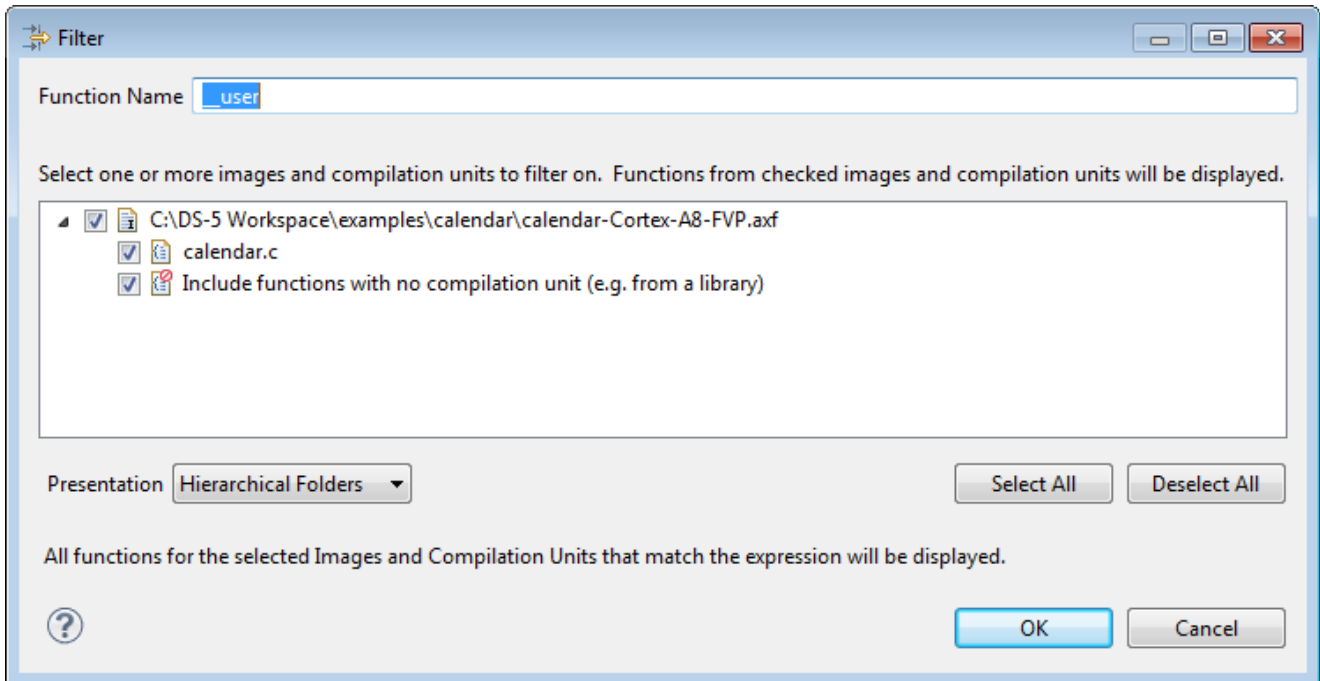


Figure 10-42 Function filter dialog box

10.37 Script Parameters dialog box

Use the **Script Parameters** dialog box to specify script parameters.

Script Parameters

Specifies parameters for the script in the text field. Parameters must be space-delimited.

Variables...

Opens the **Select Variable** dialog box, in which 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.

Enable Verbose Mode

Checking this option causes the script to run in verbose mode. This means that each command in the script is echoed to the **Commands** view.

OK

Saves the current parameters and closes the Script Parameters dialog box.

Cancel

Closes the Script Parameters dialog box without saving the changes.

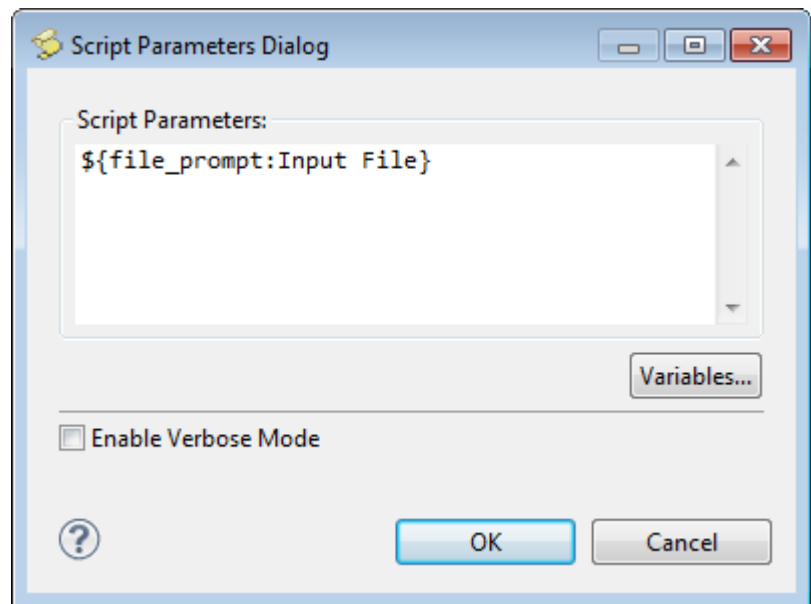


Figure 10-43 Script Parameters dialog box

10.38 Debug Configurations - Connection tab

Use the **Connection** tab in the **Debug Configurations** dialog box 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. You must select an RSE connection to the target if your Linux application debug operation is:

- **Download and debug application**
- **Start gdbserver and debug target-resident application.**

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 *Virtual File System* (VFS).

Host mount point

Specify the location of the file system on the local host. You can:

- Enter the location in the field provided.
- Click **File System...** to locate the directory in an external location from the workspace.
- Click **Workspace...** to locate the directory within your 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.

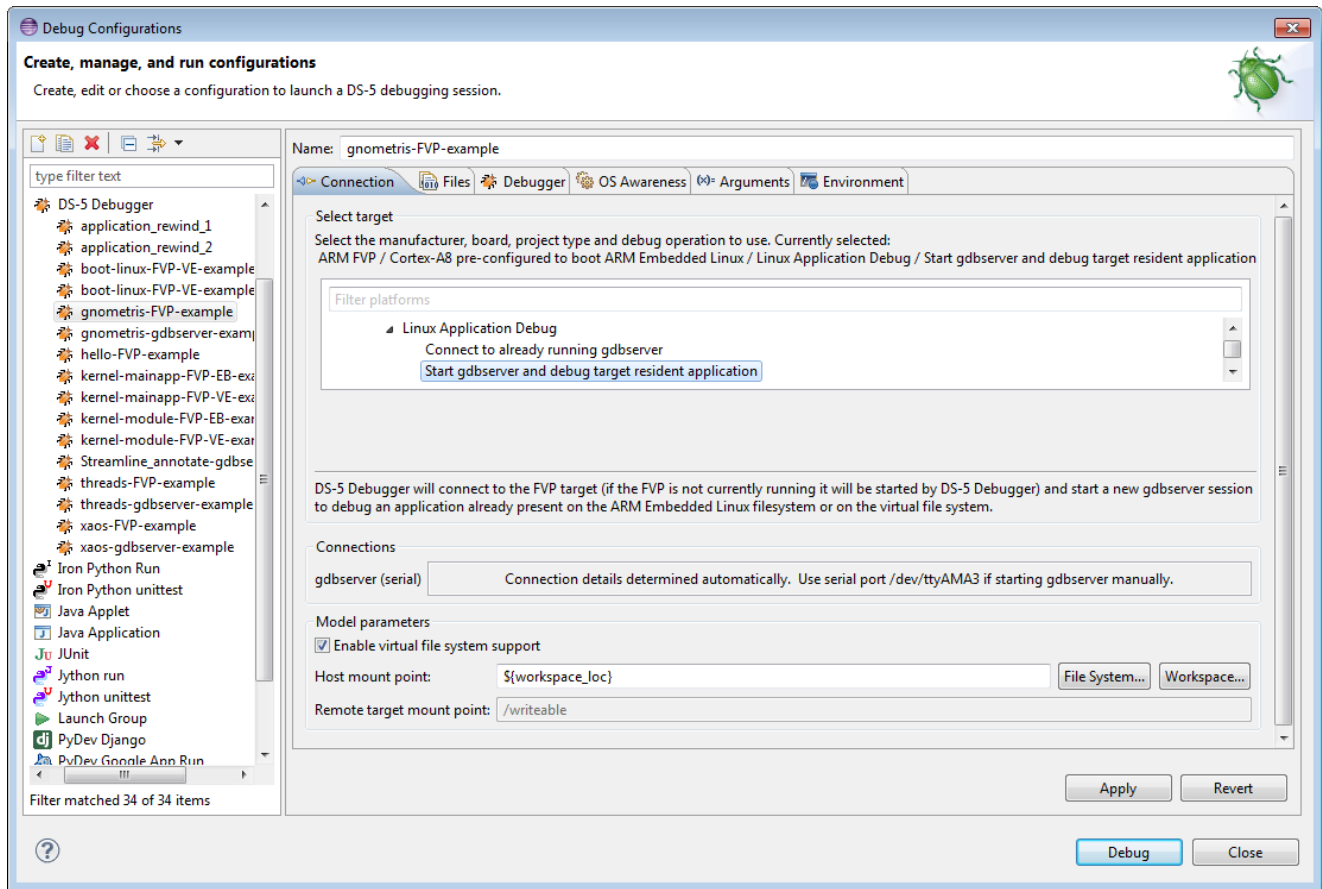


Figure 10-44 Connection tab (Shown with connection configuration for an FVP with virtual file system support enabled)

10.39 Debug Configurations - Files tab

Use the **Files** tab in the **Debug Configurations** dialog box 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.

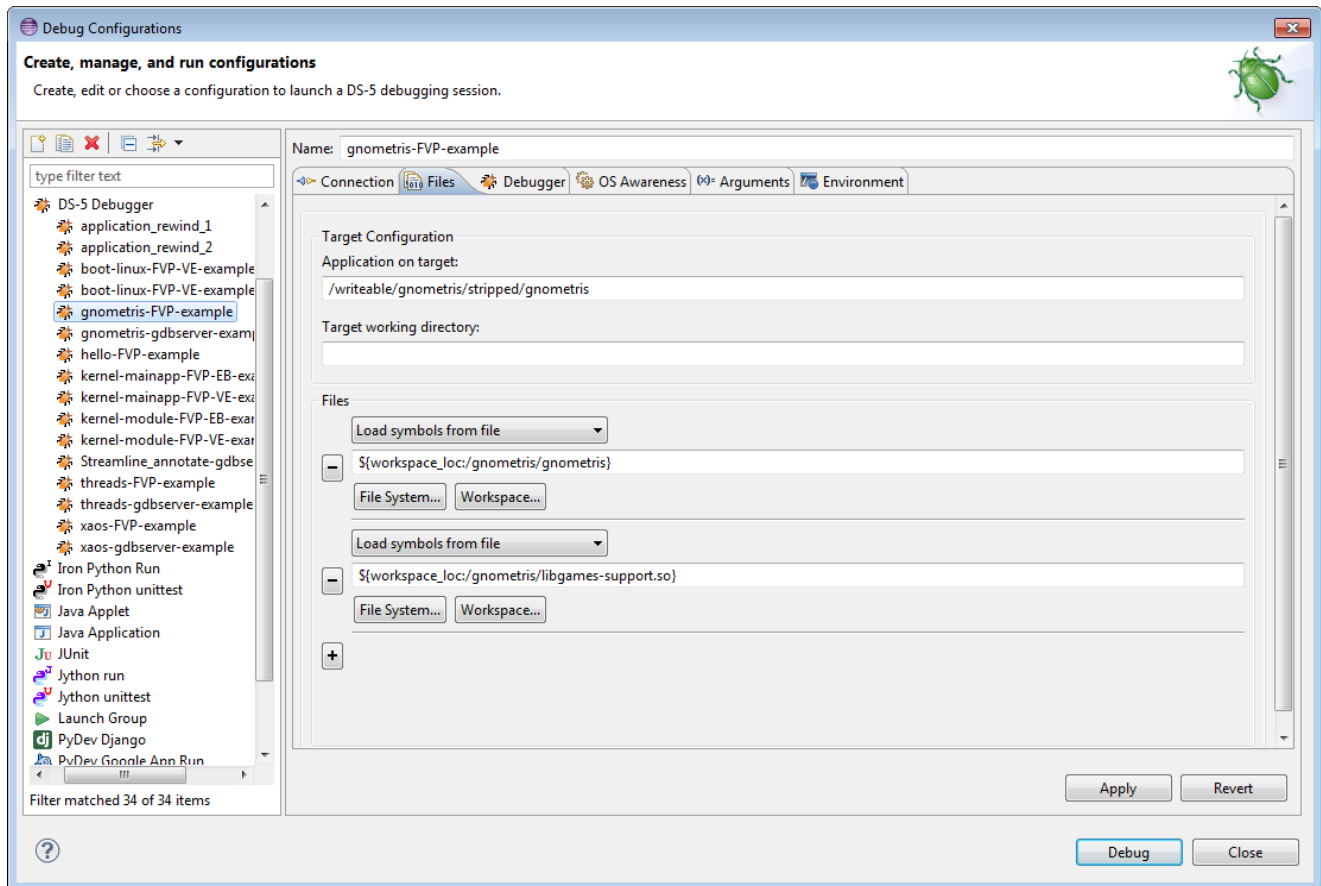


Figure 10-45 Files tab (Shown with file system configuration for an application on an FVP)

Files options summary

The Files options available depend on the debug operation you selected on the **Connection** tab. The possible 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 a project directory or sub-directory within the Eclipse workspace.

For example, to download the stripped (no debug) Gnometriz application image, select the `gnometris/stripped/gnometris` file.

Select **Load symbols** to load the debug symbols from the specified image.

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 a 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 from file

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 from file is selected 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 `gnometriz/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 file 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

[6.10.2 About debugging a Linux kernel on page 6-153.](#)

10.40 Debug Configurations - Debugger tab

Use the **Debugger** tab in the **Debug Configurations** dialog box 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 / .py)

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 / .py)

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 file 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.

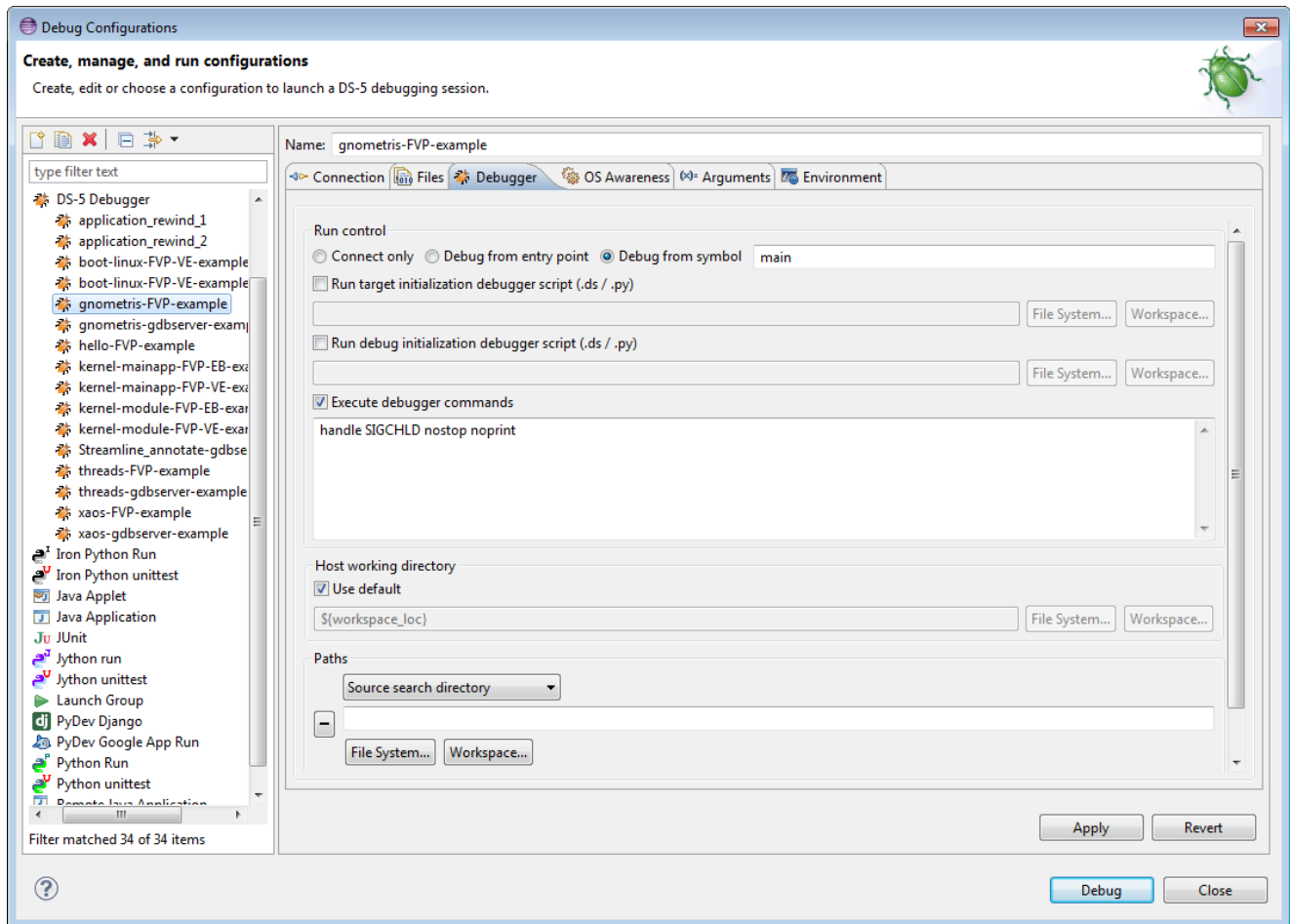


Figure 10-46 Debugger tab (Shown with settings for application starting point and search paths)

Related concepts

[6.10.2 About debugging a Linux kernel on page 6-153.](#)

10.41 Debug Configurations - OS Awareness tab

Use the **OS Awareness** tab in the **Debug Configurations** dialog box 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.

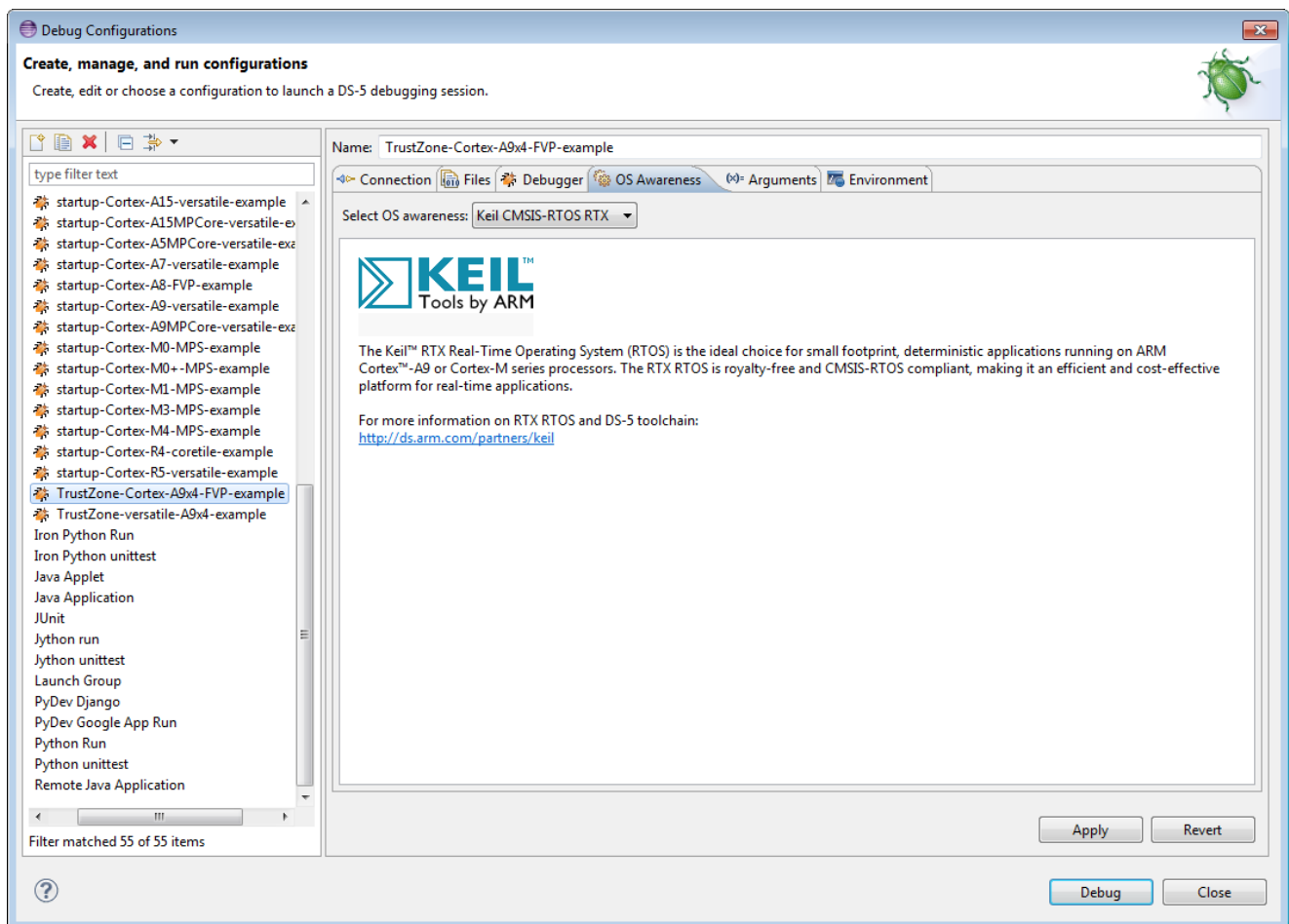


Figure 10-47 OS Awareness tab

10.42 Debug Configurations - Arguments tab

If your application accepts command-line arguments to `main()`, specify them using the **Arguments** tab in the **Debug Configurations** dialog box.

The **Arguments** tab contains the following elements:

Note

These settings only apply if the target supports semihosting and they cannot be changed while the target is running.

Program Arguments

This panel enables you to enter the arguments. Arguments are separated by spaces. They 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.

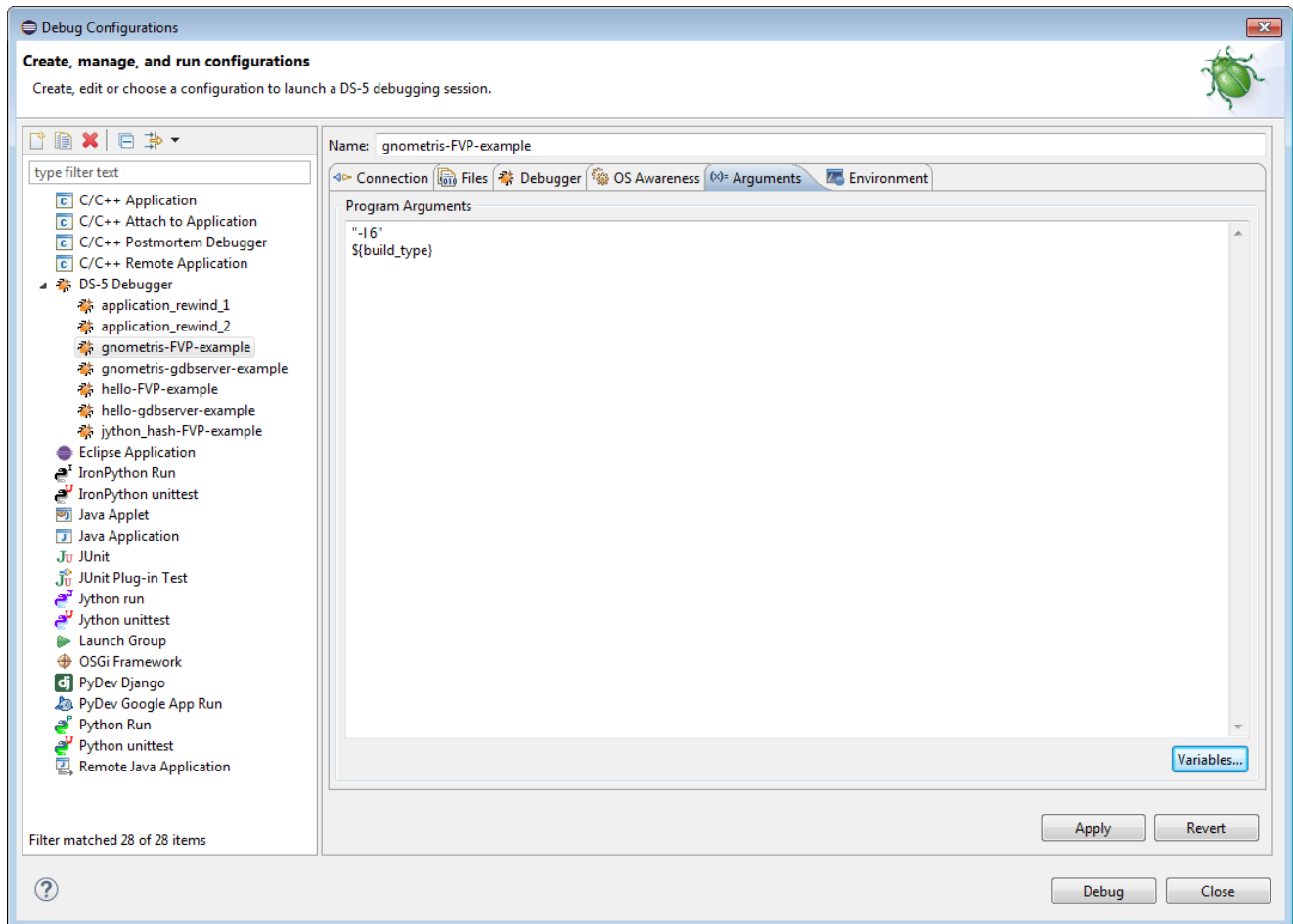


Figure 10-48 Arguments tab

Related references

- [4.3 About passing arguments to main\(\) on page 4-107.](#)
- [7.1 About semihosting and top of memory on page 7-169.](#)
- [7.2 Working with semihosting on page 7-171.](#)
- [7.3 Enabling automatic semihosting support in the debugger on page 7-172.](#)
- [7.4 Controlling semihosting messages using the command-line console on page 7-173.](#)

Related information

[DS-5 Debugger commands.](#)

10.43 Debug Configurations - Environment tab

Use the **Environment** tab in the **Debug Configurations** dialog box to create and configure the target environment variables that are passed to the application when the debug session starts.

The **Environment** tab contains the following elements:

————— **Note** —————

The settings in this tab are not used for connections that use the **Connect to already running gdbserver** debug operation.

Target environment variables to set

This panel displays the target environment variables in use by the debugger.

New...

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.

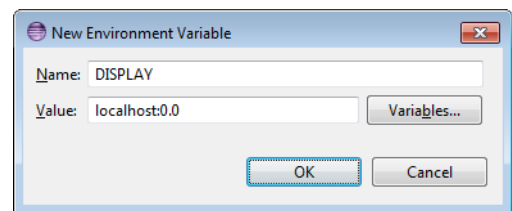


Figure 10-49 New Environment Variable dialog box

Edit...

Opens the **Edit Environment Variable** dialog box where you can edit the properties for the selected target environment variable.

Remove

Removes the selected 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.

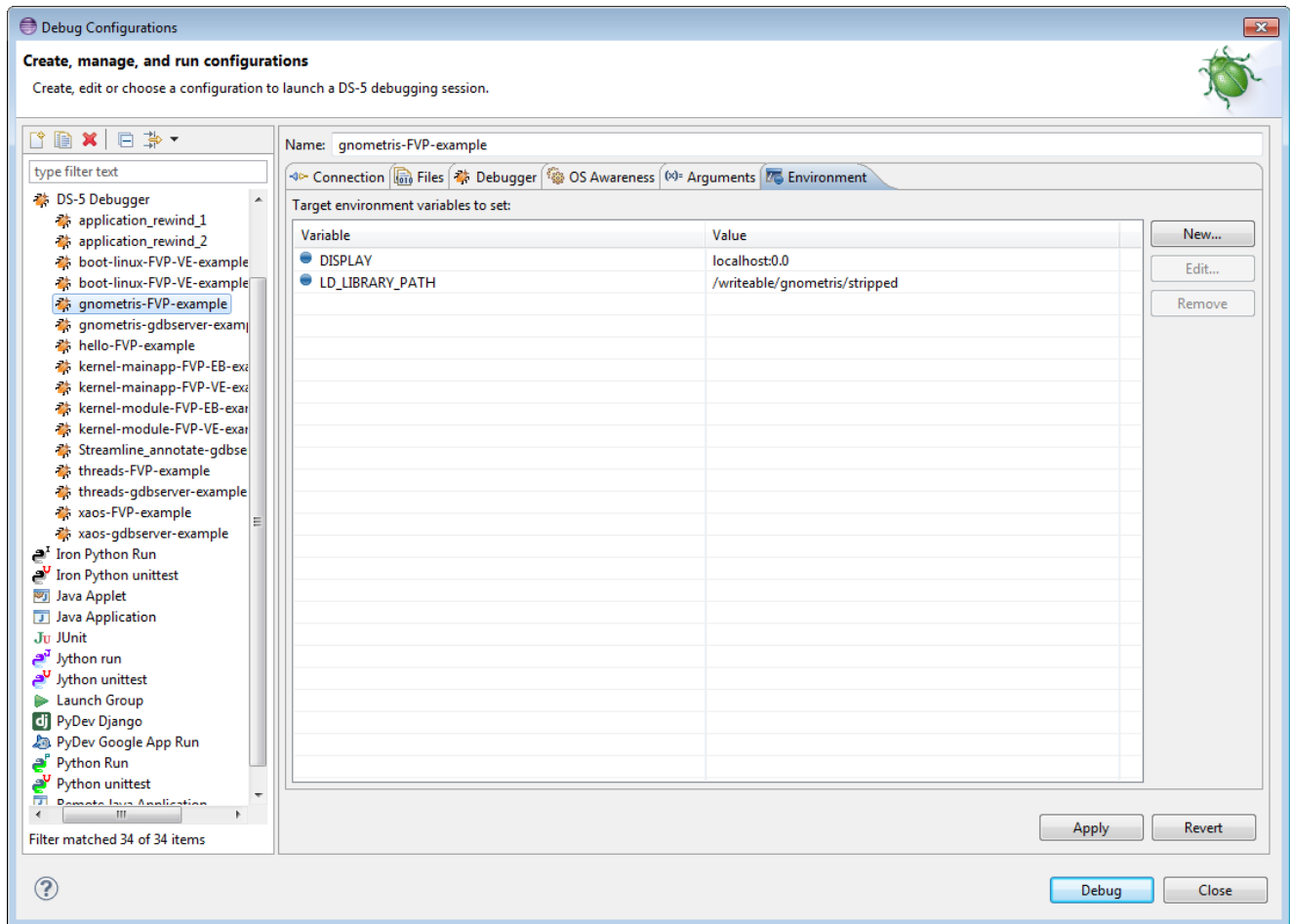


Figure 10-50 Environment tab (Shown with environment variables configured for an FVP)

10.44 DTSL Configuration Editor dialog box

Use the *Debug and Trace Services Layer* (DTSL) Configuration Editor to configure additional debug and trace settings. The configuration options available depend on the capabilities of the target. Typically, they enable configuration of the trace collection method and the trace that is generated.

A typical set of configuration options might include:

Trace Capture

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.

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 trace

Enable or disable cycle accurate trace.

Trace capture range

Specify an address range to limit the trace capture.

ITM

Enable or disable trace collection from the ITM unit.

Named DTSL configuration profiles can be saved for later use.

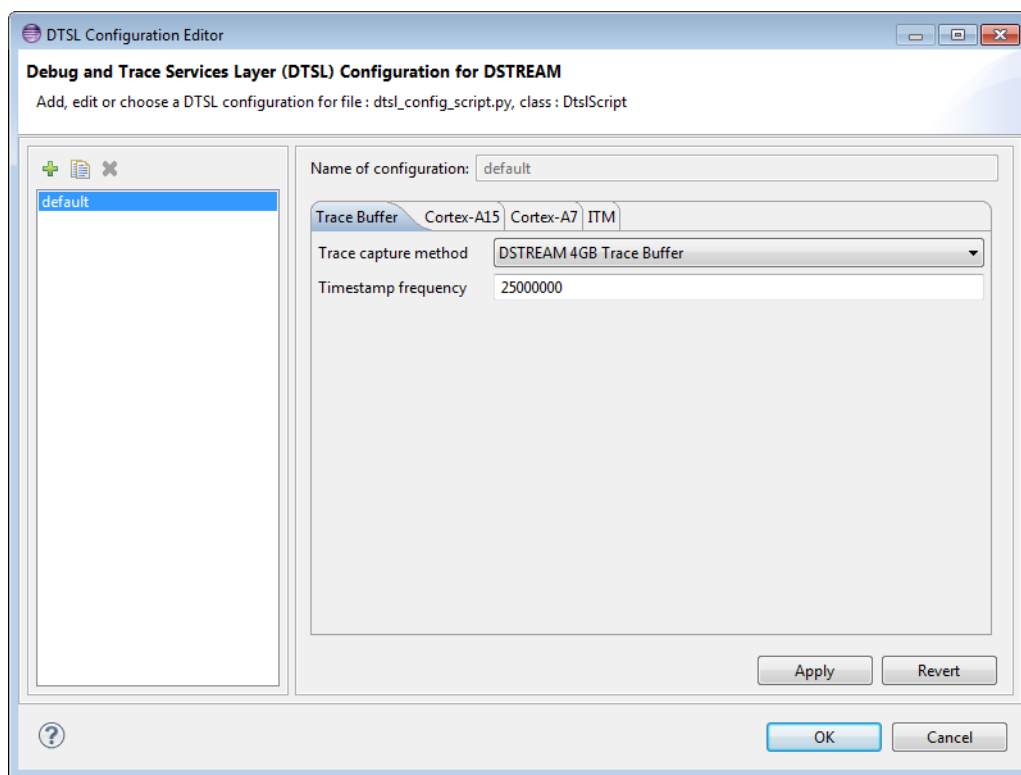


Figure 10-51 DTSL Configuration Editor (Shown with Trace capture method set to DSTREAM)

Related concepts

[6.17 About debugging caches on page 6-165.](#)

Related references

[10.19 Cache Data view on page 10-255.](#)

Related information

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

10.45 Configuration Database panel

Use the **Configuration Database** panel to manage the configuration database settings.

Default Configuration Databases

Displays the default DS-5 configuration databases.

Note

ARM recommends that you do not disable these.

User Configuration Databases

Enables you to add your own configuration database.

Add

Opens a dialog box in which you can select a configuration database to add.

Edit

Opens a dialog box in which you can modify the name and location of 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

Rebuilds the configuration database.

Test platforms...

Enables you to select which platforms to test, then tests them, and reports any errors found.

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 settings.

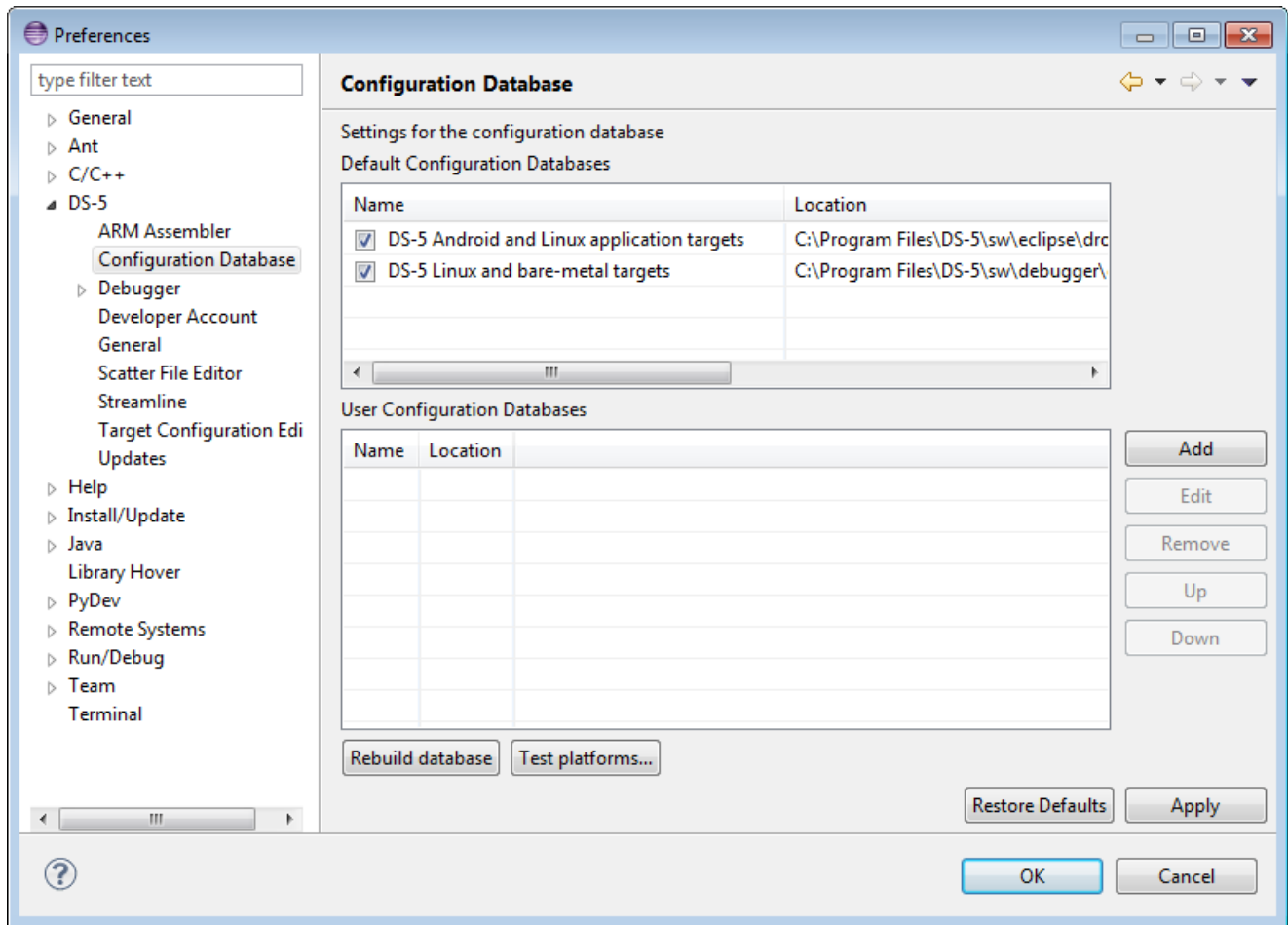


Figure 10-52 Configuration Database panel

Related concepts

[2.10 About the target configuration import utility on page 2-56.](#)

Related tasks

[2.11 Adding a new platform on page 2-58.](#)

[2.12 Adding a new configuration database to DS-5 on page 2-60.](#)

10.46 Configuration Perspective

The Configuration Perspective enables you to view and import model configurations in DS-5.

To access the Configuration Perspective, click **Window > Open Perspective > DS-5 Configuration** from the main menu.

Use the Configuration Perspective to:

- View and edit model configurations in the Model Configurations Editor.
- Import models configurations into your DS-5 Debug Configuration Database.
- Create a launch configuration for the new model configuration.

From the Configuration Perspective, you can use the **Tools** menu to access the following views:

- Debug Hardware Firmware Installer
- Debug Hardware Configure IP.

This section contains the following subsections:

- [10.46.1 Importing and connecting to a model on page 10-312.](#)
- [10.46.2 Models Configurations Editor - Model Devices and Cluster Configuration on page 10-313.](#)
- [10.46.3 Updating multiple debug hardware units on page 10-314.](#)
- [10.46.4 About platform bring-up in DS-5 on page 10-315.](#)
- [10.46.5 About the Platform Configuration Editor view on page 10-315.](#)
- [10.46.6 Creating a new platform configuration on page 10-316.](#)
- [10.46.7 Editing a platform configuration in the PCE on page 10-320.](#)
- [10.46.8 About the device hierarchy in the PCE view on page 10-326.](#)
- [10.46.9 Configuring your debug hardware unit for platform autodetection on page 10-327.](#)

10.46.1 Importing and connecting to a model

DS-5 Debugger provides built-in support for connecting to a large range of ARM FastModel products. To use any other simulation model with DS-5 Debugger you must first import it into the configuration database.

Procedure

1. Open the Configuration Perspective.
2. Select **File > New > Configuration Database** to open the **New Configuration Database** dialog.
3. Enter a name for the new database and click **Finish**.
4. In the **Project Explorer** view, right click and select **New > Model Configuration** from the context menu to open the **New Model** dialog.
5. Select the database in which to create the new model entry, then click **Next**. This opens the **Select Method for Connecting to Model** dialog.
6. Select the method for connecting to the model. The options are:
 - a. **Launch and connect to a specific model.** Click **Next** to select a specific model from the file system. Click **File** to browse for a model, for example from the DS-5 installation.
 - b. **Browse for model running on local host.** Click **Next** to browse for models running on the local host. Then click **Browse** to display the list of running models. Select the model from the **Model Browser** dialog.
7. Click **Finish** to connect to the model. This opens the Models Configuration Editor, where you can view and edit the devices in the model.
8. Click **Save** to save the changes to the model to the .mdf file.
9. Click **Import** to create DS-5 Configuration Database files. This adds the configuration to the DS-5 preferences.

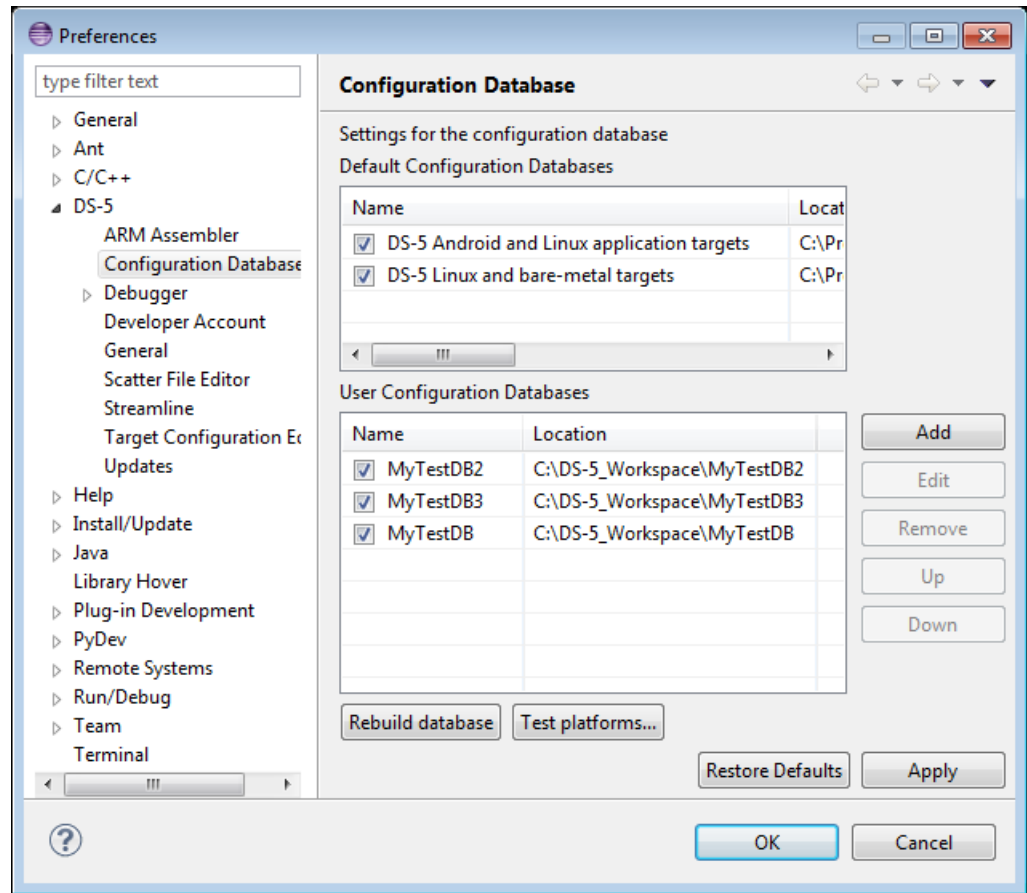


Figure 10-53 Configuration Database

10. Click **Debug** to create a new Debug Configuration entry and open the **Debug Configurations** dialog. You must configure the connection before starting the debug session.

Related references

[10.46.2 Models Configurations Editor - Model Devices and Cluster Configuration](#) on page 10-313.

10.46.2 Models Configurations Editor - Model Devices and Cluster Configuration

Use the Model Devices and Cluster Configuration tab to view and configure the devices in the model.

Executable Devices

Lists the cores available within the model. You can add, remove, or edit the available cores.

Associations

Lists the non-executable devices within the model. You can expand the associations to see how the non-executable devices are mapped. You can delete items from the associations view or add items from the list of available non-executable devices.

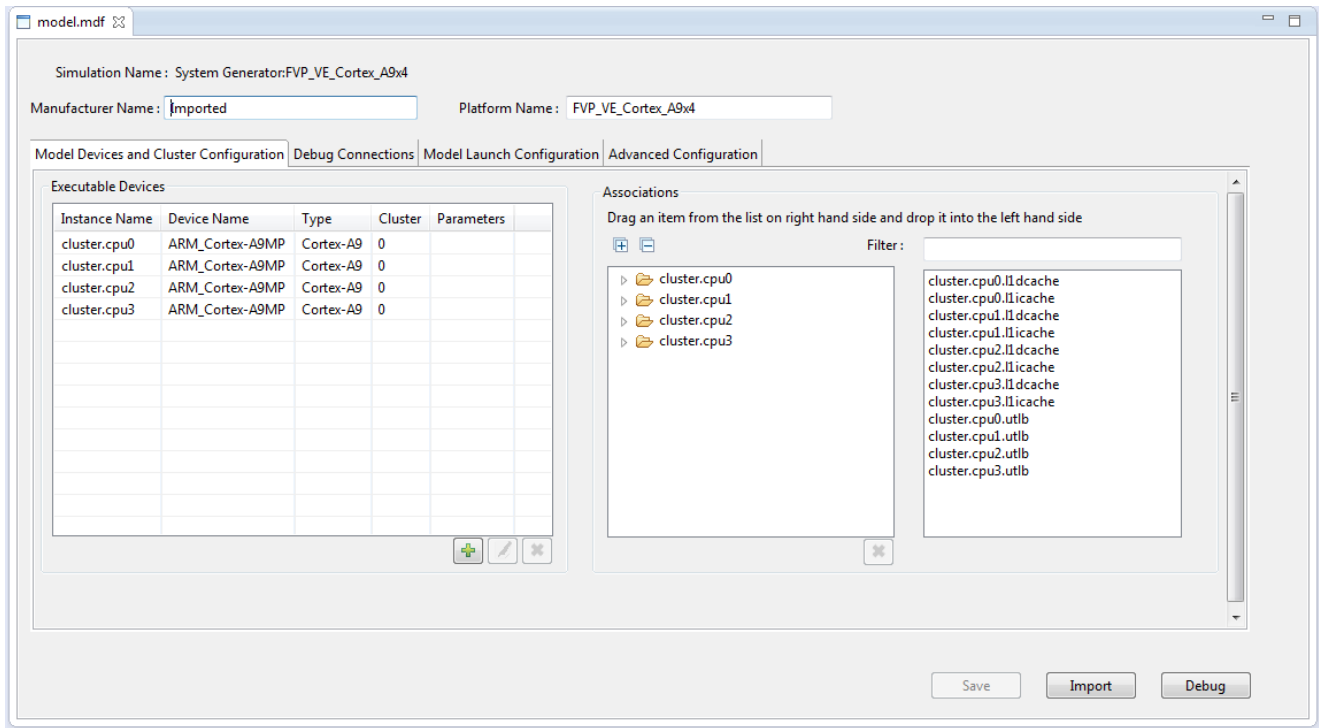


Figure 10-54 Model Devices and Cluster Configuration tab

Related tasks

[10.46.1 Importing and connecting to a model on page 10-312.](#)

10.46.3 Updating multiple debug hardware units

To update multiple debug hardware units, use the `dbghw_batchupdater` command line utility.

The command line utility, `dbghw_batchupdater`, enables you to:

- Install firmware on a group of DSTREAM or RVI units.
- View the firmware versions on a group of DSTREAM or RVI units.

The input to `dbghw_batchupdater` is a file containing a list of DSTREAM or RVI units. Each line in the input file is a string that specifies a single DSTREAM or RVI connection. Firmware images are available within a subdirectory of the DS-5 installation.

Syntax

```
dbghw_batchupdater -list file [-option]...
```

Where:

`list file` Specifies the file containing a list of DSTREAM or RVI connection strings.

`option:` `log file`

Specifies an output file to log the status of the update.

`updatefile file`

Specifies a file containing the path to the firmware.

`i`

Installs the firmware on the units. To install the firmware, you must also specify the `updatefile` option.

`v`

Lists the firmware versions.

`h`

Displays help information. This is the default if no arguments are specified.

Example 10-2 Examples

```
# Input file C:\input_file.txt contains:
# TCP:ds-sheep1
# TCP:DS-Rhubarb

# List firmware versions.
dbghw_batchupdater -list "C:\input_file.txt" -v
Versions queried on 2014-06-10 10:42:36
TCP:ds-sheep1: 4.18.0 Engineer build 3
TCP:DS-Rhubarb: 4.17.0 build 27

# Install firmware on DSTREAMs
dbghw_batchupdater.exe -list "C:\input_file.txt" -i -updatefile "C:\Program Files\DS-5\sw\debughw\firmware\ARM-RVI-4.17.0-27-base.dstream" -log out.log
```

Related references

[10.54 Debug Hardware Configure IP view on page 10-337.](#)

[10.55 Debug Hardware Firmware Installer view on page 10-339.](#)

10.46.4 About platform bring-up in DS-5

Effective debug and trace support requires that the debugger has the necessary information about the platform it needs to debug.

The complexity of modern System-on-Chip (SoC) based platforms is increasing. The debugger needs to know:

- What devices are present on the SoC.
- The type and configuration details of each device.
- The base addresses of the CoreSight components.
- The type and index of the Access Ports (AP) to access the various CoreSight components.
- How the different devices relate or connect to each other (their topology).

DS-5 Debugger can automatically detect most of this information. However, it is common that the debugger is unable to detect certain features on a complex SoC. The reasons might be:

- The SoC does not make the information available to the debugger.
- The information from the SoC might be missing when parts of the SoC are powered down.
- Devices inside the SoC might interfere with topology detection.

DS-5 Debugger provides a simple and efficient platform bring-up process for all ARM CoreSight-based SoCs.

DS-5 Debugger does not make any assumptions about the platform configuration. Hence if it only has limited information about the platform, it can only provide limited debug and trace functionality. Hence it is common for the debugger to provide limited trace functionality for certain processors on the platform. You can augment the automatically detected platform configuration by manually providing data based on your knowledge of the SoC. This enables you to provide any missing parts of the topology definition. This is preferable because the DTSL scripts can be generated based on the augmented information rather than requiring manual editing.

10.46.5 About the Platform Configuration Editor view

The Platform Configuration Editor (PCE) view enables you to create or modify debug configurations for connecting to your platform.

In the PCE view, you can:

- Review the devices on your development platform.
- Modify device information or add new devices that the debugger was unable to autodetect.

- Configure your debug hardware unit, and target-related features that are appropriate to correctly debug on your development platform.
- Review or modify the debug activities for the various processors on the platform.
- Build and save the platform configuration to an RDDI configuration file. The RDDI configuration file is used by the debugger to connect to the target processors on your development platform.

PCE enables you to easily specify the debug topology by defining the connections between the various processors, CoreSight components, and debug IP on the platform. This enables DS-5 Debugger to create the appropriate DTSL scripts that are necessary for the debug connection to the platform.

Note

DS-5 Debugger connects to the platform to autodetect the devices on the platform. DS-5 Debugger does not maintain the connection to the target after reading the device information from the platform.

10.46.6 Creating a new platform configuration

You can use the Platform Configuration Editor (PCE) within DS-5 to create debug configurations for new platforms.

Creating a new platform configuration in DS-5 requires a new Platform Configuration project in Eclipse.

Procedure

1. From the main Eclipse menu in DS-5, select **File > New > Other**. This opens the new project dialog.
2. Select **DS-5 Configuration Database > Platform Configuration** and then click **Next**.

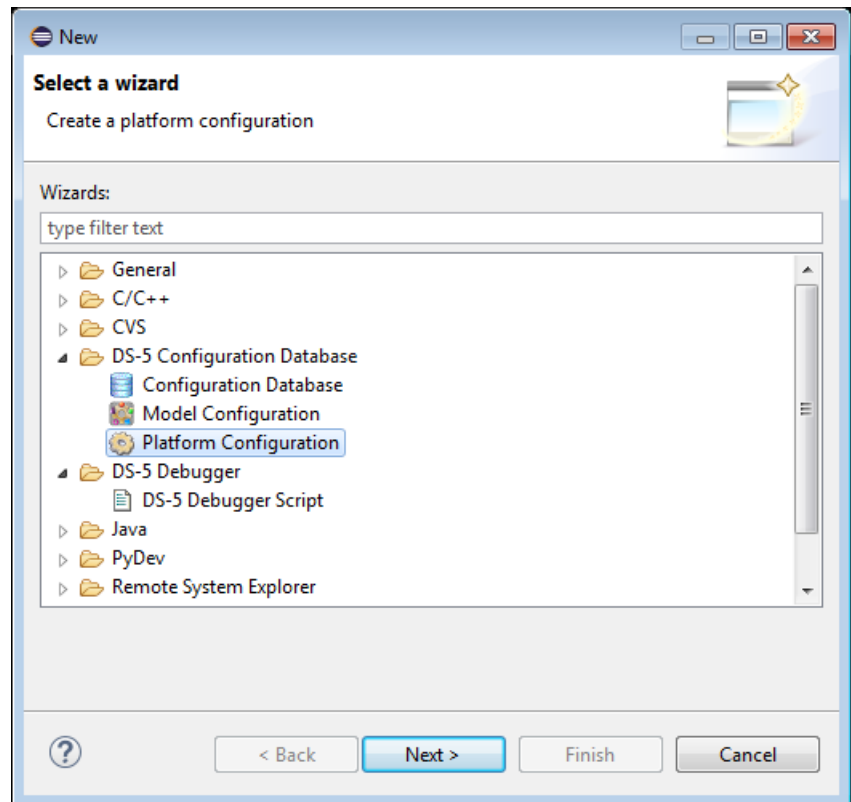


Figure 10-55 New PCE project

3. In the **New Platform** dialog box, select a configuration database, or click **Create New Database** to create a new configuration database and enter a name for your configuration database. Click **Next**. All DS-5 platform configurations must be stored in a configuration database.

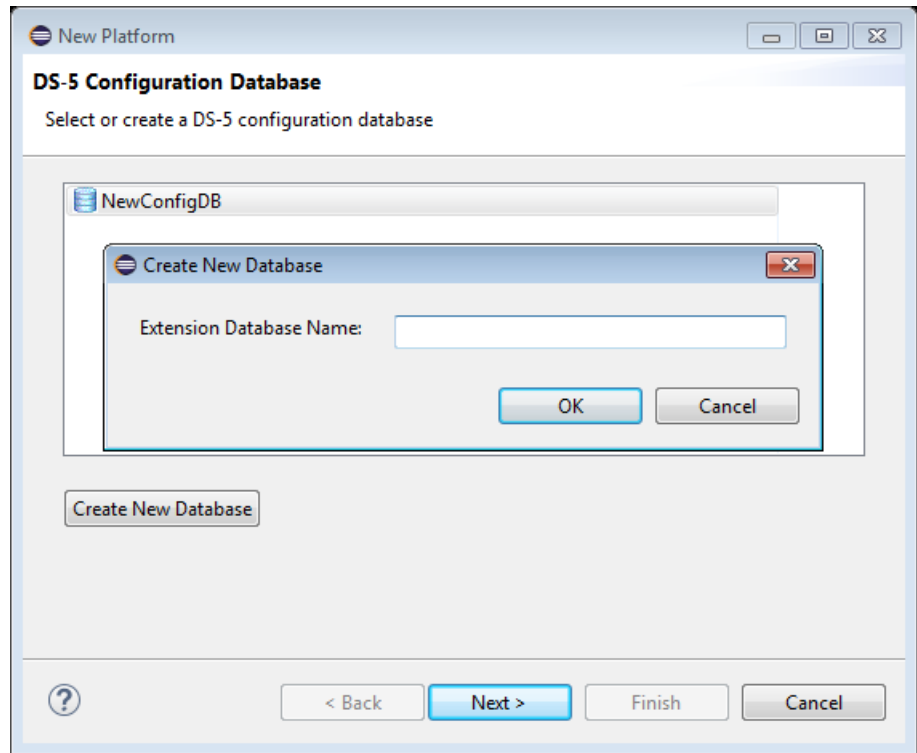


Figure 10-56 New configuration database

4. In the **Platform Information** dialog box, enter the **Platform Manufacturer**, for example ARM. Enter the **Platform Name**, for example Juno. If you want to provide a URL link to information about the platform, you can enter it in **Platform Info URL**. The URL appears in the **Debug Configurations** panel when you select a debug activity for the platform. Click **Next**.

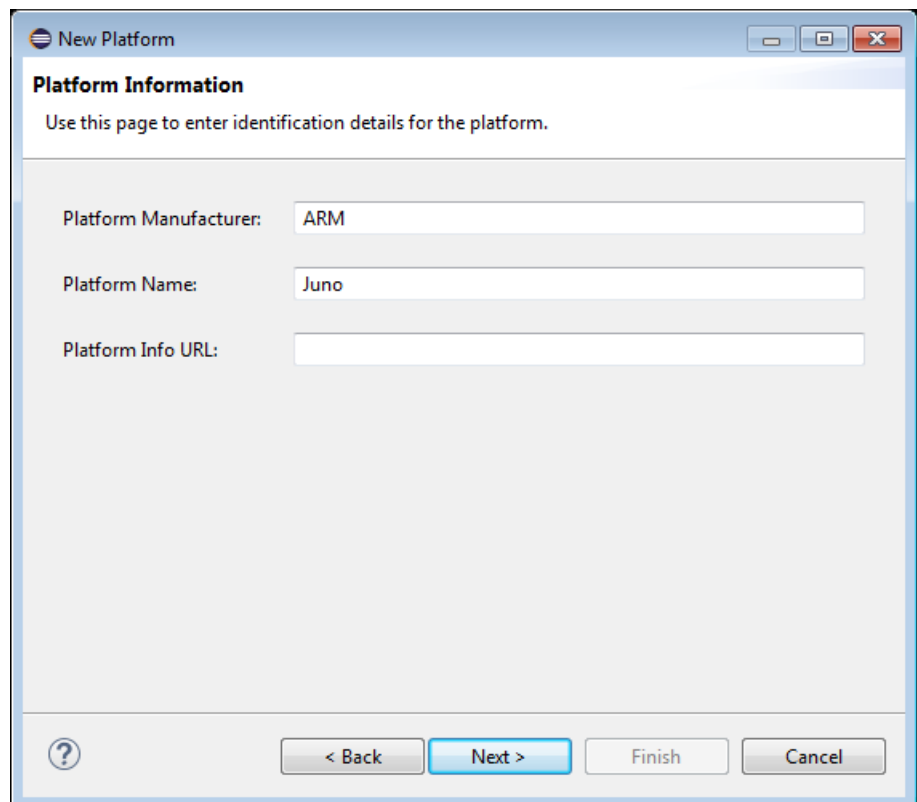


Figure 10-57 New platform information

5. This shows the **Platform Configuration Creation** dialog box. Choose the option most appropriate to how you want to create the platform configuration:

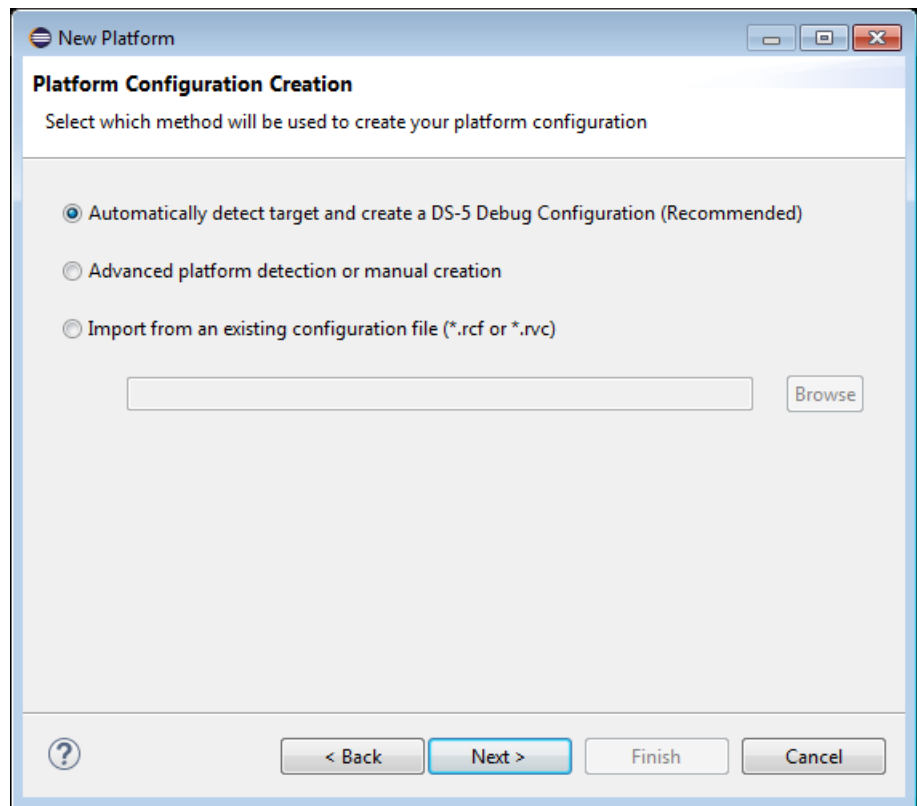


Figure 10-58 Platform creation options

- **Automatically detect target and create a DS-5 Debug Configuration.** This is the recommended option. DS-5 automatically detects the devices that are present on your platform. It then provides you the opportunity to add more devices if needed and to specify how the devices are interconnected.
 - **Advanced platform detection or manual creation.** This gives you control over the individual stages involved in reading the device information from your platform. This is useful if reading certain device information is likely to make the platform unresponsive.
 - **Import from an existing configuration file (*.rcf or *.rvc).** Use this if you already have a configuration file for the platform. You can then provide additional information about the other debug components on the platform.
6. Select the **Automatically detect target and create a DS-5 Debug Configuration** option and then click **Next**.
7. In the **Debug Vehicle Connection** dialog box, specify the **Connection Address**, for example TCP: 11.22.100.20 or click **Browse** to select the debug hardware unit that is attached to your platform.

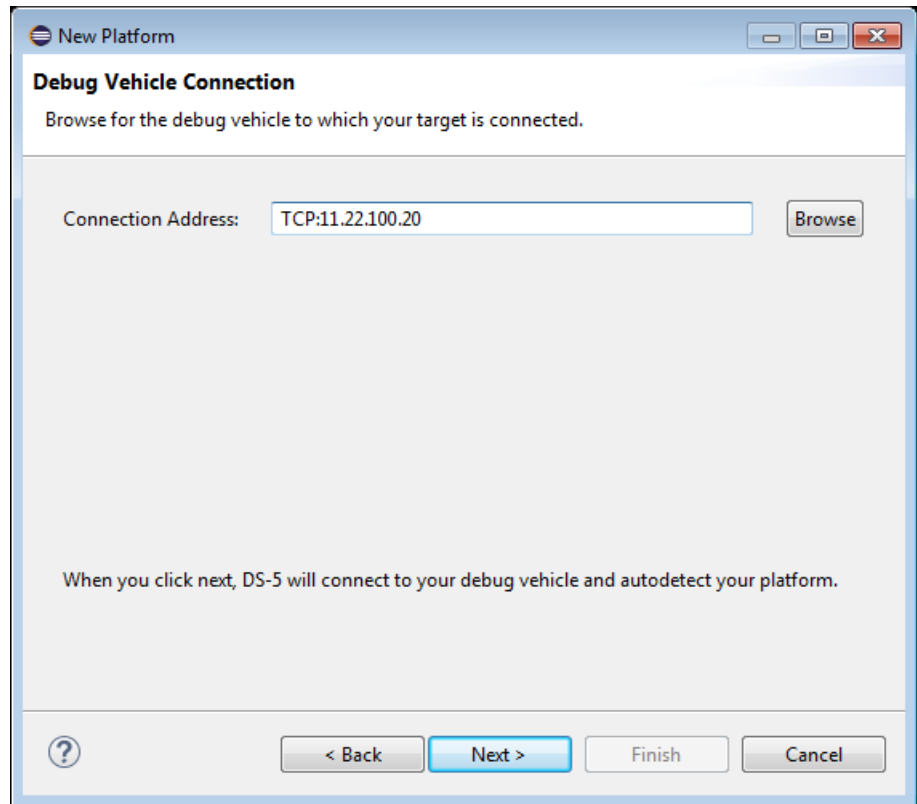


Figure 10-59 Connection address

8. DS-5 connects to the platform and reads all the device information that it can from the platform. The **Summary** dialog box shows the list of devices that DS-5 autodetected on the platform. Click **Finish** and DS-5 builds the platform configuration based on the information it has autodetected.

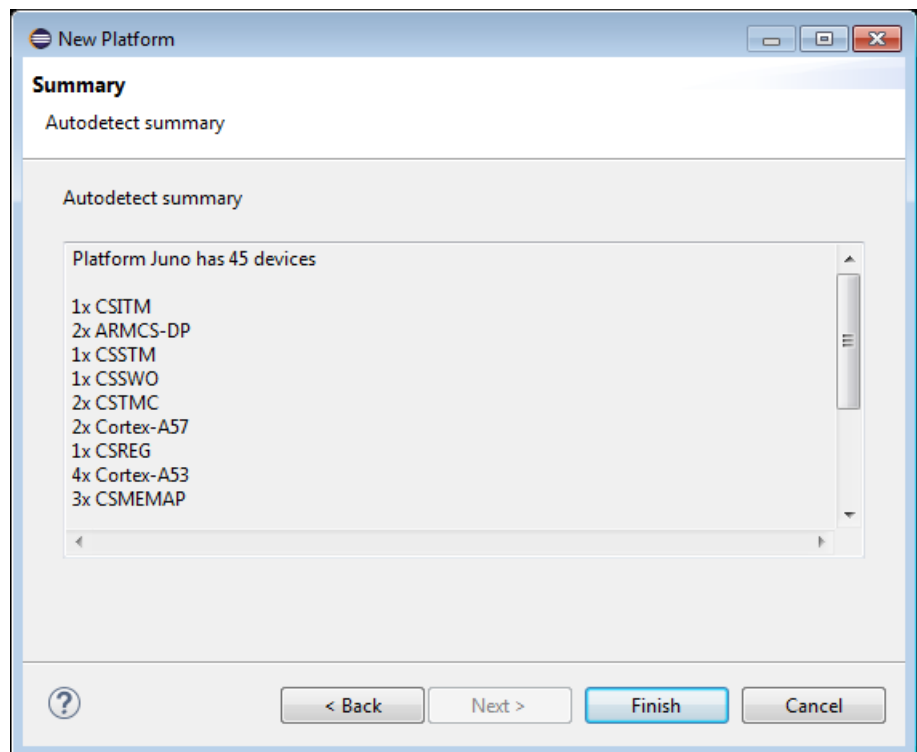


Figure 10-60 Autodetect summary

————— Note —————

DS-5 does not make assumptions about how the devices are connected if it is unable to obtain this information from the platform. In particular, DS-5 might not have obtained all the necessary information about the trace devices on the platform. You can provide this information in the PCE view.

9. DS-5 creates the platform configuration, and provides the option to review or debug the platform:
 - **Review Platform.** This displays the PCE view in DS-5. It shows all the devices and information that DS-5 knows about the platform. In the PCE view, you can provide the information about the platform that DS-5 was unable to autodetect.
 - **Debug Platform.** This displays the **Debug Configurations** dialog box for your platform. You can configure your debug activity and start a debug session on your platform.
 - **Do Nothing.** DS-5 saves the platform configuration to your configuration database. You can open this in DS-5 later to modify it or you can use it to connect to and debug the platform later.

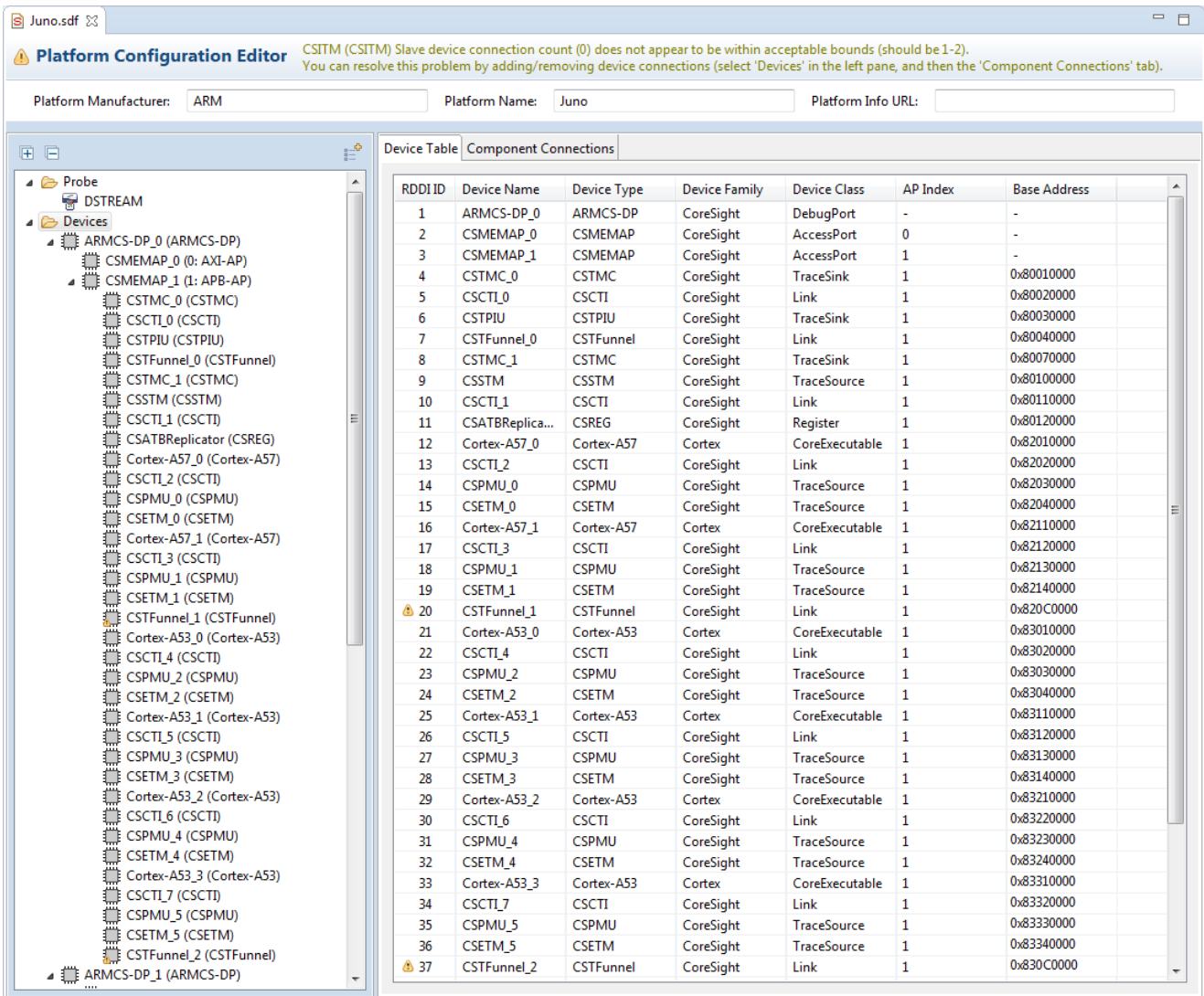


Figure 10-61 Platform Configuration view

10.46.7 Editing a platform configuration in the PCE

This shows you how to add component connections in the PCE view to support trace.

After you create a new platform configuration in DS-5, you can review the platform configuration. This shows the device summary in the PCE view. The left-hand pane contains:

Probe

This shows the configuration for your debug hardware unit, for example your DSTREAM unit. You can enter the **Connection Address** in the **Autodetect** tab if you want to autodetect the platform again after modifying the debug hardware settings.

Devices

This shows the scanchain and device hierarchy of your platform. The device hierarchy usually consists of one or more Debug Access Ports (DAP). Each DAP consists of one or more Access Ports (AP). Each AP shows the devices that have been detected through that access port.

Debug Activities

This shows the type of debug activities you can perform on the target. The debug activities are accessible from the **Debug Configurations** dialog box when you want to start a debug session.

You can use the PCE view to identify missing component connections and to add them to your platform configuration. The figure shows the configuration of an example platform in the PCE view after autodetection in DS-5.

Platform Configuration Editor - Cortex-M3 (Cortex-M3) Slave device connection count (0) does not appear to be within acceptable bounds (should be 1-2). You can resolve this problem by adding/removing device connections (select 'Devices' in the left pane, and then the 'Component Connections' tab).

Platform Manufacturer: ARM Platform Name: Juno Platform Info URL:

Device Table Component Connections

Component Connections Help

Master	Slave	Link Type	Link Details	Link Origin
CSTFunnel_0 (0x80040000)	CSTFunnel_2 (0x830C0000)	ATB	Master Interface = 0, Slave Interface = 1	Detected
CSSTI (0x80100000)	CSTFunnel_0 (0x80040000)	ATB	Master Interface = 0, Slave Interface = 2	Detected
CSETM_0 (0x82040000)	CSTFunnel_2 (0x830C0000)	ATB	Master Interface = 0, Slave Interface = 0	Detected
CSTFunnel_1 (0x820C0000)	CSTFunnel_2 (0x830C0000)	ATB	Master Interface = 0, Slave Interface = 1	Detected
CSETM_2 (0x83040000)	CSTFunnel_1 (0x820C0000)	ATB	Master Interface = 0, Slave Interface = 1	Detected
CSETM_2 (0x83040000)	CSTFunnel_2 (0x830C0000)	ATB	Master Interface = 0, Slave Interface = 1	Detected
CSETM_3 (0x83140000)	CSTFunnel_2 (0x830C0000)	ATB	Master Interface = 0, Slave Interface = 0	Detected
CSETM_4 (0x83240000)	CSTFunnel_1 (0x820C0000)	ATB	Master Interface = 0, Slave Interface = 0	Detected
CSETM_4 (0x83240000)	CSTFunnel_2 (0x830C0000)	ATB	Master Interface = 0, Slave Interface = 3	Detected
CSETM_5 (0x83340000)	CSTFunnel_2 (0x830C0000)	ATB	Master Interface = 0, Slave Interface = 0	Detected
CSETM_6 (0xE0041000)	CSTFunnel_2 (0x830C0000)	ATB	Master Interface = 0, Slave Interface = 1	Detected
CSTFunnel_3 (0xE0042000)	CSTFunnel_2 (0x830C0000)	ATB	Master Interface = 0, Slave Interface = 0	Detected
CSITM (0xE0000000)	CSTFunnel_1 (0x820C0000)	ATB	Master Interface = 0, Slave Interface = 0	Detected
CSCTI_0 (0x80020000)	CSCTI_0 (0x80010000)	CTITrigger	Trigger Out = 1	Detected
CSCTI_0 (0x80020000)	CSCTI_0 (0x80030000)	CTITrigger	Trigger Out = 3	Detected
CSCTI_0 (0x80020000)	CSCTI_1 (0x80070000)	CTITrigger	Trigger Out = 1	Detected
Cortex-A57_0 (0x82010000)	CSCTI_2 (0x82020000)	CTITrigger	Trigger DBGRESTART = 1	Detected
Cortex-A57_1 (0x82110000)	CSCTI_3 (0x82120000)	CTITrigger	Trigger DBGRESTART = 1	Detected
Cortex-A53_0 (0x83010000)	CSCTI_3 (0x82120000)	CTITrigger	Trigger DBGRESTART = 1	Detected
Cortex-A57_0 (0x82010000)	CSETM_0 (0x82040000)	CoreTrace	N/A	Detected
Cortex-A57_1 (0x82110000)	CSETM_1 (0x82140000)	CoreTrace	N/A	Detected
Cortex-A53_0 (0x83010000)	CSCTI_4 (0x83020000)	CTITrigger	Trigger DBGRESTART = 1	Detected
Cortex-A53_0 (0x83010000)	CSETM_2 (0x83040000)	CoreTrace	N/A	Detected
CSETM_2 (0x83040000)	CSCTI_4 (0x83020000)	CTITrigger	Trigger In = 6	Detected
Cortex-A53_1 (0x83110000)	CSCTI_5 (0x83120000)	CTITrigger	Trigger DBGRESTART = 1	Detected
Cortex-A53_1 (0x83110000)	CSETM_3 (0x83140000)	CoreTrace	N/A	Detected
CSETM_3 (0x83140000)	CSCTI_5 (0x83120000)	CTITrigger	Trigger In = 6	Detected
Cortex-A53_2 (0x83210000)	CSCTI_6 (0x83220000)	CTITrigger	Trigger DBGRESTART = 1	Detected
Cortex-A53_2 (0x83210000)	CSETM_4 (0x83240000)	CoreTrace	N/A	Detected
CSETM_4 (0x83240000)	CSCTI_6 (0x83220000)	CTITrigger	Trigger In = 6	Detected
Cortex-A53_3 (0x83310000)	CSCTI_7 (0x83320000)	CTITrigger	Trigger DBGRESTART = 1	Detected
Cortex-A53_3 (0x83310000)	CSETM_5 (0x83340000)	CoreTrace	N/A	Detected
CSETM_5 (0x83340000)	CSCTI_7 (0x83320000)	CTITrigger	Trigger In = 6	Detected
CSETM_0 (0x82040000)	CSCTI_2 (0x82020000)	CTITrigger	Trigger In = 6	Detected
CSETM_1 (0x82140000)	CSCTI_3 (0x82120000)	CTITrigger	Trigger In = 6	Detected

Add Link Autodetect Component Connections

Figure 10-62 Component Connections

The PCE view shows, in the title bar, that the Cortex-M3 processor does not have any slave connections. Autodetection was unable to determine the topology information for the Cortex-M3 processor. The device hierarchy on the left-hand pane shows that the Cortex-M3 processor and the trace source, CSETM_6, are under the same access port, CSSEMAP_2.

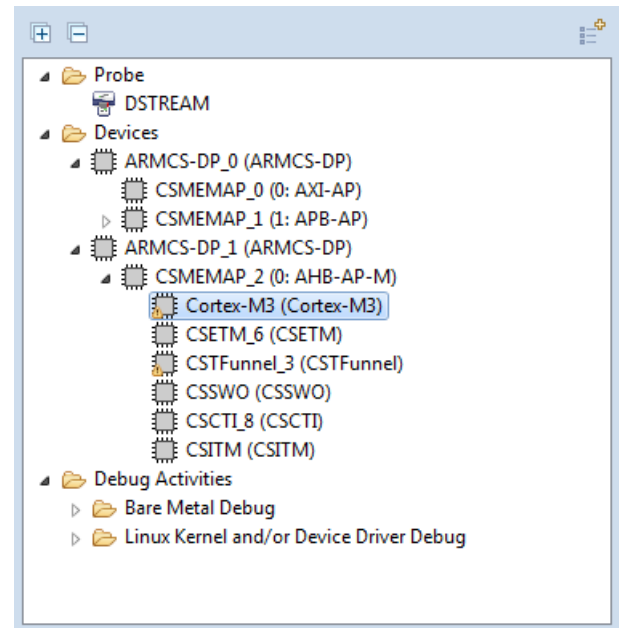


Figure 10-63 Device hierarchy

You can now add a component connection between the Cortex-M3 processor and the ETM to enable trace for the Cortex-M3 processor on the platform.

Note

To add the right components to your platform configuration you must know the device topology. This demonstrates adding components to an example platform.

Procedure

1. Select **Devices** in the left-hand pane.
2. Select the **Component Connections** tab in the right-hand pane.
3. To add a new component connection, click **Add Link**. This shows the **Add Link** dialog box.
4. Select Core Trace for the **Type**, Cortex-M3 for the **Master**, CSETM_6 for the **Slave**. Click **Ok** to add the component connection.

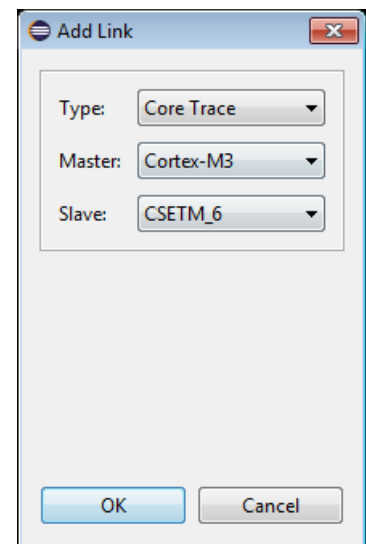


Figure 10-64 Add Core Trace

5. The PCE view shows that CSETM_6 does not have any slave connections. The device hierarchy shows that there is a CSCTI_8 component under the same access port. To add this component connection to CSETM_6, click **Add Link** and select CTI Trigger for **Type**, CSETM_6 for **Master**, and CSCTI_8 for **Slave**.

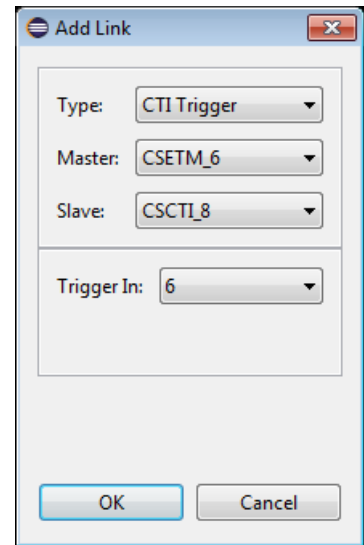


Figure 10-65 Add CTI Trigger

6. The **Component Connections** tab shows the user added component connections in the PCE view. Save the platform configuration.

Component Connections Help				
Master	Slave	Link Type	Link Details	Link Origin
CSETM_0 (0x82040000)	CSTFunnel_2 (0x830C0000)	ATB	Master Interface = 0, Slave Interface = 0	Detected
CSETM_1 (0x82140000)	CSTFunnel_2 (0x830C0000)	ATB	Master Interface = 0, Slave Interface = 0	Detected
CSTFunnel_1 (0x820C0000)	CSTFunnel_0 (0x80040000)	ATB	Master Interface = 0, Slave Interface = 0	Detected
CSTFunnel_1 (0x820C0000)	CSTFunnel_2 (0x830C0000)	ATB	Master Interface = 0, Slave Interface = 0	Detected
CSETM_2 (0x83040000)	CSTFunnel_2 (0x830C0000)	ATB	Master Interface = 0, Slave Interface = 1	Detected
CSETM_3 (0x83140000)	CSTFunnel_2 (0x830C0000)	ATB	Master Interface = 0, Slave Interface = 0	Detected
CSETM_4 (0x83240000)	CSTFunnel_2 (0x830C0000)	ATB	Master Interface = 0, Slave Interface = 1	Detected
CSETM_5 (0x83340000)	CSTFunnel_2 (0x830C0000)	ATB	Master Interface = 0, Slave Interface = 1	Detected
CSETM_6 (0xE0041000)	CSTFunnel_2 (0x830C0000)	ATB	Master Interface = 0, Slave Interface = 0	Detected
CSTFunnel_3 (0xE0042000)	CSTFunnel_2 (0x830C0000)	ATB	Master Interface = 0, Slave Interface = 3	Detected
CSCTI_0 (0x80020000)	CSTMC_0 (0x80010000)	CTITrigger	Trigger Out = 1	Detected
CSCTI_0 (0x80020000)	CSTPIU (0x80030000)	CTITrigger	Trigger Out = 3	Detected
CSCTI_0 (0x80020000)	CSTMC_1 (0x80070000)	CTITrigger	Trigger Out = 1	Detected
Cortex-A57_0 (0x82010000)	CSCTI_2 (0x82020000)	CTITrigger	Trigger DBGRESTART = 1	Detected
Cortex-A57_1 (0x82110000)	CSCTI_3 (0x82120000)	CTITrigger	Trigger DBGRESTART = 1	Detected
Cortex-A57_0 (0x82010000)	CSETM_0 (0x82040000)	CoreTrace	N/A	Detected
Cortex-A57_1 (0x82110000)	CSETM_1 (0x82140000)	CoreTrace	N/A	Detected
Cortex-A53_0 (0x83010000)	CSCTI_4 (0x83020000)	CTITrigger	Trigger DBGRESTART = 1	Detected
Cortex-A53_0 (0x83010000)	CSETM_2 (0x83040000)	CoreTrace	N/A	Detected
CSETM_2 (0x83040000)	CSCTI_4 (0x83020000)	CTITrigger	Trigger In = 6	Detected
Cortex-A53_1 (0x83110000)	CSCTI_5 (0x83120000)	CTITrigger	Trigger DBGRESTART = 1	Detected
Cortex-A53_1 (0x83110000)	CSETM_3 (0x83140000)	CoreTrace	N/A	Detected
CSETM_3 (0x83140000)	CSCTI_5 (0x83120000)	CTITrigger	Trigger In = 6	Detected
Cortex-A53_2 (0x83210000)	CSCTI_6 (0x83220000)	CTITrigger	Trigger DBGRESTART = 1	Detected
Cortex-A53_2 (0x83210000)	CSETM_4 (0x83240000)	CoreTrace	N/A	Detected
CSETM_4 (0x83240000)	CSCTI_6 (0x83220000)	CTITrigger	Trigger In = 6	Detected
Cortex-A53_3 (0x83310000)	CSCTI_7 (0x83320000)	CTITrigger	Trigger DBGRESTART = 1	Detected
Cortex-A53_3 (0x83310000)	CSETM_5 (0x83340000)	CoreTrace	N/A	Detected
CSETM_5 (0x83340000)	CSCTI_7 (0x83320000)	CTITrigger	Trigger In = 6	Detected
CSETM_0 (0x82040000)	CSCTI_2 (0x82020000)	CTITrigger	Trigger In = 6	Detected
CSETM_1 (0x82140000)	CSCTI_3 (0x82120000)	CTITrigger	Trigger In = 6	Detected
Cortex-M3 (0xE000ED00)	CSETM_6 (0xE0041000)	CoreTrace	N/A	User Added
CSETM_6 (0xE0041000)	CSCTI_8 (0xE0044000)	CTITrigger	Trigger In = 6	User Added

Figure 10-66 User added component connections

- Build the platform configuration. To build the platform configuration, right-click the project in the Project Explorer view and select **Build Platform**.

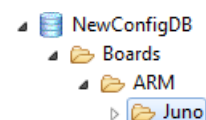


Figure 10-67 Project Explorer

- In the build platform dialog box, select **Full Debug and Trace**. This regenerates the Debug Configuration files with the user added CoreSight trace components. If you select **Debug Only**, the Debug Configuration files only contain configuration for a debug session without trace capability.

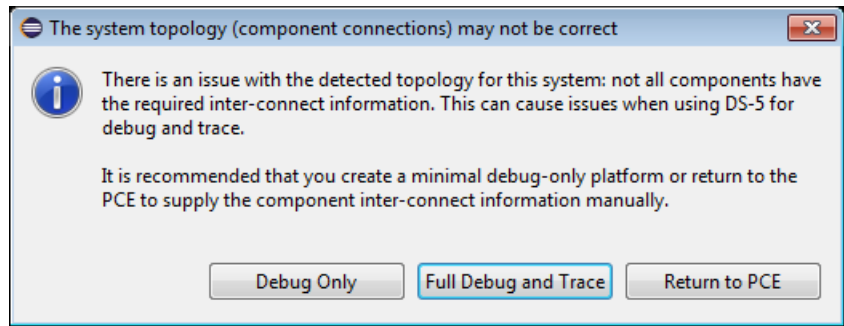


Figure 10-68 Full debug and trace

9. Open the **Debug Configurations** view by right-clicking in the Project Explorer view and selecting **Debug As > Debug Configurations...**

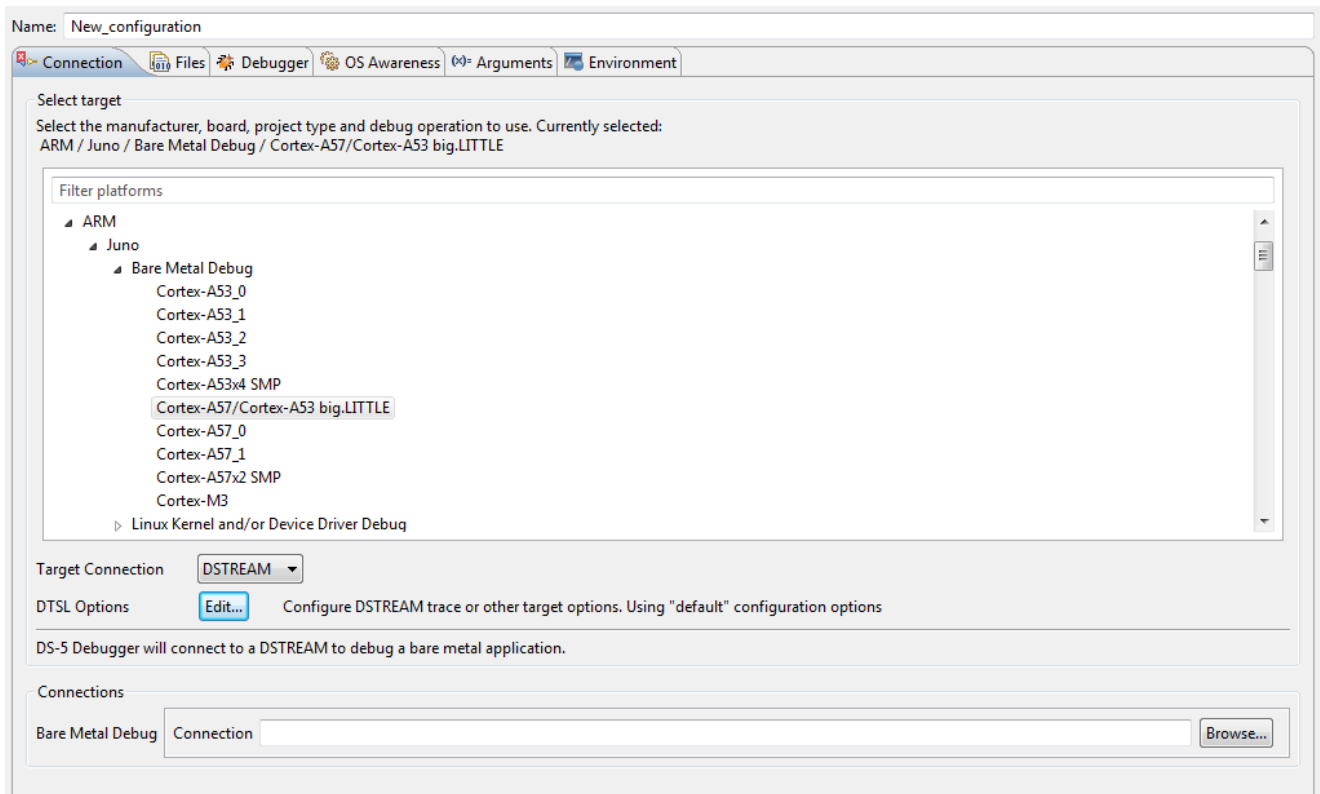


Figure 10-69 Debug Activities

10. Select your platform and debug activity from the **Connection** tab in the **Debug Configurations** dialog box.

————— **Note** —————

To see your new platform in the **Debug Configurations** list, your configuration database must be specified in **Window > Preferences > DS-5 > Configuration Database**.

11. Click **Edit** on **DTSL Options**. This shows that the new platform configuration provides trace capability for the Cortex-M3 processor.

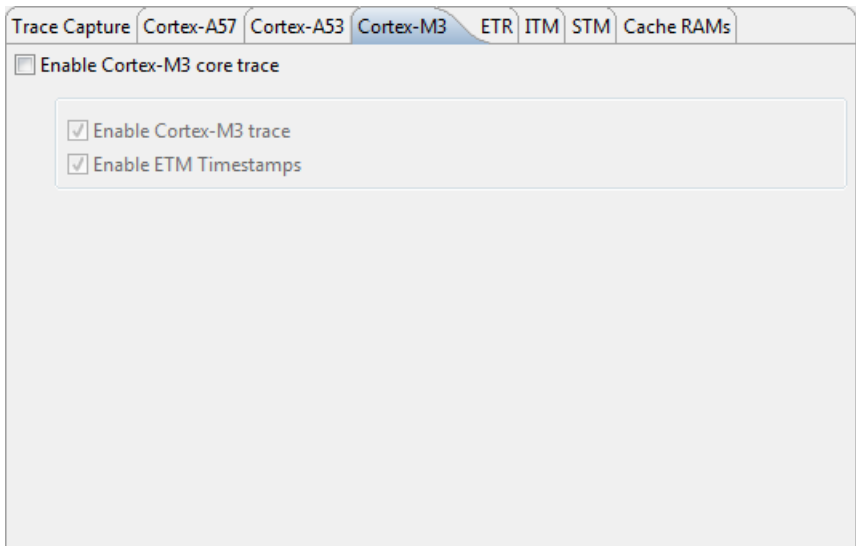


Figure 10-70 DTSL Options

10.46.8 About the device hierarchy in the PCE view

The device hierarchy in the PCE view shows the devices on the platform.

In the PCE view, you can add or remove devices to configure the platform for how you want to debug the platform. This shows the device hierarchy of an example platform. DS-5 Debugger might not autodetect all the devices on the platform. If you want to use these undetected devices in the debug session, you must first add them to the device hierarchy and configure them appropriately.

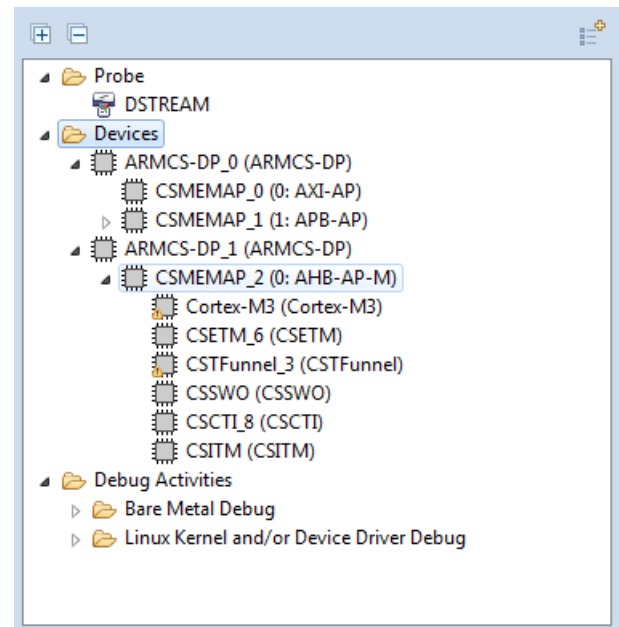


Figure 10-71 Device hierarchy

Alternatively, if you only want a simple debug session and do not need some of the devices that have been autodetected, then you can remove them from the device hierarchy.

The context menu for the device hierarchy contains:

Toggle Devices Panel

This shows or hides the devices that you can add to the device hierarchy.

Add Custom Device

This adds a device to the scanchain.

Autodetect Component Connections

This starts the autodetection to detect the connections between the various components on the platform.

Enumerate APs

This is available for Debug Access Ports (DAP) on the device hierarchy. This enumerates the Access Ports under the DAP.

Read CoreSight ROM Tables

This reads the CoreSight ROM tables to obtain more information about the devices from the various access ports. This might cause certain devices on the platform to become unresponsive. In such cases, you can unselect this option during autodetection, in the right-hand pane after selecting your debug hardware **Probe**.

Remove Device

This removes the device from the hierarchy.

You can add a device as a sibling or as a child, by drag-and-dropping from the Devices Panel to the appropriate place in the device hierarchy.

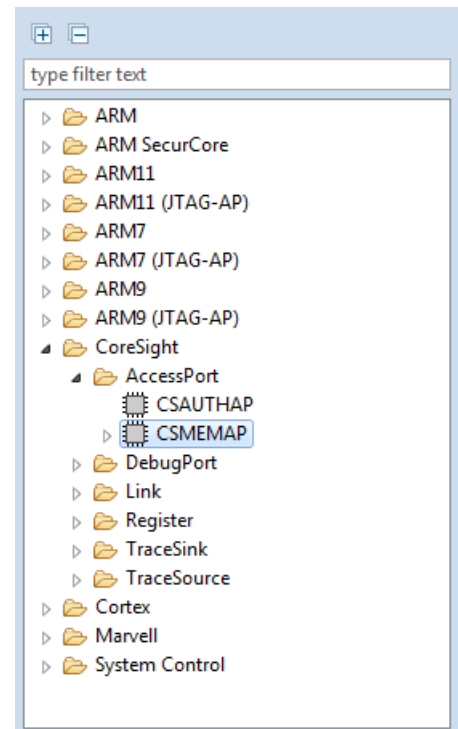


Figure 10-72 Devices Panel

Any device that you add or remove from the hierarchy changes the topology of the SoC. You must ensure that the topology is appropriate for your platform. After adding new devices, you can configure the devices in the right-hand pane in the PCE view.

10.46.9 Configuring your debug hardware unit for platform autodetection

Automatically detecting the correct configuration for your platform requires correct configuration of your debug hardware unit.

In the **Autodetect** tab of the Platform Configuration Editor (PCE) view, you can configure key settings used in autodetection, such as whether to use JTAG or Serial Wire Debug (SWD), and the clock speed.

Procedure

1. Select your debug hardware unit from the **Probe** in the left-hand pane of the PCE view.

2. Select the **Autodetect** tab from the right-hand pane.

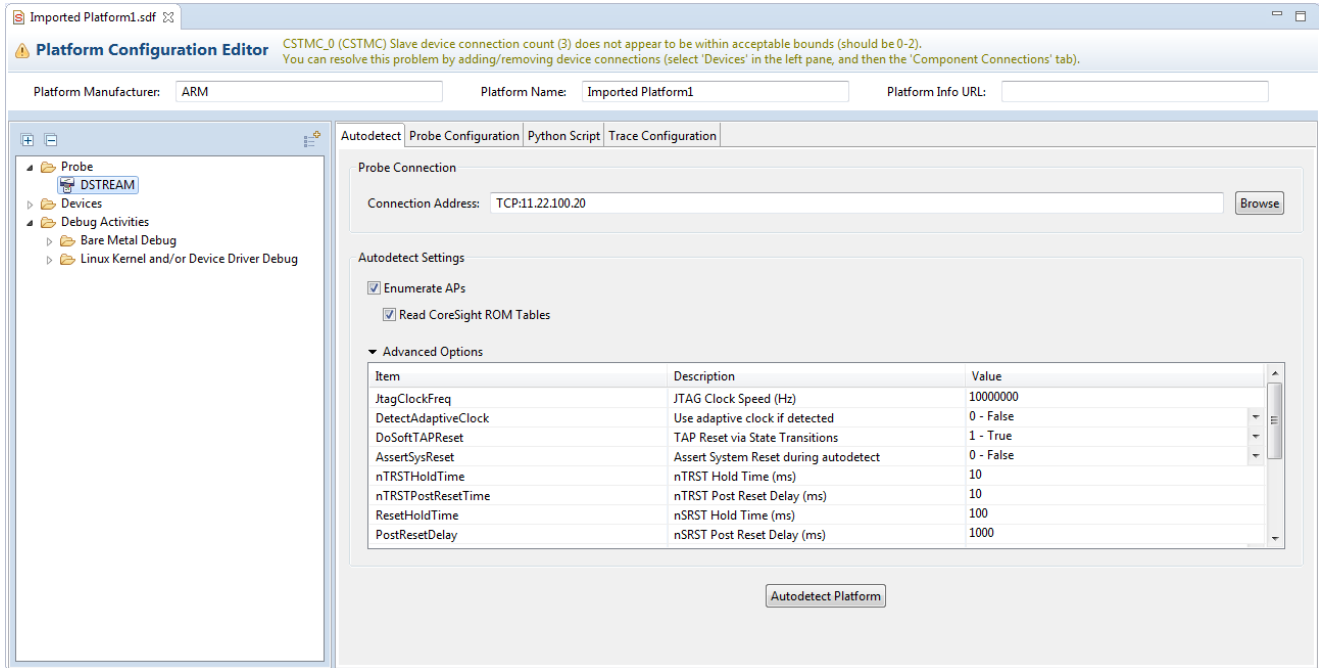


Figure 10-73 Autodetect settings

3. Enter the **Connection Address** of your debug hardware unit or click **Browse** to select one.
4. Expand **Advanced Options** to configure your debug hardware unit appropriately. Here you can set the LVDS Debug Interface Mode to either JTAG or SWD for connecting to the target. If you use a JTAG connection, then you must set the JTAG type and clock speed appropriately. Other options that are important for autodetection include the reset hold and delay times, drive strengths, reset behavior during autodetection, and cable pin configurations.

————— **Note** —————

If your target supports both JTAG and SWD, you must first enable Use SWJ Switching. The **Probe Configurations** tab provides these settings and other configuration options for your debug hardware unit.

5. Click **Autodetect Platform** to automatically detect the platform.
The debug hardware unit interrogates the scanchain at the current clock speed. If the clock speed is too high, some devices on the scanchain might not be detected. If you suspect that this is happening, decrease the clock speed.
6. Click on **Devices** in the left-hand pane to review the automatically detected platform configuration.
7. Add any missing topology links for the platform in the **Component Connections** tab.
8. Build the platform configuration by right-clicking the platform configuration project in the **Project Explorer** view and selecting **Build Platform**.

10.47 About the Remote System Explorer

Use the *Remote Systems Explorer* (RSE) perspective to connect to and work with a variety of remote systems.

It 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 them 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 an RSE 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**.

10.48 Remote Systems view

The **Remote Systems** view is a hierarchical tree view of local and remote systems.

It enables you to:

- Set up a connection to a remote target.
- 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.

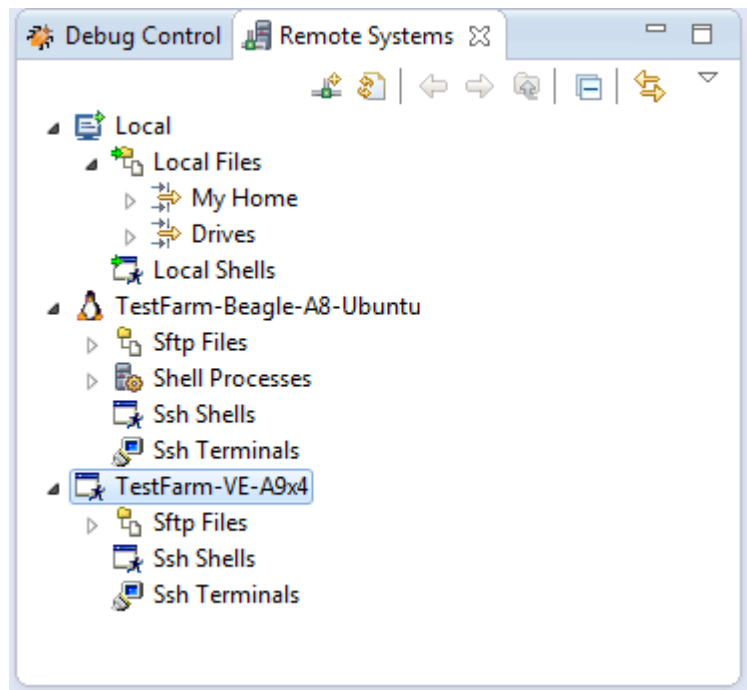


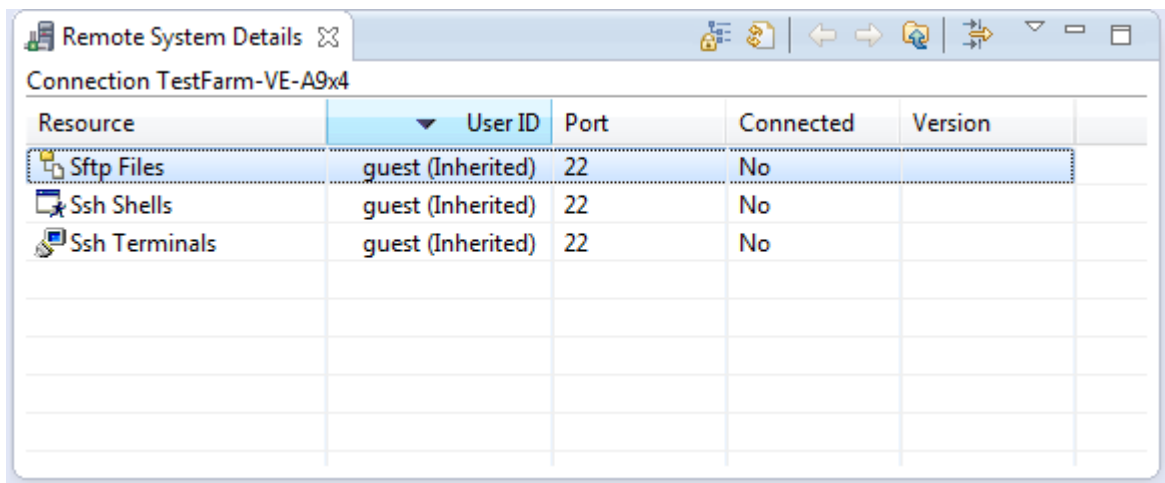
Figure 10-74 Remote Systems view

10.49 Remote System Details view

The **Remote System Details** view is a tabular view giving details about local and remote systems.

It enables you to:

- Set up a Linux connection to a remote target.
- 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.



Resource	User ID	Port	Connected	Version
Sftp Files	guest (Inherited)	22	No	
Ssh Shells	guest (Inherited)	22	No	
Ssh Terminals	guest (Inherited)	22	No	

Figure 10-75 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**.

10.50 Target management terminal for serial and SSH connections

Use the target management terminal 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.

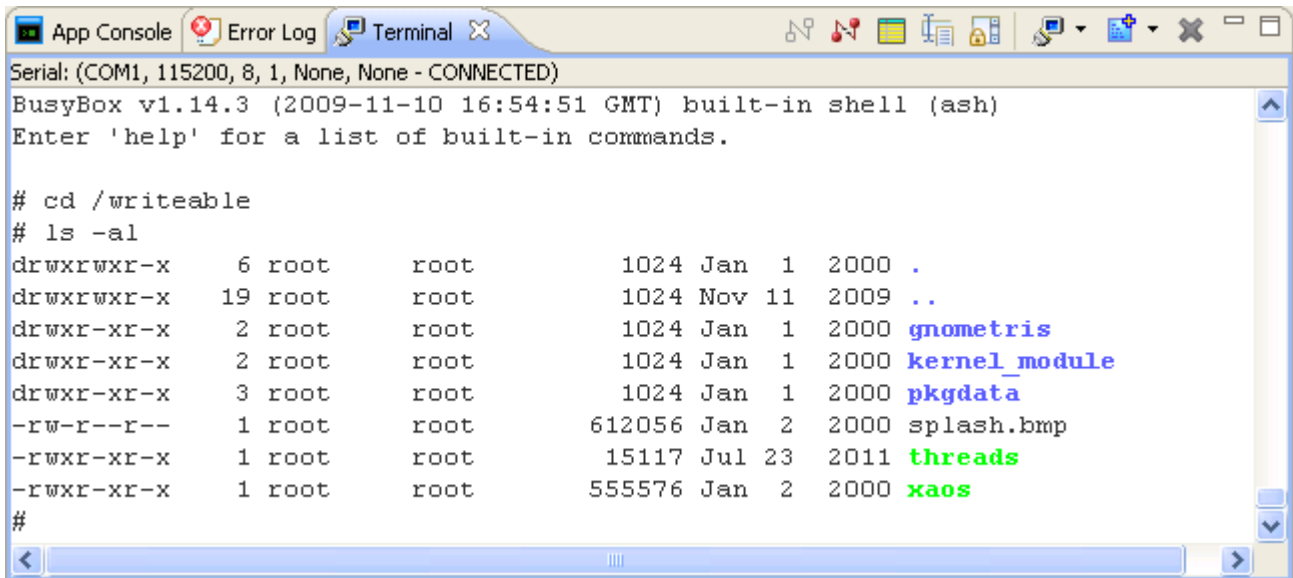


Figure 10-76 Terminal view

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 2-39.](#)

[2.6 Configuring a connection to a Linux Kernel on page 2-45.](#)

10.51 Remote Scratchpad view

Use the **Remote Scratchpad** view as an electronic clipboard. You can copy and paste or drag and drop useful files and folders into it for later use.

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 change it in the original file system.

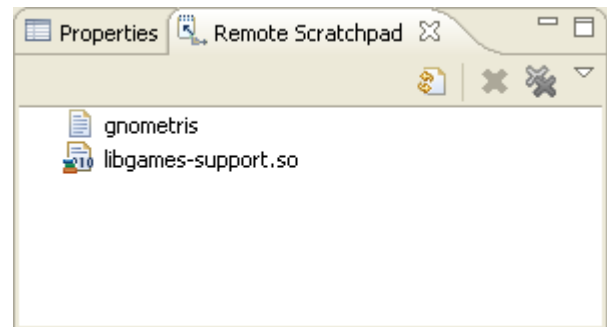


Figure 10-77 Remote Scratchpad

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**.

10.52 Remote Systems terminal for SSH connections

Use the **Remote Systems** terminal 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.

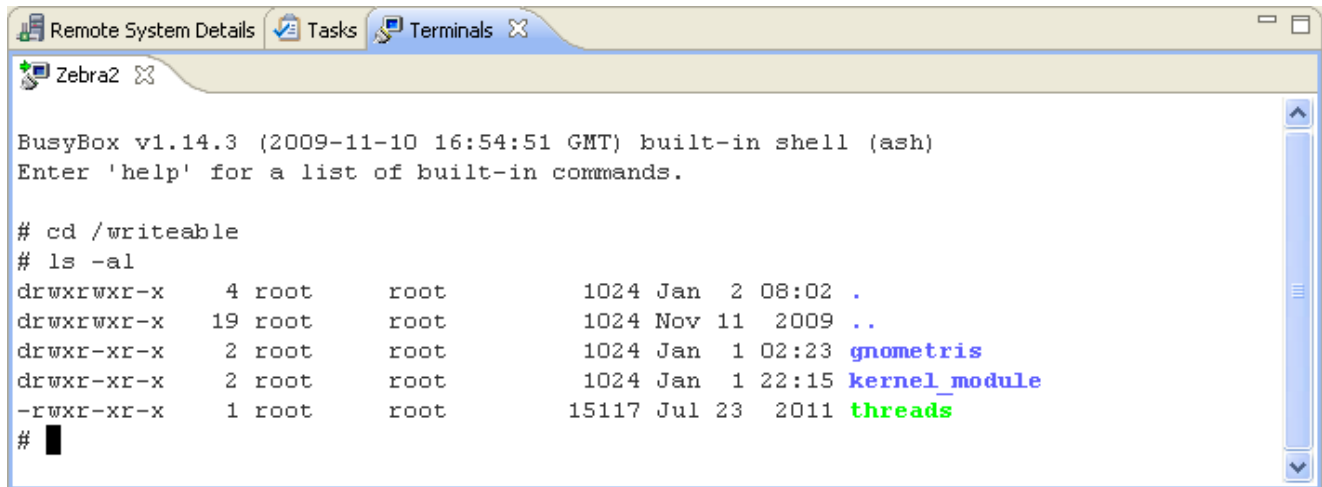


Figure 10-78 Remote Systems terminal

This terminal 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 remote system** and configure a 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.

10.53 Terminal Settings dialog box

Use the **Terminal Settings** dialog box to change the view title, encoding, connection type, and connection settings for the terminal.

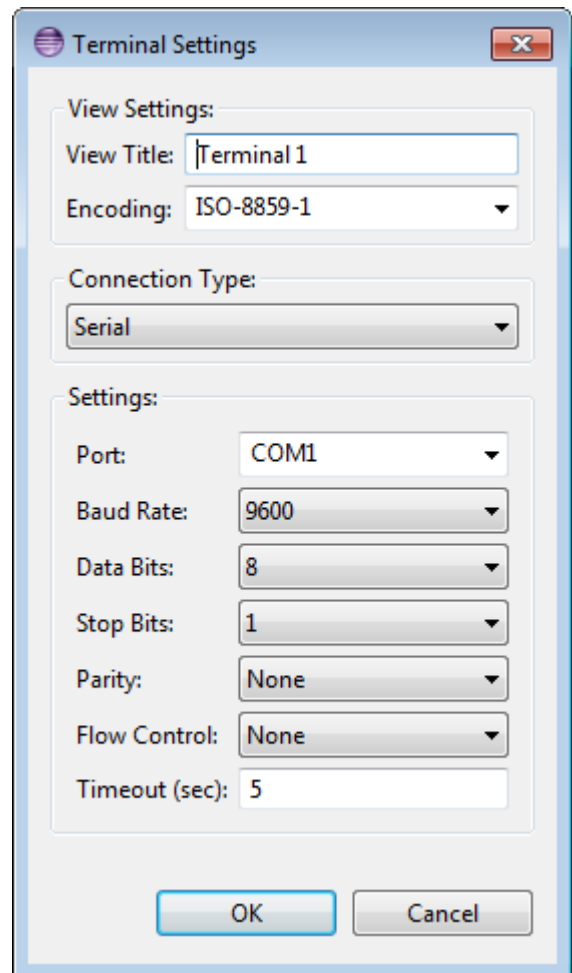


Figure 10-79 Terminal Settings dialog box

View Settings

Enables you to specify the name and encoding for the **Terminal**.

View Title

Enter a name for the **Terminal** view.

Encoding

Select the character set encoding for the terminal.

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 connection 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.

10.54 Debug Hardware Configure IP view

Use the **Debug Hardware Configure IP** view to configure Ethernet and internet protocol settings on the debug hardware units connected to the host workstation.

To access the Debug Hardware Configure IP view from the main menu, select **Window > Show View > Other > DS-5 Debugger > Debug Hardware Configure IP**.

The configuration process depends on the way in which the debug hardware unit is connected to the host computer, and whether or not your network uses Dynamic Host Configuration Protocol (DHCP). If you have connected your debug hardware unit to an Ethernet network or directly to the host computer using an Ethernet cross-over cable, you must configure the network settings before you can use the unit for debugging. You have to configure the network settings only once for each debug hardware unit.

The following connections are possible:

- Your debug hardware unit is connected to your local network that uses DHCP. In this situation, you do not have to know the Ethernet address of the unit, but you must enable DHCP.
- Your debug hardware unit is connected to your local network that does not use DHCP. In this situation, you must assign a static IP address to the debug hardware unit.

Note

If you want to connect to a debug hardware unit on a separate network, you must know the IP address of that unit.

Figure 10-80 Debug Hardware Configure IP view

Debug Hardware Configure IP view options

Ethernet/MAC Address

The Ethernet address/media access control (MAC) address of the debug hardware unit.

The address is automatically detected when you click **Browse** and select the hardware. To enter the value manually, select the **Configure New** option.

Browse

Click to display the Connection Browser dialog. Use it to browse and select the debug hardware unit in your local network or one that is connected to a USB port on the host workstation.

Identify

Click to visually identify your debug hardware unit using the indicators available on the debug hardware. On RVI™, all LEDs on the front panel flash during identification. On DSTREAM, the DSTREAM logo flashes during identification.

Restart

Click to restart the selected debug hardware unit.

Configure New

Select this option to manually configure a debug hardware unit that was not previously configured or is on a different subnet.

Ethernet Type

Select the type of Ethernet you are connecting to. **Auto-Detect** is the default option. The other available options are: **10-MBit Half Duplex**, **10-MBit Full Duplex**, **100-MBit Half Duplex**, **100-MBit Full Duplex**.

TCP/IP Settings

Host Name - The name of the debug hardware unit. This must contain only the alphanumeric characters (A to Z, a to z, and 0 to 9) and the - character, and must be no more than 39 characters long.

Get settings using DHCP - Enables or disables the Dynamic Host Configuration Protocol (DHCP) on the debug hardware unit. If using DHCP, you must specify the hostname for your debug hardware unit.

IP Address - The static IP address to use.

Default Gateway - The default gateway to use.

Subnet Mask - The subnet mask to use.

Configure

Click to apply changes to the debug hardware unit.

Related references

[10.55 Debug Hardware Firmware Installer view on page 10-339.](#)

[10.56 Connection Browser dialog box on page 10-341.](#)

10.55 Debug Hardware Firmware Installer view

Use the **Debug Hardware Firmware Installer** view to update the firmware for your debug hardware.

To access the Debug Hardware Firmware Installer view, from the main menu, select **Window > Show View > Other > DS-5 Debugger > Debug Hardware Firmware Installer**.

A debug hardware unit stores templates for each supported device. Each template defines how to communicate with the device and the settings that you can configure for that device. The templates are provided in a firmware file.

ARM periodically releases updates and patches to the firmware that is installed on a debug hardware unit. Each update or patch is released as a firmware file. These updates or patches might extend the capabilities of your debug hardware, or might fix an issue that has become apparent.

In DS-5, the latest firmware files are available at: `install_directory\sw\debughw\firmware\`

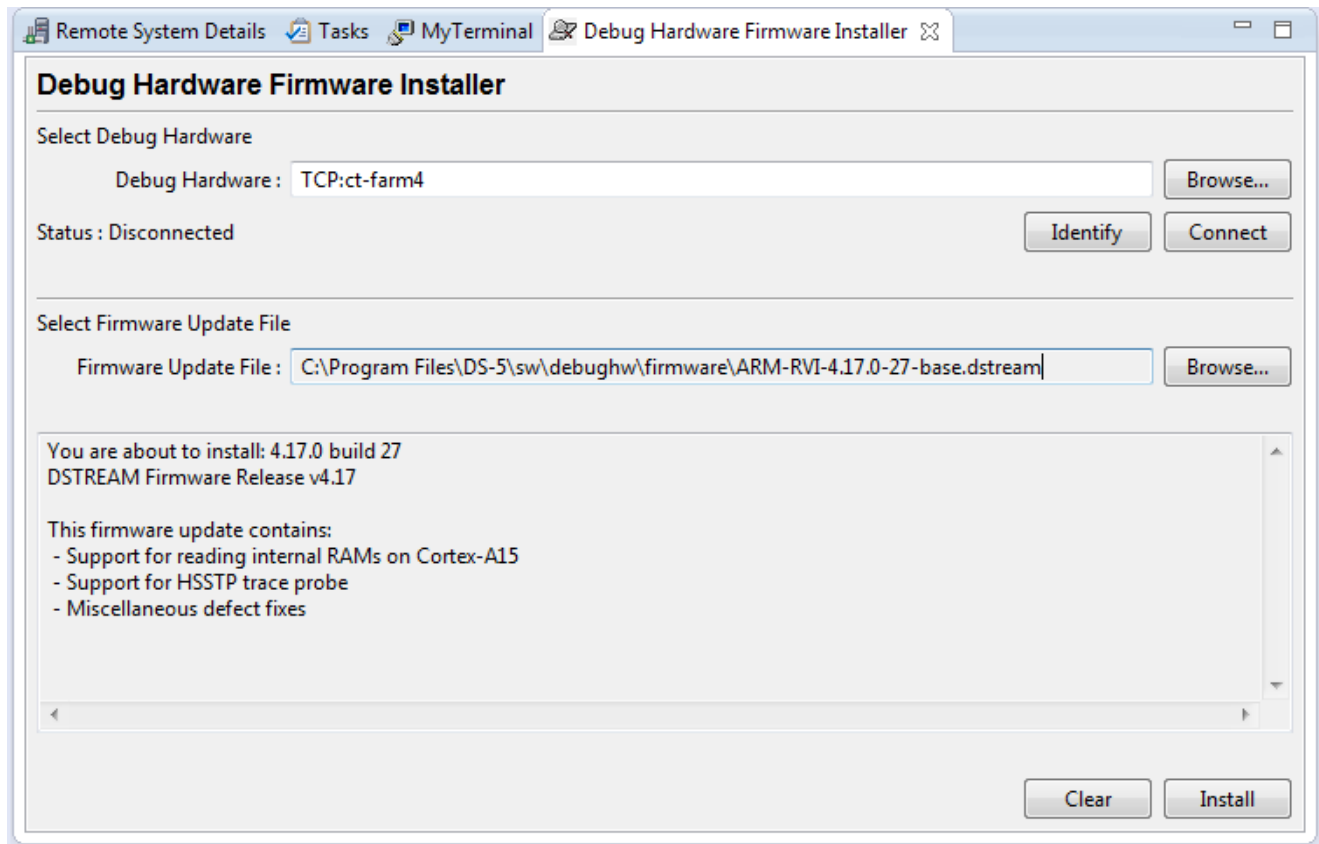


Figure 10-81 Debug Hardware Firmware Installer

Debug Hardware Firmware Installer view options

Select Debug Hardware

The currently selected debug hardware. You can either enter the IP address or host name of the debug hardware unit, or use the **Browse** button and select the debug hardware unit.

Browse

Click to display the Connection Browser dialog. Use it to browse and select the debug hardware unit in your local network or one that is connected to a USB port on the host workstation.

Identify

Click to visually identify your debug hardware unit using the indicators available on the debug hardware. On RVI, all LEDs on the front panel flash during identification. On DSTREAM, the DSTREAM logo flashes during identification.

Connect

Click to connect to your debug hardware. Once connected, the dialog shows the current firmware status.

Select Firmware Update File

Use the **Browse** button and select the firmware file. Once the file is selected, the dialog shows the selected firmware update file details.

Browse

Click to browse and select the firmware file.

Clear

Click to clear the currently selected debug hardware and firmware file.

Install

Click to install the firmware file on the selected debug hardware.

Firmware file format

Firmware files have the following syntax: `ARM-RVI-N.n.p-bld-type.unit`

`N.n.p`

Is the version of the firmware. For example, `4.5.0` is the first release of firmware version `4.5`.

`build`

Is a build number.

`Type`

Is either:

`base`

The first release of the firmware for version `N.n`.

`patch`

Updates to the corresponding `N.n` release of the firmware.

`unit`

Identifies the debug hardware unit, and is one of:

`dstream`

for a DSTREAM debug and trace unit.

`rvi`

for an RVI debug unit.

Related references

[10.54 Debug Hardware Configure IP view on page 10-337.](#)

[10.56 Connection Browser dialog box on page 10-341.](#)

[10.46.3 Updating multiple debug hardware units on page 10-314.](#)

10.56 Connection Browser dialog box

Use the **Connection Browser** dialog box to browse for and select a debug hardware unit in your local network or one that is connected to a USB port on the host workstation. When the **Connection Browser** dialog box finds a unit, it is added to the list of available units.

To view the **Connection Browser** dialog, click **Browse** from the **Debug Hardware Configure IP** or **Debug Hardware Firmware Installer** views.

To connect to the debug hardware, select the hardware from the list, and click **Select**.

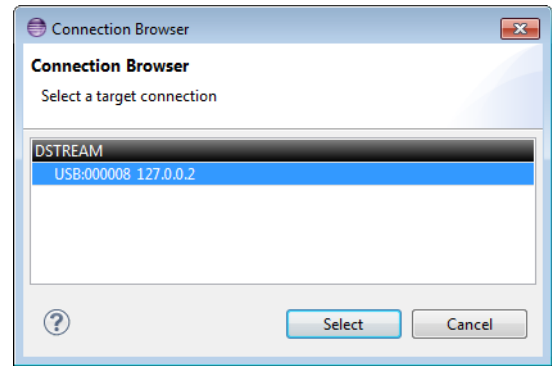


Figure 10-82 Connection Browser (Showing a USB connected DSTREAM)

Note

- If debug hardware units do not appear in the list, check your network and setup of your debug unit.
- Debug hardware units connected to different networks do not appear in the **Connection Browser** dialog box. If you want to connect to a debug hardware unit on a separate network, you must know the IP address of that unit.
- Any unit shown in light gray has responded to browse requests but does not have a valid IP address. You cannot connect to that unit by TCP/IP until you have configured it for use on your network.
- Only appropriate debug hardware units are shown. VSTREAM connections are not shown.

Related references

[10.54 Debug Hardware Configure IP view on page 10-337.](#)

[10.55 Debug Hardware Firmware Installer view on page 10-339.](#)

10.57 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
	Reverse step over		Reverse step out
	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 sections:

- [*11.1 ARM Linux problems and solutions*](#) on page 11-346.
- [*11.2 Enabling internal logging from the debugger*](#) on page 11-347.
- [*11.3 Target connection problems and solutions*](#) on page 11-348.

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 2-39.](#)

[2.6 Configuring a connection to a Linux Kernel on page 2-45.](#)

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 10-214.](#)

[7.5 Controlling the output of logging messages on page 7-174.](#)

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 2-39.](#)

[2.6 Configuring a connection to a Linux Kernel on page 2-45.](#)

Chapter 12

File-based Flash Programming in ARM DS-5

Describes the file-based flash programming options available in DS-5.

It contains the following sections:

- *12.1 About file-based flash programming in ARM® DS-5 on page 12-350.*
- *12.2 Flash programming configuration on page 12-352.*
- *12.3 Creating an extension database for flash programming on page 12-354.*
- *12.4 About using or extending the supplied ARM® Keil® flash method on page 12-355.*
- *12.5 About creating a new flash method on page 12-357.*
- *12.6 About testing the flash configuration on page 12-361.*
- *12.7 About flash method parameters on page 12-362.*
- *12.8 About getting data to the flash algorithm on page 12-363.*
- *12.9 About interacting with the target on page 12-364.*

12.1 About file-based flash programming in ARM® 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 ARM 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 possible 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

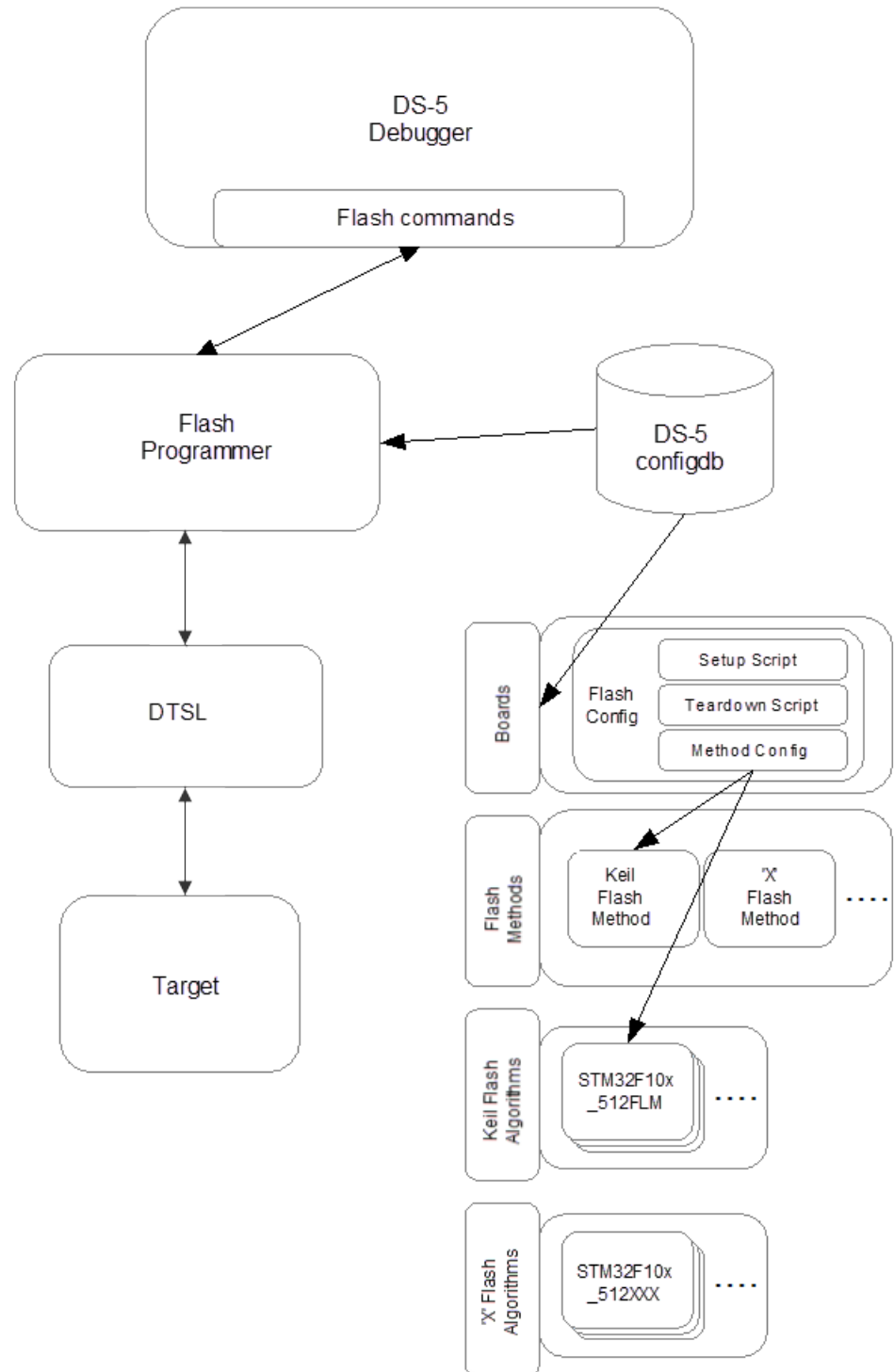


Figure 12-1 DS-5 File Flash Architecture

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>
    <!-- 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 12-354.](#)

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 ARM® Keil® flash method

DS-5 Debugger contains a full implementation of the Keil® flash programming method. This might be used to program any flash device supported by the Keil MDK product. It might 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 µVision® 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.

This section contains the following subsections:

- [12.4.1 Adding flash support to an existing platform using an existing Keil flash algorithm on page 12-355.](#)
- [12.4.2 Adding flash support to an existing target platform using a new Keil flash algorithm on page 12-356.](#)

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 must be the name of the core on the target that runs 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_ARMv7.zip` file within the `DS-5/examples` directory. Extract it to your file system and then import the `DS-5Examples\flash_algo-STM32F10x` 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 12-355.

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 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.

This section contains the following subsections:

- [12.5.1 About using the default implementation `FlashMethodv1` on page 12-357.](#)
- [12.5.2 About creating the flash method Python script on page 12-358.](#)

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>\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:x1="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 schema. 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 canceled. 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

Describes the OS awareness feature available in DS-5.

It contains the following sections:

- *13.1 About Writing operating system awareness for DS-5 Debugger* on page 13-372.
- *13.2 Creating an OS awareness extension* on page 13-373.
- *13.3 Implementing the OS awareness API* on page 13-377.
- *13.4 Enabling the OS awareness* on page 13-379.
- *13.5 Implementing thread awareness* on page 13-382.
- *13.6 Implementing data views* on page 13-385.
- *13.7 Programming advice and noteworthy information* on page 13-388.

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.

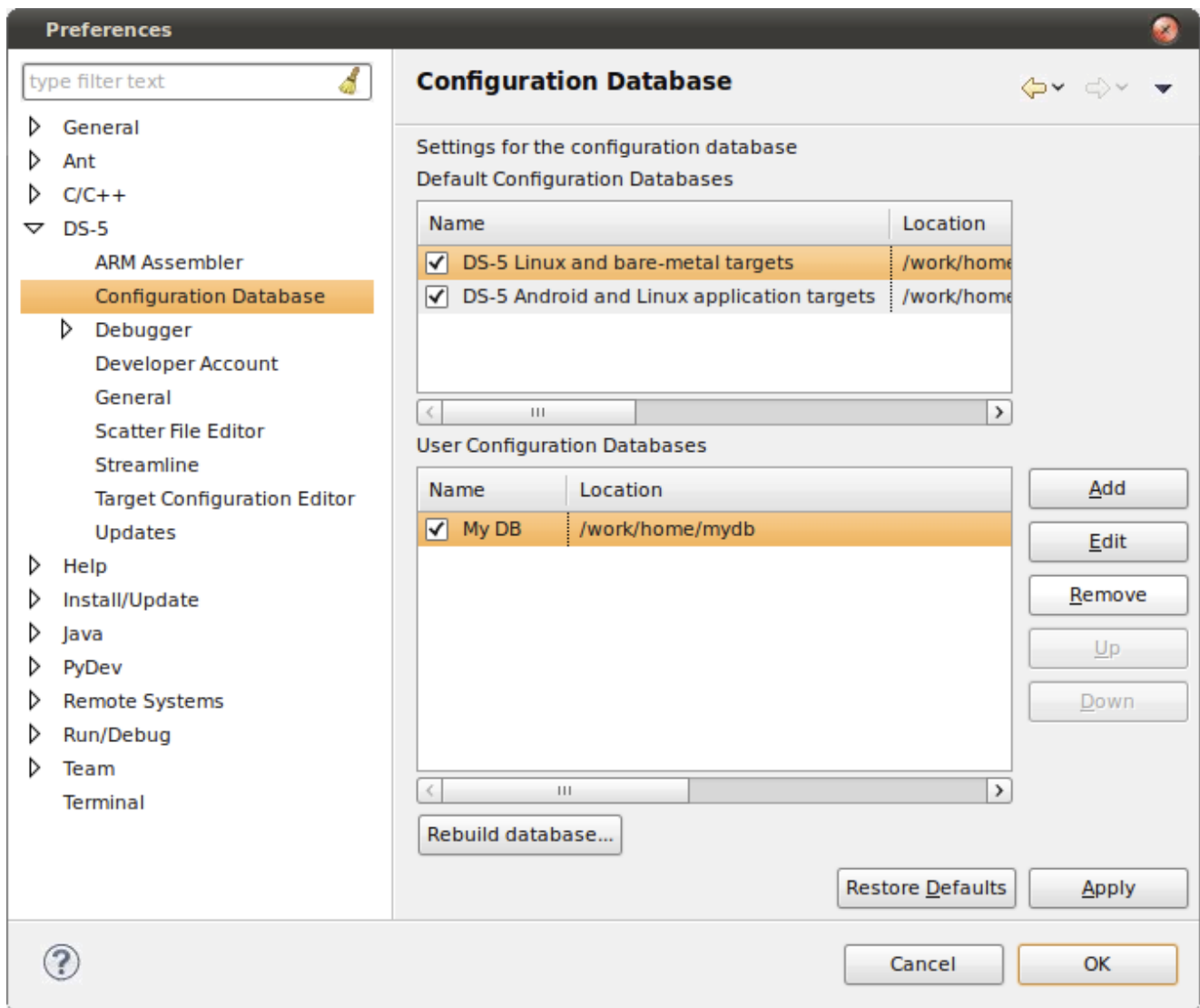


Figure 13-1 Eclipse preferences for mydb

Note

The some folder entry is represented by /work/home in Figure 1.

- 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>
```

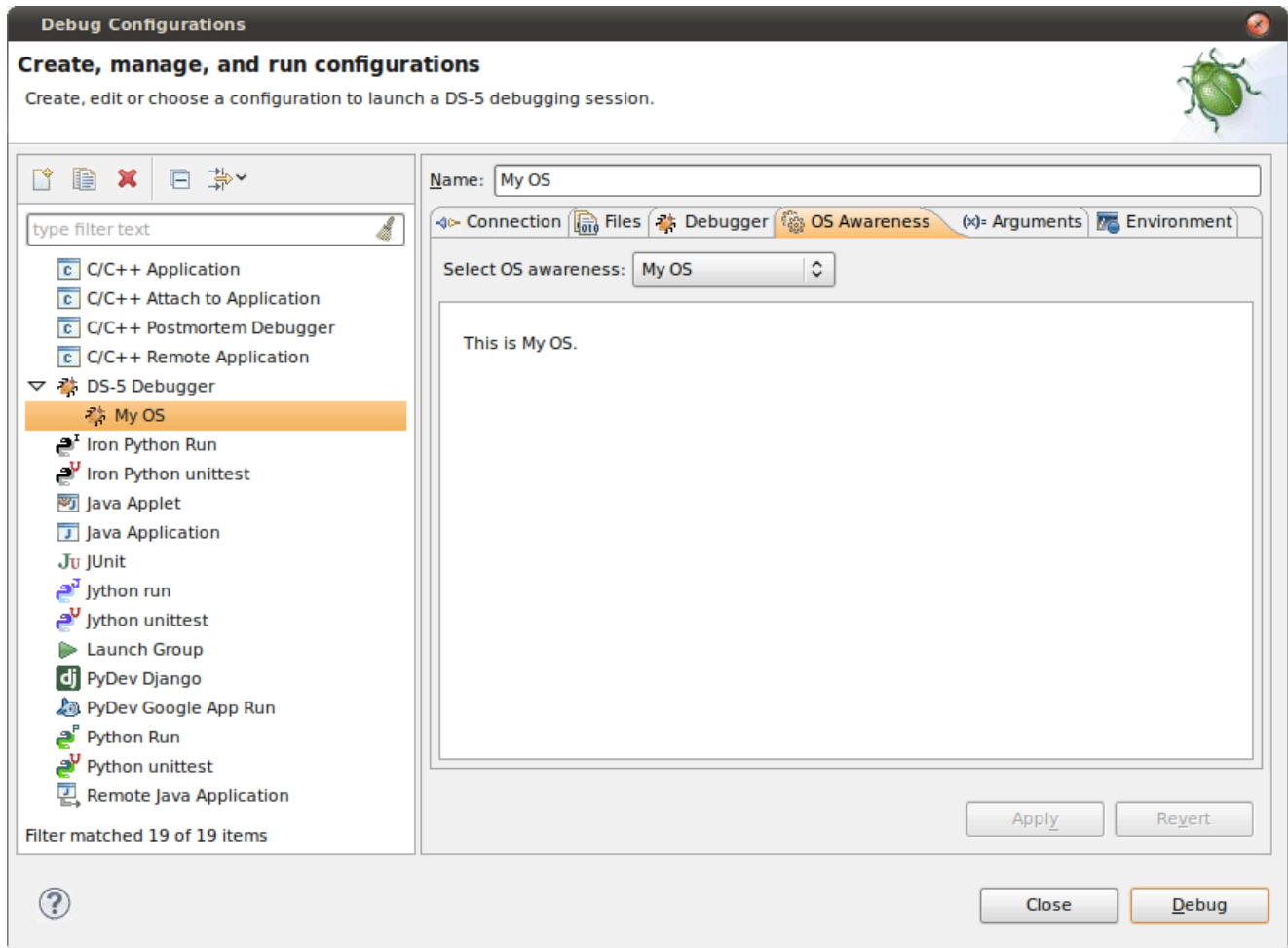
Note

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. A debug configuration with this OS awareness selected works, although this does 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**.

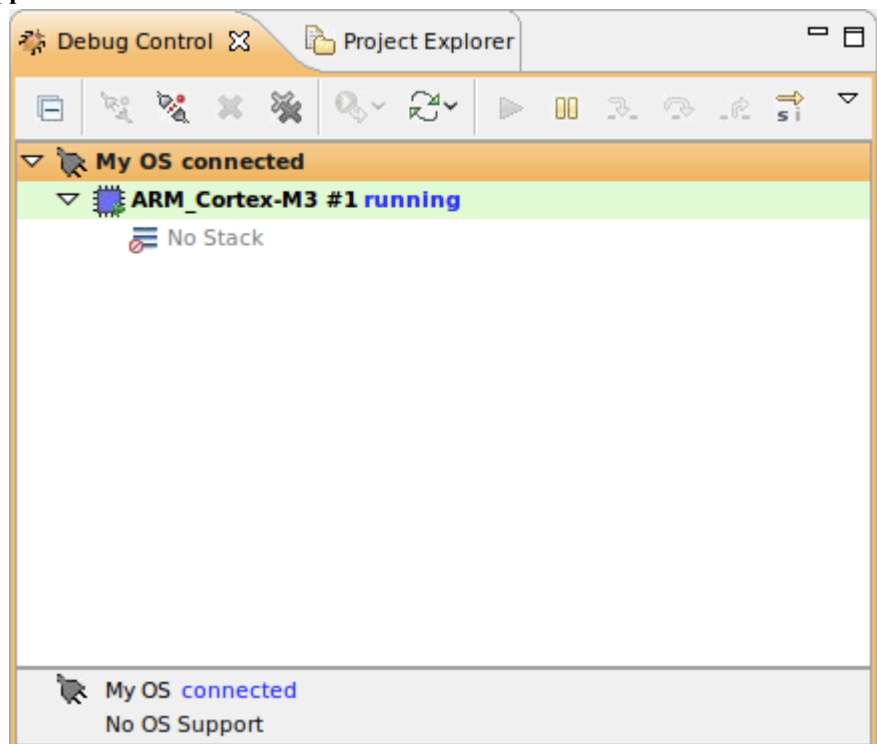


Figure 13-3 myos No OS Support

- 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()`.

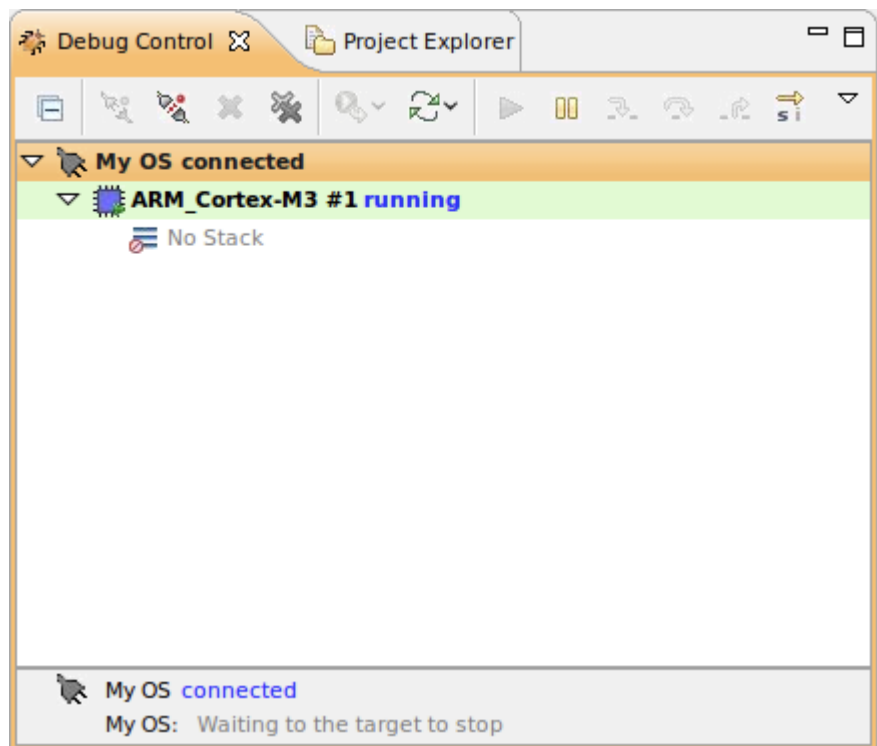


Figure 13-4 myos waiting for target to stop

- 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.

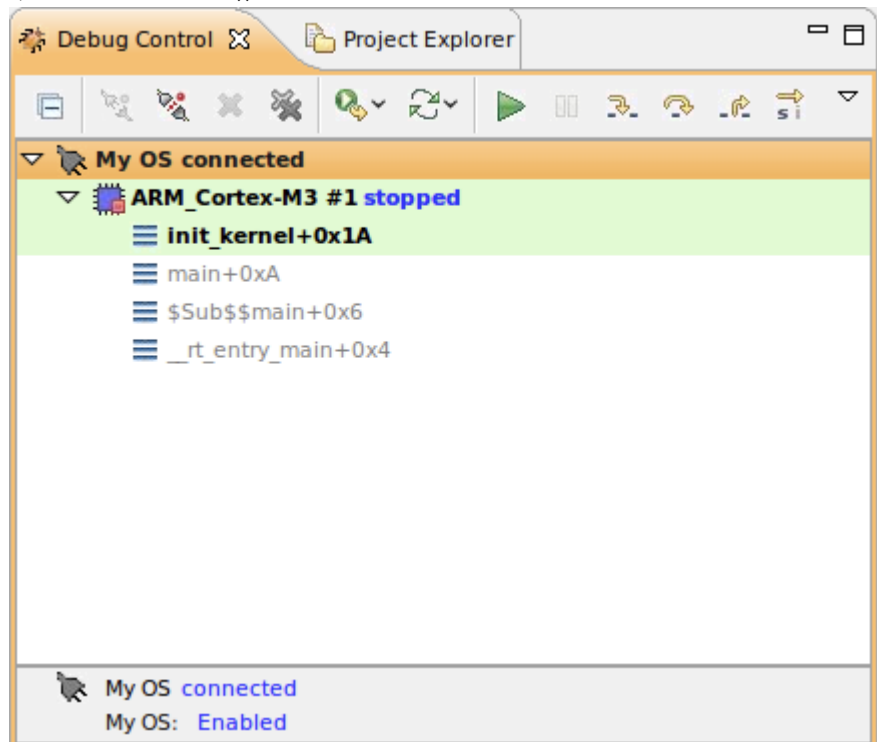


Figure 13-5 myos Enabled

- 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 initialized as `scheduler_running` is not set to 1 yet, but symbols are loaded:

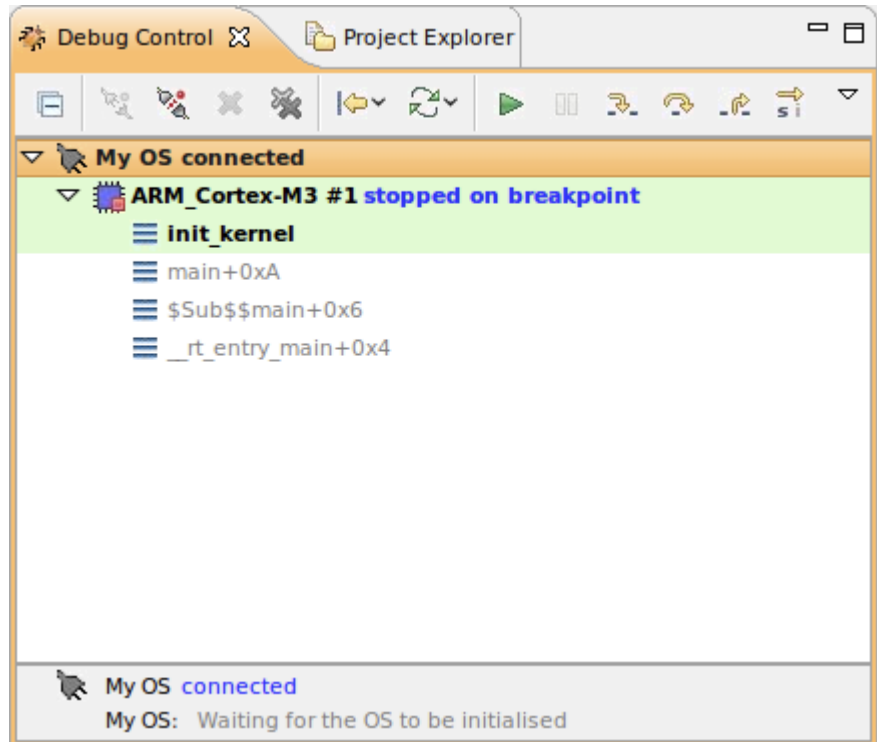


Figure 13-6 myos waiting for OS initialization

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 getAllOSContexts(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:

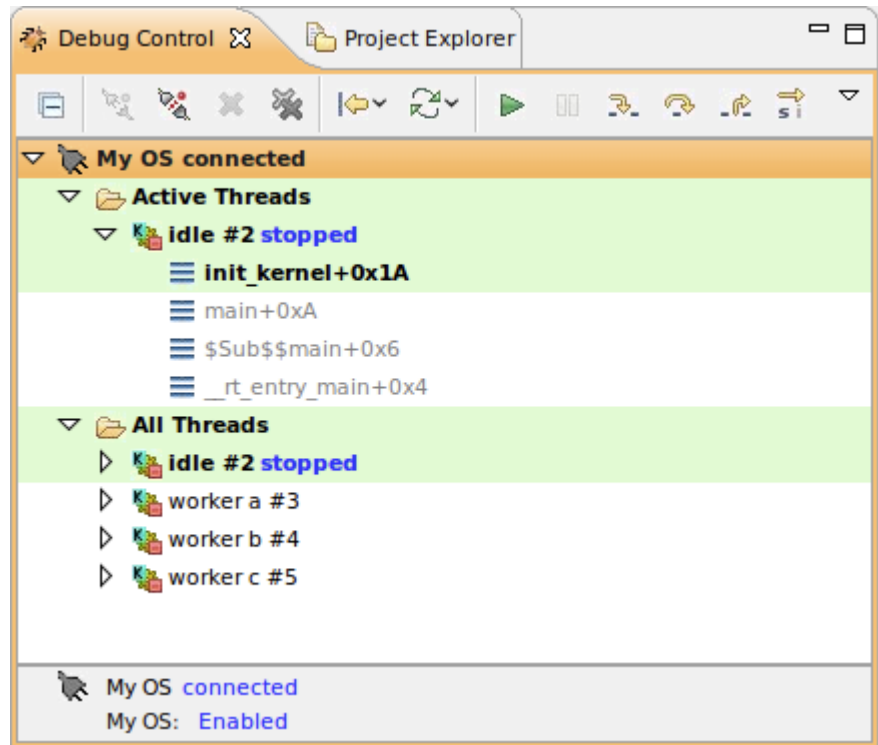


Figure 13-7 myos Debug Control view data

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:

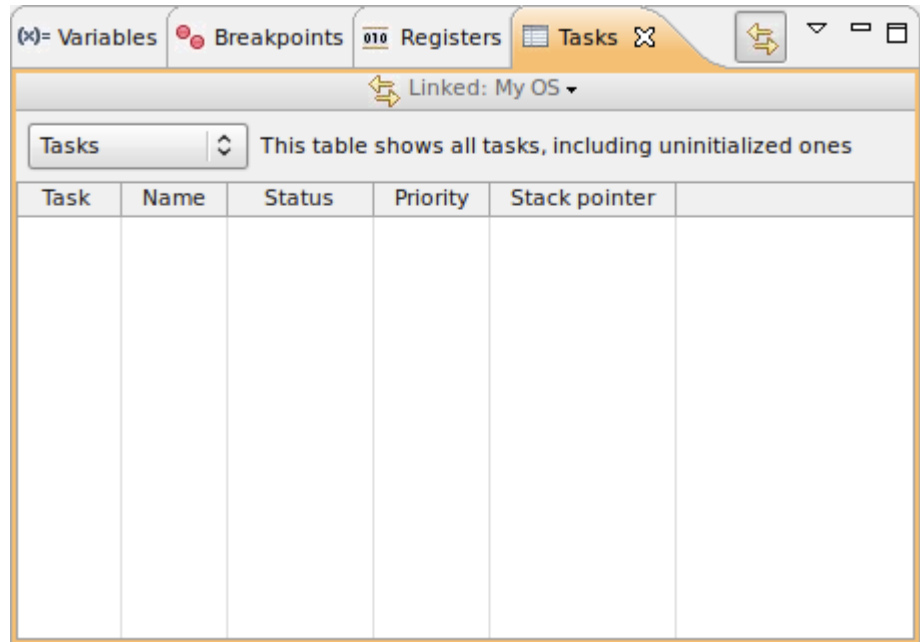


Figure 13-8 myos Empty Tasks table

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:

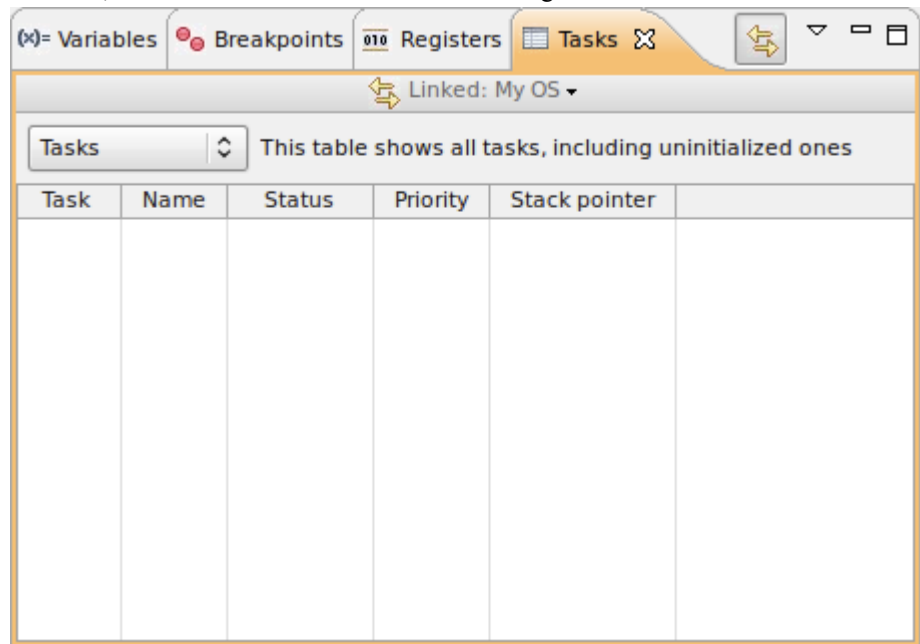


Figure 13-9 myos populated Tasks table

- 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.

Chapter 14

Debug and Trace Services Layer (DTSL)

Describes the DS-5 Debugger *Debug and Trace Services Layer* (DTSL).

DTSL is a software layer that sits between the debugger and the RDDI target access API. DS-5 Debugger uses DTSL to:

- Create target connections.
- Configure the target platform to be ready for debug operations.
- Communicate with the debug components on the target.

As a power user of DS-5 Debugger, you might need to use DTSL:

- As part of new platform support.
- To extend the capabilities of DS-5 Debugger.
- To add support for custom debug components.
- To create your own Java or Jython programs which interact with your target.

It contains the following sections:

- [14.1 Additional DTSL documentation and files on page 14-390.](#)
- [14.2 Need for DTSL on page 14-391.](#)
- [14.3 DS-5 configuration database on page 14-396.](#)
- [14.4 DTSL as used by DS-5 Debugger on page 14-402.](#)
- [14.5 Main DTSL classes and hierarchy on page 14-404.](#)
- [14.6 DTSL options on page 14-412.](#)
- [14.7 DTSL support for SMP and AMP configurations on page 14-418.](#)
- [14.8 DTSL Trace on page 14-422.](#)
- [14.9 Extending the DTSL object model on page 14-429.](#)
- [14.10 Debugging DTSL Jython code within DS-5 Debugger on page 14-434.](#)
- [14.11 DTSL in stand-alone mode on page 14-438.](#)

14.1 Additional DTSL documentation and files

Additional DTSL documents and files are provided in <DS-5 Install folder>\sw\DTSL.

The following documents are useful for understanding DTSL. Make sure you have access to them.

DTSL object level documentation

DTSL is mainly written in Java, so the documentation takes the form of Javadoc files. The DTSL Javadoc is provided both as a PDF (DTSLJavaDocs.pdf) and as HTML files (inside com.arm.debug.dtsl.docs.zip). You can view the HTML files directly in a browser, or use them from within the Eclipse environment.

Certain classes in the DTSL Javadocs are marked as Deprecated. These classes must not be used in new DTSL code. They are only provided in the documentation in case you encounter them when inspecting older DTSL code.

RDDI API documentation

DTSL is designed to use RDDI-DEBUG as its native target connection API. Some of the RDDI-DEBUG API is therefore referred to from within DTSL. For completeness, the RDDI documentation is included with the DTSL documentation.

The RDDI documentation is provided in HTML format in <DS-5 Install folder>\sw\debugger\RDDI\docs\html. To access the documentation, open index.html.

Also, make sure you have access to DTSLExamples.zip. This contains example DTSL code, in addition to the DS-5 configdb entries discussed in this document. This document assumes that you have added the examples to your Eclipse installation by importing the Eclipse projects contained in this file.

Related information

[ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition.](#)

[ARM Debug Interface Architecture Specification.](#)

14.2 Need for DTSL

DTSL addresses the growing complexity and customization of ARM-based SoCs using the CoreSight Architecture. Before the creation of DTSL, most debug tools were designed at a time when SoC debug architecture was much simpler. SoCs typically contained only one core, and if multiple cores were used, they were of different types and were accessed by dedicated debug channels. Debug tools designed during that time, including ARM debuggers, cannot easily be scaled to more modern and complex debug architectures. DTSL is therefore designed to address several problems which older debug tools cannot easily address.

This section contains the following subsections:

- [14.2.1 SoC design complexity](#) on page 14-391.
- [14.2.2 Debug flexibility](#) on page 14-392.
- [14.2.3 Integrated tool solutions](#) on page 14-392.
- [14.2.4 DS-5 Debugger architecture before DTSL](#) on page 14-392.
- [14.2.5 DS-5 Debugger architecture after DTSL](#) on page 14-393.
- [14.2.6 DS-5 Debugger connection sequence showing where DTSL fits in](#) on page 14-395.

14.2.1 SoC design complexity

The debugger must be able to handle complex SoC designs which contain many cores and many debug components. For example, the following figure shows a relatively simple SoC design containing many debug components:

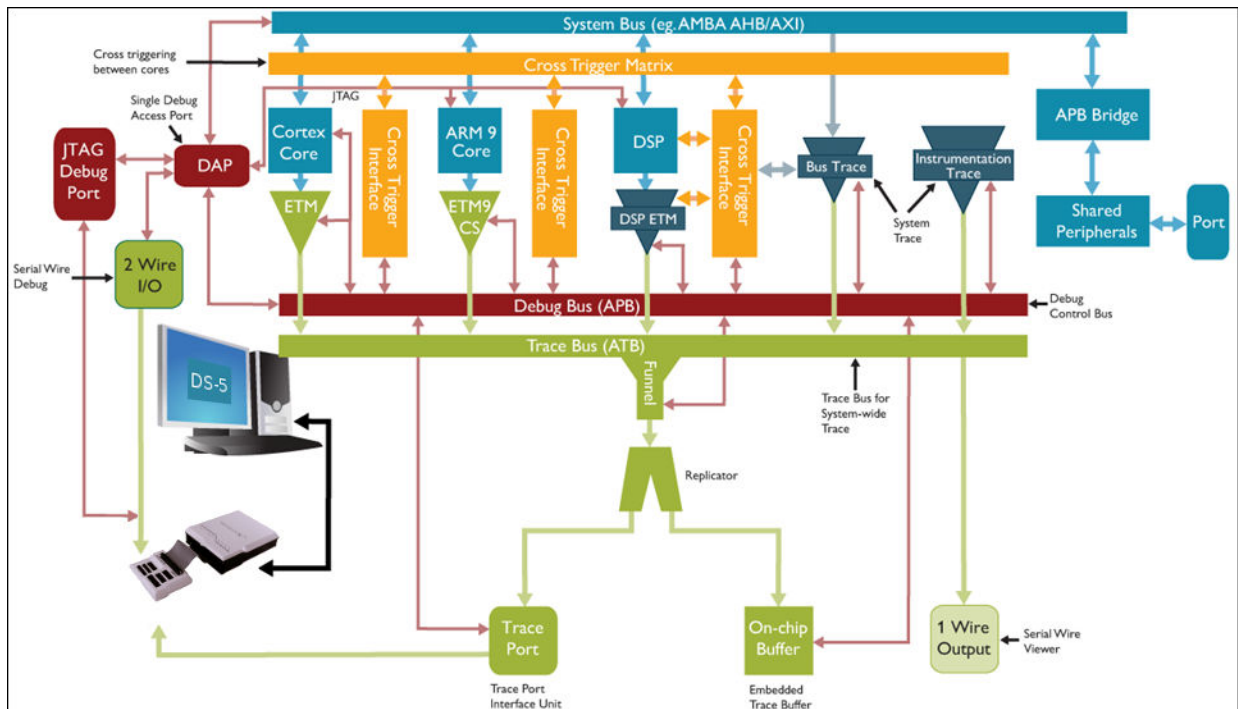


Figure 14-1 A simple CoreSight Design

Such systems are continuing to become more complicated as time goes on. For example, SoC designers might want to use multiple sub-systems which are accessed through multiple DAPs, but which are linked by multiple *Cross Trigger Interfaces* (CTIs) so that they can still be synchronized. Each sub-system would have a similar design to that shown in the figure, but with shared CTIs and possibly shared TPIU.

Because system designs are so complicated, and vary so greatly, DTSL is designed to provide a layer of abstraction between the details of a particular system and the tools which provide debugging functionality to the user. For example, a debug tool using DTSL knows that there is a source of trace data for a particular core, and can access that data, but does not have to handle the complexities of system configuration and tool set-up in order to get that data. It does not have to know how to, for example, program up CoreSight Funnels, collect trace data from a DSTREAM, or demultiplex the TPIU trace protocol.

14.2.2 Debug flexibility

DTSL is designed to address the problems associated with the following:

- SoC designers sometimes add their own components, which are not part of any ARM standard. Debug tools might need to interact with these components to access the target.
- CoreSight designs can be very flexible, and early implementations might have design issues that the debug tool needs to work around.
- CoreSight designs can contain components which can be interconnected in many ways.

14.2.3 Integrated tool solutions

CoreSight designs can contain shared debug resources which need to be managed and used from multiple tools. For example, the system might be able to generate trace from several trace sources, such as ARM cores + DSP. In legacy designs, the trace paths would be independent and each debug tool would have its own connection to the respective sub-system. In a typical CoreSight system, the trace data is merged by a Funnel component and delivered to a single trace storage device through a single interface. The trace data is then uploaded and de-multiplexed. The trace data might need to be delivered to several different debug tools, such as ARM DS-5 and DSP Debug Tool.

DTSL addresses the tool integration problem that this situation raises.

14.2.4 DS-5 Debugger architecture before DTSL

Before DTSL first became available, the early DS-5 Debugger Software stack was as shown in the following figure:

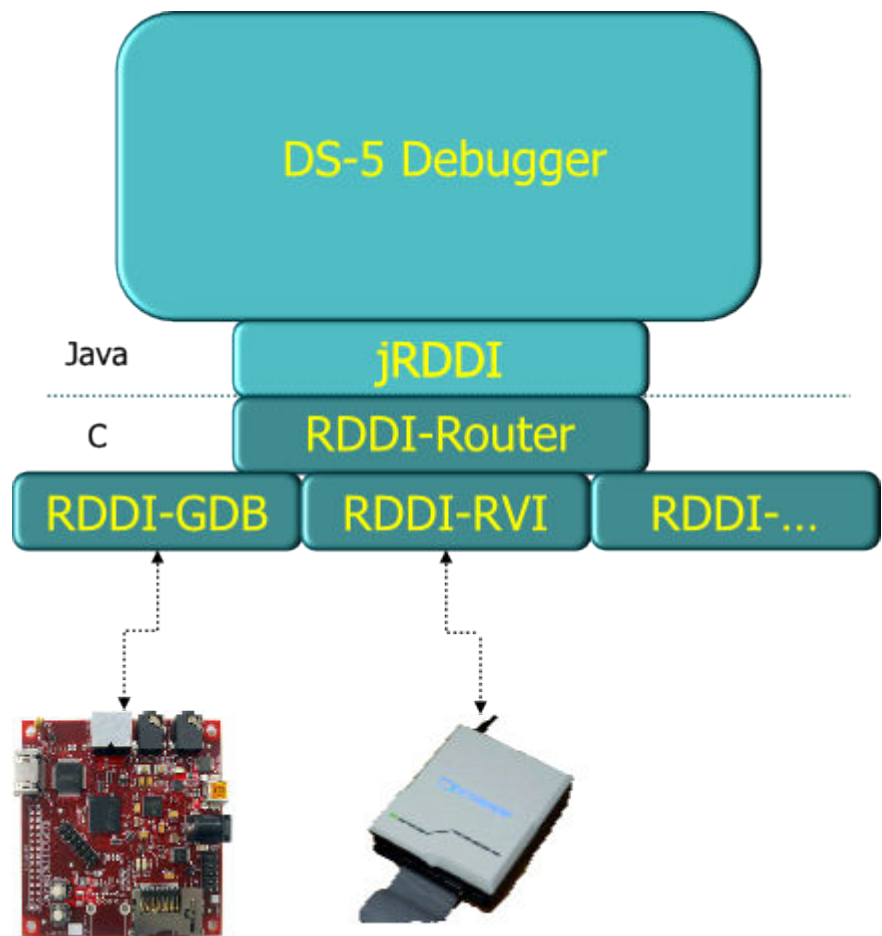


Figure 14-2 Initial DS-5 Debugger SW Stack

From the bottom upwards, the components of the debug stack are:

RDDI-DEBUG API

The debugger uses this API as its standard native connection to a debug controller such as DSTREAM/RVI, CADI Model, or gdbserver. There is an implementation of RDDI-DEBUG for each of the supported types of debug controller.

RDDI-Router API

This API is identical to RDDI-DEBUG, but it is used to 'vector' the API calls to the appropriate implementation. This is necessary because the debugger can support multiple connections and connection types simultaneously.

jRDDI

This is a Java wrapper for the C RDDI-DEBUG API. It is not a true Java layer, but nominally it is the lowest Java layer in the stack.

14.2.5 DS-5 Debugger architecture after DTSL

After DTSL was introduced, the DS-5 Debugger Software stack changed. It is now as shown in the following figure:

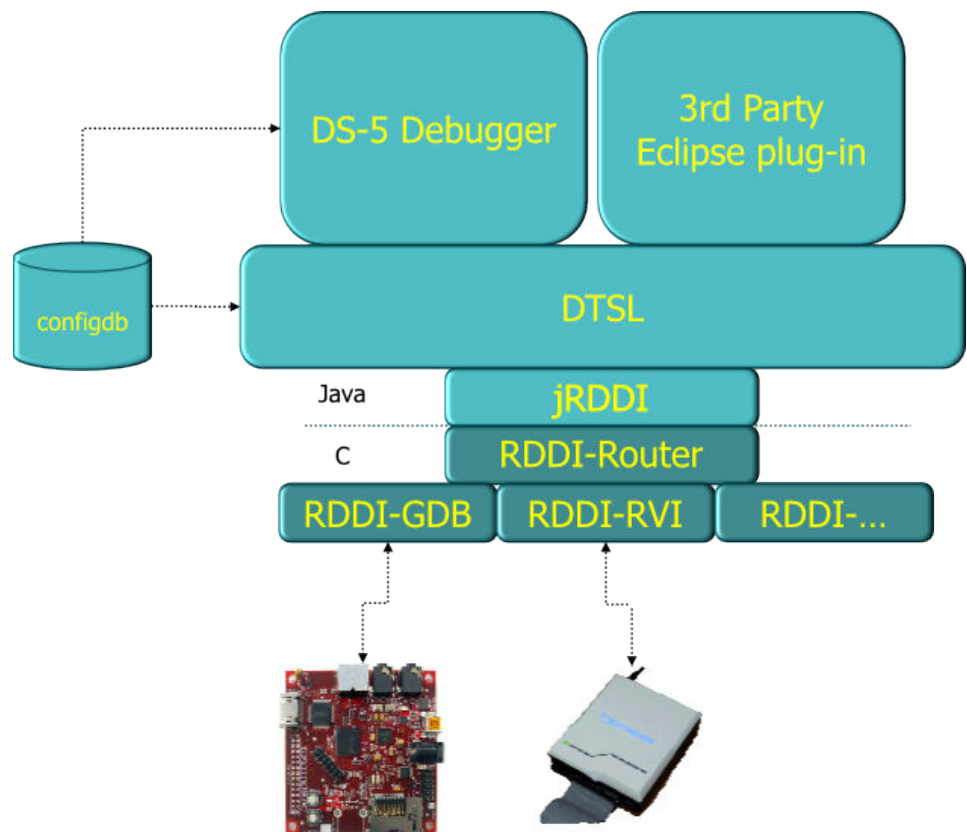


Figure 14-3 Post DTSL DS-5 Debugger SW Stack

In addition to the layers that existed before DTSL, the stack now contains a DTSL layer which does the following:

- Handles system initialization and DTSL-level component creation. This is controlled by DTSL Jython scripts, which are typically contained in a platform configuration database (configdb).

Note

Do not confuse DTSL Jython Scripting with DS-5 Debugger Jython Scripting. Both of them use Jython, but they operate at different levels in the software stack. However, a Debugger Jython Script can use DTSL functionality.

- Provides a common connection interface for several client programs.
- Delivers trace streams to several trace clients.
- Uses the existing native RDDI infrastructure.

The DS-5 Debugger uses DTSL to communicate with the lower layers of the stack. DTSL provides a set of named objects for the debugger (or another tool) to use. The object set consists of the following:

- Debug objects, which control core execution. Their interface looks very similar to the jRDDI and RDDI-DEBUG interfaces.
- Trace source interfaces, which represent target components which can generate trace data.
- Trace capture interfaces, which are used to start and stop trace collection and to provide notification events to clients.
- Other target components, such as other CoreSight devices or other third-party target devices.
- A Configuration and Connection interface, which instantiates and configures the DTSL objects and queries the configuration to allow clients to discover which top level interfaces are present.

Related concepts

[14.4.3 DTSL access from Debugger Jython scripts on page 14-403.](#)

14.2.6 DS-5 Debugger connection sequence showing where DTSL fits in

The sequence below outlines the DS-5 Debugger connection sequence when connecting to a target, and where DTSL fits in.

1. The user creates an Eclipse launch configuration by selecting a platform (board) and a debug operation from the DS-5 `configdb`. The user also specifies other debugger configuration parameters such as which file (`.axf`) to debug.
2. The user activates a launch configuration, either from the launch configuration editor or by selecting a previously prepared launch configuration.
3. The debugger launcher code locates the platform (board) entry in the DS-5 Debugger `configdb` and locates the DTSL configuration script. This script is run, and it creates the DTSL configuration.
4. The debugger connects to the DTSL configuration created by the script. It locates, by name, the object or objects identified by the debug operation specified in the `configdb` platform entry. It uses these objects to access the target device, including access to memory, registers, and execution state.

14.3 DS-5 configuration database

All use cases of DTSL potentially require the use of the DS-5 configuration database (`configdb`). ARM therefore recommends that you have a basic understanding of the configuration database.

DS-5 Debugger uses a configuration database, called `configdb`, to store information on how to connect to platforms. This information is split across several top-level locations in the database, each of which can contain the following:

Board information

Manufacturer, name, list of SoCs, list of Flash, DTSL initialization Jython scripts.

SoC information

Core information. For example, a SoC may contain Cortex-A9 + Cortex-M3.

Core information

Register sets for the core, and other information such as TrustZone support.

Flash information

Information on flash types and programming algorithms.

Common scripts (Jython)

Jython scripts which might be of use to several database entries.

This information is mainly stored as XML files, located in sub-directories of the top-level locations.

The configuration database is located at `<DS-5 install folder>\sw\debugger\configdb`.

DS-5 allows you to configure one or more extension `configdb` locations, which are typically used to add more board definitions or flash support to DS-5.

This section contains the following subsections:

- [14.3.1 Modifying DS-5 configdb on page 14-396.](#)
- [14.3.2 Configdb board files on page 14-397.](#)
- [14.3.3 About project_types.xml on page 14-397.](#)
- [14.3.4 About the keil-mcbstm32e.py script on page 14-398.](#)
- [14.3.5 DTSL script on page 14-400.](#)

14.3.1 Modifying DS-5 configdb

The DS-5 `configdb` is usually installed into a read-only location, to prevent accidental modification of the installed files. However, DS-5 allows the user to install `configdb` extensions, which can be in a writeable location. The `configdb` extensions can also override the entries in the installed directories. To modify an installed `configdb` board entry (directory), you need to copy the installed entry into your Documents folder or home directory, modify it, and tell DS-5 to add it as a `configdb` extension.

For example, to modify the Keil MCBSTM32E platform files:

1. Create a `configdb` directory in your Documents folder or in another writeable location.
2. Create a `Boards` directory inside the `configdb` directory.
3. Copy the `Keil/MSCTM32E` directory into the `Boards` directory.
4. Modify the copied `configdb` files.
5. Tell DS-5 about the new `configdb` extension. To do this:
 - a. Select **Window > Preferences**.
 - b. Expand the **DS-5** entry in the Preferences window.
 - c. Select the **Configuration Database** entry.
 - d. Click the **Add...** button to add the location of the new configuration database to the **User Configuration Databases** list.

Note

If you modify any of the XML files in a configdb directory, you must tell DS-5 to rebuild the database by clicking the **Rebuild database...** button on the DS-5 Configuration Database preferences panel.

14.3.2 Configdb board files

Within the configdb is the Boards directory. It contains one sub-directory for each board manufacturer, and a boards.xml file which optionally provides human-readable display names for boards.

For example, the Keil MCBSTM32E platform is a simple STM32E (Cortex-M3) MCU board. The main configdb files are located in <DS-5 install folder>sw\debugger\configdb\Boards\Keil\MCBSTM32E, and are as follows:

project_types.xml

This is the main XML file which describes the platform entry to DS-5.

keil-mcbstm32e.py

This is the DTSL platform configuration and setup script, implemented in Jython.

keil-mcbstm32e.rvc

This is the DSTREAM RDDI configuration file for the platform. The DS-5 Debug Hardware Configuration utility usually creates this file.

keil-mcbstm32e_flash.xml

This contains information on flash devices and algorithms, and their configuration parameters.

14.3.3 About project_types.xml

The project_types.xml file defines the project types supported for the platform. Debug operations and activities, which refer to the other files in the platform directory, are defined for each project type.

The following code is part of the project_types.xml file for the Keil MCBSTM32E platform.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!--Copyright (C) 2009-2013 ARM Limited. All rights reserved.-->
<platform_data xmlns="http://www.arm.com/project_type" xmlns:peripheral="http://
com.arm.targetconfigurationeditor" xmlns:xsi="http://www.w3.org/2001/XMLSchema"
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://keil-mcbstm32e_flash.xml</flash_config>
  <project_type_list>
    <project_type type="BARE_METAL">
      <name language="en">Bare Metal Debug</name>
      <description language="en">This allows a bare-metal debug connection.</description>
      <execution_environment id="BARE_METAL">
        <name language="en">Bare Metal Debug</name>
        <description language="en">This allows a bare-metal debug connection.</description>
        <param default="CDB://mcbstm32e.rvc" id="config_file" type="string" visible="false"/>
        <param default="CDB://mcbstm32e.py" id="dtsl_config_script" type="string"
          visible="false"/>
        <xsi:include href="../../Include/hardware_address.xml"/>
        <activity id="ICE_DEBUG" type="Debug">
          <name language="en">Debug Cortex-M3</name>
          <xsi:include href="../../Include/ulinkpro_activity_description.xml"/>
          <xsi:include href="../../Include/ulinkpro_connection_type.xml"/>
          <core connection_id="Cortex-M3" core_ref="Cortex-M3" soc="st/stm32f103xx"/>
          <param default="DebugOnly" id="dtsl_config" type="string" visible="false"/>
          <xsi:include href="../../Include/ulinkpro_browse_script.xml"/>
          <xsi:include href="../../Include/ulinkpro_setup_script.xml"/>
        </activity>
        <activity id="ICE_DEBUG" type="Debug">
          <name language="en">Debug Cortex-M3</name>
          <xsi:include href="../../Include/dstream_activity_description_bm.xml"/>
          <xsi:include href="../../Include/dstream_connection_type.xml"/>
          <core connection_id="Cortex-M3" core_ref="Cortex-M3" soc="st/stm32f103xx"/>
          <param default="DSTREAMDebugAndTrace" id="dtsl_config" type="string" visible="false"/>
          <param default="options.traceBuffer.traceCaptureDevice" id="dtsl_tracecapture_option"
            type="string" visible="false"/>
        </activity>
      </project_type>
  </project_type_list>
</platform_data>
```

```

<param default="ITM" id="eventviewer_tracesource" type="string" visible="false"/>
</activity>
</execution_environment>
</project_type>
</project_type_list>
</platform_data>

```

The XML file declares a BARE_METAL project type. BARE_METAL is a term which describes a system not running an OS, where the debug connection takes full control of the core. The file declares an execution environment within the project type, and declares debug activities within that execution environment. The code here shows only one debug activity, but each execution environment can declare several debug activities. The debug activity shown here is a debug and trace session using a DSTREAM target connection.

When DS-5 displays the debug session launcher dialog, it scans the entire configdb and builds a list of supported manufacturers and boards, and the supported project types and debug activities, and lets the user choose which one they want to use. In the following example, the user is assumed to have chosen the highlighted debug activity. When DS-5 Debugger launches the debug session, it creates a DTSL configuration and passes it the {*config_file*, *dtsl_config_script*, *dtsl_config*} property set. These parameters are used as follows:

config_file

This value is passed to the RDDI-DEBUG connection DLL or so (DS-5 uses RDDI-DEBUG as its target connection interface, and RDDI-DEBUG needs this file to tell it which devices are in the target system).

dtsl_config_script

This value tells DTSL which Jython script to use to create the DTSL configuration used by the debugger.

dtsl_config

The DTSL Jython script can contain several system configurations, defined by Jython class names which in turn are derived from the DTSLv1 object. This value tells DTSL which class to create. The MCBSTM32E Jython script contains two such classes, one for a debug and trace configuration and one for a debug-only configuration. The class name used for the highlighted example is *DSTREAMDebugAndTrace*, so in this example a Jython class named *DSTREAMDebugAndTrace* must exist in the *dtsl_config_script*.

Some of these entries have a file location prefix of CDB://. This indicates that the location is within the platform directory in the configuration database.

DTSL creates an instance of the referenced Jython class, which causes the `def __init__(self, root)` constructor to be run. After this constructor is run, the debugger expects to find a DTSL Device object whose name is the same as the name given in the core setting in the debug activity. In this example, therefore, the debugger expects to find a DTSL object named "Cortex-M3", and it directs all debug operation requests to this object.

14.3.4 About the keil-mcbstm32e.py script

The complete content of the *keil-mcbstm32e.py* file for the Keil MCBSTM32E platform is included in the *DTSLExampleConfigdb* project in *DTSLExamples.zip*. The important aspects of the script are as follows:

- The script is written in Jython. Jython is an implementation of the Python language which integrates tightly with Java. The integration is tight enough to allow the following:
 - A Jython script can contain Python classes which Java can use, and which appear to Java as though they were Java classes.
 - A Jython script can contain Python classes which can sub-class Java classes.
 - A Jython script can create Java class instances and use them.

This is why the script contains some `import` statements which import Java classes. Many of these classes are from the `com.arm.debug.dtsl` package.

- DTSL creates an instance of a class named *DSTREAMDebugAndTrace*.

- The constructor `__init__` creates all the DTSL objects required for the connection.
- The RDDI-DEBUG API, which is the native API used by the debugger for target access, assigns each device a unique device index number. The script contains lines which find the index number for a named device and assign that number to a variable. The following is an example of such a line:

```
devID = self.findDevice("Cortex-M3")
```

This line assigns the RDDI device index number for the named device “Cortex-M3” to the variable `devID`.

- The script creates a *ResetHookedDevice* object, derived from *Device*, with the name “Cortex-M3”. This is an example of how Jython can extend the standard DTSL Java classes by sub-classing them.
- The script creates an *AHBCortexMMemAPAccessor* and installs it into the Cortex-M3 object as a memory filter. This is how a custom named memory space is added to the core. When a memory access is requested with an address prefixed by “AHB”, the access is redirected to the *AHBCortexMMemAPAccessor* object which, in this case, uses the CoreSight AHB-AP to access the memory.
- The script creates DTSL objects for the CoreSight components in the SoC.
- The script creates a *DSTREAMTraceCapture* object, which the debugger uses to read trace data.
- The script declares a set of options which provide user configuration data for the script. The Eclipse debug session launcher panel displays these options so that they can be set before making a target connection. After the constructor is called, DTSL passes the option values to the class by calling its *optionValuesChanged()* method.

```
from com.arm.debug.dtsl.configurations import DTSLv1

[snip]
class ResetHookedDevice(Device):
    def __init__(self, root, devNo, name):
        Device.__init__(self, root, devNo, name)
        self.parent = root
    def systemReset(self, resetType):
        Device.systemReset(self, resetType)
        # Notify root configuration
        self.parent.postReset()

class DSTREAMDebugAndTrace(DTSLv1):
    '''A top level configuration class which supports debug and trace'''

    @staticmethod
    def getOptionList():
        '''The method which specifies the configuration options which
        the user can edit via the launcher panel |Edit...| button
        ...
        return [
            DTSLv1.tabSet(
                name='options',
                displayName='Options',
                childOptions=[
                    DSTREAMDebugAndTrace.getTraceBufferOptionsPage(),
                    DSTREAMDebugAndTrace.getETMOptionsPage(),
                    DSTREAMDebugAndTrace.getITMOptionsPage()
                ]
            )
        ]

    @staticmethod
    def getTraceBufferOptionsPage():
        # If you change the position or name of the traceCapture
        # device option you MUST modify the project_types.xml to
        # tell the debugger about the new location/name
        return DTSLv1.tabPage(
            name='traceBuffer',
            displayName='Trace Buffer',
            childOptions=[
                DTSLv1.enumOption(
                    name='traceCaptureDevice',
                    displayName='Trace capture method',
                    defaultValue='DSTREAM',
                    values=[
                        ('none', 'No trace capture device'),
                        ('DSTREAM', 'DSTREAM 4GB Trace Buffer')
                    ]
                ),
                DTSLv1.booleanOption(
```

```

        name='clearTraceOnConnect',
        displayName='Clear Trace Buffer on connect',
        defaultValue=True
    ),
    DTSLv1.booleanOption(
        name='startTraceOnConnect',
        displayName='Start Trace Buffer on connect',
        defaultValue=True
    ),
    DTSLv1.enumOption(
        name='traceWrapMode',
        displayName='Trace full action',
        defaultValue='wrap',
        values=[
            ('wrap', 'Trace wraps on full and continues to store data'),
            ('stop', 'Trace halts on full')
        ]
    )
]
)
]
)

[snip]
def __init__(self, root):
    '''The class constructor'''
    # base class construction
    DTSLv1.__init__(self, root)
    # create the devices in the platform
    self.cores = []
    self.traceSources = []
    self.reservedATBIDs = {}
    self.createDevices()
    self.setupDSTREAMTrace()
    for core in self.cores:
        self.addDeviceInterface(core)

    def createDevices(self):
# create MEMAP
        devID = self.findDevice("CSMEMAP")
        self.AHB = CortexM_AHBAP(self, devID, "CSMEMAP")
        # create core
        devID = self.findDevice("Cortex-M3")
        self.cortexM3 = ResetHookedDevice(self, devID, "Cortex-M3")
        self.cortexM3.registerAddressFilters(
            [AHBCortexMMemAPAccessor("AHB", self.AHB, "AHB bus accessed via AP_0")])
        self.cores.append(self.cortexM3)
        # create the ETM disabled by default - will enable with option
        devID = self.findDevice("CSETM")
        self.ETM = V7M_ETMTraceSource(self, devID, 1, "ETM")
        self.ETM.setEnabled(False)
        self.traceSources.append(self.ETM)
        # ITM disabled by default - will enable with option
        devID = self.findDevice("CSITM")
        self.ITM = V7M_ITMTraceSource(self, devID, 2, "ITM")
        #self.ITM = M3_ITM(self, devID, 2, "ITM")
        self.ITM.setEnabled(False)
        self.traceSources.append(self.ITM)
        # TPIU
        devID = self.findDevice("CSTPIU")
        self.TPIU = V7M_CSTPIU(self, devID, "TPIU", self.AHB)
        # DSTREAM
        self.DSTREAM = DSTREAMTraceCapture(self, "DSTREAM")
        self.DSTREAM.setTraceMode(DSTREAMTraceCapture.TraceMode.Continuous)

```

14.3.5 DTSL script

The DTSL script defines the DTSL options using a set of static methods. The option definitions must be available before creating an instance of the configuration class.

To display and modify the DTSL options before connecting, use the Eclipse launcher panel. To display and modify the DTSL options during a DS-5 debug session, use the command line or the Debug Control view.

In Windows 7, the DTSL options values are persisted in your workspace under the directory C:\Users\<user>\Documents\DS-5 Workspace\metadata\plugins\com.arm.ds\DTSL. In this directory there is a sub-directory for the platform, in which there is another sub-directory for the debug operation. Within the debug operation directory there are one or more .dtslprops files, whose names match the names option sets in the DTSL Options Dialog. These files are standard Java properties files. The

following is the default properties file for the Keil MCBSTM32E Platform, Bare Metal Project, Debug and Trace Debug operation:

```
options.ETM.cortexM3coreTraceEnabled=true
options.ITM.itmTraceEnabled=true
options.ITM.itmTraceEnabled.itmowner=Target
options.ITM.itmTraceEnabled.itmowner.target.targetITMATBID=2
options.ITM.itmTraceEnabled.itmowner.debugger.DWTENA=true
options.ITM.itmTraceEnabled.itmowner.debugger.PRIVMASK.[15\:8]=true
options.ITM.itmTraceEnabled.itmowner.debugger.PRIVMASK.[23\:16]=true
options.ITM.itmTraceEnabled.itmowner.debugger.PRIVMASK.[31\:24]=true
options.ITM.itmTraceEnabled.itmowner.debugger.PRIVMASK.[7\:0]=true
options.ITM.itmTraceEnabled.itmowner.debugger.STIMENA=0xFFFFFFFF
options.ITM.itmTraceEnabled.itmowner.debugger.TSENA=true
options.ITM.itmTraceEnabled.itmowner.debugger.TSPrescale=none
options.traceBuffer.traceCaptureDevice.clearTraceOnConnect=true
options.traceBuffer.traceCaptureDevice.startTraceOnConnect=true
options.traceBuffer.traceCaptureDevice.traceWrapMode=wrap
options.traceBuffer.traceCaptureDevice=DSTREAM
```

The names of the options exactly match the name hierarchy defined in the DTSL script (see the full DTSL script source code to create the configuration options).

When DS-5 Debugger displays the options, it calls the *getOptionList()* method in the DTSL configuration class to retrieve a data description of the options. It matches these options with the persisted values from the *.dtslprops* file and transforms this data into an interactive dialog type display for the user. When the user saves the options, the *.dtslprops* file is updated. After the DTSL configuration instance is created, DTSL calls the *optionValuesChanged()* method to inform the instance of the configuration settings values. During the debug session, the user can change any option which is marked with an *isDynamic=True* property.

Related references

[14.6 DTSL options on page 14-412.](#)

14.4 DTSL as used by DS-5 Debugger

This section contains the following subsections:

- [14.4.1 Eclipse debug session launcher dialog on page 14-402.](#)
- [14.4.2 Connecting to DTSL on page 14-402.](#)
- [14.4.3 DTSL access from Debugger Jython scripts on page 14-403.](#)

14.4.1 Eclipse debug session launcher dialog

The Launcher dialog scans the entire DS-5 configdb, including extension directories, and builds a list of supported manufacturers and boards, along with the supported project types and debug activities, as declared in the `project_types.xml` files in each Board directory. This list is displayed to the user, who then chooses the combination of manufacturer, board, project type and debug activity they want to use. When the user chooses a `{Manufacturer, Board, Project Type, Debug Operation}`, the launcher inspects the referenced `{dtsl_config_script, dtsl_config}`, that is, the script and the class, to see if any DTSL options are specified. If so, the Launcher displays an **Edit...** button so that the user can change the values for the DTSL options.

14.4.2 Connecting to DTSL

To use DTSL, a client must create a *DTSLConnection* object using the *DTSL ConnectionManager* class (`com.arm.debug.dtsl.ConnectionManager`). *ConnectionManager* has static methods that allow the *DTSLConnection* object to be created from a set of connection parameters. After a *ConnectionManager* object is obtained, calling its `connect()` method creates the *DTSLConfiguration* object which contains all the target component objects.

When the *DTSL ConnectionManager* class creates a new *DTSLConnection*, it assigns a unique key to it. It constructs this key from the connection properties:

- `dtsl_config_script`: the absolute path to the DTSL Jython script
- `dtsl_config`: the Jython DTSL class name
- `config_file`: the absolute path to the RDDI configuration file
- `dtsl_config_options`: optional DTSL options (a hash of the content of the DTSL options file)
- `rddi_retarget_address`: optional re-target address for the RDDI configuration
- possibly other input.

If the *DTSL ConnectionManager* detects an attempt to connect to an already existing *DTSLConnection* (that is, the connection key matches an existing *DTSLConnection* instance) then DTSL returns the already existing instance. There can only be one *DTSLConnection* with any given key.

A *DTSLConnection* can also be created by obtaining an existing DTSL instance key and requesting a connection to that instance. Both DS-5 Debugger and third-party Eclipse plugins can therefore connect to an existing *DTSLConnection* instance. If the DS-5 Debugger creates the *DTSLConnection* instance for a platform, then a third-party plugin can connect to the same instance by one of two methods:

- Use an identical set of connection properties.
- Arrange to get the *DTSLConnection* instance key from the debugger, and use that key to make the connection.

DTSL reference-counts connections to a platform instance and only closes the *DTSLConnection* instance when all clients have disconnected.

14.4.3 DTSL access from Debugger Jython scripts

DTSL uses Jython scripting to create the DTSL configuration. The configuration typically stores objects for each debug component in the target system.

DS-5 Debugger also uses Jython scripting, but at a different level, to DTSL, in the debugger software stack. In debugger scripting, the debugger provides an object interface to the debugger features. For example, a debugger script can:

- load .axf files
- determine the current execution context
- read registers
- set breakpoints
- control execution.

These operations cause operations on the objects in the DTSL configuration, but there is not always a direct mapping from debugger operations to DTSL object operations. This is especially true for SMP systems.

Sometimes, however, it makes sense for a debugger script to access low level DTSL objects. For example, a user with in-depth CoreSight experience might want to manually program up a PTM sequencer, or directly drive CTI inputs. In such cases, the debugger script can get the DTSL configuration, locate the objects of interest and call their methods directly. Although this is a very powerful feature, it must be used with care, because the debugger has no way of knowing that such operations have taken place. In many cases this does not matter, especially if the DTSL objects being used are not directly used by the debugger. However, more care is required when directly accessing core objects used by the debugger.

The following is an example of how a debugger Jython script might get access to a DTSL object called “PTM”:

```
from arm_ds.debugger_v1 import Debugger
from com.arm.debug.dtsl import ConnectionManager
from com.arm.debug.dtsl.interfaces import IConfiguration

# Connect to DS-5 Debugger
debugger = Debugger()
assert isinstance(debugger, Debugger)
if not debugger.isConnected():
    return
# Access the underlying DTSL configuration
dtslConnectionConfigurationKey = debugger.getConnectionConfigurationKey()
dtslConnection = ConnectionManager.openConnection(dtslConnectionConfigurationKey)
dtslConfiguration = dtslConnection.getConfiguration()
assert isinstance(dtslConfiguration, IConfiguration)
deviceList = dtslConfiguration.getDevices()
for device in deviceList:
    assert isinstance(device, IDevice)
    if device.getName() == "PTM":
        ...
```

14.5 Main DTSL classes and hierarchy

There are four basic types of object that DTSL exposes to the Debugger or third-party plugin:

- Connection and Configuration objects, which implement the *IConnection* and *IConfiguration* interfaces respectively.
- Device objects, which implement the *IDevice* interface. Cores, and most CoreSight components, are of this type. If a device needs a connection type operation, which most devices do, then it also implements *IDeviceConnection* (see the *ConnectableDevice* object).
- TraceSource objects, which typically implement both the *IDevice* and *ITraceSource* interfaces. ETM and PTM objects are of this type.
- Trace capture devices, which typically implement the *ITraceCapture* interface. These objects give access to a trace capture device such as a DSTREAM or an ETB.

This section contains the following subsections:

- [14.5.1 DTSL configuration objects](#) on page 14-404.
- [14.5.2 DTSL device objects](#) on page 14-405.
- [14.5.3 CoreSight device component register IDs](#) on page 14-406.
- [14.5.4 DTSL trace source objects](#) on page 14-406.
- [14.5.5 DTSL trace capture objects](#) on page 14-408.
- [14.5.6 Memory as seen by a core device](#) on page 14-409.
- [14.5.7 Physical memory access via CoreSight](#) on page 14-409.
- [14.5.8 DTSL MEM-AP support](#) on page 14-410.
- [14.5.9 Linking MEM-AP access to a core device](#) on page 14-410.

14.5.1 DTSL configuration objects

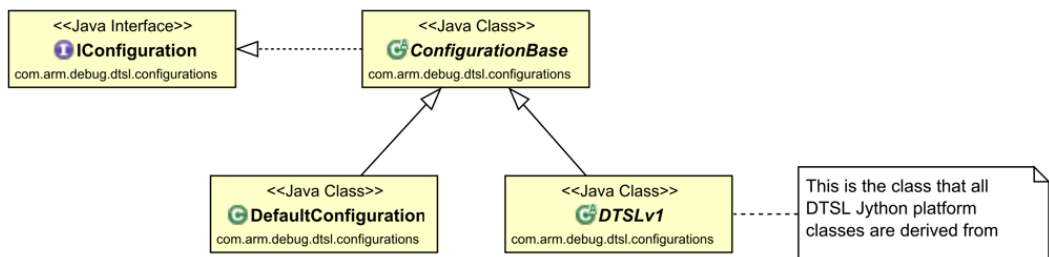


Figure 14-4 DTSL Configuration class hierarchy

The *DTSLConnection* object is the top level DTSL object that allows access to all the other DTSL objects using the platform configuration. Specifically, the *DTSLConnection* allows access to the *ConfigurationBase* instance, for example *DTSLv1*, which allows access to the rest of the DTSL objects. The content of the platform configuration depends on the associated *ConnectionParameters* set.

If the *ConnectionParameters* instance does not specify a DTSL configuration script, then an object of type *DefaultConfiguration* is created. The configuration content is constructed by creating a *Device* object for each device known to RDDI-DEBUG. For DSTREAM, this means that a *Device* object is created for each device declared in the *.rvc* file, but for other kinds of RDDI this might come from a different data set. This allows for a simple connection to a platform with direct connections to any target devices specified in the RDDI configuration file.

If the *ConnectionParameters* instance does specify a DTSL configuration script, then that script is run to create an instance of a configuration object derived from *DTSLv1*. When the configuration script is

run, it is expected to populate the configuration with the set of known device objects, trace sources and trace capture devices.

Note

- ARM recommends using a configuration script to create a DTSL configuration, because it allows much greater flexibility when creating devices.
- DTSLv1 is named as such to show that the configuration is using the V1 interface and object set. This is the current set. If ARM changes the interface and object set, then it might start using DTSLv2. This allows ARM to maintain backwards compatibility, but also to move forward with new or modified interfaces.

14.5.2 DTSL device objects

Device objects are used to interface to any target component that has an RDDI-DEBUG interface. Such components are typically cores or CoreSight devices. All *Device* objects implement the *IDevice* interface, which closely matches the RDDI-DEBUG native interface.

The following is a code sequence from a DTSL Jython script to create the *Device* object for a Cortex-A8 core:

```
1. devID = self.findDevice("Cortex-A8")
2. self.cortexA8 = ConnectableDevice(self, devID, "Cortex-A8")
3. self.addDeviceInterface(self.cortexA8)
```

Line 1 locates the device ID (RDDI-DEBUG device index number) for the named device from the RDDI configuration. Line 2 creates the DTSL *ConnectableDevice* object. Line 3 adds the device object to the DTSL configuration.

The following figure shows part of the Device class hierarchy:

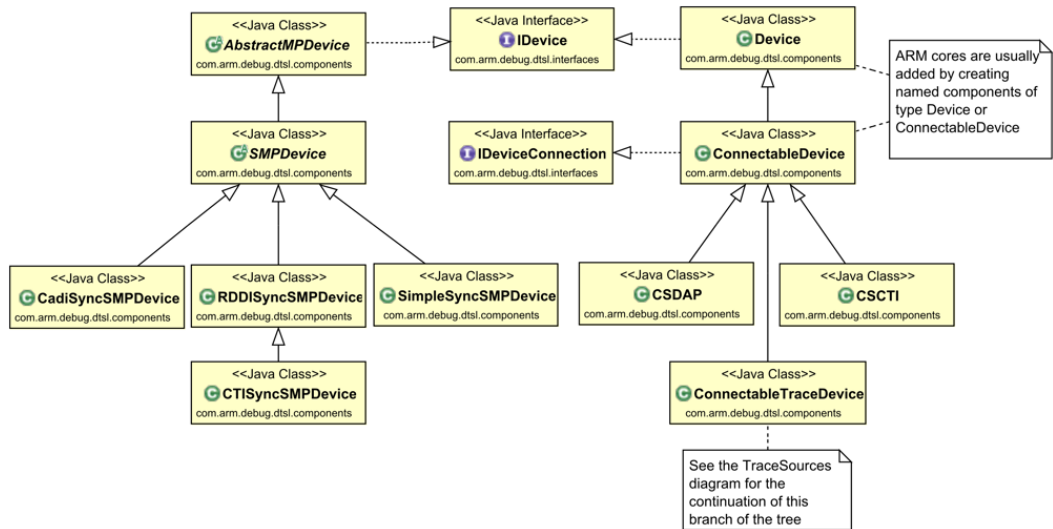


Figure 14-5 DTSL Device object hierarchy

Note

The figure shows the main components used for cores and core clusters.

14.5.3 CoreSight device component register IDs

The documentation for a CoreSight component lists its component registers and their address offsets. For example, the CoreSight STM component has a Trace Control and Status Register called `STMTCR` which has an offset of `0xE80`. To access this register through the *IDevice* interface, you need to know its register ID. To determine the ID, divide the documented offset by four. For example, the register ID for the `STMTCR` register is `0x3A0`, which is `0xE80/4`.

14.5.4 DTSL trace source objects

These objects represent sources of trace data within the platform. These could be ARM devices such as:

- ETM
- PTM
- ITM
- STM
- MTB (previously known as BBB)
- custom trace components that output data onto the CoreSight ATB.

These devices must implement the *ITraceSource* interface to be recognized as a trace source and to provide ATB ID information. They typically also implement *IDevice*. Most of these types of device only implement the register access methods from *IDevice* to allow configuration and control of the device, and they usually have a partner class which defines the names of the registers supported. For example, the *STMTraceSource* class has a partner class called *STMRegisters* which, for convenience, defines the STM register set IDs and many of their bit fields.

The class hierarchy for trace source objects is shown in the following figure:

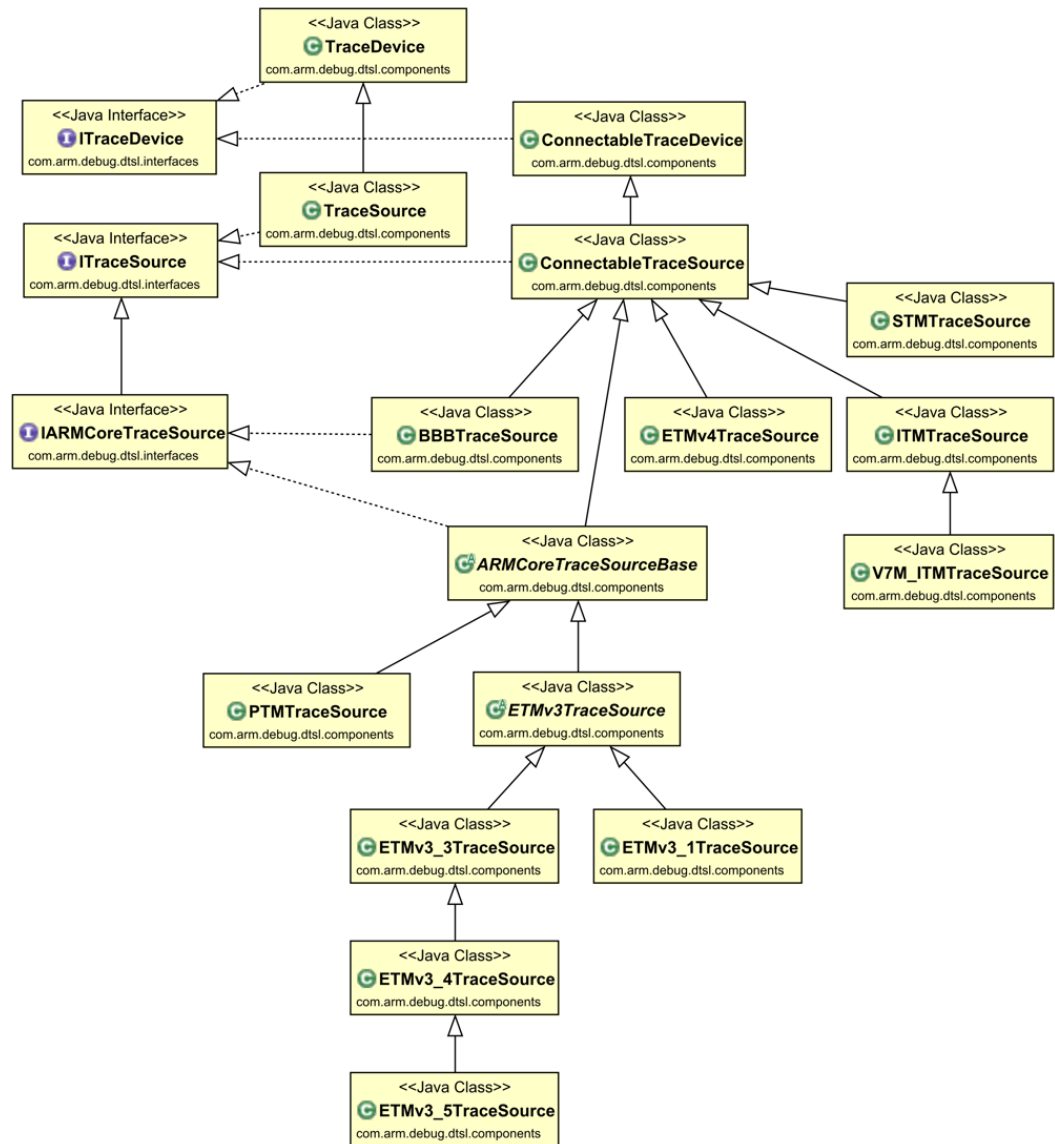


Figure 14-6 DTSL Trace Source class hierarchy

When implementing new trace source objects, you can choose to base them on *TraceDevice*, *ConnectableTraceDevice*, *TraceSource*, or *ConnectableTraceSource*. The choice depends on whether the source needs a connection, and whether it can identify itself in the trace stream with a source ID. As shown in the figure, all the standard ARM trace sources are derived from *ConnectableTraceSource*. This is because they are real devices which can be connected to for configuration, and which have ATB IDs to identify themselves in the received trace stream.

The following is a typical code sequence from a DTSL Jython script to create an ETM trace source:

```

1. devID = self.findDevice("CSETM")
2. etmATBID = 1
3. self.ETM = ETMv3_3TraceSource(self, devID, etmATBID, "ETM")

```

Line 1 locates the CSETM device ID (RDDI-DEBUG device index number) from the RDDI configuration. Line 2 assigns the ATB ID to be used for the ETM. Line 3 creates the DTSL *ETMv3_3TraceSource* object and names it “ETM”. If there are multiple ETMs in the platform, they should have different names, such as “ETM_1” and “ETM_2”, or “ETM_Cortex-A8” and “ETM_Cortex-M3”.

After creating the trace source objects, you must inform any trace capture device about the set of trace source objects to associate with it. This allows the client program to locate the ATB ID for the source of interest and request delivery of trace data for that source.

Related concepts

[14.5.5 DTSL trace capture objects on page 14-408.](#)

14.5.5 DTSL trace capture objects

Trace capture objects are responsible for storing and delivering trace data. Some trace capture devices reside on the platform itself, such as CoreSight ETB, TMC/ETB and TMC/ETR. Some trace capture devices capture trace into off-platform storage, such as DSTREAM with its 4GB trace buffer.

The following figure shows the class hierarchy and interfaces for Trace Capture device.

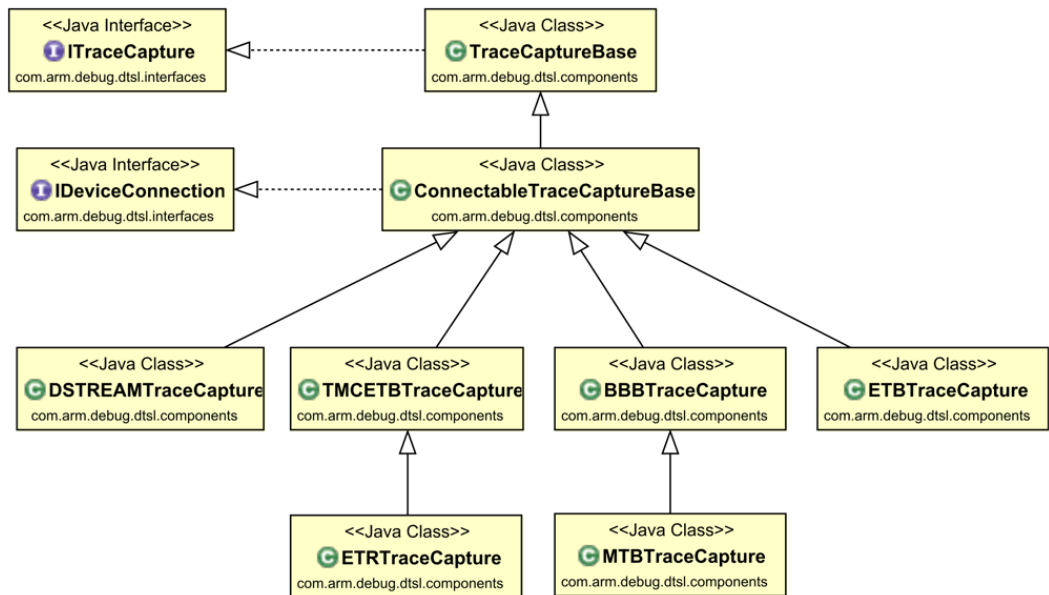


Figure 14-7 DTSL Trace Capture Objects

The following is a typical code sequence from a DTSL Jython script to create an ETB trace capture device:

```

1. devID = self.findDevice("CSETB")
2. self.ETB = ETBTraceCapture(self, devID, "ETB")
3. self.ETB.setFormatterMode(FormatterMode.BYPASS)
4. self.ETB.addTraceSource(self.ETM, self.coretexA8.getID())
5. self.addTraceCaptureInterface(self.ETB)
6. self.setManagedDevices([self.ETM, self.ETB])

```

Line 1 locates the ETB device ID (number) from the RDDI configuration (.rvic file). Line 2 creates the *ETBTraceCapture* object with the name “ETB”. Line 3 configures the formatter mode of the ETB. Line 4 adds an ETM object, such as that created by the code sequence in [14.5.4 DTSL trace source objects on page 14-406](#), to the set of trace sources to associate with the trace capture device. This should be done for all trace source objects which deliver trace to the trace capture device. To associate the ETM with a core, the code uses a version of the *addTraceSource()* method which allows it to associate the core by its ID. Line 5 adds the trace capture device to the DTSL configuration. Line 6 tells DTSL to automatically manage connection and disconnection to and from the ETM and ETB devices.

When a client program has a reference to the DTSL configuration object, it can query it for its set of trace capture devices. For each trace capture device, it can find out which trace sources feed into the trace capture device.

14.5.6 Memory as seen by a core device

When a DTSL configuration creates DTSL device objects for ARM cores, target memory can be accessed by performing memory operations on the device objects. This is how the DS-5 Debugger typically accesses memory during a debug session. However, such memory accesses have certain characteristics and are restricted in certain ways:

- For most ARM cores, memory cannot be accessed through the core when the core is executing.
- For cores with an MMU, the address used to access memory through the memory access methods of a device is the address as seen from the point of view of the core. This means that if the MMU is enabled, then the address is a virtual address, and it undergoes the same address translation as if it had been accessed by an instruction executed by the core. This is usually what a DTSL client, such as a debugger, wants to happen, so that it can present the same view of memory as that which the core sees when executing instructions.
- For cores with enabled caches, the data returned by the memory access methods of a device is the same as would be returned by a memory access by an instruction executed on the core. This means that if the data for the accessed address is currently in a cache, then the cached data value is returned. This value might be different from the value in physical memory. This is usually what a DTSL client, such as a debugger, wants to happen, so that it can present the same view of memory as that which the core sees when executing instructions.

14.5.7 Physical memory access via CoreSight

Although CoreSight does not require it, most CoreSight implementations provide a direct way to access the bus or buses of the target system. They do this by providing a *Memory Access Port* (MEM-AP) which is accessed through the CoreSight DAP. There are several types of MEM-AP depending on the type of the system bus. The three main types are APB-AP, AHB-AP, and AXI-AP, which provide access to APB, AHB, and AXI bus types respectively. Each of these access ports implements the CoreSight MEM-AP interface.

The following figure shows a simple, but typical, arrangement of MEM-APs:

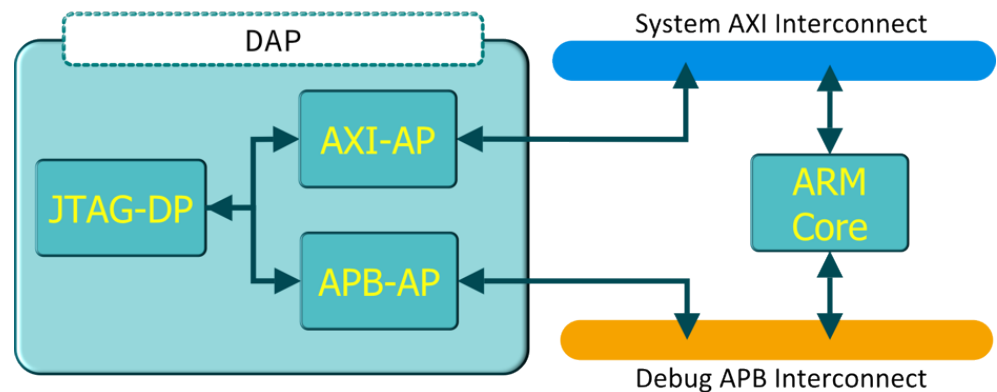


Figure 14-8 MEM-AP Access Ports

To allow direct memory access through one of the MEM-APs, a DTSL configuration can create device objects for the MEM-APs themselves. When the memory access methods are called on such devices, the memory access is directed straight onto the system bus, completely bypassing the core or cores.

Note

The memory access is not processed by the core MMU (so there is no core MMU address translation), and bypasses any cache in the core, which might result in a different value being observed to that observed by the core.

14.5.8 DTSL MEM-AP support

DTSL provides special classes for MEM-AP support. The following figure shows the class hierarchy:

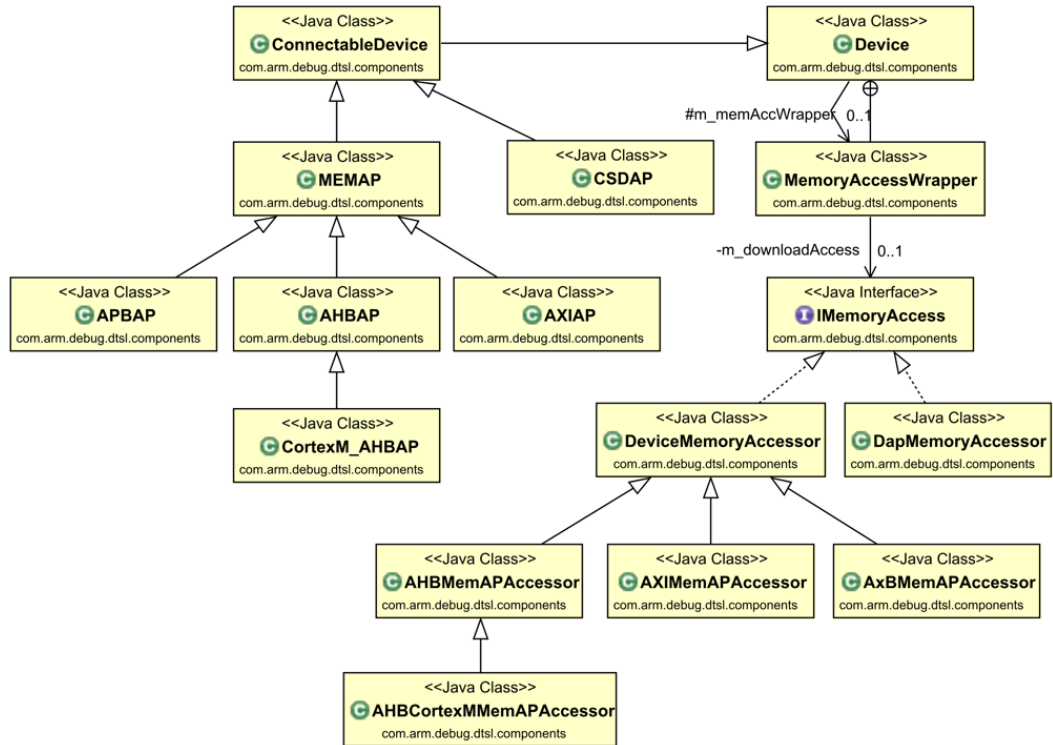


Figure 14-9 MEM-AP Class Hierarchy

The figure shows two main class trees. These are the MEM-AP tree and the *DeviceMemoryAccessor* tree. The DTSL configuration typically creates objects for one or more of the MEM-AP class types, suitably specialized for the actual bus type.

In the MCBSTM32E example, there is an AHB-AP which can be used to access memory directly. In the case of Cortex-M3, this bus is also used to access the CoreSight debug components, but for non-Cortex-M cores it is more typical for there to be a separate APB-AP for debug component access. The significant lines of the DTSL configuration script are similar to the following:

```
devID = self.findDevice("CSMEMAP")
self.AHB = CortexM_AHBAP(self, devID, "CSMEMAP")
```

In this case, the RDDI-DEBUG configuration has a device called CSMEMAP, which associates with a *CortexM_AHBAP* DTSL object. This object is derived from a DTSL *Device*, and so has memory access methods available.

If a client is aware of such DTSL devices, then it can use them to access memory directly.

14.5.9 Linking MEM-AP access to a core device

Not all clients are directly aware of MEM-AP type devices. DS-5 Debugger is an example of such a client. To allow such clients to make use of MEM-AP devices, named address space filters can be added to any DTSL Device object. The purpose of the address space filter is to tell the Device object that, if it sees a memory access with a known address space name, it should carry out the access through another DTSL device, rather than through the core. For example, we can add an address space filter to the Cortex-M3 DTSL Device which detects memory accesses to an address with an address space of "AHB". When it detects such an access, it performs the access using the AHB device, instead of going through

the Cortex-M3. For DS-5 Debugger, this means that the user can prefix an address with AHB: (for example, AHB:0x20000000), and the access is performed using the AHB-AP.

The following code shows how the address space filter is added to the Cortex-M3 object:

```
devID = self.findDevice("CSMEMAP")
self.AHB = CortexM_AHBAP(self, devID, "CSMEMAP")
devID = self.findDevice("Cortex-M3")
self.cortexM3 = ResetHookedDevice(self, devID, "Cortex-M3")
self.cortexM3.registerAddressFilters(
    [AHBCortexMMemAPAccessor("AHB", self.AHB, "AHB bus accessed via AP_0")])
```

Any number of address filters can be added, but each filter name (DS-5 Debugger address prefix) must be unique.

To determine the supported address spaces for an object which implements *IDevice*, call the *getAddressSpaces()* method. When a client matches against an address space, it can map the address space to a rule parameter which is passed into the *IDevice* memory access methods. The rule parameter is then used to direct the memory access to the appropriate device.

14.6 DTSL options

On many platforms, the debug components allow configuration of their properties. For example, in some CoreSight PTM components, the generation of timestamps within the trace data stream can be turned on or off. Such options are typically accessed and changed by using the DTSL objects that were created as part of the DTSL configuration. For example, the DTSL *PTMTraceSource* object has a *setTimestampingEnabled()* method to control timestamping. In this way, the DTSL objects that a DTSL configuration holds can expose a set of configuration options that you might want to modify. You might also want to create an initial option set to be applied at platform connection time, and then change some of those options after connecting, during a debug session. For example, this allows the PTM timestamp option to have a user setting applied at connection time, while also allowing you to turn the timestamps on and off during a debug session.

This section contains the following subsections:

- [14.6.1 DTSL option classes on page 14-412.](#)
- [14.6.2 DTSL option example walk-through on page 14-413.](#)
- [14.6.3 Option names and hierarchy on page 14-414.](#)
- [14.6.4 Dynamic options on page 14-416.](#)
- [14.6.5 Option change notification on page 14-416.](#)
- [14.6.6 Option change notification example walk-through on page 14-416.](#)

14.6.1 DTSL option classes

To support the concept of DTSL options, a DTSL configuration can expose a set of option objects. These objects allow a client to query the option set and their default values, and to modify the option values before and after connecting. The option objects are arranged hierarchically, and grouped in a way that allows them to be presented in a GUI. The option set must be available before connecting, so the options are exposed by a static method *getOptionList()* on the DTSLv1 derived class within a Jython script.

Note

The *getOptionList()* static method is not part of any defined Java interface. The DTSL configuration script manager uses Jython introspection at run time to determine whether the method exists.

The object set returned from *getOptionList()* should be an array of option objects. It is very common to partition the set of options into logical groups, each of which has its own tab page within a *TabSet*. Each tab page contains the options for its associated group.

The following figure shows the supported options types and class hierarchy:

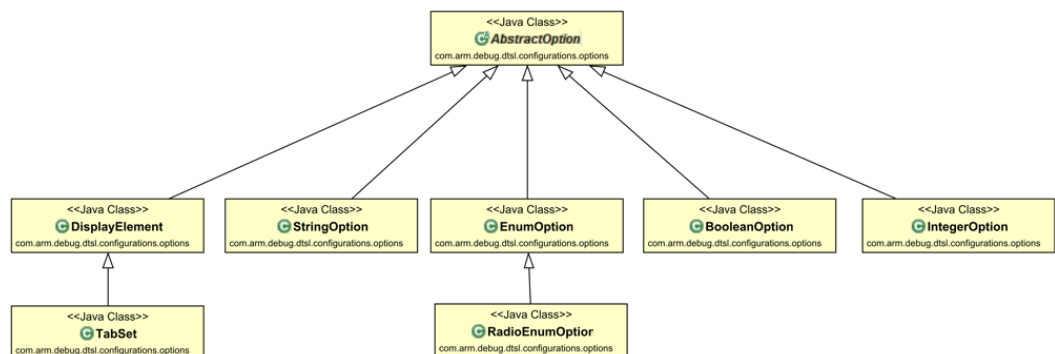


Figure 14-10 DTSL Option Classes

The DTSLv1 class provides many static helper methods for creating the options, and it is more usual for these methods to be used rather than directly creating the objects.

14.6.2 DTSL option example walk-through

The following is a simplified example from the Keil MCBSTM32E platform Jython script:

```

1. class DSTREAMDebugAndTrace(DTSLv1):
2.     '''A top level configuration class which supports debug and trace'''
3.
4.     @staticmethod
5.     def getOptionList():
6.         '''The method which specifies the configuration options which
7.         the user can edit via the launcher panel |Edit...| button
8.         '''
9.         return [
10.            DTSLv1.tabSet(
11.                name='options',
12.                displayName='Options',
13.                childOptions=[
14.                    DSTREAMDebugAndTrace.getTraceBufferOptionsPage(),
15.                    DSTREAMDebugAndTrace.getETMOptionsPage(),
16.                    DSTREAMDebugAndTrace.getITMOptionsPage()
17.                ]
18.            )
19.        ]

```

Line 4 marks the method as a static method of the containing class. This allows it to be called before an instance of the class exists. It also implies that any methods that are called are also static methods, because there is no self (this) associated with an instance of the class. Line 5 defines the static method with the name *getOptionList*. If this static method is present, then the configuration has options, otherwise it does not. Line 10 creates a *TabSet* object with name 'options', display name 'Options', and an array of child options, which in this example are each created by calling another static method.

Note

You might find it helpful to provide child options using several static methods. This prevents the nesting level of brackets from becoming too deep and difficult to understand, and makes it easier for you to avoid using the wrong type of bracket in the wrong place.

The following code extract shows the *getTraceBufferOptionsPage* method:

```

1.     @staticmethod
2.     def getTraceBufferOptionsPage():
3.         return DTSLv1.tabPage(
4.             name='traceBuffer',
5.             displayName='Trace Buffer',
6.             childOptions=[
7.                 DTSLv1.enumOption(
8.                     name='traceCaptureDevice',
9.                     displayName='Trace capture method',
10.                    defaultValue='none',
11.                    values=[
12.                        ('none', 'No trace capture device'),
13.                        ('DSTREAM', 'DSTREAM 4GB Trace Buffer')
14.                    ]
15.                ),
16.                DTSLv1.booleanOption(
17.                    name='clearTraceOnConnect',
18.                    displayName='Clear Trace Buffer on connect',
19.                    defaultValue=True
20.                ),
21.                DTSLv1.booleanOption(
22.                    name='startTraceOnConnect',
23.                    displayName='Start Trace Buffer on connect',
24.                    defaultValue=True
25.                ),
26.                DTSLv1.enumOption(
27.                    name='traceWrapMode',
28.                    displayName='Trace full action',
29.                    defaultValue='wrap',
30.                    values=[
31.                        ('wrap', 'Trace wraps on full and continues to store data'),
32.                        ('stop', 'Trace halts on full')
33.                    ]
34.                )

```

```
35.     ]
36. )
```

Note

The code uses nesting and indentation to help keep track of closing bracket types.

Line 3 creates a tab page named 'traceBuffer', which has an array of child options. These child options are displayed on the tab page within a GUI. Working through the child options might help you understand how they are displayed to the user. Line 7 creates an enum option. This is an option whose value is one of a set of pre-defined values, and which is typically presented to the user as a drop down list box. The list box shows the pre-defined values, and the user selects one of them. The values are given as pairs of strings. The first string is the internal value, and the second string is the text displayed to the user. Lines 16 to 21 create boolean options. These are options which are true or false, or on or off, and are usually shown to the user as a check box GUI element.

The following figure shows how DS-5 renders the tab set and tab page:

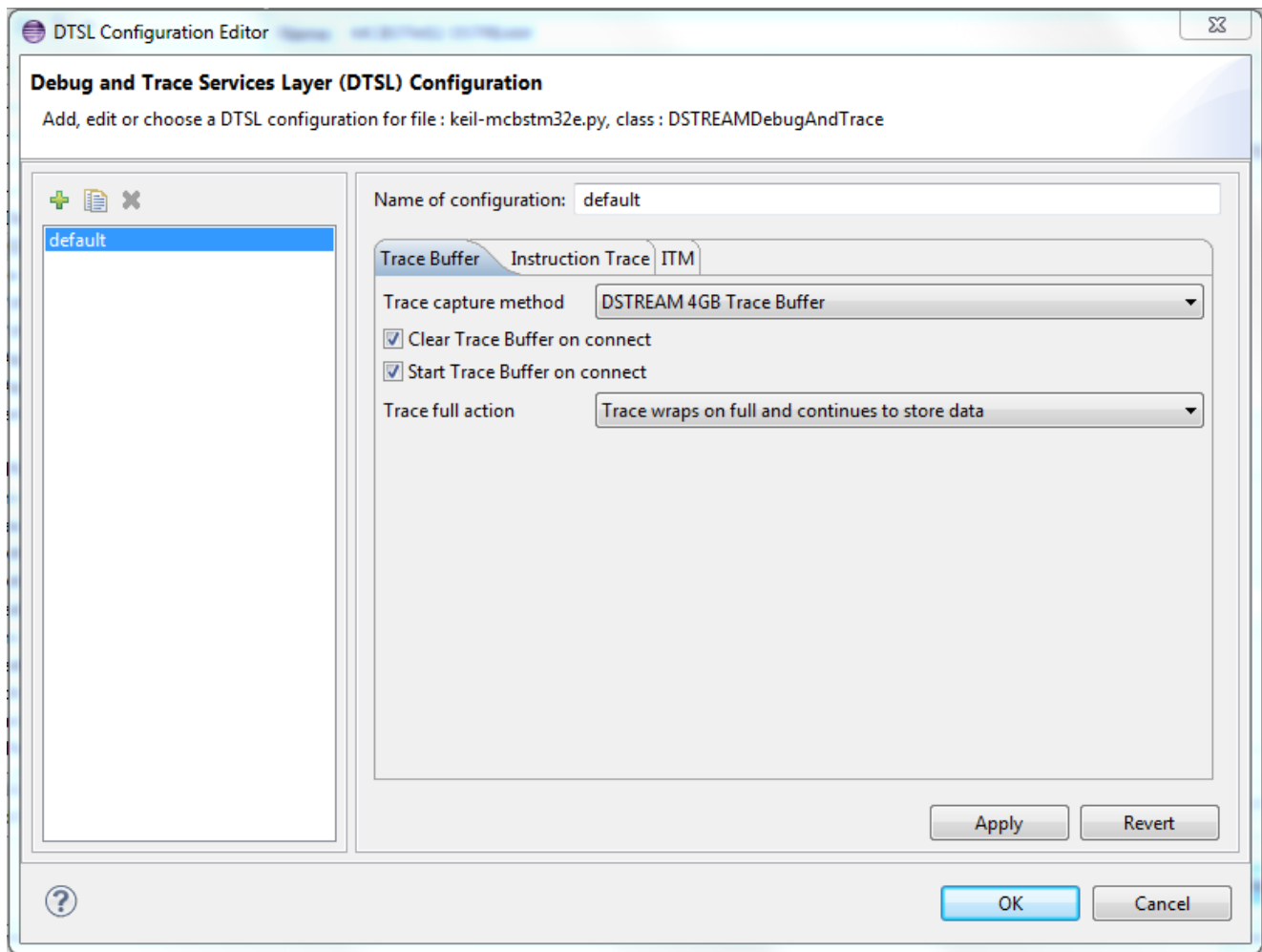


Figure 14-11 DSTREAM Trace Options

For more examples, see the full source code for the Keil example in the DTSLEExampleConfigdb project.

14.6.3 Option names and hierarchy

All options are part of an option hierarchy. Starting at the outermost level, the *TabSet* object is usually named 'options'. All other options are then created in a *childOptions* path, starting from this outermost

level. The 'name path' for an option consists of all the internal names (not the display names) in the hierarchy between the outermost level and the option in question, joined by the '.' character. For example, in the previous code samples, the option which indicates the currently selected trace capture device has the name path 'options.traceBuffer.traceCaptureDevice'. The components of this name path, joined by '.', are as follows:

options

The internal name of the outermost *TabSet*.

traceBuffer

The internal name of the child option for the trace buffer tab page object.

traceCaptureDevice

The internal name of the *EnumOption* for the currently selected trace capture device.

The full path name is important for at least three reasons:

- It can be used from the DS-5 Debugger command line, to read or modify the option value, using the commands `show dtsl-options` or `set dtsl-options`.
- It can be used in the `project_types.xml` file to direct the DS-5 Debugger to relevant options, such as which trace capture device to use (if any).
- It can be used in the `getOptionValue` and `setOptionValue` methods of the configuration, to read or modify an option's value.

Note

The full path option name is case sensitive.

Here is an example output from the `show dtsl-options` command to see the list of available DTSL options and their current values.

```
Command: show dtsl-options
dtsl-options options.ETM.cortexM3coreTraceEnabled: value is "true"
dtsl-options options.ITM.itmTraceEnabled: value is "true"
dtsl-options options.ITM.itmTraceEnabled.itmowner: value is "Target"
                                                    (read only)
dtsl-options options.ITM.itmTraceEnabled.itmowner.debugger.DWTENA: value is "true"
dtsl-options options.ITM.itmTraceEnabled.itmowner.debugger.PRIVMASK.[15:8]: value is "true"
dtsl-options options.ITM.itmTraceEnabled.itmowner.debugger.PRIVMASK.[23:16]: value is "true"
dtsl-options options.ITM.itmTraceEnabled.itmowner.debugger.PRIVMASK.[31:24]: value is "true"
dtsl-options options.ITM.itmTraceEnabled.itmowner.debugger.PRIVMASK.[7:0]: value is "true"
dtsl-options options.ITM.itmTraceEnabled.itmowner.debugger.STIMENA: value is "0xFFFFFFFF"
dtsl-options options.ITM.itmTraceEnabled.itmowner.debugger.TSENA: value is "true"
dtsl-options options.ITM.itmTraceEnabled.itmowner.debugger.TSPrescale: value is "none"
dtsl-options options.ITM.itmTraceEnabled.itmowner.target.targetITMATBID: value is "2"
                                                    (read only)
dtsl-options options.traceBuffer.clearTraceOnConnect: value is "true"
                                                    (read only)
dtsl-options options.traceBuffer.startTraceOnConnect: value is "true"
                                                    (read only)
dtsl-options options.traceBuffer.traceCaptureDevice: value is "DSTREAM"
                                                    (read only)
dtsl-options options.traceBuffer.traceWrapMode: value is "wrap"
                                                    (read only)
dtsl-options options.ucProbe.ucProbeEnabled: value is "false"
dtsl-options options.ucProbe.ucProbeEnabled.PORT: value is "9930"
                                                    (read only)
```

Here is an example of the `set dtsl-options` command to change the current value of any non read-only option:

```
Command: set dtsl-options options.ITM.itmTraceEnabled false
DTSL Configuration Option "options.ITM.itmTraceEnabled" set to false
```

14.6.4 Dynamic options

Some option values can be modified dynamically, after connecting to the platform. For a DS-5 Debug session, this means the option can be changed during the debug session, using either the DS-5 Debugger command line or the **DTSL Options...** menu selection with the Debug Control View.

Not all options can be modified after connecting. For example, the trace capture device cannot typically change during the debug session, although the option to enable ITM trace can change. Even if an option can be changed, it might not apply the change immediately. For example, most trace-related dynamic options apply changes only when tracing is started or restarted.

To mark an option as dynamic, add the `'isDynamic=True'` parameter to the option constructor. For example, the ITM option to generate timestamps could be created as follows:

```
DTSLv1.booleanOption(
    name='TSENA',
    displayName = 'Enable differential timestamps',
    defaultValue=True,
    isDynamic=True
)
```

When DS-5 Debugger displays the options during a debug session, it only allows the dynamic options to be changed. All the options are shown to the user, but the non-dynamic ones are grayed out and cannot be changed.

14.6.5 Option change notification

Shortly after the DTSL configuration instance (the object derived from DTSLv1) is created, the option values are given to the instance by calling its *optionValuesChanged* method. This method inspects the current option values and configures the platform components accordingly.

Note

The *optionValuesChanged* method is called after the constructor is called, but before the DTSL components are connected to the target platform. This means that the DTSL objects can be configured with their settings, but cannot send the settings to the target components.

If the options are changed during a debug session, then the *optionValuesChanged* method is called again, to inform the DTSL components that the options have changed.

Note

Currently, the call to the *optionValuesChanged* method does not indicate which options have changed. A future version of DTSL will address this.

14.6.6 Option change notification example walk-through

These Jython code snippets are from the Keil MCBSTM32E platform Jython script and the *DSTREAMDebugAndTrace* class:

```
1. def optionValuesChanged(self):
2.     '''Callback to update the configuration state after options are changed.
3.         This will be called:
4.         * after construction but before device connection
5.         * during a debug session should the user change the DTSL options
6.     '''
7.     obase = "options"
8.     if self.isConnected():
9.         self.updateDynamicOptions(obase)
10.    else:
11.        self.setInitialOptions(obase)
```


Line 1 declares the *optionValuesChanged* method, which is called to tell the DTSL components that the options have changed. Line 7 assigns the top level options name path value. Lines 8 to 11 call one of two methods depending on whether the configuration is connected yet.

```

1.  def setInitialOptions(self, obase):
2.      '''Takes the configuration options and configures the
3.         DTSL objects prior to target connection
4.         Param: obase the option path string to top level options
5.         ...
6.     if self.traceDeviceIsDSTREAM(obase):
7.         self.setDSTREAMTraceEnabled(True)
8.         self.setDSTREAMOptions(obase+".traceBuffer")
9.         obaseETM = obase+".ETM"
10.        obaseITM = obase+".ITM"
11.        self.setETMEnabled(self.getOptionValue(
12.                               obaseETM+".cortexM3coreTraceEnabled"))
13.        self.reservedATBIDs = {}
14.        self.setITMEnabled(self.getOptionValue(obaseITM+".itmTraceEnabled"))
15.        obaseITMOwner = obaseITM+".itmTraceEnabled.itmowner"
16.        if self.debuggerOwnsITM(obaseITMOwner):
17.            self.setITMOwnedByDebugger(True);
18.            self.setITMOptions(obaseITMOwner+".debugger")
19.        else:
20.            self.setITMOwnedByDebugger(False);
21.            self.reservedATBIDs["ITM"] =
22.                self.getOptionValue(obaseITMOwner+".target.targetITMATBID")
23.            self.updateATBIDAssignments()
24.        else:
25.            self.setDSTREAMTraceEnabled(False)
26.            self.setETMEnabled(False)
27.            self.setITMEnabled(False)

```

In this code example, note the following:

- The value for an option is retrieved using the *self.getOptionValue* method, which takes the full option path name to the option value.
- The code builds up the full option path names, which allows the options to be moved more easily. This can be seen in the way that the *obaseITMOwner* value is constructed and passed to the *self.setITMOptions* method. This allows *self.setITMOptions* to be written without having to hard code the full option name path into it. Instead, it only needs to know the path extensions from the passed base to determine its option values.

For completeness, the following shows the dynamic option update method:

```

1.  def updateDynamicOptions(self, obase):
2.      '''Takes any changes to the dynamic options and
3.         applies them. Note that some trace options may
4.         not take effect until trace is (re)started
5.         Param: obase the option path string to top level options
6.         ...
7.     if self.traceDeviceIsDSTREAM(obase):
8.         obaseETM = obase+".ETM"
9.         self.setETMEnabled(self.getOptionValue(
10.                               obaseETM+".cortexM3coreTraceEnabled"))
11.        obaseITM = obase+".ITM"
12.        if self.getOptionValue(obaseITM+".itmTraceEnabled"):
13.            self.setITMEnabled(True)
14.            obaseITMOwner = obaseITM+".itmTraceEnabled.itmowner"
15.            if self.debuggerOwnsITM(obaseITMOwner):
16.                self.setITMOptions(obaseITMOwner+".debugger")
17.        else:
18.            self.setITMEnabled(False)

```

For the dynamic option changes, only the options marked as dynamic need inspecting.

————— Note —————

The option values are passed on to the corresponding DTSL objects, but the option changes might not be applied immediately. In many cases, the change only applies when execution or trace is next started. Whether the option change is applied immediately is determined by the implementation of the DTSL object.

14.7 DTSL support for SMP and AMP configurations

From the point of view of DS-5 Debugger, *Symmetric Multi Processing* (SMP) refers to a set of architecturally identical cores which are tightly coupled together and used as a single multi-core execution block. From the point of view of the debugger, they must be started and halted together.

In larger systems, there may be several SMP sets, each of which is referred to as a cluster. Typically, a cluster is a set of 4 cores in an SMP configuration. All cores in the SMP cluster also have the same view of memory and run the same image.

From the point of view of DS-5 Debugger, *Asymmetric Multi Processing* (AMP) refers to a set of cores which are operating in an uncoupled manner. The cores can be of different architectures { Cortex-A8, Cortex-M3 }, or of the same architecture but not operating in an SMP configuration. From the point of view of the debugger, it depends on the application whether the cores need to be started or halted together.

From the point of view of DTSL, the cores in the set (SMP or AMP) are part of the same configdb platform. Using `project_types.xml`, the platform exposes a set of debug operations which cover the supported use cases. All of these use cases must be provided for by the same Jython DTSL configuration class. This is because, although there can be multiple clients using DTSL (for example, one debugger controlling a Cortex-A8 and another controlling a Cortex-M3), there is only one set of target debug hardware (for example, only one TPIU). There must therefore be a single DTSL instance in control of the debug hardware.

In SMP systems, there is usually a hardware mechanism which keeps the set of cores at the same execution state. Some AMP systems must also have synchronized execution state, and the multi-client, single DTSL instance architecture supports this. The single DTSL instance is always aware of the target execution state, and can typically arrange for a single execution state between all AMP cores.

This section contains the following subsections:

- [14.7.1 AMP systems and synchronized execution on page 14-418.](#)
- [14.7.2 Execution synchronization levels on page 14-419.](#)
- [14.7.3 Software synchronization on page 14-419.](#)
- [14.7.4 Tight synchronization on page 14-419.](#)
- [14.7.5 Hardware synchronization on page 14-419.](#)
- [14.7.6 SMP states on page 14-420.](#)
- [14.7.7 Use of CTI for SMP execution synchronization on page 14-420.](#)

14.7.1 AMP systems and synchronized execution

If a platform contains multiple cores, then when the first DTSL client connects, the DTSL configuration creates devices for all of the cores. The client uses the devices for the cores it wants to control. When a second client connects to the same platform, it must present an identical set of connection parameters. The DTSL connection manager therefore returns the same DTSL configuration instance that was created by the first client connection. The second client can use the devices for the cores it wants to control. In this way, two clients can use the same DTSL configuration instance, including any DTSL options.

If execution synchronization is not required, a simple DTSL configuration is enough, with core execution state being independent. However, if synchronized execution state is required, then the created object model must provide this. The execution synchronization can be implemented with features in the hardware, or by creating a software object model hierarchy which arranges for a shared execution state.

14.7.2 Execution synchronization levels

The level at which DTSL can perform synchronized execution status depends heavily on both the execution controller (for example, the JTAG control box) and the on-chip debug hardware. However, there are roughly three different levels (or qualities) of synchronization:

- Software synchronization
- Tight synchronization
- Hardware synchronization

14.7.3 Software synchronization

This is the lowest level or quality of synchronization. 'Software' refers to the DTSL software running on the host debugger computer. At this level, the synchronization is of the following form:

Synchronized start

This is achieved by asking each device to start executing, by calling the *go()* method on each device in turn.

Synchronized stop

This is achieved by asking each device to stop executing, by calling the *stop()* method on each device in turn. If one device is seen to be stopped (by DTSL receiving a `RDDI_EVENT_TYPE.RDDI_PROC_STATE_STOPPED` stopped event), then the DTSL configuration must request all other devices to stop.

This synchronization is done on the host computer, so there can be hundreds of milliseconds between each core actually stopping. Whether this is a problem depends on how the target application handles other cores not responding (if they communicate with each other at all).

14.7.4 Tight synchronization

With tight synchronization, the execution controller (JTAG box such as DSTREAM) can manage the synchronization. This can typically be further divided into several sub-levels of support:

- The execution controller supports the `RDDI Debug_Synchronize()` call. In this case, the synchronized start and stop functionality is implemented in the execution controller. The controller is much 'closer' to the target system, so it can typically synchronize down to sub-millisecond intervals.
- The execution controller can define one or more sets of cores which form a cluster. When any one of the cores in a set is seen to halt execution, the others are automatically halted. This typically provides synchronized stop down to a sub-millisecond interval. DSTREAM supports this technique.
- The execution controller supports `Debug_Synchronize()`, but cannot define clusters. In this case, the DTSL configuration must be written so that if it sees any core in a synchronized group as halted, it issues the `RDDI Debug_Synchronize()` call to halt the others in the group. In a group of several devices, the time interval between the first halting and the others halting may be hundreds of milliseconds, but the interval between the others halting is typically sub-millisecond.

14.7.5 Hardware synchronization

With hardware synchronization, the target provides synchronization features on-chip. This is typically the case for ARM CoreSight systems that use the *Cross Trigger Interface* (CTI) to propagate debug halt and go requests between cores. This ability relies on the hardware design implementing this feature, and so might not be available on all CoreSight designs.

Related concepts

[14.7.7 Use of CTI for SMP execution synchronization on page 14-420.](#)

14.7.6 SMP states

For SMP systems, DTSL presents a single device to the client (see *SMPDevice* and its relations in the DTSL Java docs), and the client controls execution state through this device. This *SMPDevice* is a 'front' for the set of real devices which form an SMP group. When the *SMPDevice* reports the execution state to the client, there is the possibility of inconsistent states. Ideally, for an SMP group, all the cores have the same state, either executing or halted. In practice, this might not be the case. To allow for this possibility, the *SMPDevice* can report an inconsistent state to the client (debugger). This represents the case when not all cores are in the same state. Normally, DTSL provides a time window within which it expects all cores to get into the same state. If all cores become consistent within this time window, then DTSL reports a consistent state to the client, otherwise it reports an inconsistent state. This allows the client to reflect the true state of the system to the user, but still allows the state to be reported as consistent if consistency is achieved at some future time.

14.7.7 Use of CTI for SMP execution synchronization

Cross Trigger Interface (CTI) is part of the ARM *Embedded Cross Trigger* (ECT) system. Each component in the system can have a CTI which allows inputs and outputs to be routed (or channeled) between the components. The channeling is done by the *Cross Trigger Matrix* (CTM), which is part of the ECT. The CTM supports a fixed number of channels onto which the CTIs can output or input signals. There may be many signals in the system which can be routed between components, and the CTIs can be told which signals to route by assigning them to a channel.

For example, in many systems, each core in the SMP group has a CTI connected to the following signals:

Table 14-1 CTI Signal Connections

Name	Direction	Purpose
DBGTRIGGER	Output from core to CTI	Indicates that the core is going to enter debug state (is going to stop executing)
EDBGRQ	Input to core from CTI	An external request for the core to enter debug state (stop executing)
DBGRESTART	Input to core from CTI	An external request for the core to exit debug state (start executing)

For synchronized execution to work, the DTSL configuration assigns two channels, one of which is for stop requests and the other of which is for start requests. The CTI or CTIs are configured to connect the above signals onto these channels.

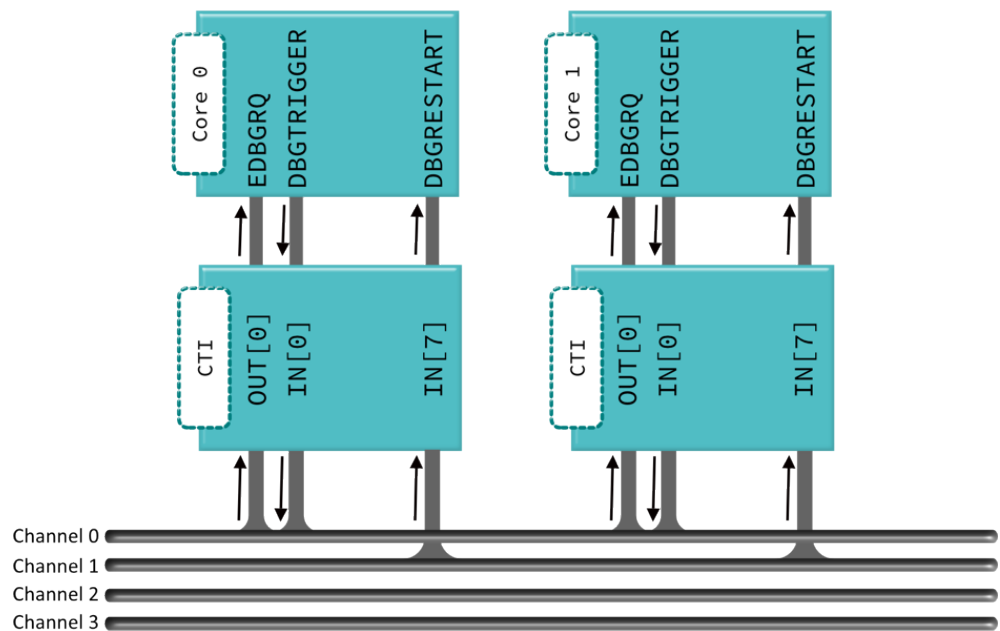


Figure 14-12 Example use of CTI for H/W execution synchronization

With this configuration:

- When a debug tool wants to halt all cores, it sends a CTI pulse. Sending a pulse from any CTI onto the stop channel sends a EDBGREQ to all cores, which causes them to halt. This provides the synchronized stop functionality for stop requests instigated by the debug tool.
- When any core halts (hits a breakpoint), the DBGTRIGGER signal outputs onto the stop channel and sends a EDBGREQ signal to all the other cores, which causes them to halt. This provides the synchronized stop functionality for breakpoints and watchpoints, for example.
- When all cores are ready to restart, sending a pulse from any CTI onto the start channel sends a DBGRESTART signal to all cores. This provides the synchronized start functionality.

The convention for DTSL configurations is that channel 0 is used for the stop channel and channel 1 is used for the start channel. DTSL configuration scripts usually allow this to be modified by changing the following constants, which are assigned near the top of the configuration script:

```
CTM_CHANNEL_SYNC_STOP = 0 # use channel 0 for sync stop
CTM_CHANNEL_SYNC_START = 1 # use channel 1 for sync start
```

14.8 DTSL Trace

DTSL is designed to support many different trace capture devices, such as DSTREAM, ETB, TMC/ETB and TMC/ETR. It is also possible to extend DTSL to support other trace capture devices. Each of these capture devices can present its data to DTSL in a different format.

Within a platform, trace data can originate from several trace sources. This data is mixed together into the data stream which the trace capture device collects. For simplicity, trace clients (software packages which receive or read trace data from DTSL) are usually designed based on the assumption that the only trace data they receive from the trace source is data which they know how to decode. For example, if a trace client knows how to decode PTM data, then it only expects to receive PTM data when it reads trace data from DTSL.

This section contains the following subsections:

- [14.8.1 Platform trace generation on page 14-422.](#)
- [14.8.2 DTSL trace decoding on page 14-423.](#)
- [14.8.3 DTSL decoding stages on page 14-423.](#)
- [14.8.4 DTSL trace client read interface on page 14-424.](#)
- [14.8.5 Supporting multiple trace capture devices on page 14-425.](#)
- [14.8.6 Decoding STM STPv2 output on page 14-426.](#)
- [14.8.7 Example STM reading code on page 14-426.](#)
- [14.8.8 STM objects on page 14-427.](#)
- [14.8.9 DTSL client time-stamp synchronization support on page 14-428.](#)

14.8.1 Platform trace generation

The following figure shows a simplified diagram of trace generation within a platform. There are several trace sources outputting trace data onto a trace bus. The bus takes the data through a frame generator and outputs it to a trace capture device.

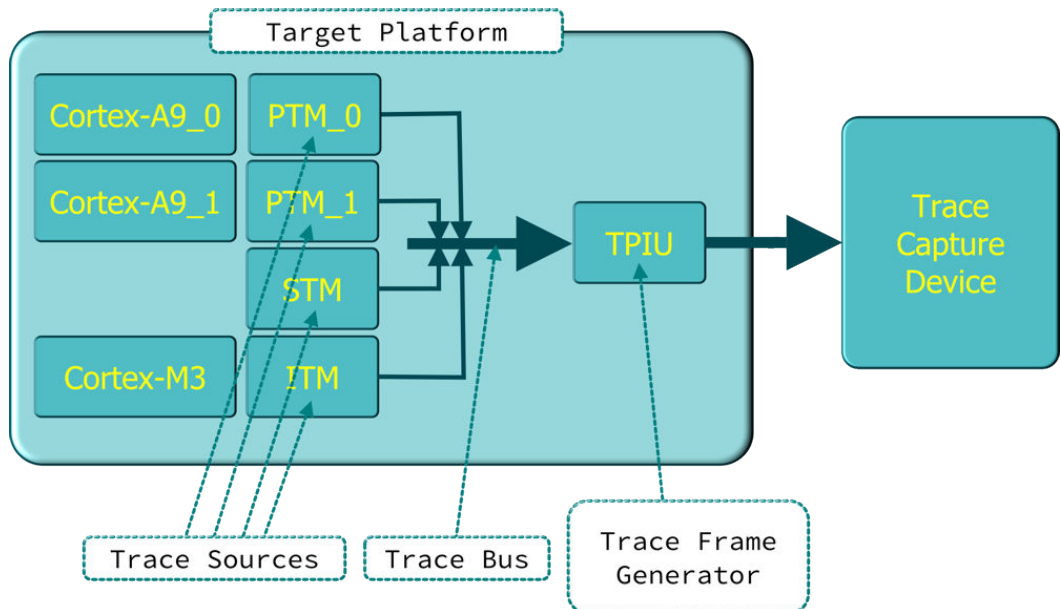


Figure 14-13 Trace Generation

14.8.2 DTSL trace decoding

To process the raw trace data from the trace capture device into a format which is suitable for trace clients to consume, DTSL pushes the raw trace data through a pipeline of trace decoders. The following figure shows an example of this flow for DSTREAM trace data:

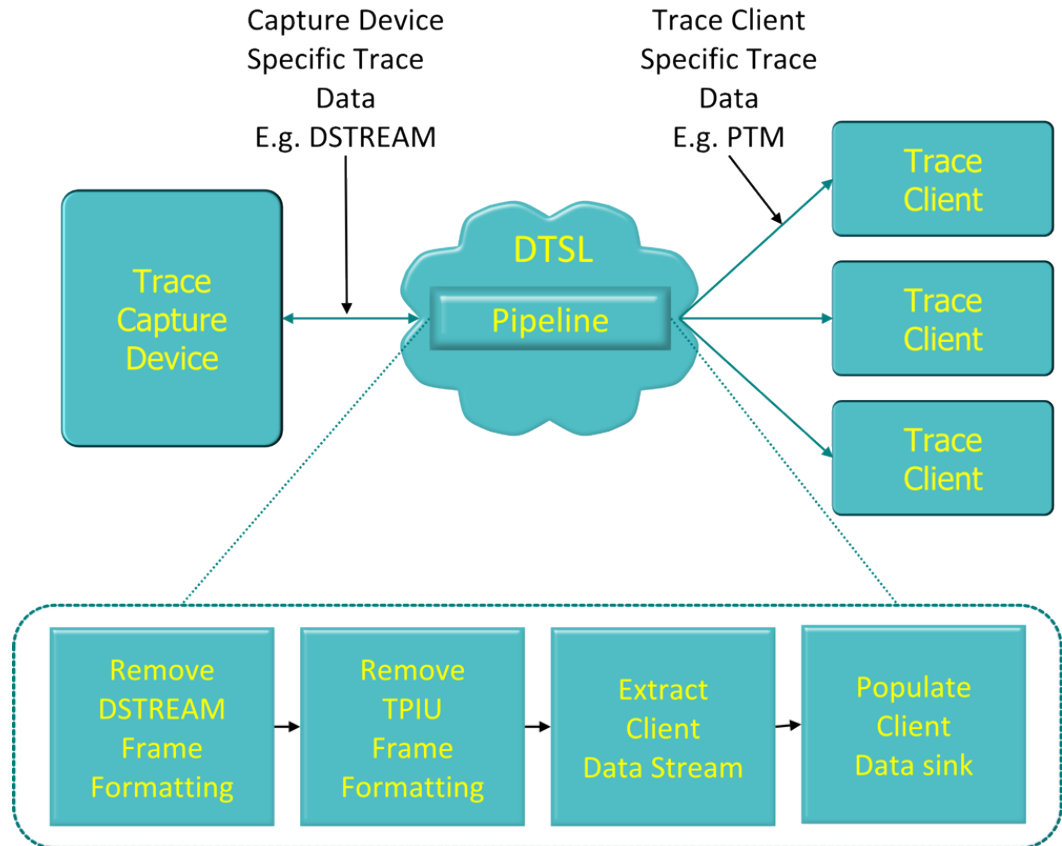


Figure 14-14 DTSL Trace Decoding Stages for DSTREAM

The number and type of the pipeline decoding blocks depends on the format of the trace capture device and the trace data format. However, the final stage should always be to place client compatible data (raw trace source data) into a *DataSink*, ready for the trace client to consume.

14.8.3 DTSL decoding stages

The minimal pipeline decoder does nothing to the data from the trace capture device, except to write it into a *DataSink* for the trace client to read. You can use this pipeline when you know that the data from the trace capture device is already in the format required by a trace client. For example, if you have a disk file which contains raw PTM trace data (that is, the data exactly as output from the PTM onto the system ATB, which you might have captured from a simulation), then you can create a PTM-file-based trace capture device. The decoding pipeline would contain only a *DataSink*, into which you would write the content of the file. The PTM trace client could then read the PTM data directly from the *DataSink*.

For less straightforward pipelines, a chain of objects must be constructed, each of which must implement the *IDataPipelineStage* interface.

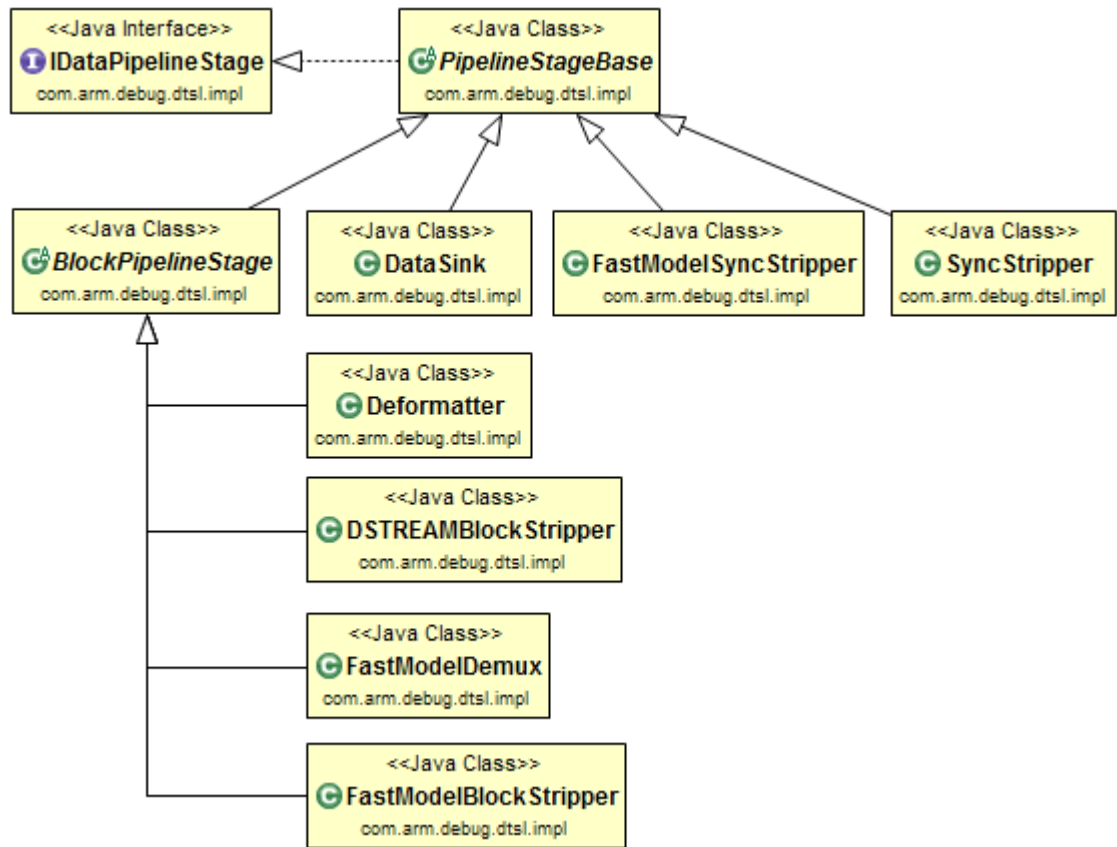


Figure 14-15 DTSL Trace Pipeline Hierarchy

For ARM based trace systems which use a TPIU Formatter (CoreSight ETB, TMC/ETB and TMC/ETR), two further pipeline stages must be added. These are the *SyncStripper* and *Deformatter* stages, which remove the TPIU sync frames and extract data for a particular trace source ID respectively.

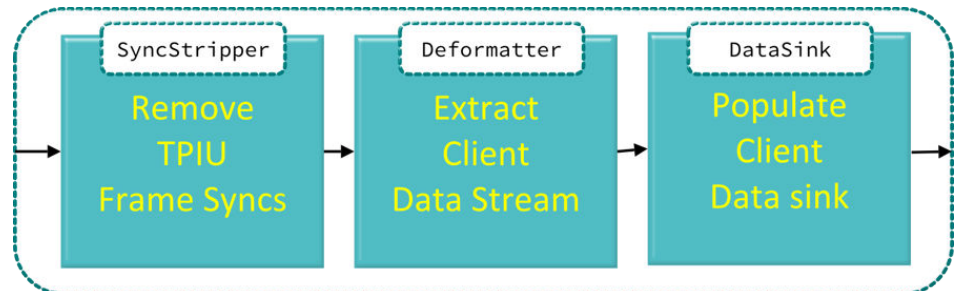


Figure 14-16 ETB Trace Decode Pipeline Stages

By implementing new pipeline stages, it is possible to provide trace support for any trace capture device, as long as the final output stage can be reduced to data which is compatible with the expectations of a trace client.

14.8.4 DTSL trace client read interface

Before a trace client can read trace data, it must get a *TraceSourceReader* object from the trace capture device. In practice, this means querying the DTSL configuration for the correct trace capture device, several of which might be available within a configuration, and calling the *borrowSourceReader()* method to get an appropriate *TraceSourceReader*. Trace data can then be retrieved from the *TraceSourceReader*. When it finishes reading trace data, the client must then return the

TraceSourceReader object to the trace capture device. This is so that the trace capture device knows when there are no clients reading trace, and therefore when it is free to start, or restart, trace collection.

14.8.5 Supporting multiple trace capture devices

A DTSL configuration can contain several trace capture devices. The following are some possible reasons for this:

- The target platform contains several CoreSight ETB components.
- The target platform can output to an external DSTREAM device, in addition to an internal CoreSight ETB.

In some cases, there can only be one active trace capture device. In this case, you can choose whether to use the DSTREAM or the ETB. In other cases, there can be several trace capture devices active at the same time. This is common when the platform contains multiple clusters of cores, each of which outputs trace to its own CoreSight ETB.

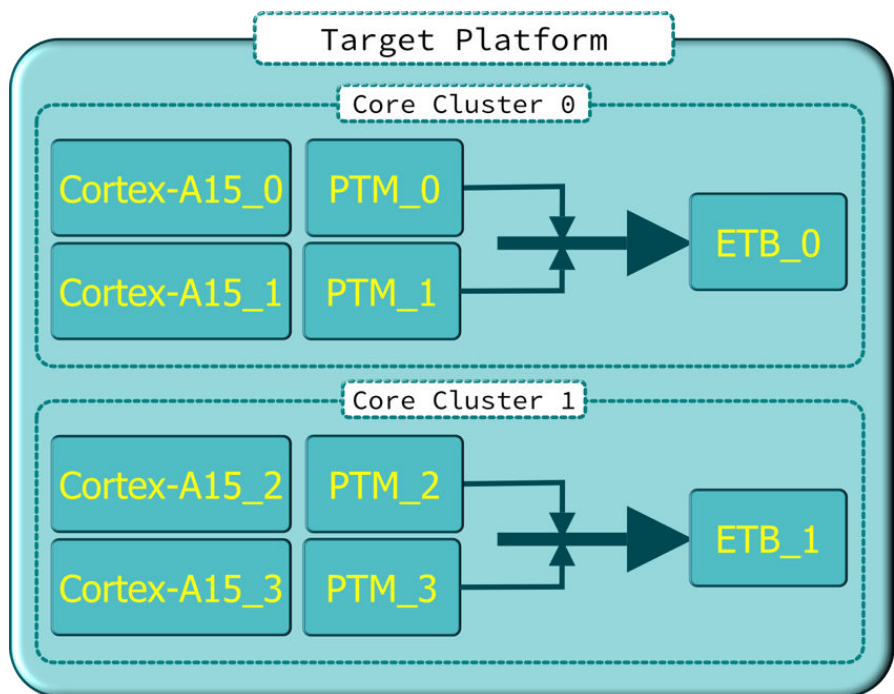


Figure 14-17 Example of Multiple Trace Capture Devices

To allow trace clients to receive trace data from a trace source, the DTSL configuration can be told about the association between a trace source and one or more trace capture devices. In the figure, for example, the trace source named PTM_2 is associated with the trace capture device named ETB_1. If a client wants to display the trace data for PTM_2, it can ask DTSL which trace capture devices are associated with that trace source, and direct its trace read requests to the correct trace capture device.

Note

It is possible for a trace source to be associated with multiple trace capture devices, such as an internal ETB and an external DSTREAM. In such cases, you might need to provide more information to the client about which trace capture device to use when reading trace data.

14.8.6 Decoding STM STPv2 output

The ARM STM CoreSight component generates a STPv2 compatible data stream as its output. The STPv2 specification is a MIPI Alliance specification which is not publicly available. To allow clients to consume STM output, DTSL has a built-in decoder which turns STPv2 into an STM object stream.

To consume STM output, a client should do the following:

- Create an object which implements the *ISTMSourceMatcher* interface. This object tells the decoder which STM master IDs and channel IDs to decode. The STM package includes three implementations of the *ISTMSourceMatcher* interface. These are *STMSourceMatcherRange*, *STMSourceMatcherSet*, and *STMSourceMatcherSingle*. If none of these implementations covers your needs, you can also create a new class which implements the *ISTMSourceMatcher* interface.
- Create an *STMChannelReader* object, specifying the trace capture device object and the source matcher object.
- Create an object which implements the *ISTMObjectReceiver* interface, to receive the STM objects.
- When trace data is available, get hold of an *ISourceReader* object. Pass this, along with the *ISTMObjectReceiver* object, to the *read* method on the *STMChannelReader* object. The *read* method decodes the trace into an STM object stream, and passes these objects to the *ISTMObjectReceiver*.

Related concepts

[14.8.4 DTSL trace client read interface on page 14-424.](#)

14.8.7 Example STM reading code

The following is some Java code which shows an example of STM Object reading. This could also be implemented in Jython.

In this case, the *STMTraceReader* object implements the *ISTMObjectReceiver* interface itself. This means that the code can pass the object to the *STMChannelReader read* method as the class to receive the *STMObjects*.

The example code creates an *STMSourceMatcherRange* object with a parameter set which matches against all Masters and Channels.

The STPv2 packet protocol outputs a SYNC packet which allows the decoder to synchronize the binary data stream to the start of a packet. When decoding arbitrary data streams, the decoder needs to synchronize to the stream before starting to decode the STPv2 packets. Once the stream is synchronized, there is no need to resynchronize, as long as contiguous data is being decoded.

The decoder has two ways to handle errors in the STPv2 packets stream:

- Throw an *STMDecodeException* or an *STMParseException*. The advantage of this method is that the errors are detected immediately, but the disadvantage is that you cannot continue processing STPv2 packets (there is no way to resume decoding after the last error position).
- Insert an *STMDecodeError* object into the generated *STMObject* set. The advantage of this method is that the full data stream is decoded, but the disadvantage is that the error is not processed by the client until the generated *STMDecodeError* is processed.

```
/**
 * Class to read STM trace data and to get it processed into a
 * text stream.
 */
public class STMTraceReader implements ISTMObjectReceiver {
    /**
     * The trace device - ETB or DSTREAM or ....
     */
    private ITraceCapture traceDevice;
    /**
     * A list of STMObjects that gets generated for us
     */
    private List<STMObject> stmObjects;
```

```
[snip - other attribute declarations]

    public void decodeSTMTrace() {
        STMSourceMatcherRange stmSourceMatcher = new STMSourceMatcherRange(0, 128, 0, 65535);
        STMChannelReader stmChannelReader = new STMChannelReader(
            "STM Example",
            this.traceDevice,
            stmSourceMatcher);
        ISourceReader reader = this.traceDevice.borrowSourceReader(
            "STM Reader", this.stmStreamID);
        if (reader != null) {
            try {
[snip - code to figure out if trace is contiguous from last read.]
                if (!traceDataIsContiguous) {
                    stmChannelReader.reSync();
                }
[snip - code to figure out how much trace to read and from where. Also assign values to
nextPos[] and readSize.]
                this.stmObjects.clear();
                try {
                    stmChannelReader.read(nextPos[0], readSize, this, nextPos, reader);
                } catch (DTSLEException e) {
                    System.out.println("Caught DTSLEException during STPv2 decode:");
                    System.out.println(e.getMessage());
                    stmChannelReader.reSync();
                }
            } catch (DTSLEException e) {
                System.out.println("DTSLEException:");
                e.printStackTrace();
            }
            finally {
                /* Must return the trace reader to DTSL so that it knows we have finished
reading
                */
                this.traceDevice.returnSourceReader(reader);
            }
        }
    }

    /* (non-Javadoc)
     * @see
com.arm.debug.dtsl.decoders.stm.stmobjects.ISTMObjectReceiver#write(com.arm.debug.dtsl.decode
rs.stm.stmobjects.STMObject)
     */
    @Override
    public boolean write(STMObject stmObject) {
        this.stmObjects.add(stmObject);
        return true;
    }
}
```

14.8.8 STM objects

The following figure shows the STM object model. All objects generated by the decoder are derived from *STMObject*. All *STMObjects* can contain a timestamp (*STMTimestamp*) if one was generated, otherwise the timestamp attribute is null.

The most common form of object generated is the *STMData* objects, which can hold multiple 4-bit, 8-bit, 16-bit, 32-bit, or 64-bit data payloads. Each data packet can also have a marker attribute, in which case it holds only one data payload.

Not all STM object types can be generated from an ARM STM component. {*STMTime*, *STMXSync*, *STMTrig*, *STMUser*} are not generated in ARM STM output.

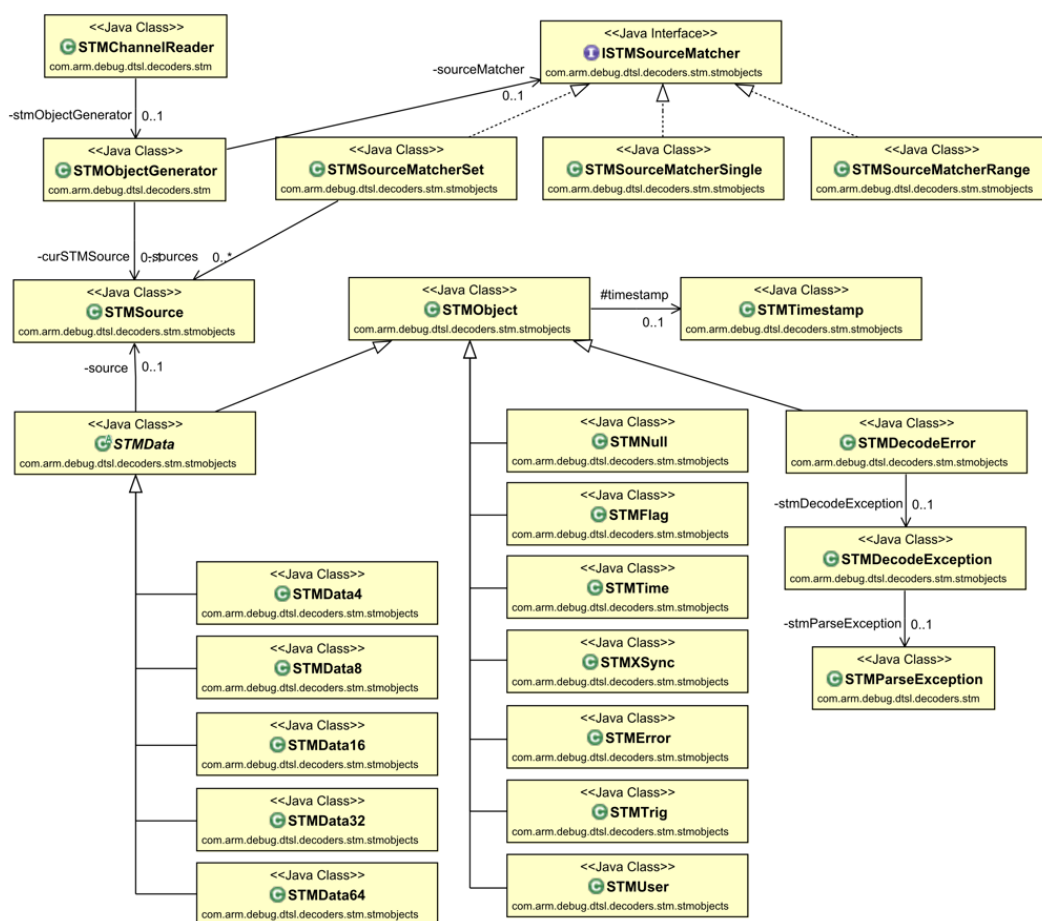


Figure 14-18 STM Object Model

14.8.9 DTSL client time-stamp synchronization support

Some trace sources have the concept of time-stamping certain trace records or events. If all trace sources are using a common, synchronized time system, then it is possible to synchronize all the client trace displays about the same real-time location. To support this, DTSL allows trace clients to request view synchronization at a particular time value. When DTSL receives such a request, it passes it on to all registered trace clients. The trace clients can receive the request and reposition their displays to show data at or around the requested time.

For a client to use the time-stamp synchronization facility, it must register an observer with the DTSL configuration. An observer is an object which implements the *ITraceSyncObserver* interface. See `ConfigurationBase.registerTraceSyncObserver` for details of how to register an observer. If, after registering an observer, the trace client requests time-stamp synchronization, then the observer receives an object. This object implements either the *ITraceSyncEvent* interface or the *ITraceSyncDetailedEvent* interface. The *ITraceSyncEvent* interface only allows reading the requested time value. The *ITraceSyncDetailedEvent* interface, however, extends this, by identifying the trace capture device and buffer location which contained the time position from the point of view of the requesting client. This might be useful to the receiving client as a hint to where they can start searching for the time value in their own trace stream.

If a client wants to request other clients to synchronize to a time value, it must use one of the `ConfigurationBase.reportTraceSyncEvent` methods.

14.9 Extending the DTSL object model

For most platform configurations, the DTSL configuration class creates standard Java DTSL components, such as CoreSight devices or ARM cores, represented as Device objects. Sometimes, the behavior of these standard components needs to be changed, or new DTSL components need to be created.

DS-5 Debugger uses the Java components that the DTSL configuration script creates. Since there is a high level of integration between Java and Jython, the DTSL configuration can create new Jython objects which extend the standard Java DTSL objects. And DS-5 Debugger can also use these Jython objects to access the target platform. This is because of the very tight integration between Java and Jython. This way of modifying behavior is straightforward if you are familiar with object oriented techniques, especially in Java. The only new technique might be the way in which a Java object can be modified by extending it in Jython. This is possible because Jython code is compiled down to Java byte code, so the system does not know whether the code was written in Java or Jython.

This section contains the following subsections:

- [14.9.1 Hooking device reset behavior on page 14-429.](#)
- [14.9.2 Adding a new trace capture device on page 14-430.](#)

14.9.1 Hooking device reset behavior

For a DSTREAM class device, the default operation for a System Reset request is to drive nSRST on the JTAG connector. On some platforms, this pin is not present on the JTAG connector, and so some other method must be used to perform the reset. Sometimes, the reset is performed by writing to some other system component, such as a System Reset Controller device. If this is not available, another approach is to cause a system watchdog timeout, which in turn causes a system reset. Whichever approach is taken, the default reset behavior must be modified.

The following code sequence is an example of this:

```

1. from com.arm.debug.dtsl.components import ConnectableDevice
2. [snip]
3. class ResetHookedDevice(ConnectableDevice):
4.     def __init__(self, root, devNo, name):
5.         ConnectableDevice.__init__(self, root, devNo, name)
6.         self.parent = root
7.
8.     def systemReset(self, resetType):
9.         ConnectableDevice.systemReset(self, resetType)
10.        # Notify root configuration
11.        self.parent.postReset()
12.
13. class DSTREAMDebugAndTrace(DTSLv1):
14.     '''A top level configuration class which supports debug and trace'''
15.
16.
17.     [snip]
18.
19.     def setupPinMUXForTrace(self):
20.         '''Sets up the IO Pin MUX to select 4 bit TPIU trace'''
21.         addrDBGMCU_CR = 0xE0042004
22.         value = self.readMem(addrDBGMCU_CR)
23.         value |= 0xE0 # TRACE_MODE=11 (4 bit port), TRACE_IOEN=1
24.         self.writeMem(addrDBGMCU_CR, value)
25.
26.     def enableSystemTrace(self):
27.         '''Sets up the system to enable trace
28.         For a Cortex-M3 system we must make sure that the
29.         TRCENA bit (24) in the DEMCR registers is set.
30.         NOTE: This bit is normally set by the DSTREAM Cortex-M3
31.         template - but we set it ourselves here in case
32.         no one connects to the Cortex-M3 device.
33.         ...
34.         addrDEMCR = 0xE000EDFC
35.         bitTRCENA = 0x01000000
36.         value = self.readMem(addrDEMCR)
37.         value |= bitTRCENA

```

```

38.         self.writeMem(addrDEMCR, value)
39.
40.     def postReset(self):
41.         '''Makes sure the debug configuration is re-instated
42.           following a reset event
43.         '''
44.         if self.getOptionValue("options.traceBuffer.traceCaptureDevice") == "DSTREAM":
45.             self.setupPinMUXForTrace()
46.             self.enableSystemTrace()

```

Line 3 declares a Jython class called *ResetHookedDevice*, which is derived from the standard DTSL Java *Device* class. The *ConnectableDevice* class implements the *IDevice* interface. One of the methods in the *IDevice* interface is *systemReset()*:

```

/**
 * Reset system
 * @param resetType
 *     one of the following values
 *     <ul>
 *     <li>{@link RDDI#RDDI_RST_PERFORM}</li>
 *     <li>{@link RDDI#RDDI_RST_ASSERT}</li>
 *     <li>{@link RDDI#RDDI_RST_DEASSERT}</li>
 *     </ul>
 * @throws DTSLException On error
 */
public abstract void systemReset(int resetType) throws DTSLException;

```

The *ResetHookedDevice* class provides a new implementation for the *systemReset* method. If the DTSL configuration class gives DTSL an instance of the *ResetHookedDevice* class, instead of an instance of the *ConnectableDevice* class, then when DS-5 Debugger calls the *systemReset* method, the new method is called instead of the old one:

```

# create core
devID = self.findDevice("Cortex-M3")
self.cortexM3 = ResetHookedDevice(self, devID, "Cortex-M3")
self.addDeviceInterface(self.cortexM3)

```

The code sequence creates a *ResetHookedDevice* instance and gives it the name “Cortex-M3”. When the debugger asks DTSL for the “Cortex-M3” device, it gives it the *ResetHookedDevice* instance. This device was derived from *Device*, so it gets all the methods that the *Device* class implements, but the old *systemReset* method is replaced with the new one.

When deriving a new class from an existing one, you can replace as many of its *IDevice* methods as you want. Your new implementation of an *IDevice* method can call the existing implementation within the *ConnectableDevice* object by calling *ConnectableDevice.<method-name>()*.

See the DTSL Java docs for full *IDevice* documentation.

The constructor for the *ResetHookedDevice* class stores a reference to the DTSL configuration which created it. When a reset is requested, it calls the existing *ConnectableDevice* reset code and tells the parent configuration that a reset has taken place. In this case, the configuration reconfigures the pin mux settings of the device to make sure that external trace is enabled.

Related concepts

[14.5.2 DTSL device objects on page 14-405.](#)

14.9.2 Adding a new trace capture device

DS-5 Debugger has built in support for reading trace data from DSTREAM, ETB, TMC/ETM and TMC/ETR devices. Adding support for a new trace capture device is not very difficult, however, and can be done entirely with DTSL Jython scripts.

The DTSL trace capture objects class hierarchy shows that all DTSL trace capture objects are derived from the *ConnectableTraceCaptureBase* class. This base class implements two interfaces, *ITraceCapture* and *IDeviceConnection*. *ITraceCapture* defines all the methods that relate to controlling and reading trace data from a capture device, and *IDeviceConnection* defines the methods

for a component that needs to be connected to. The *ConnectableTraceCaptureBase* class contains stub implementations for all the methods in both interfaces.

To create a new trace capture class:

1. Create a new class derived from the *ConnectableTraceCaptureBase* class, or the *TraceCaptureBase* class if appropriate.
2. Implement the class constructor, making sure to call the base class constructor in your implementation.
3. Override the *startTraceCapture()* and *stopTraceCapture()* methods. The default implementations of these methods throw an exception when DTSL calls them, so you must override them to avoid this.
4. Override the *getCaptureSize()* method to return the size of raw trace data in the device.
5. Override the *getSourceData()* method to return trace data for a specified trace source.
6. If your trace device requires a connection, override the *connect()*, *disconnect()*, and *isConnected()* methods.
7. In your platform DTSL Jython script, create an instance of your new trace capture device class and add it to the DTSL configuration.

The following example Jython code implements a new trace capture device which reads its trace data from an ETB dump file (the raw content of an ETB buffer). It is assumed that this code is in *FileBasedTraceCapture.py*.

```
from java.lang import Math
from com.arm.debug.dtsl.impl import DataSink
from com.arm.debug.dtsl.impl import Deformatter
from com.arm.debug.dtsl.impl import SyncStripper
from com.arm.debug.dtsl.components import ConnectableTraceCaptureBase
from com.arm.debug.dtsl.configurations import ConfigurationBase
import sys
import os
import jarray

class FileBasedTraceCaptureDevice(ConnectableTraceCaptureBase):
    """
    Base class for a trace capture device which just returns
    a fixed data set from a file. The amount of trace data captured
    is just the size of the file.
    """
    def __init__(self, configuration, name):
        """Construction
        Params: configuration
                the top level DTSL configuration (the
                class you derived from DTSLv1)
                name
                the name for the trace capture device
        """
        ConnectableTraceCaptureBase.__init__(self, configuration, name)
        self.filename = None
        self.fileOpened = False
        self.hasStarted = False
        self.trcFile = None

    def setTraceFile(self, filename):
        """Sets the file to use as the trace data source
        Params: filename
                the file containing the trace data
        """
        self.filename = filename

    def connect(self):
        """We interpret connect() as an opening of the trace data file
        """
        self.trcFile = file(self.filename, 'rb')
        self.fileOpened = True
        self.fileSize = os.path.getsize(self.filename)

    def disconnect(self):
        """We interpret disconnect() as a closing of the trace data file
        """
        if self.trcFile != None:
            self.trcFile.close()
        self.fileOpened = False
        self.fileSize = 0

    def isConnected(self):
```

```

        return self.fileOpened

    def startTraceCapture(self):
        self.hasStarted = True

    def stopTraceCapture(self):
        self.hasStarted = False

    def getMaxCaptureSize(self):
        return self.fileSize

    def setMaxCaptureSize(self, size):
        return self.getMaxCaptureSize()

    def getCaptureSize(self):
        return self.fileSize

    def getNewCaptureSize(self):
        return self.getCaptureSize()

    def hasWrapped(self):
        return True

class ETBFileBasedTraceCaptureDevice(FileBasedTraceCaptureDevice):
    """
    Creates a trace capture device which returns ETB trace
    data from a file.
    """
    def __init__(self, configuration, name):
        """Construction
        Params: configuration
                the top level DTSL configuration (the
                class you derived from DTSLv1)
                name
                the name for the trace capture device
        """
        FileBasedTraceCaptureDevice.__init__(self, configuration, name)

    def getSourceData(self, streamID, position, size, data, nextPos):
        """Reads the ETB trace data from the file
        Params: streamID
                for file formats which contain multiple
                streams, this identifies the stream for which
                data should be returned from
                position
                the byte index position to read from
                size
                the max size of data (in bytes) we should return
                data
                where to write the extracted data
                nextPos
                an array into which we set entry [0] to the
                next position to read from i.e. the position parameter
                value which will return data that immediately follows
                the last entry written into data
        """
        # We assume that size is small enough to allow to read an entire
        # data block in one operation
        self.trcFile.seek(position)
        rawdata = jarray.array(self.trcFile.read(size), 'b')
        nextPos[0] = position+size
        dest = DataSink(0, 0, size, data)
        # We assume the file contains TPIU frames with sync sequences
        # So we set up a processing chain as follows:
        # file data -> strip syncs -> de formatter -> to our caller
        deformatter = Deformatter(dest, streamID)
        syncStripper = SyncStripper(deformatter)
        syncStripper.forceSync(True)
        syncStripper.push(rawdata)
        syncStripper.flush()
        return dest.size()

```

We can use the new trace capture device in the platform DTSL Jython code:

```

from FileBasedTraceCapture import ETBFileBasedTraceCaptureDevice
[snip]
    self.etbFileCaptureDevice = ETBFileBasedTraceCaptureDevice(self, 'ETB(FILE)')
    self.etbFileCaptureDevice.setTraceFile('c:\\etbdump.bin')
    self.addTraceCaptureInterface(self.etbFileCaptureDevice)

```

We can add it to the configuration as though it were an ETB or DSTREAM device.

Related concepts

[14.5.5 DTSL trace capture objects](#) on page 14-408.

14.10 Debugging DTSL Jython code within DS-5 Debugger

When DS-5 connects to a platform, it automatically loads the platform Jython script and creates an instance of the configuration class. The Jython scripts which are shipped with DS-5 should not contain any errors, but if you create your own scripts, or make modifications to the installed scripts, then you might introduce errors. These errors have two common forms:

- Syntax or import errors
- Functional errors.

This section contains the following subsections:

- [14.10.1 DTSL Jython syntax errors on page 14-434.](#)
- [14.10.2 Errors reported by the launcher panel on page 14-434.](#)
- [14.10.3 Errors reported at connection time on page 14-435.](#)
- [14.10.4 DTSL Jython functional errors on page 14-435.](#)
- [14.10.5 Walk-through of a DTSL debug session on page 14-435.](#)
- [14.10.6 Starting a second instance of DS-5 for Jython debug on page 14-436.](#)
- [14.10.7 Preparing the DTSL script for debug on page 14-436.](#)
- [14.10.8 Debugging the DTSL code on page 14-437.](#)

14.10.1 DTSL Jython syntax errors

These can occur in two situations:

1. Attempting to change the DTSL options from within the Launcher Panel.
2. Attempting to connect the DS-5 Debugger to the platform.

14.10.2 Errors reported by the launcher panel

These errors usually appear in the area where the **Edit...** button for the DTSL options would normally appear, replacing it with a message:

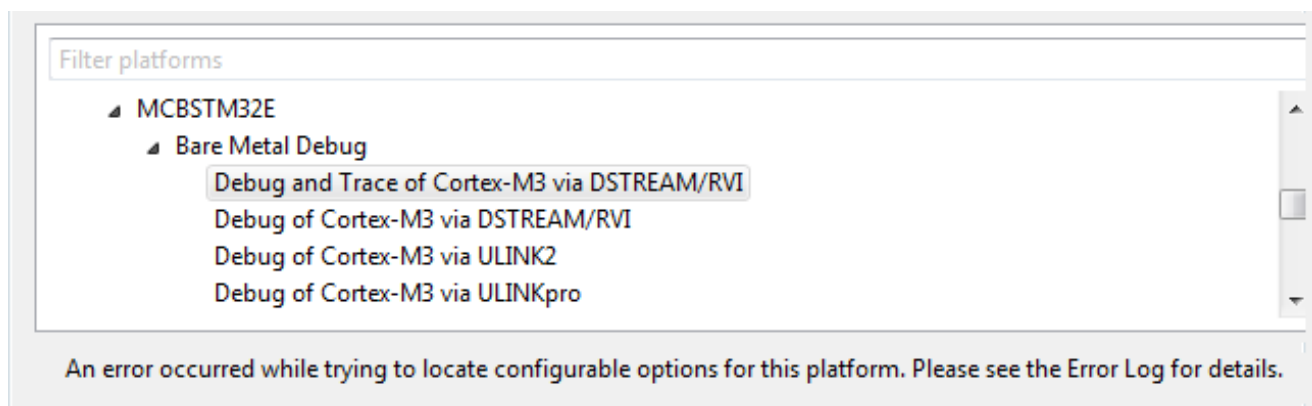


Figure 14-19 Launcher panel reporting DTSL Jython script error

To find the cause of the error, try inspecting the Error Log. If the Error Log is not visible, select **Window > Show View > Error Log** to show it.

The following is an example of some Error Log text:

```
Python error in script \\NAS1\DTSL\configdb\Boards\Keil\MCBSTM32E\keil-mcbstm32e.py at line
11: ImportError: cannot import name V7M_ETMTraceSource when creating configuration
DSTREAMDebugAndTrace
```

After resolving any issues, close and reopen the Launcher Panel to make DS-5 reinspect the Jython script. If an error still occurs, you get more entries in the Error Log. If the error is resolved, then the **Edit...** button for the DTSL options will appear as normal.

14.10.3 Errors reported at connection time

If you try to connect to a platform which contains an error in its Jython script, DS-5 displays an error dialog indicating the cause of the error:

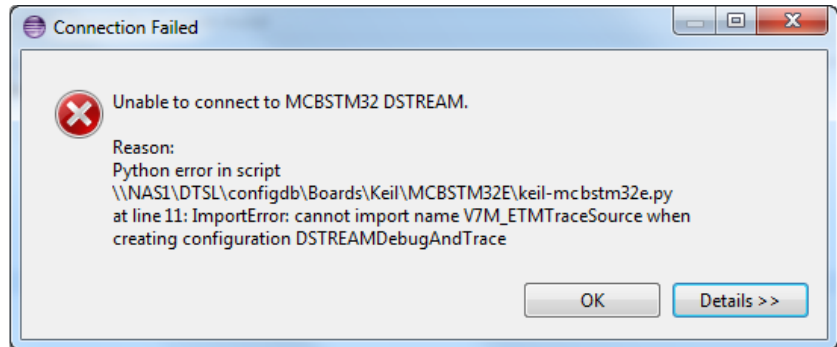


Figure 14-20 Connection Error Dialog

————— Note —————

Sometimes, the error message shown in the dialog might not be helpful, especially for run-time errors rather than syntax or import errors. DS-5 also places an entry in the Error Log, so that you can inspect the error after dismissing the error dialog. This error log entry might contain more information. You can typically find this information by scrolling down the Exception Stack Trace until you see the error reported at the point the Jython code was run.

After editing the Jython script to resolve any issues, try connecting again.

————— Note —————

You do not need to tell DS-5 that the configdb has changed when you make changes only to Jython scripts.

14.10.4 DTSL Jython functional errors

If the Jython script error you are tracking down cannot be resolved by code inspection, then you might need to use a Jython debugger. For some use cases, you can use the debugger which is built in to DS-5 as part of PyDev. Other use cases, however, display modal dialog boxes within DS-5, preventing the use of the same instance of DS-5 to debug the scripts. ARM therefore recommends that you use another instance of DS-5, or another Eclipse installation which also contains the PyDev plugin or plugins.

————— Note —————

Although you can run multiple instances of Eclipse at the same time, the instances cannot use the same Workspace.

14.10.5 Walk-through of a DTSL debug session

Make sure that DS-5 is using your intended workspace.

The debug session involves modifying a DTSL Jython script, so make sure that you are using a writeable copy of the DS-5 configdb.

Related concepts

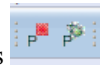
[14.3.1 Modifying DS-5 configdb on page 14-396.](#)

14.10.6 Starting a second instance of DS-5 for Jython debug

When you start a second instance of DS-5, with the first instance still running, you are asked to use a different workspace. Choose a suitable location for this second workspace.

In this second instance of DS-5, switch to the PyDev perspective. To enable the toolbar buttons that allow you to start and stop the PyDev debug server:

- Select **Window > Customize Perspective...**
- Click the **Command Groups Availability** tab.
- Scroll down through the Available Command Groups and select the PyDev Debug entry.
- Click the **Tool Bar Visibility** tab.
- Make sure that the PyDev Debug entry, and the two End Debug Server and Start Debug Server entries, are selected.



On the toolbar, you should see two new icons to stop and start the debug server.

Click the green P-bug icon to start the PyDev debug server. You should see a console view reporting the port number on which the debugger is listening (5678 by default). The DS-5 instance is ready to accept remote debug connections.

Switch to the Debug perspective. Eclipse does not automatically switch to the Debug perspective when a connection is made to the PyDev debugger. So if you do not switch to the Debug perspective yourself, then you cannot notice the connection.

14.10.7 Preparing the DTSL script for debug

When a Jython script is being debugged, it is normally launched by PyDev, and PyDev can optionally create a debug session for the script. When DS-5 launches the Jython script, however, this does not happen. This is not a problem, however, because the script itself can register with the PyDev debugger after it is launched. To do this in your script:

- Extend the import list for the script to import pydevd. If you are using a second DS-5 instance to host the PyDev debugger, then add the following to the top of the DTSL script:

```
import pydevd
```

If you are using another Eclipse (non-DS-5) to host the PyDev debugger, then import the pydevd from that Eclipse instance. Locate the pydev plugin pysrc directory and add its path to the import path before importing pydevd. For example, if the Eclipse is installed in C:\Users\john\eclipse, then the code would be as follows:

```
import sys;
sys.path.append(r'C:\Users\john\eclipse\plugins\org.python.pydev_2.7.4.2013051601\pysrc')
import pydevd
```

where pydev_xyz depends on the version of pydev installed within Eclipse.

- Insert the following line at the location where you want the PyDev debugger to gain control of the script:

```
pydevd.settrace(stdoutToServer=True, stderrToServer=True)
```

This causes a break into the debugger at that location, and redirects all standard output from the script to the debugger console. This allows you to place print statements into the script and see them in the

debugger, whereas normally DS-5 would discard any such print output. Good places to insert this statement are:

- In the constructor (`__init__`) for the DTSL configuration class.
- In the `optionValuesChanged` method.

The function documentation for the `settrace` call in pydev 2.7.4 is as follows:

```
def settrace(host=None, stdoutToServer=False, stderrToServer=False, port=5678, suspend=True,
            trace_only_current_thread=True):
    '''Sets the tracing function with the pydev debug function and initializes needed
    facilities.
    @param host: the user may specify another host, if the debug server is not in the
        same machine (default is the local host)
    @param stdoutToServer: when this is true, the stdout is passed to the debug server
    @param stderrToServer: when this is true, the stderr is passed to the debug server
        so that they are printed in its console and not in this process console.
    @param port: specifies which port to use for communicating with the server (note that
        the server must be started in the same port).
        @note: currently it's hard-coded at 5678 in the client
    @param suspend: whether a breakpoint should be emulated as soon as this function
        is called.
    @param trace_only_current_thread: determines if only the current thread will be
        traced or all future threads will also have the tracing enabled.
    ...
```

14.10.8 Debugging the DTSL code

In your main instance of DS-5 (not the PyDev debug instance), launch the connection to the platform. When the DTSL script reaches the `settrace` call, the second DS-5 instance halts the execution of the script immediately after the call. This allows you to use the PyDev debugger for tasks such as stepping through the code, examining variables, and setting breakpoints. While you are debugging, your main DS-5 instance waits for the Jython script to complete.

14.11 DTSL in stand-alone mode

DTSL is commonly used by DS-5 Debugger, both within the Eclipse environment and in the console version of the debugger. However, it can also be used in 'stand-alone' mode, completely outside of Eclipse. This allows you to use the DTSL API to take care of the target connection and configuration when writing your program. The rest of your program can concentrate on the main function of your application.

DTSL is mainly written in Java and Jython. There are therefore two kinds of stand-alone program, those written in Jython and those in Java. The `DTSLExamples.zip` file contains examples of both kinds of program, which you can look at to help you decide the best route for your application. The programs are easy to compare because they both do essentially the same things.

This section contains the following subsections:

- [14.11.1 Comparing Java with Jython for DTSL development on page 14-438.](#)
- [14.11.2 DTSL as used by a stand-alone Jython program on page 14-439.](#)
- [14.11.3 Installing the Jython example within Eclipse on page 14-439.](#)
- [14.11.4 Running the Jython program on page 14-440.](#)
- [14.11.5 Invoking the Jython program on page 14-440.](#)
- [14.11.6 About the Jython program on page 14-440.](#)
- [14.11.7 DTSL as used by a stand-alone Java program on page 14-441.](#)
- [14.11.8 Installing the Java example within Eclipse on page 14-441.](#)
- [14.11.9 Running the Java program on page 14-442.](#)
- [14.11.10 Invoking the Java program on page 14-443.](#)
- [14.11.11 About the Java program on page 14-443.](#)

14.11.1 Comparing Java with Jython for DTSL development

The advantages of Java are:

- The Javadoc for DTSL is directly available, which helps greatly when writing DTSL Java programs in environments such as Eclipse.
- Java programs seem to have faster start-up times than Jython programs.
- The Eclipse Java development environment might be considered more mature than the Python PyDev Eclipse development environment.

The advantages of Jython are:

- There are probably more people familiar with Python than with Java.
- Python is not a statically-typed language. So it is easier to write Python code without always having to create variables of specific types.

The disadvantages of Jython are:

- There is no DTSL Javadoc support, because the PyDev editor does not understand how to extract the Javadoc information from the Java `.jar` files.
- Python is not a statically-typed language, so it is hard for the PyDev editor to know the type of a variable. Using `'assert isinstance(<variable>, <type>)'` works around this to an extent, and this code appears many times in the example. After the PyDev editor sees it, it knows the type of the variable and so can provide code completion facilities. However, you still do not get access to the Javadoc within the editor. If you want to access the Javadoc, you must do it by some other method, such as through a web browser.
- Jython can be slower than Java. For example, if Jython is used as part of a trace decoding pipeline, it can significantly slow down trace processing.

14.11.2 DTSL as used by a stand-alone Jython program

The example Jython application demonstrates how to do the following:

- Create a DTSL configuration instance for the requested platform.
- Connect to a core device, such as a Cortex-M3 or other such ARM core.
- Perform the following operations:
 - Get control of the core following a reset.
 - Read and write registers on the core.
 - Read and write memory through the core.
 - Single step instructions on the core.
 - Start and stop core execution.

The example application connects to and controls the ARM core only. However, it can just as easily connect to any of the devices in the configuration, such as CoreSight components (PTM or ETB), and configure and control those devices as well.

Note

The example is a complete stand-alone application. It cannot be run when a DS-5 Debugger connection is made to the same target. However, a DS-5 Debugger Jython script can access the DTSL configuration. If you do this, take care not to interfere with the debugger.

14.11.3 Installing the Jython example within Eclipse

Make sure you have Jython installed. To download Jython, and for installation instructions, go to <http://www.jython.org/>. This document is written with reference to Jython 2.5.3, but later versions should also work.

Make sure you have the PyDev plugin installed into your Eclipse IDE. To download PyDev, and for installation instructions, go to <http://pydev.org/>. Make sure you configure PyDev to know about the Jython version you have installed.

The example project DTSLPythonExample is in the DTSLExamples.zip file. You can import DTSLPythonExample directly into your Eclipse workspace. You must also import DTSL.zip into your Eclipse workspace. After importing them, change the project configuration to refer to your DTSL library location:

1. Select the DTSLPythonExample within the Eclipse Project Explorer, right click it, and select **Properties**.
2. Select 'PyDev – PYTHONPATH' from the properties list.
3. Click the **String Substitution Variables** tab.
4. Replace the DTSL variable value with the path to your DTSL library location.

The example project also contains two launch configurations for running the program. One configuration uses the DS-5 configdb board specification, and the other refers directly to the files in the configdb. The project contains a configdb extension, which contains the Keil MCBSTM32 entries compatible with this project.

Note

If you use your own Eclipse (non DS-5) installation, then you must set the DS-5 installation location within the DS-5 preferences. This value is used within the provided launch configurations.

The readme.txt file contained within the project has more information.

Related references

[14.1 Additional DTSL documentation and files on page 14-390.](#)

14.11.4 Running the Jython program

To run the example in Eclipse:

1. Import the supplied launch configurations.
2. Modify the program arguments to refer to your installed DS-5 location.
3. Run or debug the application.

To run the example use:

- `dtslexample.bat` from Windows
- `dtslexample` from Linux.

Before running the file, edit it and change the program parameters to suit the target system you are connecting to. You might need to make the following changes:

- Change the location of `jython.bat` to match your Jython installation. DS-5 does contain part of a Jython installation, but it lacks the main `jython.bat` executable, so you must install your own.
- Change the defined location of the Eclipse workspace.
- Change the location of the DS-5 configuration database to include the database installed by DS-5 and any further extensions you require (the location within a workspace of `DTSLEExampleConfigdb\configdb` is an extension required to run the example).
- Change the connection address for the DSTREAM box to match your box. If you are using a USB connection then the code `--connectionAddress "USB"` can be left unchanged, but if you are using a TCP connection then you must change it to be of the form `--connectionAddress "TCP:<host-name|ip-address>"`, for example `--connectionAddress "TCP:DS-Tony"` or `--connectionAddress "TCP:192.168.1.32"`.
- Change the manufacturer to match the directory name of your platform in the Boards sub-directory of the DS-5 config database.
- Change the board name to match the name of the board directory within the manufacturer directory.
- Change the debug operation to match one of the activity names contained in a bare metal debug section of the `project_types.xml` file. For example:

```
<activity id="ICE_DEBUG" type="Debug">
  <name language="en">Debug Cortex-M3</name>
```

When you run the `dtslexample.py` script, it connects to the target and runs through a series of register, memory, and execution operations. By default, the script assumes that there is RAM at `0x20000000`, and that there is 64KB of it. This is correct for the Keil MCBSTM32 board. To change these values, use the `--ramStart` and `--ramSize` options.

14.11.5 Invoking the Jython program

For information on the full set of program arguments, run the program with the `--help` parameter.

There are two ways to invoke the program:

- Specify the DTSL connection properties directly, using the `{--rddiConfigFile, --dtslScript, --dtslClass, --connectionType, --connectionAddress, --device}` parameters.
- Specify the DS-5 configdb parameters (equivalent to using the Eclipse launcher) using the `{--configdb, --manufacturer, --board, --debugOperation, --connectionType, --connectionAddress}` parameters, and let the program extract the DTSL connection properties from the DS-5 configdb.

14.11.6 About the Jython program

- The main program is in the `dtslexample.py` source file.
- The project is set up to use the DTSL libraries from the DTSL Eclipse project.

- The DTSL interaction flow is as follows:
 1. Connecting to DTSL. This involves forming the *ConnectionParameters* set and passing it to the DTSL static `ConnectionManager.openConnection()` method. See the Python method `connectToDTSL()` for details.
 2. Accessing the DTSL connection configuration and locating the DTSL object with the name requested in either:
 - the `--device` parameter
 - the core specified in the DS-5 configdb platform debug operation.
 3. Connecting to the core located in step 2.
 4. Performing the operations on the core, which is represented by a DTSL object that implements the *IDevice* interface. The DTSL Javadoc lists the full set of operations available on such an object. The example uses some of the more common operations, but does not cover all of them.
 5. Disconnecting from the core.
 6. Disconnecting from DTSL.
- The *IDevice* interface is a Java interface, so there are some operations which take Java parameters such as *StringBuilder* objects. This is not a problem for Jython because you can create such Java objects within your Jython program. Most of the memory operations use Java `byte[]` arrays to transport the data. Interfacing these between Jython and Java is relatively simple, but be sure to inspect the example code carefully if you want to understand how to do this.
- The *IDevice* Java interface wraps the RDDI-DEBUG C interface thinly, which means that many of the RDDI constants are used directly rather than being wrapped. This is why the example uses constants such as `RDDI_ACC_SIZE`, `RDDI_ACC_DEF`.

14.11.7 DTSL as used by a stand-alone Java program

The example Java application shows you how to do the following:

- Create a DTSL configuration instance for the requested platform.
- Connect to a core device, such as a Cortex-M3 or other such ARM core.
- Perform the following operations:
 - Get control of the core following a reset.
 - Read and write registers on the core.
 - Read and write memory through the core.
 - Single step instructions on the core.
 - Start and stop core execution.

The example application connects to and controls the ARM core only. However, it can just as easily connect to any of the devices in the configuration, such as CoreSight components (PTM or ETB), and configure and control those devices as well.

————— **Note** —————

The example is a complete stand-alone application. It cannot be run when a DS-5 Debugger connection is made to the same target. However, a DS-5 Debugger Jython script can access the DTSL configuration. If you do this, take care not to interfere with the debugger.

14.11.8 Installing the Java example within Eclipse

The example project `DTSLJavaExample` is in the `DTSLExamples.zip` file. You can import `DTSLJavaExample` directly into your Eclipse workspace. You must also import `DTSL.zip` into your Eclipse workspace. After importing it, change the project configuration to refer to your DTSL library location:

1. Select the DTSLJavaExample within the Eclipse Project Explorer, right click it, and select **Properties**.
2. Select 'Java Build Path' from the properties list.
3. Click the **Libraries** tab.
4. Replace all the referenced DTSL\libs .jar files with new entries which have the correct paths.

The example project also contains two launch configurations for running the program. One configuration uses the DS-5 configdb board specification, and the other refers directly to the files in the configdb. The project contains a configdb extension, which contains the Keil MCBSTM32 entries compatible with this project.

Note

If you use your own Eclipse (non DS-5) installation, then you must set the DS-5 installation location within the DS-5 preferences. This value is used within the provided launch configurations.

The readme.txt file contained within the project has more information.

Related references

[14.1 Additional DTSL documentation and files on page 14-390.](#)

14.11.9 Running the Java program

To run the example in Eclipse:

1. Import the supplied launch configurations.
2. Modify the program arguments to refer to your installed DS-5 location.
3. Run or debug the application.

To run the example use:

- dtslexample.bat from Windows
- dtslexample from Linux.

Before running the batch file, edit it and change the program parameters to suit the target system you are connecting to. You might need to make the following changes:

- Change the defined location of the Eclipse workspace.
- Change the location of the DS-5 configuration database to include the database installed by DS-5.
- Change the connection address for the DSTREAM box to match your box. If you are using a USB connection then the code --connectionAddress "USB" can be left unchanged, but if you are using a TCP connection then you must change it to be of the form --connectionAddress "TCP:<host-name|ip-address>", for example --connectionAddress "TCP:DS-Tony" or --connectionAddress "TCP:192.168.1.32".
- Change the manufacturer to match the directory name of your platform in the Boards sub-directory of the DS-5 config database.
- Change the board name to match the name of the board directory within the manufacturer directory.
- Change the debug operation to match one of the activity names contained in a bare metal debug section of the project_types.xml file. For example:

```
<activity id="ICE_DEBUG" type="Debug">
  <name language="en">Debug Cortex-M3</name>
```

When you run the DTSExample.java program, it connects to the target and runs through a series of register, memory, and execution operations. By default, the program assumes that there is RAM at 0x20000000, and that there is 64KB of it. This is correct for the Keil MCBSTM32 board. To change these values, use the --ramStart and --ramSize options.

14.11.10 Invoking the Java program

For information on the full set of program arguments, run the program with the `--help` parameter.

There are two ways to invoke the program:

- Specify the DTSL connection properties directly, using the `{--rddiConfigFile, --dtslScript, --dtslClass, --connectionType, --connectionAddress, --device}` parameters.
- Specify the DS-5 configdb parameters (equivalent to using the Eclipse launcher) using the `{--configdb, --manufacturer, --board, --debugOperation, --connectionType, --connectionAddress}` parameters, and let the program extract the DTSL connection properties from the DS-5 configdb.

14.11.11 About the Java program

- The main program is in the `DTSLExample.java` source file.
- The project is set up to use the DTSL libraries from the DTSL Eclipse project.
- The DTSL interaction flow is as follows:
 1. Connecting to DTSL. This involves forming the *ConnectionParameters* set and passing it to the DTSL static `ConnectionManager.openConnection()` method. See the *connectToDTSL()* method for details.
 2. Accessing the DTSL connection configuration and locating the DTSL object with the name requested in either:
 - the `--device` parameter
 - the core specified in the DS-5 configdb platform debug operation.
 3. Connecting to the core located in step 2.
 4. Performing the operations on the core, which is represented by a DTSL object that implements the *IDevice* interface. The DTSL Javadoc lists the full set of operations available on such an object. The example uses some of the more common operations, but does not cover all of them.
 5. Disconnecting from the core.
 6. Disconnecting from DTSL.
- The *IDevice* Java interface (used for all target devices) wraps the RDDI-DEBUG C interface thinly, which means that many of the RDDI constants are used directly rather than being wrapped. That is why the example uses constants such as `RDDI_ACC_SIZE.RDDI_ACC_DEF`.